OCTOTOOLS: AN AGENTIC FRAMEWORK WITH EXTENSIBLE TOOLS FOR COMPLEX REASONING

Pan Lu*, Bowen Chen*, Sheng Liu*, Rahul Thapa, Joseph Boen, James Zou
* Equal contribution Stanford University

Website: https://octotools.github.io

Code Demo Tool Cards Visualization Video

ABSTRACT

Solving complex reasoning tasks involve visual understanding, domain knowledge retrieval, numerical calculation, and multi-step reasoning. Existing methods augment foundation models (FMs) with external tools but are restricted to specialized domains, limited tool types, or require additional training data. In this paper, we introduce OctoTools, a training-free, user-friendly, and easily extensible open-source agentic framework designed to tackle complex reasoning across diverse domains. OctoTools introduces standardized tool cards to encapsulate tool functionality, a planner for both high-level and low-level planning, and an executor to carry out tool usage. We validate OctoTools' generality across 16 diverse tasks, achieving substantial average accuracy gains of 9.3% over GPT-4o. Furthermore, OctoTools also outperforms AutoGen, GPT-Functions and LangChain by up to 10.6% when given the same set of tools. Through comprehensive analysis and ablations, OctoTools demonstrates advantages in task planning, effective tool usage, and multi-step problem solving.

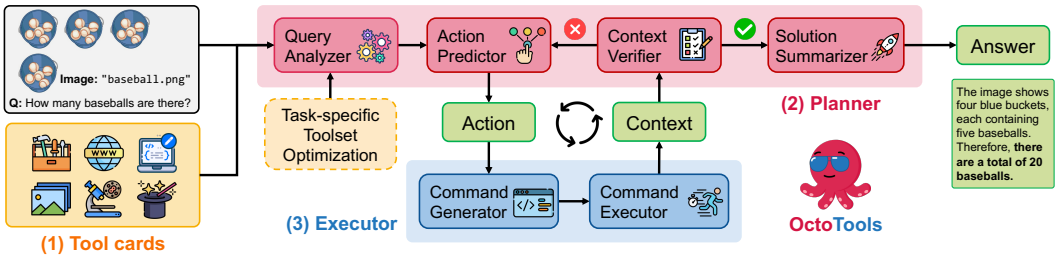


Figure 1: The overall framework of OctoTools. (1) *Tool cards* define tool-usage metadata and encapsulate heterogeneous tools, enabling training-free integration of new tools without additional training or framework refinement. (2) The *planner* governs both high-level and low-level planning to address the global objective and refine actions step by step. (3) The *executor* instantiates tool calls by generating executable commands and save structured results in the context. The final answer is summarized from the full trajectory in the context. Furthermore, the *task-specific toolset optimization* algorithm learns to select a beneficial subset of tools for downstream tasks.

1 INTRODUCTION

Foundation models (FMs) in natural language (Brown et al., 2020; Chowdhery et al., 2022; OpenAI, 2023b) have made rapid progress on tasks such as summarization, translation (Thoppilan et al., 2022), code generation (Nakano et al., 2021), and math problem solving (Shuster et al., 2022). However, complex reasoning tasks that involve multiple steps, logical decomposition, or specialized domain knowledge remains challenging. For example, solving a visual riddle may require fine-grained image understanding and text-based reasoning, while a math or chemistry question can require thorough computations or domain expertise. Existing prompting methods often fail to orchestrate these varied processes into a coherent chain of reasoning (Yao et al., 2022).

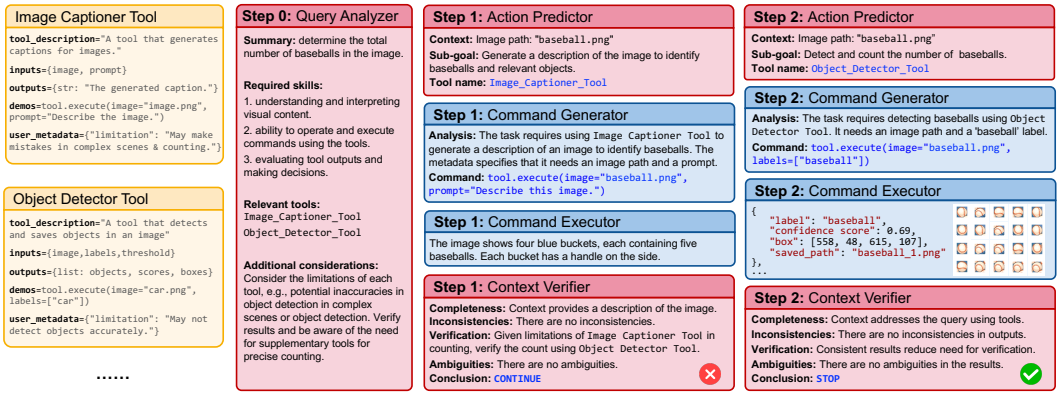


Figure 2: The demonstration of the self-contained example from Figure 1. We visualize the tool cards for selected tools, the initial plan, and two steps in which the planner and the executor orchestrate low-level planing and tool usage. See §H.1 for details and §H for more examples.

A promising direction to address these challenges is to *augment* FMs with *external tools*. By offloading specialized subtasks (e.g., web queries, Python-based calculations, and specialized scientific tools) to dedicated modules, FMs can focus on higher-level planning and synthesis. Several frameworks have explored such tool usage, from those relying on extensive supervised data and fine-tuning (Schick et al., 2023; Liu et al., 2023), to static solutions without refinement (Lu et al., 2023), and those limited to one specialized domain of tools (Nakano et al., 2021; Tao et al., 2023; Hu et al., 2024). Although these methods perform well on specific tasks, they still face challenges that hinder general widespread use. Many require substantial training with curated data, which limits their adaptability to new domains. Others are designed for a particular domain (Bran et al., 2023; Kang & Kim, 2024; Li et al., 2024a; Schmidgall et al., 2024) or cannot easily support multi-step problem-solving (Lu et al., 2023), restricting their generality.

In this paper, we propose OctoTools, a *training-free* (i.e., it does not require updating model weights), *user-friendly*, and *extensible* agentic framework for tackling *complex reasoning* tasks across diverse domains (Figure 1 and Figure 2). A key feature of OctoTools is the concept of *tool cards*, standardized wrappers that encapsulate heterogeneous tools (e.g., Python calculators, web search APIs, and domain-specific modules), along with metadata such as input-output formats, usage constraints, and best practices that delineate ideal use cases. This standardized design enables easy integration, replacement, or expansion of tools—unlike approaches requiring painstaking re-engineering for each new tool (Lu et al., 2023; Hu et al., 2024).

Building on these tool cards, OctoTools employs a dedicated *planner* that governs both high-level and low-level planning. Given a user query, the planner proposes a tentative global plan for how various tools might be employed. At each step, it generates a text-based *action* (including sub-goals and tool selection) conditioned on the evolving *context*. A separate *executor* instantiates tool calls by converting this textual action into an executable command, running the corresponding tool, and updating the context with the results. By separating strategic planning from command generation, OctoTools reduces errors and increases transparency.

An additional challenge in agentic systems is determining which subset of tools to enable for a given domain. Although providing many tools can be beneficial, enabling them all may introduce noise or slow performance (Lumer, 2024; Fore et al., 2024; Paramanayakam et al., 2024). To address this, we propose a lightweight *toolset optimization* algorithm that identifies a more useful subset of tools for each task based on validation performance, ultimately improving both accuracy and efficiency.

While recent general agent frameworks also allow FMs to use external tools autonomously, they often focus on high-level abstractions (LangChain, 2024), limited observability of intermediate decisions (OpenAI, 2023a), or multi-agent collaboration features (AutoGen, 2024), with less emphasis on enhancing complex reasoning and *quantitatively* benchmarking downstream task performance. In contrast, we systematically evaluate the entire agentic workflow of OctoTools across diverse tasks, providing in-depth analyses of when and how tool-based reasoning succeeds in complex reasoning.

We conduct large-scale experiments across 16 diverse reasoning benchmarks, spanning general vision, mathematical, scientific, medical, and agentic domains. As summarized in Figure 3, OctoTools substantially outperforms other baselines, achieving an average accuracy gain of 9.3% over zero-shot prompting by GPT-4o and 7.7% over chain-of-thought (CoT) prompting, as well as up

to 10.6% improvement compared to existing agentic frameworks when given the same tools (AutoGen, 2024; OpenAI, 2023a; LangChain, 2024). Detailed analyses show that OctoTools effectively combines multi-step planning and specialized tool usage, with each dimension providing distinct improvements. For tasks requiring intricate calculations or specialized knowledge, we found tool usage is particularly beneficial; for tasks requiring reasoning decomposition, we found multi-step planning offers significant gains.

Furthermore, our ablation studies offer insights into OctoTools's performance under different conditions. Overall, the average accuracy tends to improve as the maximum number of steps increases. Without any toolset optimization, simply enabling all tools in the toolset yields 57.4% accuracy, which still surpasses the setup with only the base tool by 3.5%, suggesting a degree of generalization as the toolset expands. Learning the optimal toolset for specific tasks raises the overall performance to 58.9%, indicating the benefit of further optimization. ,When using a weaker FM (GPT-4o-mini) as the base engine, OctoTools maintains a strong average gain of 7.1%.

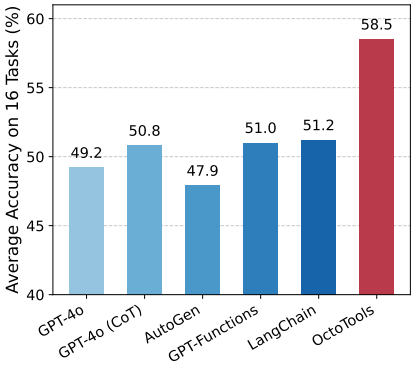


Figure 3: Accuracy scores across 16 benchmarks. OctoTools achieves an average accuracy gain of 9.3% over GPT-4o and 7.3% over LangChain.

Our contributions are as follows: (1) We propose OctoTools, a training-free, extensible agentic framework that enables FMs to call external tools in multiple steps, without the need for additional training or fine-tuning. (2) We introduce a comprehensive *planner-executor* paradigm with standardized *tool cards*, which can be easily customized or expanded for new domains. (3) We conduct large-scale experiments on 16 diverse benchmarks and show that OctoTools improves performance by a sizable margin compared to baseline prompting and other agentic frameworks. (4) We provide in-depth analyses and ablations on how multi-step reasoning and tool usage contribute to performance, offering practical guidance for future agent development.

2 Related Work

Tool-Augmented Foundation Models. A promising direction for enhancing FMs involves offloading specialized subtasks to external tools such as search engines (Komeili et al., 2022; Thoppilan et al., 2022; Shuster et al., 2022; Yao et al., 2022), web browsers (Nakano et al., 2021), calculators (Cobbe et al., 2021; Thoppilan et al., 2022), translation systems (Thoppilan et al., 2022), or Python interpreters (Gao et al., 2023). Broadly, these methods either rely on large-scale fine-tuning or human supervision to teach FMs how to invoke tools (Schick et al., 2023; Komeili et al., 2022; Nakano et al., 2021; Thoppilan et al., 2022) or use few-shot prompts for single tools in narrowly defined tasks (Yao et al., 2022; Lazaridou et al., 2022; Gao et al., 2023). In contrast, OctoTools is a *training-free* framework that integrates diverse tools through standardized *tool cards* and employs a planner-executor paradigm to manage multi-step reasoning. Because new tools can be introduced without re-training, OctoTools offers a more *extensible* and *modular* approach to tool usage.

Foundation Model Agents. A growing body of work leverages FMs as autonomous agents that make decisions and invoke tools in multi-step workflows. Some agents use closed-source models with hand-engineered prompts (Chen et al., 2023; Wang et al., 2024a), while others fine-tune FMs on curated data that learn when and how to call tools (Liu et al., 2023; Tao et al., 2023; Zhang et al., 2024). These frameworks often face limitations. For example, although specialized agent frameworks achieve strong performance in particular domains (e.g., chemistry (Bran et al., 2023), vision (Li et al., 2024a; Hu et al., 2024), materials science (Kang & Kim, 2024), or medical imaging (Schmidgall et al., 2024)), they typically lack generality across diverse tasks. Additionally, some systems are constrained by narrow capabilities with static planning (Lu et al., 2023) and multi-step reasoning (Hu et al., 2024). Recently, general-purpose agent platforms such as AutoGen (AutoGen, 2024), GPT-Functions (OpenAI, 2023a), and LangChain (LangChain, 2024) have emerged, but they have seen less emphasis on complex reasoning and rigorous benchmarking across diverse downstream tasks. In contrast, OctoTools combines the flexibility of such platforms with a dedicated planner and executor to handle multi-step decision-making.

3 THE OCTOTOOLS FRAMEWORK

We propose OctoTools, an open-source, versatile, and user-friendly agent-toolbox framework for complex reasoning tasks, as illustrated in Figure 1 and Figure 2. Given a user query $q \in \mathcal{Q}$ and a pretrained language model $\text{LM}_\theta(\cdot)$, a naive approach would generate an output directly as $y \sim \text{LM}_\theta(q)$, providing a single-step response. In contrast, our OctoTools framework introduces a structured, multi-step process that leverages external tools to tackle queries effectively.

Specifically, OctoTools contains a set of *tools* $\mathcal{D} = \{d_i\}_{i=1}^n$ and associated metadata $\mathcal{M} = \{m_i\}_{i=1}^n$, where n is the number of available tools. Given a query, a *planner* (based on a language model) first generates a *tentative plan* from a high-level perspective, indicating how these tools can be used to address the query, which forms the initial *context* s_0. From this plan, the planner determines the initial *action* a_1 for tool usage, specifying which tool d_1 to use, the relevant context, and a sub-goal. An *executor* (also powered by a language model) then converts the planner's text-based action a_1 into a machine-executable *command* o_t, which is run to obtain intermediate results r_1. These results, along with the original action, update the context to $s_1 := (a_1, o_1, r_1)$. This process constitutes one step in our framework.

This process repeats, with the planner iteratively refining its actions based on the evolving context until it either finds a complete solution or inference limits (*e.g.*, time or steps) are reached. After T steps, the framework produces a full *trajectory* $(s_0, s_1, \ldots, s_T)$, stored in a structured manner in the context. The planner then uses this trajectory to generate the final solution to the original query.

OctoTools provides a robust and effective framework for solving complex tasks through sub-goal decomposition and systematical tool usage. Standardized *tool cards* encapsulate functionality (§3.1), the *planner* orchestrates both high-level and low-level task planning (§3.2), and the *executor* instantiates tool calls for each sub-goal (§3.3). The following sections detail the logic of each component, with implementation details provided in §E.

3.1 TOOL CARDS

To enable seamless interaction among tools, the planner, and the executor, the toolbox serves as a fundamental building block of OctoTools. In contrast to previous work (Lu et al., 2023; Hu et al., 2024) that rely on careful adaptions and tuning to support new tools, OctoTools leverages *tool cards*, which encapsulate the specifications of tools in a structured manner.

Each *tool card* represents a tool d alongside its metadata m (as shown in Figure 2). This metadata includes the tool's name, version, input and output types, and command demonstrations. It may contain additional usage constraints (*e.g.*, user-defined "*limitations*" and "*best practices*"), which provide developers' insights to guide the planner and executor. For example, `Image_Captioner_Tool` includes "*it may make mistakes in complex scenes*" and "*consider using other tools for verification*" in the user metadata (§F.4). See §F for details for featured tools.

To ensure consistent interactions, every tool card implements two standard functions. `execute()` encapsulates the tool's primary functionality, e.g., generating code snippet or performing object detection. Executed results are stored in a structured format to support different output types, e.g., generated code, detected objects, stored files (see §F). `get_metadata()` allows the planner and executor to dynamically evaluate the tool's capabilities and understand usage constraints.

The design of each tool card is modular relative to the framework, enabling users to integrate diverse tools without modifying the underlying framework or agent logic. New tool cards can be added, replaced, or updated with low effort. Consequently, OctoTools remains robust and extensible, even as tasks grow in complexity or scope.

3.2 PLANNER MODULE

Planner initialization. The planner inspects the *toolbox* and loads each *tool card* as defined in §3.1. Formally, it constructs a set $\{d_i\}_{i=1}^n$ of n available tools, each with metadata m that describes its input-output schema and usage constraints. Rather than enabling the full toolset, one might enable a subset $\mathcal{D}^* = \{d_i\}_{i=1}^k$ chosen based on expert insights or optimized with a small set of examples (See §3.4 and §5.2 for details and experimental study).

Query analysis and action prediction. Given a query q, the planner formulates a *high-level*, tentative plan for tool usage based on its initialization. As shown in Figure 1 and Figure 2, the high-level plan summarizes the query objective, analyzes the required skills, identifies relevant tools, and includes additional considerations to highlight the need for verification. The *high-level* task plan provides a global perspective on the final objective, ensuring that each subsequent sub-goal remains aligned with the original query.

Subsequently, at step t, an *action predictor* produces an action a_t that combines the planner-suggested sub-goal (e.g., *"detect baseballs in the image"*), the selected tool d (e.g. `Object_Detector`), and the relevant context. This *low-level* plan refines and executes each sub-goal in real time, adapting to new information or feedback at each step.

Context verification and solution summarization. After each command is executed, a *context verifier* checks whether the problem can be solved given the current context. It verifies completeness (e.g., whether all sub-goals are satisfied) and identifies any ambiguities. If the problem remains incomplete, the planner continues the next iteration of the cycle by predicting the next action a_{t+1}.

Once the verifier concludes that the query has been solved, a separate *solution summarizer* compiles the final answer from the trajectory $(s_0, s_1, \ldots, s_T)$. This stage integrates intermediate tool outputs, traces reasoning steps, and presents a concise, user-friendly summary as the final solution.

3.3 Executor Module

Command prediction. Prior work (Lu et al., 2023; Hu et al., 2024) often expects a single language model both for planning each step (i.e., which tool to use) and for generating the corresponding executable command. This dual responsibility can overload the model and lead to errors, especially when dealing with complex or environment-specific code (Bran et al., 2023; Li et al., 2024b; Ji et al., 2024). To mitigate these issues, OctoTools introduce a *command generator* that interprets the planner's text-based actions and produces executable code.

Given the action a_t predicted by the planner, the *command generator* (powered by a language model) creates a low-level command o_t in the form of an executable Python script, which calls the tool d_t with necessary inputs and performs any required data preparation. This step bridges the abstract action specified in a_t and the concrete tool call. By separating decision-making from code generation, each component of the system can concentrate on its specialized role.

Command execution. Once an executable command is generated, it must be run in an environment that may involve dependencies, external libraries, or resource access (e.g., file systems). Directly coupling execution with planning poses security and maintainability challenges, especially if the planner is not capable of managing code execution details.

In OctoTools, an *command executor* runs the generated command o_t in a Python environment, obtaining a result r_t. This may include tool outputs, logs, or error messages. The executor then adds the current context of this step $s_t := (a_t, o_t, r_t)$ to the agent's current trajectory $(s_0, s_1, \ldots, s_{t-1})$. The trajectory preserves a clear history of the actions taken, the code produced, and the results obtained.

3.4 Task-specific Toolset Optimization

The OctoTools toolbox contains a diverse set of tools covering different modalities and skills. By leveraging structured tool cards and robust planning capabilities, OctoTools demonstrates strong generality when all available tools are enabled across different tasks (see §3.4). However, when a small set of validation examples are available for a task, configuring a *task-specific* subset of tools can further enhance efficiency and effectiveness.

To this end, we propose an automated algorithm to optimize the toolset configuration for each task. Given n available tools in the toolbox, the total number of possible subsets is $O(2^n)$, which is prohibitively large. To make this tractable, we employ a greedy search strategy that reduces the complexity to $O(n)$. Our approach proceeds in three stages.

Stage 1: Baseline setup. We first establish a baseline performance by enabling the base toolset in the toolbox, denoted $\mathcal{D}_{\text{base}}$. This base set represents a minimal starting toolset, which can be pre-defined by the user.

Stage 2: Individual tool evaluation. Next, we measure each candidate tool d_i by enabling it alongside the base toolset. For each d_i, we form an augmented subset $\mathcal{D}_i = \mathcal{D}_{\text{base}} \cup \{d_i\}$ and compute the accuracy difference on the set of validation examples

$$\Delta_{d_i} = \text{Acc}(\mathcal{D}_i) - \text{Acc}(\mathcal{D}_{\text{base}}). \tag{1}$$

If $\Delta_{d_i} > 0$, we consider the tool d_i beneficial for the target task.

Stage 3: Optimized toolset selection. Finally, we aggregate all tools that yield positive improvements and combine them with the default set to form the optimized toolset:

$$\mathcal{D}^* = \mathcal{D}_{\text{base}} \cup \{ d_i \mid \Delta_{d_i} > 0\}. \tag{2}$$

This set contains all tools that individually demonstrate a performance gain over the baseline, ensuring a customized yet efficient configuration for the downstream task. While this selection does not guarantee global optima, we observe overall improvements over simply using all tools (see §5.2).

4 Experiments

4.1 Experimental Setups

To demonstrate the generality of our OctoTools framework, we conduct comprehensive evaluations on 16 diverse benchmarks spanning two modalities, five domains, and four reasoning types, as shown in Table 1. These benchmarks encompass a wide range of complex reasoning tasks, including visual understanding, numerical calculation, knowledge retrieval, and multi-step reasoning.

For general visual reasoning, we include AlgoPuzzleVQA (Ghosal et al., 2024), Hallusion-VD (Guan et al., 2024), PuzzleVQA (Chia et al., 2024), and VQA 2.0 (Goyal et al., 2017). For mathematical reasoning, we use Game of 24 (Lile, 2024), Omni-MATH (Gao et al., 2024), CLEVR-Math (Lindström & Abraham, 2022), and MathVista (Lu et al., 2024b). For scientific reasoning, we adopt GPQA (Rein et al., 2023), MMLU-Pro (Wang et al., 2024b), and SciFIBench (Roberts et al., 2024). To evaluate models in specialized medical domains, we test on MedQA (Jin et al., 2021), PathCLS (Sun et al., 2025), PathVQA (He et al., 2020), and SLAKE (Liu et al., 2021) Additionally, we incorporate GAIA-Text, a textual subset of the GAIA (Mialon et al., 2023) benchmark designed to evaluate agentic frameworks with tool-calling capabilities. See §C.1 for details.

For each benchmark, we sampled 100 examples to construct a validation set for toolset optimization (§3.4) and ablation studies (§5), and set aside a held-out test set of 200 examples for final evaluation. To mitigate randomness, we report the average accuracy with standard deviation across three trials for all experiments, including ablation studies. Details on experimental setups are provided in §C.

We created a diverse array of tools in the toolbox for our experiments. The base toolset consists of `Generalist_Solution_Generator`, a tool built on top of the base FM that takes as input a specialized prompt generated by the executor, such as "*Describe the image in detail*" or "*Generate stepwise solutions for the math problem*". This tool allows for general step-by-step reasoning without requiring external tools but lacks domain specificity. The toolbox also includes image perception tools such as `Image_Captioner`, web search APIs like `Google_Search`, a code generator `Python_Code_Generator`, and specialized tools like `Path_Generalist_Classifier` for pathology image classification. Among them, `Relevant_Patch_Zoomer` is a unique tool that takes a textual query and returns zoomed-in quarter patches, which provide localized details for fine-grained visual reasoning scenarios. See §C.2 and §F for more details.

4.2 Main Results

We compare the performance of our framework after toolset optimization (denoted as OctoTools) against three baselines: (1) zero-shot, where the base FM directly answers queries without additional prompting, (2) chain-of-thought (CoT), where the base FM is prompted to "Think step by step" to generate step-by-step reasoning, and (3) OctoTools$_{\text{base}}$, which uses only the base

Datasets	Modality	Domain					0-shot	CoT	OctoTools$_{base}$	OctoTools	Δ (0-shot)	Δ (CoT)
AlgoPuzzleVQA	Vision	General	✓			✓	41.3 ±0.3	42.7 ±1.0	44.0 ±0.9	**48.7** ±0.3	+7.4	+6.0
Hallusion-VD	Vision	General	✓				52.0 ±1.0	53.3 ±2.1	59.0 ±0.0	**63.3** ±2.9	+11.3	+10.0
PuzzleVQA	Vision	General	✓			✓	52.2 ±1.0	54.0 ±1.3	59.3 ±0.8	**61.0** ±0.5	+8.8	+7.0
VQA 2.0	Vision	General	✓			✓	50.3 ±1.0	48.7 ±0.3	47.2 ±0.8	**54.5** ±0.0	+4.2	+5.8
Game of 24	Text	Mathematical		✓		✓	22.2 ±2.5	33.3 ±1.5	37.8 ±3.3	**44.7** ±2.8	+22.5	+11.4
Omni-MATH	Text	Mathematical		✓	✓	✓	27.0 ±0.0	29.3 ±1.3	30.2 ±0.6	**32.2** ±0.8	+5.2	+2.9
CLEVR-Math	Vision	Mathematical	✓	✓			64.5 ±3.0	75.2 ±1.5	68.8 ±0.8	**79.0** ±0.9	+14.5	+3.8
MathVista	Vision	Mathematical	✓	✓	✓	✓	59.3 ±0.8	59.5 ±1.5	63.0 ±1.3	**64.3** ±1.0	+5.0	+4.8
GPQA	Text	Scientific		✓	✓	✓	53.7 ±1.9	52.3 ±2.0	53.7 ±2.5	**54.7** ±1.3	+1.0	+2.4
MMLU-Pro	Text	Scientific			✓	✓	71.7 ±0.3	70.3 ±0.6	71.5 ±1.3	**73.7** ±1.3	+2.0	+3.4
SciFIBench	Vision	Scientific	✓		✓		72.5 ±0.0	75.0 ±0.9	77.3 ±0.8	**78.3** ±0.6	+5.8	+3.3
MedQA	Text	Medical			✓		84.5 ±1.0	84.8 ±0.6	**92.8** ±0.6	91.5 ±1.8	+7.0	+6.7
PathCLS	Vision	Medical	✓		✓		36.0 ±0.9	37.5 ±1.8	37.0 ±1.8	**58.2** ±1.3	+22.2	+20.7
PathVQA	Vision	Medical	✓		✓	✓	32.0 ±1.8	27.8 ±1.8	43.5 ±2.6	**49.2** ±1.2	+17.2	+21.4
SLAKE	Vision	Medical	✓		✓	✓	59.3 ±1.0	60.3 ±0.6	59.2 ±1.8	**63.8** ±1.4	+4.5	+3.5
GAIA-Text	Text	Agentic		✓	✓	✓	8.7 ±0.8	8.4 ±0.5	9.7 ±0.9	**18.4** ±1.2	+9.7	+10.0
Average (%)	-	-	-	-	-	-	49.2	50.8	53.4	**58.5**	**+9.3**	**+7.7**

Table 1: Main results across 16 benchmarks spanning different modalities, domains, and required reasoning skills: visual understanding (), numerical calculation (), knowledge retrieval (), and multi-step reasoning (). Performance gains (Δ) are computed for OctoTools relative to both zero-shot (0-shot) and chain-of-thought (CoT) baselines, with OctoTools achieving 58.5% average accuracy and improvements of 9.3% and 7.7% respectively.

tool without external integrations. Unless otherwise specified, all results presented below use `gpt-4o-2024-08-06` as the base model.

Table 1 and Figure 3 summarize performance across all 16 benchmarks. OctoTools achieves consistent gains, outperforming zero-shot and CoT baselines by 9.3% and 7.7% on average, respectively. OctoTools$_{base}$ also demonstrates improvements over zero-shot (4.2%) and CoT (2.6%), indicating that our framework's step-by-step reasoning contributes significantly to performance, independent of external tool integration. Figure 11 visualizes gains over zero-shot. Detailed analyses of these gains are presented in §4.4.

4.3 Comparisons with Other Agent Frameworks

We compare three commonly used general AI agent frameworks: GPT-Functions (OpenAI, 2023a), LangChain (LangChain, 2024), and AutoGen (AutoGen, 2024). GPT-Functions enables GPT-4o to call user-specified tools via function calling. LangChain is a framework providing multi-agent collaboration, long-term memory, and tool usage. AutoGen is a recent agentic framework that creates multiple autonomous agents with tool usage.

For a standardized comparison of each system's ability to plan and use tools over multiple steps, we configure all agent frameworks, including OctoTools, to use the same underlying model (GPT-4o) and hyperparameters. They share the same toolset (with the same implementations of tools), a maximum reasoning budget of 10 steps, and a time budget of 300 seconds. See §C for further details.

Datasets	AutoGen	GPT-Functions	LangChain	OctoTools
AlgoPuzzleVQA	44.0 ±1.0	44.5 ±0.5	42.7 ±2.8	**48.7** ±0.3
Hallusion-VD	52.7 ±4.7	57.0 ±1.7	53.7 ±3.1	**63.3** ±2.9
Puzzle VQA	40.0 ±2.3	52.5 ±2.8	53.5 ±7.8	**61.0** ±0.5
VQA 2.0	46.0 ±1.0	45.5 ±0.9	54.0 ±1.0	**54.5** ±0.0
Game of 24	24.2 ±2.4	34.5 ±2.3	18.3 ±4.1	**44.7** ±2.8
Omni-MATH	28.5 ±1.3	22.8 ±1.8	29.7 ±0.6	**32.2** ±0.8
CLEVR-Math	69.5 ±3.9	71.2 ±1.0	69.2 ±4.6	**79.0** ±0.9
MathVista	24.7 ±2.5	54.5 ±2.0	55.7 ±0.3	**64.3** ±1.0
GPQA	48.7 ±2.9	45.8 ±2.6	52.2 ±1.2	**54.7** ±1.3
MMLU-Pro	65.0 ±2.5	65.8 ±2.4	70.3 ±1.2	**73.7** ±1.3
SciFIBench	70.0 ±2.2	68.8 ±3.2	77.0 ±0.5	**78.3** ±0.6
MedQA	83.7 ±2.8	84.8 ±0.3	73.7 ±0.6	**91.5** ±1.8
PathCLS	58.0 ±1.3	58.2 ±0.6	56.3 ±1.3	**58.2** ±1.3
PathVQA	42.7 ±0.8	42.8 ±2.3	45.7 ±4.4	**49.2** ±1.2
SLAKE	62.2 ±1.8	59.7 ±1.9	59.3 ±0.8	**63.8** ±1.4
GAIA-Text	6.3 ±0.8	7.9 ±0.8	7.6 ±1.2	**18.4** ±1.2
Average (%)	47.9	51.0	51.2	**58.5**

Figure 4: Comparison with agent frameworks using the same toolbox. OctoTools achieves superior performance with an average accuracy of 58.5%, outperforming the next best baseline by 7.3%. Results are averaged over three trials.

Table 4 and Figure 12 show the performance of OctoTools compared to other agent frameworks across all 16 benchmarks. Overall, OctoTools outperforms other agent frameworks, achieving an average accuracy gain of 10.6% over AutoGen, 7.5% over GPT-Functions, and 7.3% over LangChain. See §4.4 for detailed analysis.

4.4 Analysis of Performance Gains

OctoTools includes three aspects for improvement in complex reasoning: *task planning*, *external tool calling*, and *multi-step problem solving*. Our analysis disentangles these benefits as follows.

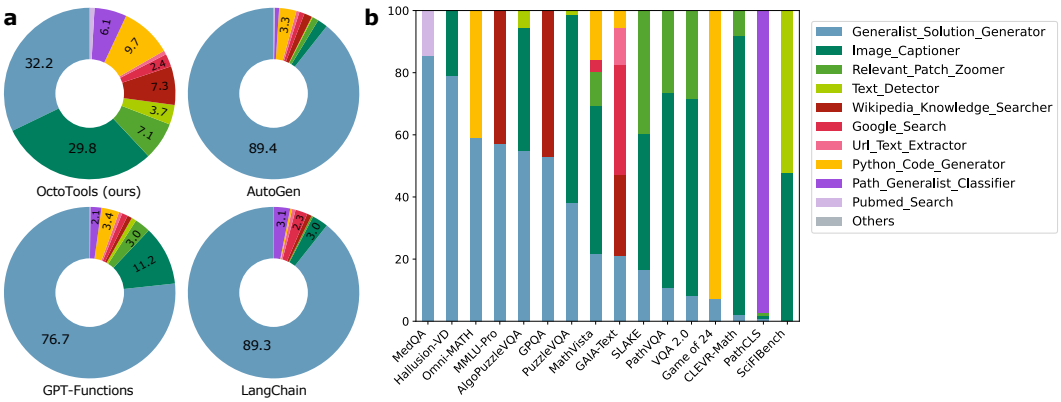


Figure 5: **a.** Tool usage distribution in our OctoTools framework and agent baselines by averaging results from 16 tasks. **b.** Tool usage distribution across 16 tasks in OctoTools. OctoTools takes advantage of different external tools to address task-specific challenges.

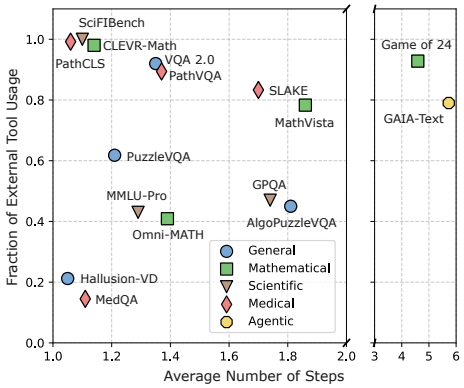


Figure 6: Benchmark distribution across average # steps and fraction of external tool usage.

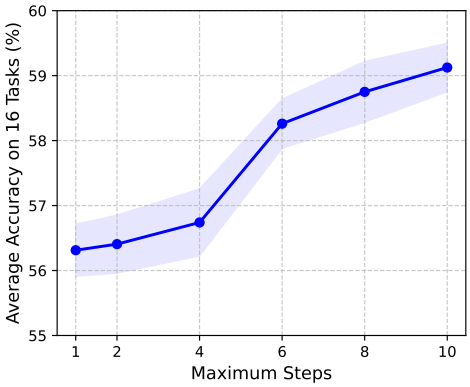


Figure 7: Average accuracy across 16 benchmarks w.r.t. maximum allowed reasoning steps.

Tool usage distribution. Figure 5 (a) shows the tool usage distribution for OctoTools and the agent baselines, averaged over 16 tasks. OctoTools takes advantages of *task planning* by calling both a base tool to decompose the query into subtasks (32.2%) and external specialized tools (67.8%) for reasoning skills such as fine-grained image understanding, domain knowledge access, precise calculation, and other domain-specific tasks. In contrast, the other agent baselines exhibit limitations in task planning for external tool usage, with external tool usage rates of 10.6% for AutoGen, 23.3% for GPT-Functions, and 10.7% for LangChain.

Figure 5 (b) illustrates the tool usage distribution of OctoTools across the 16 tasks. OctoTools adapts to each task by selecting the most suitable tools. For instance, OctoTools employs five different tools to address the diverse challenges in GAIA-Text. In MathVista, a benchmark featuring mathematical reasoning problems in visual context, it uses `Relevant_Patch_Zoomer` for local vision perception, `Google_Search` for web queries, and `Python_Code_Generator` for precise calculations.

External tool usage *v.s.* multiple steps. Figure 6 shows how tasks are distributed based on the average number of steps and the fraction of external tool usage in OctoTools. Tasks with a high average number of steps indicates that *multi-step problem solving* is valuable, while a high proportion of external tool usage highlight the benefits of *external tool calling*. Notably, two tasks score highly on both dimensions: Game of 24, which frequently calls `Python_Code_Generator` to explore and verify arithmetic expressions that evaluate to 24, and GAIA-Text, which requires multiple tools to address a variety of underlying skills.

5 Ablation Study

5.1 Number of Maximum Allowed Steps

We explore how the behavior and performance of OctoTools change under different maximum step limits. We report the average over three trials, with results summarized in Figure 7. Overall, performance tends to improve as the maximum number of steps increases, highlighting the benefit of

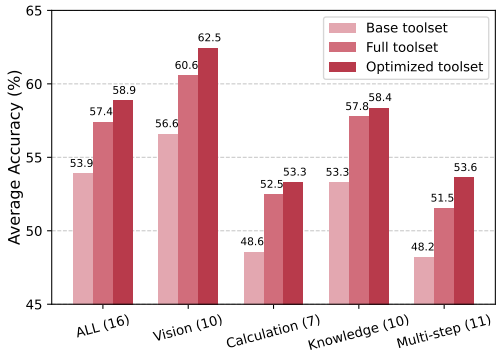


Figure 8: Performance under three toolset strategies in OctoTools across all 16 tasks and various categories (the number in parentheses indicates # tasks in each category).

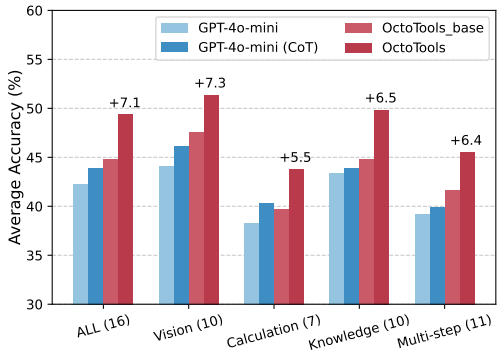


Figure 9: Performance of OctoTools on 16 tasks and various categories using a weaker FM, GPT-4o-mini. The number in parentheses indicates # tasks in each category.

longer chains of multi-step reasoning. Detailed results and analysis for individual tasks are provided in §D. Additionally, we found that averaged across 16 tasks, running OctoTools with GPT-4o typically takes less than $5 for 100 queries with 10 maximum allowed steps.

5.2 Toolset Optimization

To investigate the benefits of our toolset optimization algorithm (presented in §3.4 and Algorithm 1), we evaluate OctoTools under three toolset strategies: (1) OctoTools$_{\text{base}}$, i.e., the setup with only the base `Generalist_Solution_Generator` tool, (2) OctoTools with the *full* toolset, where all possible tools are enabled, and (3) OctoTools with the *optimized* toolset. Figure 8 presents results across all 16 tasks and different reasoning categories.

On average, the optimized toolset achieves 58.9% accuracy, outperforming the OctoTools$_{\text{base}}$ (53.9%) by 5.0%. Meanwhile, simply enabling all tools yields 57.4% accuracy, which still outperforms OctoTools$_{\text{base}}$ by 3.5%. Although toolset optimization provides the higher overall performance, our framework is sufficiently versatile that enabling all tools also offers a substantial improvement, helping ensure generalization as we scale up the toolset.

While enabling all available tools might appear advantageous, prior work (Lumer, 2024; Fore et al., 2024; Paramanayakam et al., 2024) has shown that it can sometimes degrade performance. Our findings suggest that external feedback (e.g., performance on a validation set or expert insights) helps identify when a certain tool improves performance. Additionally, selectively enabling tools avoids unnecessary overhead and improves efficiency. A promising next step is to explore a query-based approach that dynamically refines the toolset for each query.

5.3 Using a Weaker FM

Additionally, we evaluate our framework using a weaker FM (GPT-4o-mini) as the base engine. Similar to the analysis with GPT-4o, we compare OctoTools against the zero-shot and CoT baselines on the validation sets of 16 benchmarks. As shown in Figure 9, OctoTools generally outperforms the baselines across all 16 tasks and various categories. Task-specific analyses are provided in §D.

6 Conclusion

In this paper, we introduced OctoTools, a training-free, extensible agentic framework for complex reasoning. OctoTools employs standardized *tool cards* to facilitate seamless integration of diverse tools and a dedicated planner-executor workflow that separates high-level planning over multiple steps from low-level planning and command generation within each step. Through extensive experiments on 16 diverse benchmarks, OctoTools consistently outperforms baselines, achieving average accuracy gains of up to 9.3% over GPT-4o and up to 10.6% over strong agentic frameworks. Our in-depth analysis shows that OctoTools' improvements stem from dynamic task planning, effective tool usage, and multi-step problem decomposition. Future work includes test-time inference at the query level, extending multi-agent collaboration, and exploring specialized domains.

ACKNOWLEDGMENTS

This work is partially supported by the Hoffman-Yee Research Grants program at Stanford HAI and Chan Zuckerberg Initiative. We thank members of the Zou group for helpful discussions and insightful feedback.

CONTRIBUTION STATEMENT

PL and RT started the project. PL completed the early framework. PL, BC refined the framework. PL, BC, and SL contributed to experiments and paper writing. Correspondence to: Pan Lu (panlu@stanford.edu), James Zou (jamesz@stanford.edu).

REFERENCES

AutoGen. Autogen. https://github.com/microsoft/autogen, 2024. 2, 3, 7

Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. Forest-of-thought: Scaling test-time compute for enhancing llm reasoning. *arXiv preprint arXiv:2412.09078*, 2024. 16

Andres M Bran, Sam Cox, Oliver Schilter, Carlo Baldassari, Andrew D White, and Philippe Schwaller. Chemcrow: Augmenting large-language models with chemistry tools. *arXiv preprint arXiv:2304.05376*, 2023. 2, 3, 5

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020. 1

Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 9650–9660, 2021. 18

Wei-Ge Chen, Irina Spiridonova, Jianwei Yang, Jianfeng Gao, and Chunyuan Li. Llava-interactive: An all-in-one demo for image chat, segmentation, generation and editing. *arXiv preprint arXiv:2311.00571*, 2023. 3

Yew Ken Chia, Vernon Toh Yan Han, Deepanway Ghosal, Lidong Bing, and Soujanya Poria. PuzzleVQA: Diagnosing multimodal reasoning challenges of language models with abstract visual patterns. *arXiv preprint arXiv:2403.13315*, 2024. 6, 17

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022. 1

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. 3

Michael Fore, Simranjit Singh, and Dimitrios Stamoulis. Geckopt: Llm system efficiency via intent-based tool selection. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pp. 353–354, 2024. 2, 9

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, et al. Omni-MATH: A universal olympiad level mathematic benchmark for large language models. *arXiv preprint arXiv:2410.07985*, 2024. 6, 17

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023. 3

Deepanway Ghosal, Vernon Toh Yan Han, Chia Yew Ken, and Soujanya Poria. Are language models puzzle prodigies? algorithmic puzzles unveil serious challenges in multimodal reasoning. *arXiv preprint arXiv:2403.03864*, 2024. 6, 16

Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh. Making the v in vqa matter: Elevating the role of image understanding in visual question answering. In *Proceedings of the IEEE Conference on Computer Vision and pattern Recognition*, pp. 6904–6913, 2017. 6, 17

Tianrui Guan, Fuxiao Liu, Xiyang Wu, Ruiqi Xian, Zongxia Li, Xiaoyu Liu, Xijun Wang, Lichang Chen, Furong Huang, Yaser Yacoob, et al. HallusionBench: an advanced diagnostic suite for entangled language hallucination and visual illusion in large vision-language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14375–14385, 2024. 6, 16

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025. 16

Xuehai He, Yichen Zhang, Luntian Mou, Eric Xing, and Pengtao Xie. PathVQA: 30000+ questions for medical visual question answering. *arXiv preprint arXiv:2003.10286*, 2020. 6, 17

Yushi Hu, Weijia Shi, Xingyu Fu, Dan Roth, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, and Ranjay Krishna. Visual sketchpad: Sketching as a visual chain of thought for multimodal language models. *arXiv preprint arXiv:2406.09403*, 2024. 2, 3, 4, 5

Zhenlan Ji, Daoyuan Wu, Pingchuan Ma, Zongjie Li, and Shuai Wang. Testing and understanding erroneous planning in llm agents through synthesized user inputs. *arXiv preprint arXiv:2404.17833*, 2024. 5

Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, 11(14):6421, 2021. 6, 17, 19

Yeonghun Kang and Jihan Kim. Chatmof: an artificial intelligence system for predicting and generating metal-organic frameworks using large language models. *Nature Communications*, 15(1): 4705, 2024. 2, 3

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022. 16

Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8460–8478, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.579. URL https://aclanthology.org/2022.acl-long.579. 3

Inc LangChain. LangChain. https://github.com/langchain-ai/langchain, 2024. 2, 3, 7

Angeliki Lazaridou, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. Internet-augmented language models through few-shot prompting for open-domain question answering. *arXiv preprint arXiv:2203.05115*, 2022. 3

Binxu Li, Tiankai Yan, Yuanting Pan, Jie Luo, Ruiyang Ji, Jiayuan Ding, Zhe Xu, Shilong Liu, Haoyu Dong, Zihao Lin, et al. Mmedagent: Learning to use medical tools with multi-modal agent. *arXiv preprint arXiv:2407.02483*, 2024a. 2, 3

Zelong Li, Wenyue Hua, Hao Wang, He Zhu, and Yongfeng Zhang. Formal-LLM: Integrating formal language and natural language for controllable llm-based agents. *arXiv preprint arXiv:2402.00798*, 2024b. 5

Nathan Lile. Math twenty four (24s game) dataset. https://huggingface.co/datasets/nlile/24-game, 2024. 6, 17

Adam Dahlgren Lindström and Savitha Sam Abraham. CLEVR-Math: A dataset for compositional language, visual and mathematical reasoning. *arXiv preprint arXiv:2208.05358*, 2022. 6, 17

Bo Liu, Li-Ming Zhan, Li Xu, Lin Ma, Yan Yang, and Xiao-Ming Wu. SLAKE: A semantically-labeled knowledge-enhanced dataset for medical visual question answering. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 1650–1654. IEEE, 2021. 6, 17

Shilong Liu, Hao Cheng, Haotian Liu, Hao Zhang, Feng Li, Tianhe Ren, Xueyan Zou, Jianwei Yang, Hang Su, Jun Zhu, et al. Llava-plus: Learning to use tools for creating multimodal agents. *arXiv preprint arXiv:2311.05437*, 2023. 2, 3

Ming Y Lu, Bowen Chen, Drew FK Williamson, Richard J Chen, Ivy Liang, Tong Ding, Guillaume Jaume, Igor Odintsov, Long Phi Le, Georg Gerber, et al. A visual-language foundation model for computational pathology. *Nature Medicine*, 30:863–874, 2024a. 18

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 35, 2023. 2, 3, 4, 5

Pan Lu, Hritik Bansal, Tony Xia, Jiacheng Liu, Chunyuan Li, Hannaneh Hajishirzi, Hao Cheng, Kai-Wei Chang, Michel Galley, and Jianfeng Gao. MathVista: Evaluating mathematical reasoning of foundation models in visual contexts. In *International Conference on Learning Representations (ICLR)*, 2024b. 6, 17, 64

Elias Lumer. Toolshed: Scale tool-equipped agents with advanced rag-tool fusion and tool knowledge bases. *arXiv preprint arXiv:2410.14594*, 2024. 2, 9

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023. 6, 18, 73

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021. 1, 2, 3

OpenAI. Function calling - openai. https://platform.openai.com/docs/guides/function-calling, 2023a. 2, 3, 7

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023b. 1

Varatheepan Paramanayakam, Andreas Karatzas, Iraklis Anagnostopoulos, and Dimitrios Stamoulis. Less is more: Optimizing function calling for llm execution on edge devices. *arXiv preprint arXiv:2411.15399*, 2024. 2, 9

Ethan Perez, Patrick Lewis, Wen-tau Yih, Kyunghyun Cho, and Douwe Kiela. Unsupervised question decomposition for question answering. *arXiv preprint arXiv:2002.09758*, 2020. 16

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. *arXiv preprint arXiv:2311.12022*, 2023. 6, 17

Tianhe Ren, Yihao Chen, Qing Jiang, Zhaoyang Zeng, Yuda Xiong, Wenlong Liu, Zhengyu Ma, Junyi Shen, Yuan Gao, Xiaoke Jiang, Xingyu Chen, Zhuheng Song, Yuhong Zhang, Hongjie Huang, Han Gao, Shilong Liu, Hao Zhang, Feng Li, Kent Yu, and Lei Zhang. Dino-x: A unified vision model for open-world object detection and understanding, 2024. URL https://arxiv.org/abs/2411.14347. 18

Jonathan Roberts, Kai Han, Neil Houlsby, and Samuel Albanie. SciFIBench: Benchmarking large multimodal models for scientific figure interpretation. *arXiv preprint arXiv:2405.08807*, 2024. 6, 17

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023. 2, 3

Samuel Schmidgall, Rojin Ziaei, Carl Harris, Eduardo Reis, Jeffrey Jopling, and Michael Moor. Agentclinic: a multimodal agent benchmark to evaluate ai in simulated clinical environments. *arXiv preprint arXiv:2405.07960*, 2024. 2, 3

Kurt Shuster, Jing Xu, Mojtaba Komeili, Da Ju, Eric Michael Smith, Stephen Roller, Megan Ung, Moya Chen, Kushal Arora, Joshua Lane, et al. Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage. *arXiv preprint arXiv:2208.03188*, 2022. 1, 3

Simeng Sun, Yang Liu, Shuohang Wang, Chenguang Zhu, and Mohit Iyyer. Pearl: Prompting large language models to plan and execute actions over long documents. *arXiv preprint arXiv:2305.14564*, 2023. 16

Yuxuan Sun, Hao Wu, Chenglu Zhu, Sunyi Zheng, Qizi Chen, Kai Zhang, Yunlong Zhang, Dan Wan, Xiaoxiao Lan, Mengyue Zheng, et al. PathMMU: A massive multimodal expert-level benchmark for understanding and reasoning in pathology. In *European Conference on Computer Vision*, pp. 56–73. Springer, 2025. 6, 17, 19

Heyi Tao, Sethuraman TV, Michal Shlapentokh-Rothman, Derek Hoiem, and Heng Ji. Webwise: Web interface control and sequential exploration with large language models. *arXiv preprint arXiv:2310.16042*, 2023. 2, 3

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022. URL https://arxiv.org/abs/2201.08239. 1, 3

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024a. 3

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *arXiv preprint arXiv:2406.01574*, 2024b. 6, 17

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 16

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. 1, 3, 16

Yuxiang Zhang, Jing Chen, Junjie Wang, Yaxin Liu, Cheng Yang, Chufan Shi, Xinyu Zhu, Zihao Lin, Hanwen Wan, Yujiu Yang, et al. Toolbehonest: A multi-level hallucination diagnostic benchmark for tool-augmented large language models. *arXiv preprint arXiv:2406.20015*, 2024. 3

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022. 16

APPENDIX CONTENTS

A Algorithm in OctoTools **16**

B Additional Related Work **16**

C Experimental Details **16**

C.1 Benchmark Datasets 16

C.2 Tools Used in Our Experiments 18

C.3 Additional Tools for Exploration Study 18

D Experimental Results **19**

E Configurations in OctoTools **25**

E.1 Query Analyzer 25

E.2 Action Predictor 26

E.3 Command Predictor 27

E.4 Context Verifier 30

E.5 Solution Summarizer 31

F Tool Cards in OctoTools **32**

F.1 ArXiv Paper Searcher Tool 32

F.2 Generalist Solution Generator Tool 35

F.3 Google Search Tool 37

F.4 Image Captioner Tool 39

F.5 Path Generalist Classifier Tool 41

F.6 Pubmed Search Tool 43

F.7 Python Code Generator Tool 45

F.8 Relevant Patch Zoomer Tool 47

F.9 Text Detector Tool 50

F.10 URL Text Extractor Tool 53

F.11 Wikipedia Knowledge Searcher Tool 55

G Additional Tool Cards in OctoTools **57**

G.1 Object Detector Tool 57

G.2 Advanced Object Detector Tool 60

G.3 Nature News Fetcher Tool 62

H Experimental Examples **64**

H.1 Demonstration Example 64

H.2 Example for Multi-step Mathematical Reasoning 69

H.3 Example for Agentic Reasoning 73
H.4 Example for Medical Reasoning 79
H.5 Example for Pathology Diagnosis 82
H.6 Example for Scientific Reasoning 84
H.7 Example for Fine-grained Visual Reasoning 87
H.8 Exploration Example for Literature Review 90

A ALGORITHM IN OCTOTOOLS

Algorithm 1 Task-specific Toolset Optimization

Input: Toolbox $\mathcal{D} = \{d_i\}_{i=1}^{n}$, base toolset $\mathcal{D}_{\text{base}}$
Output: Optimized toolset $\mathcal{D}^*$
Stage 1: Baseline setup
$\text{Acc}_{\text{baseline}} \leftarrow \text{Acc}(\mathcal{D}_{\text{base}})$
# Stage 2: Individual tool evaluation
for each d_i in D **such that** $d_i \notin \mathcal{D}_{\text{base}}$ **do**
  $\mathcal{D}_i \leftarrow \mathcal{D}_{\text{base}} \cup \{d_i\}$
  $\text{Acc}_i \leftarrow \text{Acc}(\mathcal{D}_i)$
  $\Delta_{d_i} \leftarrow \text{Acc}_i - \text{Acc}_{\text{baseline}}$
  if $\Delta_{d_i} > 0$ **then**
    $\mathcal{D}_{\text{beneficial}} \leftarrow \mathcal{D}_{\text{beneficial}} \cup \{d_i\}$
  end if
end for
# Stage 3: Select optimized toolset
$\mathcal{D}^* \leftarrow \mathcal{D}_{\text{beneficial}} \cup \mathcal{D}_{\text{base}}$
Return $\mathcal{D}^*$

B ADDITIONAL RELATED WORK

Foundation Models in Complex Tasks. When faced with multi-step problems, a common strategy is to break down a question into simpler sub-questions and solve them step by step. Early work approached decomposition with unsupervised or weakly supervised models (Perez et al., 2020; Khot et al., 2022), and more recent research has explored prompting techniques for step-by-step reasoning, including Chain-of-Thought (Wei et al., 2022), Least-to-Most (Zhou et al., 2022), ReAct (Yao et al., 2022), Pearl (Sun et al., 2023), Forest-of-Thought (Bi et al., 2024), and rStar-Math(Guan et al., 2025). While these methods significantly improve FMs' single-model reasoning capabilities, they primarily rely on the latent capacity of foundation models without external validation or targeted tool usage. In contrast, OctoTools systematically combines multi-step decomposition (via an iterative planner) with specialized tools (encapsulated by *tool cards*) and an executor for reliable, context-aware function calling. This design makes it easy to incorporate domain-specific functionalities and check intermediate steps with external modules, thus improving both correctness and versatility in tackling challenging tasks.

C EXPERIMENTAL DETAILS

C.1 BENCHMARK DATASETS

Here, we report further details of each of the 16 benchmark we used in this study. Unless specified otherwise, a validation set of 100 examples and a test set of 200 examples were sampled from each dataset. For benchmarks with fewer than 300 total samples, the test set consisted of all remaining samples not used for validation. No additional preprocessing was performed for open-ended questions. For multiple choice questions, choices were enumerated using capital letters and each concatenated to the question following a new line character. Any subsetting of the original datasets are also described below.

C.1.1 GENERAL DOMAIN BENCHMARKS

AlgoPuzzleVQA (Ghosal et al., 2024) is a set of geometric puzzles that require both visual understanding, language understanding, and complex algorithmic reasoning that are difficult for base VLMs.

Hallusion-VD is a subset of HallusionBench (Guan et al., 2024), a benchmark to test visual understanding through optical and geometric illusions. For our experiments we use the Visual Dependent

subset of HallusionBench, consisting of questions where the visual context is required to give a definitive answer.

PuzzleVQA (Chia et al., 2024) is a dataset of puzzle instances based on abstract, geometric patterns that test understanding and reasoning based on colors, numbers, sizes, and shapes.

VQA 2.0 (Goyal et al., 2017) consists of open-ended questions about images that require fine-grained visual understanding.

C.1.2 Mathematical Benchmarks

Game of 24 (Lile, 2024) is based on the classic arithmetic game of 24 (also known as 24, the 24 numbers game, *etc.*). The puzzle involves using four numbers and basic arithmetic operations (addition, subtraction, multiplication, division) to construct an expression that evaluates to 24. For example, given the numbers 4, 9, 10, and 13, one valid solution is "(10 - 4) $\times$ (13 - 9) = 24". Solving the puzzle requires numerical calculation skills as well as multiple attempts to verify proposed solutions. Each puzzle is presented as open-ended questions.

Omni-MATH (Gao et al., 2024) is a text-only mathematical benchmark consisting of open-ended competition-level problems at the Olympiad level, requiring advanced mathematical knowledge and reasoning.

CLEVR-Math (Lindström & Abraham, 2022) consists of multimodal math word problems involving addition and subtraction. Each problem contains a textual description and an image illustrating the scenario. A combination of language, visual, and mathematical reasoning is required to solve these word problems.

MathVista (Lu et al., 2024b) is benchmark designed to combine challenges from diverse mathematical and visual tasks. The queries are a mix of multiple choice and open-ended questions and require numerical computation, fine-grained visual understanding, and multi-step reasoning.

C.1.3 Scientific Benchmarks

GPQA or Graduate Level Google-Proof Q&A Benchmark (Rein et al., 2023) is a set of challenging text-only multiple choice questions written by domain experts in biology, physics, and chemistry designed to be "extremely difficult".

MMLU-Pro (Wang et al., 2024b) is a text-only benchmark consisting of challenging, reasoning-focused multiple choice questions that require general scientific knowledge and complex reasoning.

SciFIBench (Roberts et al., 2024) is a benchmark of multiple choice questions for scientific figure interpretation. Queries involve understanding and extracting information from scientific figures.

C.1.4 Medical Benchmarks

MedQA (Jin et al., 2021) consists of text-only multiple choice questions curated from professional medical board exams. Questions cover general medical and clinical knowledge and reasoning.

PathCLS is a subset of PathMMU (Sun et al., 2025) that consists of multiple choice questions based on hematoxylin and eosin (H&E)-stained histopathology microscopy images reformulated from well-known pathology classification datasets. These questions generally involve disease diagnosis based on histopathology images.

PathVQA (He et al., 2020) is a visual question-answering dataset curated from pathology-related image-caption pairs sourced from textbooks, spanning multiple tissue types and stains. All questions were treated as open-ended in our evaluation.

SLAKE or Semantically-Labeled Knowledge-Enhanced Dataset (Liu et al., 2021) is a radiology visual question-answering dataset. The associated images span X-ray images, computed tomography (CT) scans, and magnetic resonance imaging (MRI). All questions were treated as open-ended in our evaluation. Though the authors of the dataset released object detection labels and segmentation masks for each image, these were excluded in our evaluation for the purpose of increasing difficulty.

C.1.5 AGENTIC BENCHMARK

GAIA-Text (Mialon et al., 2023) is a benchmark specifically designed to evaluate general AI assistants and agents, requiring abilities such as multi-step reasoning, web browsing, and generally tool-use proficiency. The questions are designed to be difficult for base FMs. We use the text-only subset of this dataset in our experiments.

C.2 TOOLS USED IN OUR EXPERIMENTS

We implemented 11 tools in the toolbox for our experiments. Here, we provide detailed descriptions of each tool. See §F for the complete tool cards of each tool and usage examples.

Arxiv Paper Searcher (§F.1) searches arXiv[1], an open-access pre-print repository, for abstracts and links that match a given query.

Generalist Solution Generator (§F.2) is an instance of the OctoTools base FM and acts as the default reasoning engine if the agent decides not to use an external tool.

Google Search (§F.3) uses the Google custom search API[2] to search the web and return links and a summary of each result.

Image Captioner (§F.4) is an instance of the base FM prompted for generating text descriptions of input images.

Path Generalist Classifier (§F.5) is a tool for performing general classification of H&E-stained pathology microscopy images. The tool relies on CONCH (Lu et al., 2024a), a pretrained vision-language foundation model for pathology, for performing zero-shot classification of pathology image patches.

Pubmed Search (§F.6) retrieves relevant article abstracts from PubMed based on a text query. The retrieval is performed using the PubMed and NCBI APIs[3].

Python Code Generator (§F.7) generates and executes Python code given a query and returns the execution result. The code generation is performed by an instance of the base FM prompted for Python code generation.

Relevant Patch Zoomer (§F.8) is an instance of the base FM that, given a query, decides which regions of the image to zoom into (among the four quadrants and the center patch) and saves the zoomed patches.

Text Detector (§F.9) detects multilingual text within an image by calling the EasyOCR tool for text detection[4].

URL Text Extractor (§F.10) visits web pages given the URL and returns the text content of the page.

Wikipedia Knowledge Searcher (§F.11) searches Wikipedia using the MediaWiki API[5] and returns articles matching a given query.

C.3 ADDITIONAL TOOLS FOR EXPLORATION STUDY

We also provide several additional tools for exploration, as follows:

Object Detector (§G.1) performs object detection on an image given a list of object labels to detect, using the Grounding DINO model (Caron et al., 2021). Due to the standardized design of tool cards, this tool can be upgraded to the **Advanced Object Detector** (§G.2), which uses DINO-X (Ren et al., 2024), a more recent version powered by API calls.[6]

[1] https://arxiv.org/
[2] https://developers.google.com/custom-search/v1/introduction
[3] https://www.ncbi.nlm.nih.gov/books/NBK25501/
[4] https://github.com/JaidedAI/EasyOCR
[5] https://www.mediawiki.org/wiki/API
[6] https://github.com/IDEA-Research/DINO-X-API

Nature News Fetcher (§G.3) retrieves the latest news articles from the science journal *Nature*.[7] An example in §H.8 demonstrates how this tool can be used to obtain the latest research trends for a given topic.

D EXPERIMENTAL RESULTS

Gains from decompositions *v.s.* tool usage. To assess the performance gains from *tool calling*, we compute for each benchmark the difference between OctoTools, *i.e.*, with external tools, and $\text{OctoTools}_{\text{base}}$, *i.e.*, without external tools: $\Delta_{\text{tools}} = \text{Acc}_{\text{OctoTools}} - \text{Acc}_{\text{OctoTools}_{\text{base}}}$. Similarly, to assess the performance gains from *decomposing* the problem into *multiple steps*, we compute the difference between $\text{OctoTools}_{\text{base}}$ and the 0-shot baseline: $\Delta_{\text{decom}} = \text{Acc}_{\text{OctoTools}_{\text{base}}} - \text{Acc}_{\text{0-shot}}$.

For each benchmark, the performance gains from these two dimensions are visualized in Figure 10. We observe that the tasks can be broadly categorized into three groups: (1) Tasks far above the $y = x$ diagonal benefit more from step decomposition, and a specialized tool is either not needed or not available. (2) Tasks far below the diagonal benefit more from one or more specialized tools that are well-suited for the task, while multi-step reasoning is less critical. (3) Tasks along the diagonal benefit from both aspects. This analysis highlights that AI agent frameworks can improve performance in different ways depending on the specific skills demanded by each task, and that OctoTools is a versatile framework for achieving such improvements across a diverse set of tasks.

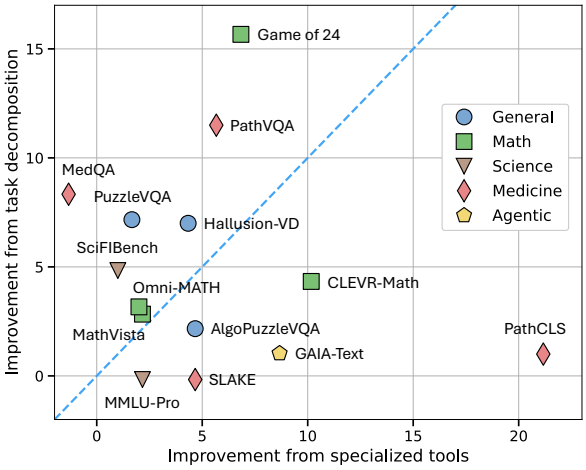


Figure 10: Benchmark distribution across two dimensions. Tasks that show high improvement from task decomposition likely require multi-step reasoning, while tasks that show high improvement from specialized tools likely require specialized skills.

Optimized tool sets. Table 2 shows the optimized tool sets across 16 tasks from Algorithm 1. The toolset optimization method in OctoTools is able to find diverse optimal tool sets for different tasks. In general, the `Image_Captioner` and `Relevant_Patch_Zoomer` tools are very commonly used in vision benchmarks, with the former being used in all vision benchmarks and the latter being used in 6 out of the 10. The `Python_Code_Generator` is represented in 3 out of the 4 mathematical domain benchmarks. The `Wikipedia_Knowledge_Searcher` is represented in all the scientific domain benchmarks. We also see that highly domain-specific tools are represented in their corresponding use cases, such as `Pubmed_Search` in a general medical benchmark (Jin et al., 2021) and `Path_Generalist_Classifier` in a pathology classification benchmark (Sun et al., 2025).

Steps taken for different tasks. Figure 18 shows the distribution of the number of steps taken. OctoTools is able to adapt to each task by applying different sets of tools and constructing chains of reasoning as needed.

[7] https://www.nature.com/latest-news

Benchmarks	Modality	Domain	Generalist_Solution_Generator	Image_Captioner	Relevant_Patch_Zoomer	Text_Detector	Wikipedia_Knowledge_Searcher	Google_Search	URL_Text_Extractor	ArXiv_Paper_Searcher	Python_Code_Generator	Path_Generalist_Classifier	Pubmed_Search
AlgoPuzzleVQA	Vision	General	✓	✓		✓							
Hallusion-VD	Vision	General	✓	✓									
PuzzleVQA	Vision	General	✓	✓									
VQA 2.0	Vision	General	✓	✓	✓								
Game of 24	Text	Mathematical	✓								✓		
Omni-MATH	Text	Mathematical	✓								✓		
CLEVR-Math	Vision	Mathematical	✓	✓	✓								
MathVista	Vision	Mathematical	✓	✓	✓			✓			✓		
GPQA	Text	Scientific	✓				✓						
MMLU-Pro	Text	Scientific	✓				✓						
SciFIBench	Vision	Scientific	✓	✓		✓	✓			✓			
MedQA	Text	Medical	✓										✓
PathCLS	Vision	Medical	✓	✓	✓							✓	
PathVQA	Vision	Medical	✓	✓	✓								
SLAKE	Vision	Medical	✓	✓	✓								
GAIA-Text	Text	Agentic	✓				✓	✓	✓		✓		

Table 2: Optimized tool sets for each benchmark following our Algorithm1. A ✓indicates that the tool is used for that benchmark.

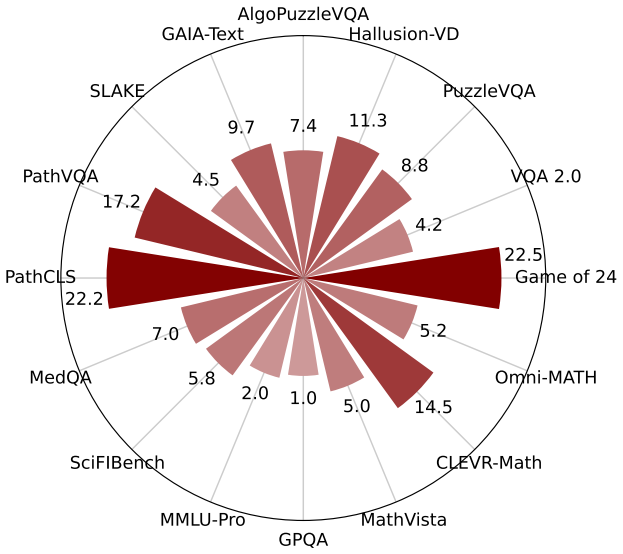


Figure 11: Performance gains across different benchmarks from our OctoTools framework over the base GPT-4o model.

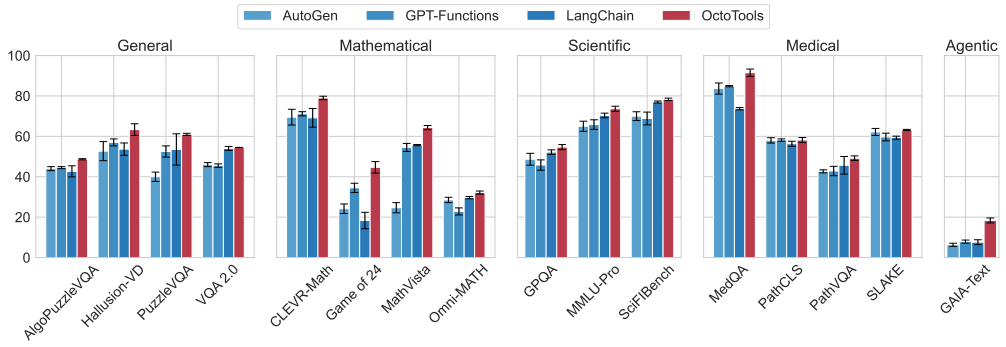


Figure 12: **Performance ours vs. other agents**. Our system consistently outperforms agent baselines across all benchmarks. Bar values represent accuracy and error bars represent standard deviation.

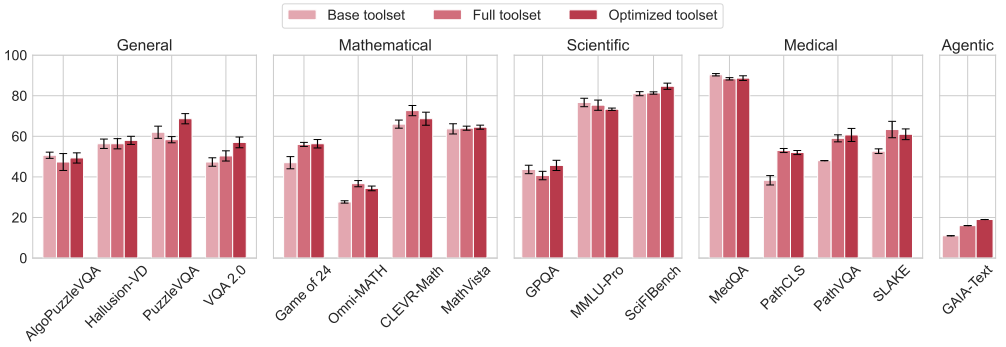


Figure 13: **Performance with vs. without tool selection**. While toolset optimization increases performance over using the full toolset in most tasks, even without it, our system achieves similar performance by naively enabling all possible tools. Bar values represent accuracy and error bars represent standard deviation.

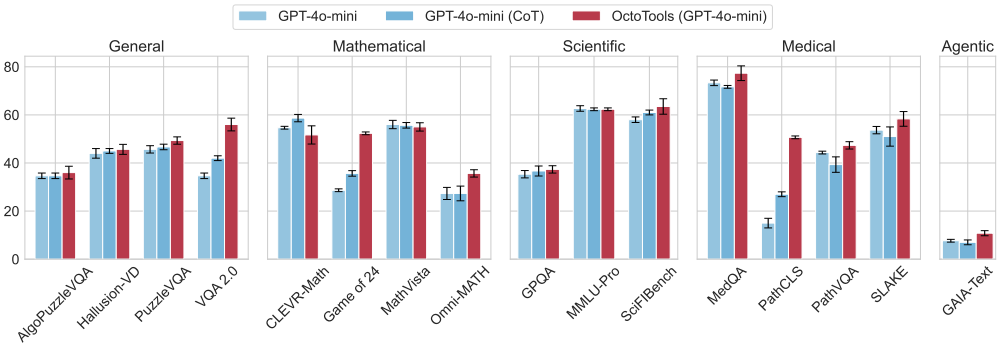


Figure 14: **Performance on a weaker FM (GPT-4o-mini)**. We observe similar trends using OctoTools with a weaker base FM. Bar values represent accuracy and error bars represent standard deviation.

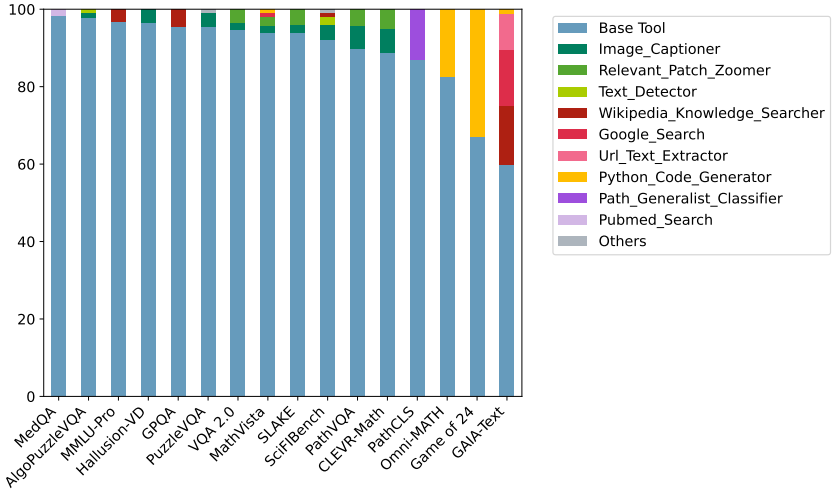


Figure 15: **Distribution of tools usage.** Frequency of tools used by the AutoGen agent for each benchmark.

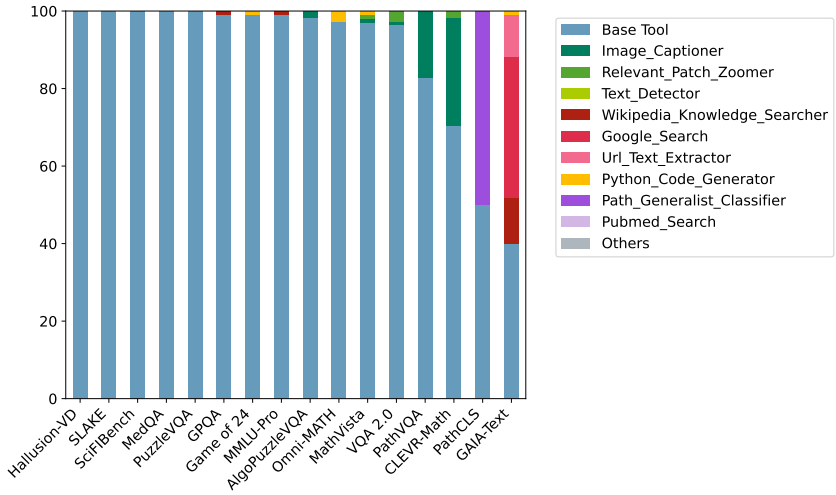


Figure 16: **Distribution of tools usage.** Frequency of tools used by the LangChain agent for each benchmark.

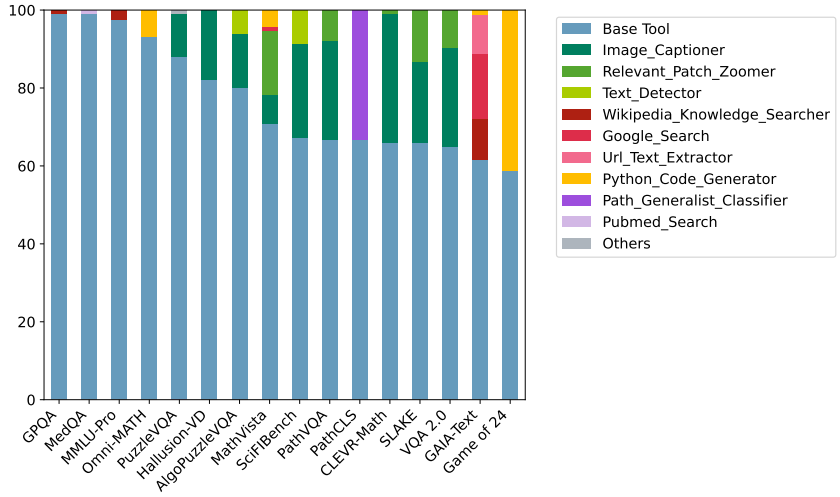


Figure 17: **Distribution of tools usage.** Frequency of tools used by the GPT-Functions agent for each benchmark.

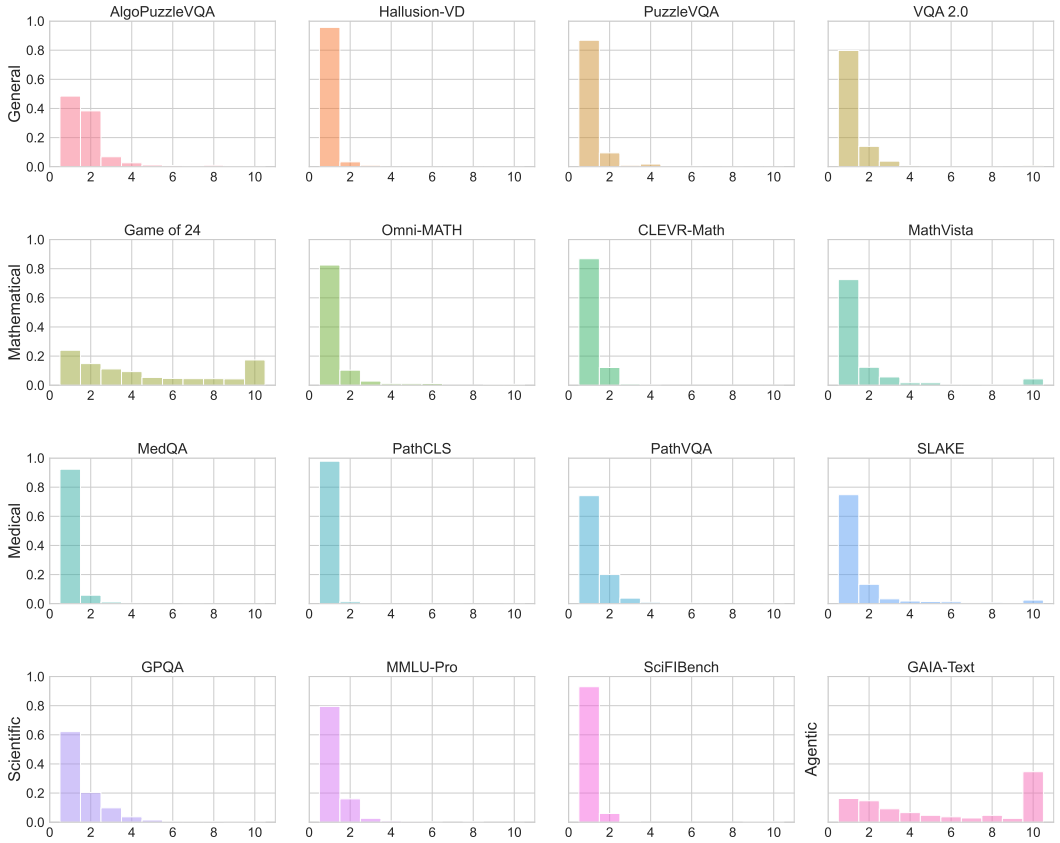


Figure 18: **Distribution of number of steps used.**

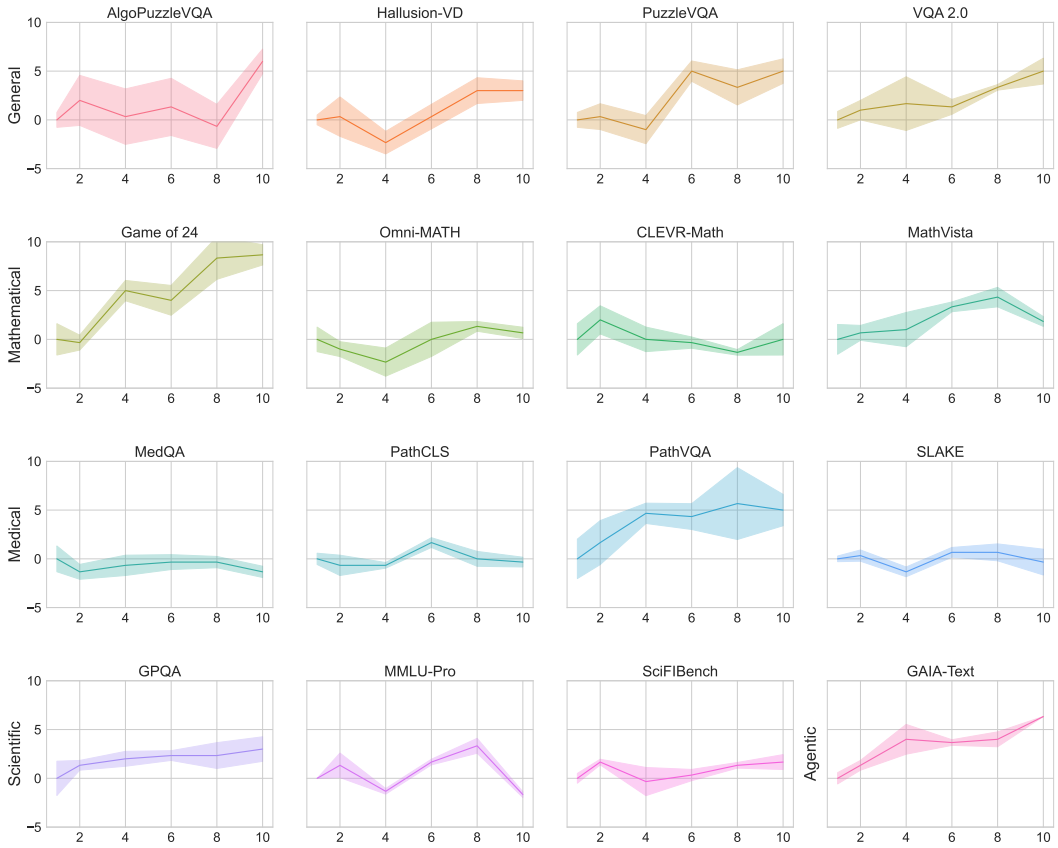


Figure 19: **Accuracy vs number of maximum steps.** The change in accuracy from a maximum step of 1 is plotted. Most benchmarks improve in performance with the number of allowed steps.

E CONFIGURATIONS IN OCTOTOOLS

E.1 QUERY ANALYZER

Prompt for Query Analyzer

Task: Analyze the given query with accompanying inputs and determine the skills and tools needed to address it effectively.

Available tools: {`self.available_tools`}

Metadata for the tools: {`self.toolbox_metadata`}

Image: {`image_info`}

Query: {`question`}

Instructions:
1. Carefully read and understand the query and any accompanying inputs.
2. Identify the main objectives or tasks within the query.
3. List the specific skills that would be necessary to address the query comprehensively.
4. Examine the available tools in the toolbox and determine which ones might relevant and useful for addressing the query. Make sure to consider the user metadata for each tool, including limitations and potential applications (if available).
5. Provide a brief explanation for each skill and tool you've identified, describing how it would contribute to answering the query.

Your response should include:
1. A concise summary of the query's main points and objectives, as well as content in any accompanying inputs.
2. A list of required skills, with a brief explanation for each.
3. A list of relevant tools from the toolbox, with a brief explanation of how each tool would be utilized and its potential limitations.
4. Any additional considerations that might be important for addressing the query effectively.
Please present your analysis in a clear, structured format.

E.2 ACTION PREDICTOR

Prompt for Action Predictor

Task: Determine the optimal next step to address the given query based on the provided analysis, available tools, and previous steps taken.

Query: {question}
Image: {image}
Query Analysis: {query_analysis}
Available Tools: {self.available_tools}
Tool Metadata: {self.toolbox_metadata}
Previous Steps and Their Results: {memory.get_actions()}
Current Step: {step_count} in {max_step_count} steps
Remaining Steps: {max_step_count - step_count}

Instructions:
1. Analyze the context thoroughly, including the query, its analysis, any image, available tools and their metadata, and previous steps taken.
2. Determine the most appropriate next step by considering:
   - Key objectives from the query analysis
   - Capabilities of available tools
   - Logical progression of problem-solving
   - Outcomes from previous steps
   - Current step count and remaining steps
3. Select ONE tool best suited for the next step, keeping in mind the limited number of remaining steps.
4. Formulate a specific, achievable sub-goal for the selected tool that maximizes progress towards answering the query.

Output Format:
<justification>: detailed explanation of why the selected tool is the best choice for the next step, considering the context and previous outcomes.
<context>: MUST include ALL necessary information for the tool to function, structured as follows:
   * Relevant data from previous steps
   * File names or paths created or used in previous steps (list EACH ONE individually)
   * Variable names and their values from previous steps' results
   * Any other context-specific information required by the tool
<sub_goal>: a specific, achievable objective for the tool, based on its metadata and previous outcomes. It MUST contain any involved data, file names, and variables from Previous Steps and Their Results that the tool can act upon.
<tool_name>: MUST be the exact name of a tool from the available tools list.

Rules:
   - Select only ONE tool for this step.
   - The sub-goal MUST directly address the query and be achievable by the selected tool.
   - The Context section MUST include ALL necessary information for the tool to function, including ALL relevant file paths, data, and variables from previous steps.
   - The tool name MUST exactly match one from the available tools list: {self.available_tools}.
   - Avoid redundancy by considering previous steps and building on prior results.

Example (do not copy, use only as reference):
<justification>: [Your detailed explanation here]
<context>: Image path: "example/image.jpg", Previous detection results: [list of objects]
<sub_goal>: Detect and count the number of specific objects in the image "example/image.jpg"
<tool_name>: Object_Detector_Tool

E.3 COMMAND PREDICTOR

Prompt for Command Predictor

Task: Generate a precise command to execute the selected tool based on the given information.

Query: {question}
Image: {image}
Context: {context}
Sub-Goal: {sub_goal}
Selected Tool: {tool_name}
Tool Metadata: {tool_metadata}

Instructions:

Instructions:
1. Carefully review all provided information: the query, image path, context, sub-goal, selected tool, and tool metadata.
2. Analyze the tool's input_types from the metadata to understand required and optional parameters.
3. Construct a command or series of commands that aligns with the tool's usage pattern and addresses the sub-goal.
4. Ensure all required parameters are included and properly formatted.
5. Use appropriate values for parameters based on the given context, particularly the Context field which may contain relevant information from previous steps.
6. If multiple steps are needed to prepare data for the tool, include them in the command construction.

Output Format:
<analysis>: a step-by-step analysis of the context, sub-goal, and selected tool to guide the command construction.
<explanation>: a detailed explanation of the constructed command(s) and their parameters.
<command>: the Python code to execute the tool, which can be one of the following types:
a. A single line command with execution = tool.execute().
b. A multi-line command with complex data preparation, ending with execution = tool.execute().
c. Multiple lines of execution = tool.execute() calls for processing multiple items.

```
python
<your command here>
```

Rules:
1. The command MUST be valid Python code and include at least one call to tool.execute().
2. Each tool.execute() call MUST be assigned to the execution variable in the format execution = tool.execute(...).
3. For multiple executions, use separate execution = tool.execute() calls for each execution.
4. The final output MUST be assigned to the execution variable, either directly from tool.execute() or as a processed form of multiple executions.
5. Use the exact parameter names as specified in the tool's input_types.
6. Enclose string values in quotes, use appropriate data types for other values (e.g., lists, numbers).
7. Do not include any code or text that is not part of the actual command.
8. Ensure the command directly addresses the sub-goal and query.
9. Include ALL required parameters, data, and paths to execute the tool in the command itself.
10. If preparation steps are needed, include them as separate Python statements before the tool.execute() calls.

Prompt for Command Prediction (Continued)

Examples (Not to use directly unless relevant):

Example 1 (Single line command):
<analysis>: The tool requires an image path and a list of labels for object detection.
<explanation>: We pass the image path and a list containing "baseball" as the label to detect.
<command>:

```
```
python
execution = tool.execute(image="path/to/image", labels=["baseball"])
```
```

Example 2 (Multi-line command with data preparation):
<analysis>: The tool requires an image path, multiple labels, and a threshold for object detection.
<explanation>: We prepare the data by defining variables for the image path, labels, and threshold, then pass these to the `tool.execute()` function.
<command>:

```
```
python
image = "path/to/image"
labels = ["baseball", "football", "basketball"]
threshold = 0.5
execution = tool.execute(image=image, labels=labels, threshold=
    threshold)
```
```

Example 3 (Multiple executions):
<analysis>: We need to process multiple images for baseball detection.
<explanation>: We call the tool for each image path, using the same label and threshold for all.
<command>:

```
```
python
execution = tool.execute(image="path/to/image1", labels=["baseball"],
     threshold=0.5)
execution = tool.execute(image="path/to/image2", labels=["baseball"],
     threshold=0.5)
execution = tool.execute(image="path/to/image3", labels=["baseball"],
     threshold=0.5)
```
```

Prompt for Command Predictor (Continued)

Some Wrong Examples:

<command>:

```
```
python
execution1 = tool.execute(query="...")
execution2 = tool.execute(query="...")
```
```

Reason: only `execution = tool.execute` is allowed, not `execution1` or `execution2`.

<command>:

```
```
python
urls = [
    "https://example.com/article1",
    "https://example.com/article2"
]

execution = tool.execute(url=urls[0])
execution = tool.execute(url=urls[1])
```
```

Reason: The command should process multiple items in a single execution, not separate executions for each item.

Remember: Your <command> field MUST be valid Python code including any necessary data preparation steps and one or more `execution = tool.execute()` calls, without any additional explanatory text. The format `execution = tool.execute` must be strictly followed, and the last line must begin with `execution = tool.execute` to capture the final output.

E.4 CONTEXT VERIFIER

Prompt for Verifier

Task: Thoroughly evaluate the completeness and accuracy of the memory for fulfilling the given query, considering the potential need for additional tool usage.

Query: {question}
Image: {image_info}
Available Tools: {self.available_tools}
Toolbox Metadata: {self.toolbox_metadata}
Initial Analysis: {query_analysis}
Memory (tools used and results): {memory.get_actions()}

Detailed Instructions:
1. Carefully analyze the query, initial analysis, and image (if provided):
   - Identify the main objectives of the query.
   - Note any specific requirements or constraints mentioned.
   - If an image is provided, consider its relevance and what information it contributes.

2. Review the available tools and their metadata:
   - Understand the capabilities and limitations and best practices of each tool.
   - Consider how each tool might be applicable to the query.

3. Examine the memory content in detail:
   - Review each tool used and its execution results.
   - Assess how well each tool's output contributes to answering the query.

4. Critical Evaluation (address each point explicitly):
   a) Completeness: Does the memory fully address all aspects of the query?
      - Identify any parts of the query that remain unanswered.
      - Consider if all relevant information has been extracted from the image (if applicable).
   b) Unused Tools: Are there any unused tools that could provide additional relevant information?
      - Specify which unused tools might be helpful and why.
   c) Inconsistencies: Are there any contradictions or conflicts in the information provided?
      - If yes, explain the inconsistencies and suggest how they might be resolved.
   d) Verification Needs: Is there any information that requires further verification due to tool limitations?
      - Identify specific pieces of information that need verification and explain why.
   e) Ambiguities: Are there any unclear or ambiguous results that could be clarified by using another tool?
      - Point out specific ambiguities and suggest which tools could help clarify them.

5. Final Determination: Based on your thorough analysis, decide if the memory is complete and accurate enough to generate the final output, or if additional tool usage is necessary.

Response Format:
<analysis>: Provide a detailed analysis of why the memory is sufficient. Reference specific information from the memory and explain its relevance to each aspect of the task. Address how each main point of the query has been satisfied.
<stop_signal>: Whether to stop the problem solving process and proceed to generating the final output.
   * "True": if the memory is sufficient for addressing the query to proceed and no additional available tools need to be used. If ONLY manual verification without tools is needed, choose "True".
   * "False": if the memory is insufficient and needs more information from additional tool usage.

E.5 SOLUTION SUMMARIZER

Prompt for Solution Summarizer

Task: Generate the final output based on the query, image, and tools used in the process.

Query: {`question`}
Image: {`image_info`}
Actions Taken: {`memory.get_actions()`}

Instructions:
1. Review the query, image, and all actions taken during the process.
2. Consider the results obtained from each tool execution.
3. Incorporate the relevant information from the memory to generate the step-by-step final output.
4. The final output should be consistent and coherent using the results from the tools.

Output Structure:
Your response should be well-organized and include the following sections:
1. Summary:
   - Provide a brief overview of the query and the main findings.

2. Detailed Analysis:
   - Break down the process of answering the query step-by-step.
   - For each step, mention the tool used, its purpose, and the key results obtained.
   - Explain how each step contributed to addressing the query.

3. Key Findings:
   - List the most important discoveries or insights gained from the analysis.
   - Highlight any unexpected or particularly interesting results.

4. Answer to the Query:
   - Directly address the original question with a clear and concise answer.
   - If the query has multiple parts, ensure each part is answered separately.

5. Additional Insights (if applicable):
   - Provide any relevant information or insights that go beyond the direct answer to the query.
   - Discuss any limitations or areas of uncertainty in the analysis.

6. Conclusion:
   - Summarize the main points and reinforce the answer to the query.
   - If appropriate, suggest potential next steps or areas for further investigation.

F TOOL CARDS IN OCTOTOOLS

F.1 ARXIV PAPER SEARCHER TOOL

ArXiv Paper Searcher Tool: Metadata

```
tool_name="ArXiv_Paper_Searcher_Tool",

tool_description="A tool that searches arXiv for papers based on a
    given query.",

input_types={
   "query": "str - The search query for arXiv papers.",
   "size": "int - The number of results per page (25, 50, 100, or 200)
       . If None, use 25.",
   "max_results": "int - The maximum number of papers to return (
       default: 25). Should be less than or equal to 100."
},

output_type="list - A list of dictionaries containing paper
    information.",

demo_commands=[
   {
      "command": 'execution = tool.execute(query="tool agents with
          large language models")',
      "description": "Search for papers about tool agents with large
          language models."
   },
   {
      "command": 'execution = tool.execute(query="quantum computing",
          size=100, max_results=50)',
      "description": "Search for quantum computing papers, with 100
          results per page, returning a maximum of 50 papers."
   },
   {
      "command": 'execution = tool.execute(query="machine learning",
          max_results=75)',
      "description": "Search for machine learning papers, returning a
          maximum of 75 papers."
   },
],

user_metadata={
   "valid_sizes": [25, 50, 100, 200],
   "base_url": "https://arxiv.org/search/"
}
```

ArXiv Paper Searcher Tool: Example 1

Input (query): enhance mathematical reasoning with large language models

Output (article 1, search date: Jan 14, 2025):

Title: Enhancing Reasoning through Process Supervision with Monte Carlo Tree Search
Authors: Shuangtao Li, Shuaihao Dong, Kexin Luan, Xinhan Di, Chaofan Ding
Abstract: Large language models (LLMs) have demonstrated their remarkable capacity across a variety of tasks. ... Furthermore, models trained on one dataset also exhibit improved performance on the other, showing the transferability of the enhanced reasoning ability.
Link: https://arxiv.org/abs/2501.01478

Output (article 2, search date: Jan 14, 2025):

Title: Open Eyes, Then Reason: Fine-grained Visual Mathematical Understanding in MLLMs
Authors: Shan Zhang, Aotian Chen, Yanpeng Sun, Jindong Gu, Yi-Yu Zheng, Piotr Koniusz, Kai Zou, Anton van den Hengel, Yuan Xue
Abstract: Current multimodal large language models (MLLMs) often underperform on mathematical problem-solving tasks that require fine-grained visual understanding. ... Our findings emphasize the importance of incorporating fine-grained visual understanding into MLLMs and provide a promising direction for future research.
Link: https://arxiv.org/abs/2501.06430

Output (structured result):


```
[
    {
        "title": "Enhancing Reasoning through Process Supervision with
            Monte Carlo Tree Search",
        "authors": "Shuangtao Li, Shuaihao Dong, Kexin Luan, Xinhan Di,
             Chaofan Ding",
        "abstract": "Large language models (LLMs) have demonstrated
            their remarkable capacity across a variety of tasks. ...
            Furthermore, models trained on one dataset also exhibit
            improved performance on the other, showing the
            transferability of the enhanced reasoning ability.",
        "link": "https://arxiv.org/abs/2501.01478"
    },
    {
        "title": "Open Eyes, Then Reason: Fine-grained Visual
            Mathematical Understanding in MLLMs",
        "authors": "Shan Zhang, Aotian Chen, Yanpeng Sun, Jindong Gu,
            Yi-Yu Zheng, Piotr Koniusz, Kai Zou, Anton van den Hengel,
            Yuan Xue",
        "abstract": "Current multimodal large language models (MLLMs)
            often underperform on mathematical problem-solving tasks
            that require fine-grained visual understanding. ... Our
            findings emphasize the importance of incorporating fine-
            grained visual understanding into MLLMs and provide a
            promising direction for future research.",
        "link": "https://arxiv.org/abs/2501.06430"
    }
]
```

ArXiv Paper Searcher Tool: Example 2

Input (query): automated scientific discovery

Output (article 1, search date: Jan 14, 2025):

Title: BoxingGym: Benchmarking Progress in Automated Experimental Design and Model Discover
Authors: Kanishk Gandhi, Michael Y. Li, Lyle Goodyear, Louise Li, Aditi Bhaskar, Mohammed Zaman, Noah D. Goodman
Abstract: Understanding the world and explaining it with scientific theories is a central aspiration of artificial intelligence research. Proposing theories, designing experiments to test them, and then revising them based on data are fundamental to scientific discovery. ... We find that augmenting the LLM-based agent with an explicit statistical model does not reliably improve these results.
Link: https://arxiv.org/abs/2501.01540

Output (article 2, search date: Jan 14, 2025):

Title: Automating the Search for Artificial Life with Foundation Models
Authors: Akarsh Kumar, Chris Lu, Louis Kirsch, Yujin Tang, Kenneth O. Stanley, Phillip Isola, David Ha
Abstract: With the recent Nobel Prize awarded for radical advances in protein discovery, foundation models (FMs) for exploring large combinatorial spaces promise to revolutionize many scientific fields. ... This new paradigm promises to accelerate ALife research beyond what is possible through human ingenuity alone.
Link: https://arxiv.org/abs/2412.17799

Output (structured result):


```
[
    {
        "title": "BoxingGym: Benchmarking Progress in Automated
            Experimental Design and Model Discovery",
        "authors": "Kanishk Gandhi, Michael Y. Li, Lyle Goodyear,
            Louise Li, Aditi Bhaskar, Mohammed Zaman, Noah D. Goodman",
        "abstract": "Understanding the world and explaining it with
            scientific theories is a central aspiration of artificial
            intelligence research. ... We find that augmenting the LLM-
            based agent with an explicit statistical model does not
            reliably improve these results.",
        "link": "https://arxiv.org/abs/2501.01540"
    },
    {
        "title": "Automating the Search for Artificial Life with
            Foundation Models",
        "authors": "Akarsh Kumar, Chris Lu, Louis Kirsch, Yujin Tang,
            Kenneth O. Stanley, Phillip Isola, David Ha",
        "abstract": "With the recent Nobel Prize awarded for radical
            advances in protein discovery, foundation models (FMs) for
            exploring large combinatorial spaces promise to
            revolutionize many scientific fields. ... This new paradigm
             promises to accelerate ALife research beyond what is
            possible through human ingenuity alone.",
        "link": "https://arxiv.org/abs/2412.17799"
    }
]
```

F.2 GENERALIST SOLUTION GENERATOR TOOL

Generalist Solution Generator Tool: Metadata

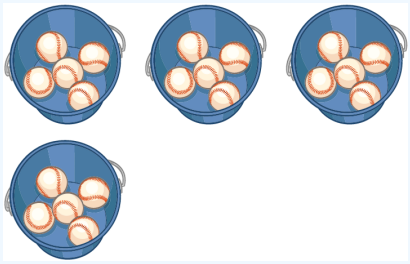
```
tool_name="Generalist_Solution_Generator_Tool",

tool_description="A generalized tool that takes query from the user as
     prompt, and answers the question step by step to the best of its
    ability. It can also accept an image.",

input_types={
   "prompt": "str - The prompt that includes query from the user to
       guide the agent to generate response (Examples: 'Describe this
       image in detail')",
   "image": "str - The path to the image file if applicable (default:
       None)",
},

output_type="str - The generated response to the original query prompt
    ",

demo_commands=[
   {
      "command": 'execution = tool.execute(prompt="Summarize the
          following text in a few lines")',
      "description": "Generate a short summary given the prompt from
          the user."
   },
   {
      "command": 'execution = tool.execute(prompt="Explain the mood of
           this scene.", image="path/to/image1.png")',
      "description": "Generate a caption focusing on the mood using a
          specific prompt and image."
   },
   {
      "command": 'execution = tool.execute(prompt="Give your best
          coordinate estimate for the pacemaker in the image and
          return (x1, y1, x2, y2)", image="path/to/image2.png")',
      "description": "Generate bounding box coordinates given the
          image and prompt from the user. The format should be (x1, y1
          , x2, y2)."
   },
   {
      "command": 'execution = tool.execute(prompt="Is the number of
          tiny objects that are behind the small metal jet less than
          the number of tiny things left of the tiny sedan?", image="
          path/to/image2.png")',
      "description": "Answer a question step by step given the image."
   }
],
```

Generalist Solution Generator Tool: Metadata (Continued)

```
user_metadata = {
   "limitation": "The Generalist_Solution_Generator_Tool may provide
       hallucinated or incorrect responses.",
   "best_practice": "Use the Generalist_Solution_Generator_Tool for
       general queries or tasks that don't require specialized
       knowledge or specific tools in the toolbox. For optimal results
       :\n\n"
   "1) Provide clear, specific prompts.\n"
   "2) Use it to answer the original query through step by step
       reasoning for tasks without complex or multi-step reasoning.\n"
   "3) For complex queries, break them down into subtasks and use the
       tool multiple times.\n"
   "4) Use it as a starting point for complex tasks, then refine with
       specialized tools.\n"
   "5) Verify important information from its responses.\n"
   "6) For image-related tasks, ensure the image path is correct and
       the prompt is relevant to the image content."
}
```

Generalist Solution Generator Tool: Example 1

Input (query): Describe the image in detail.
Input (image): `"baseballs.png"`



Output (response):
The image shows four blue buckets, each containing five baseballs. The baseballs are white with red stitching, and the buckets have handles on the sides. The arrangement of the baseballs within each bucket is slightly different, but each bucket consistently holds five baseballs. The background is plain, emphasizing the buckets and their contents.

Output (structured result):

```
{
   "The image shows four blue buckets, each containing five baseballs
       . The baseballs are white with red stitching, and the buckets
       have handles on the sides. The arrangement of the baseballs
       within each bucket is slightly different, but each bucket
       consistently holds five baseballs. The background is plain,
       emphasizing the buckets and their contents."
}
```

F.3 GOOGLE SEARCH TOOL

Google Search Tool: Metadata

```
tool_name="Google_Search_Tool",

tool_description="A tool that performs Google searches based on a
    given text query.",

input_types={
   "query": "str - The search query to be used for the Google search."
       ,
   "num_results": "int - The number of search results to return (
       default: 10).",
},

output_type="list - A list of dictionaries containing search result
    information.",

demo_commands=[
   {
      "command": 'execution = tool.execute(query="Python programming")
          ',
      "description": "Perform a Google search for 'Python programming'
           and return the default number of results."
   },
   {
      "command": 'execution = tool.execute(query="Machine learning
          tutorials", num_results=5)',
      "description": "Perform a Google search for 'Machine learning
          tutorials' and return 5 results."
   },
]
```

Google Search Tool: Example 1

Input (query): nobel prize winners in chemistry 2024

Output (result 1, search date: Jan 14, 2025):
Title: Nobel Prize in Chemistry Laureates
URL: https://www.nobelprize.org/prizes/chemistry/
Snippet: The Nobel Prize in Chemistry 2024 was awarded with one half to David Baker "for computational protein design" and the other half jointly to Demis Hassabis and ...

Output (result 2, search date: Jan 14, 2025):
Title: NSF congratulates laureates of the 2024 Nobel Prize in chemistry ...
URL: https://new.nsf.gov/news/nsf-congratulates-laureates-2024-nobel-prize-chemistry
Snippet: Oct 9, 2024 ... The U.S. National Science Foundation congratulates David Baker, Demis Hassabis and John Jumper on being awarded the 2024 Nobel Prize in ...

Output (result 3, search date: Jan 14, 2025):
Title: Press release: The Nobel Prize in Chemistry 2024 - NobelPrize.org
URL: https://www.nobelprize.org/prizes/chemistry/2024/press-release/
Snippet: Oct 9, 2024 ... David Baker has succeeded with the almost impossible feat of building entirely new kinds of proteins. Demis Hassabis and John Jumper have ...

Output (structured result):

```
[
    {
        "title": "Nobel Prize in Chemistry Laureates",
        "url": "https://www.nobelprize.org/prizes/chemistry/",
        "snippet": "The Nobel Prize in Chemistry 2024 was awarded with
            one half to David Baker ``for computational protein design
            '' and the other half jointly to Demis Hassabis and ..."
    },
    {
        "title": "NSF congratulates laureates of the 2024 Nobel Prize
            in chemistry ...",
        "url": "https://new.nsf.gov/news/nsf-congratulates-laureates
            -2024-nobel-prize-chemistry",
        "snippet": "Oct 9, 2024 ... The U.S. National Science
            Foundation congratulates David Baker, Demis Hassabis and
            John Jumper on being awarded the 2024 Nobel Prize in ..."
    },
    {
        "title": "Press release: The Nobel Prize in Chemistry 2024 -
            NobelPrize.org",
        "url": "https://www.nobelprize.org/prizes/chemistry/2024/press-
            release/",
        "snippet": "Oct 9, 2024 ... David Baker has succeeded with the
            almost impossible feat of building entirely new kinds of
            proteins. Demis Hassabis and John Jumper have ..."
    }
]
```

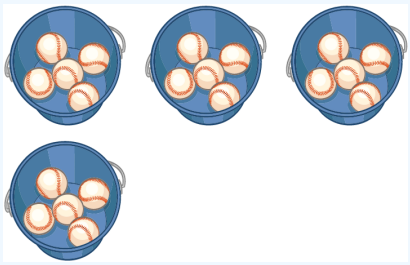
F.4 IMAGE CAPTIONER TOOL

Image Captioner Tool: Metadata

```
tool_name="Image_Captioner_Tool",

tool_description="A tool that generates captions for images using
    OpenAI's multimodal model.",

input_types={
   "image": "str - The path to the image file.",
   "prompt": "str - The prompt to guide the image captioning (default:
        'Describe this image in detail.').",
},

output_type="str - The generated caption for the image.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.png")'
          ,
      "description": "Generate a caption for an image using the
          default prompt and model."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          prompt="Explain the mood of this scene.")',
      "description": "Generate a caption focusing on the mood using a
          specific prompt and model."
   }
],

user_metadata = {
   "limitation": "The Image_Captioner_Tool provides general image
       descriptions but has limitations: 1) May make mistakes in
       complex scenes, counting, attribute detection, and
       understanding object relationships. 2) Might not generate
       comprehensive captions, especially for images with multiple
       objects or abstract concepts. 3) Performance varies with image
       complexity. 4) Struggles with culturally specific or domain-
       specific content. 5) May overlook details or misinterpret
       object relationships. For precise descriptions, consider: using
        it with other tools for context/verification, as an initial
       step before refinement, or in multi-step processes for
       ambiguity resolution. Verify critical information with
       specialized tools or human expertise when necessary."
}
```

Image Captioner Tool: Example 1

Input (image): `"baseballs.png"`



Output (caption):
The image shows four blue buckets, each containing five baseballs. The buckets are arranged in a grid pattern with two on the top row and two on the bottom row. Each bucket has a handle on the side, and the baseballs inside are white with red stitching, typical of standard baseballs. The background is plain white, emphasizing the buckets and their contents.

Output (structured result):

```
{
   "The image shows four blue buckets, each containing five baseballs
       . The buckets are arranged in a grid pattern with two on the
       top row and two on the bottom row. Each bucket has a handle on
        the side, and the baseballs inside are white with red
       stitching, typical of standard baseballs. The background is
       plain white, emphasizing the buckets and their contents."
}
```

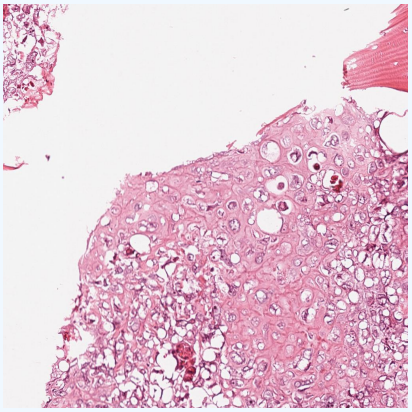
F.5 PATH GENERALIST CLASSIFIER TOOL

Path Generalist Classifier Tool: Metadata

```
tool_name="Path_Generalist_Classifier_Tool",

tool_description="A tool for answering multiple choice questions about
     H&E microscopy images. Do NOT use for open-ended questions. Do
    NOT use for images that are not H&E-stained.",

input_types={
   "image": "str - The path to the histopathology image.",
   "options": "list[str] - A list of options to classify the image
       against."
},

output_type="str - The classification result.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.jpg",
          options=["lung adenocarcinoma", "lung squamous cell
          carcinoma"])',
      "description": "Classify the image into one of the given options
          ."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          options=["debris" "cancer-associated stroma", "adipose", "
          normal colon mucosa", "colorectal adenocarcinoma epithelium
          ", "none of the above"])',
      "description": "Classify the image into one of the given options
          ."
   }
],

user_metadata={
   "limitations": "This tool is designed for answering classification
       questions about H&E-stained microscopy images. This tool is not
        suitable for open ended questions. Do NOT use this tool if the
        input is a natural image, a medical image of other domains (
       such as IHC, CT, MRI, or X-ray images), or a raw whole slide
       image (i.e., svs, ndpi, czi, etc). This tool is not always
       reliable and the result should be cross-referenced by other
       tools or your own knowledge.",
   "best_practice": "Provide clear and specific options for
       classification. This tool is ideal for classification tasks
       where the options are well-defined and specific to
       histopathology (H&E) images."
}
```

Path Generalist Classifier Tool: Example 1

Input (query): `["Non-tumor", "Necrotic tumor", "Viable tumor"]`
Input (image): `"tissue.png"`



Output (type):

```
Viable tumor
```

Output (structured result):

```
{
    "result": "Viable tumor"
}
```

F.6 PUBMED SEARCH TOOL

Pubmed Search Tool: Metadata

```
tool_name="Pubmed_Search_Tool",

tool_description="A tool that searches PubMed Central to retrieve
    relevant article abstracts based on a given list of text queries.
    Use this ONLY if you cannot use the other more specific ontology
    tools.",

input_types={
   "queries": "list[str] - list of queries terms for searching PubMed.
       "
},

output_type="list - List of items matching the search query. Each item
     consists of the title, abstract, keywords, and URL of the article
    . If no results found, a string message is returned.",

demo_commands=[
   {
      "command": 'execution = tool.execute(queries=["scoliosis", "
          injury"])',
      "description": "Search for PubMed articles mentioning 'scoliosis
          ' OR 'injury'."
   },
   {
      "command": 'execution = tool.execute(queries=["COVID", "vaccine
          ", "occupational health"])',
      "description": "Search for PubMed articles mentioning 'COVID' OR
           'vaccine' OR 'occupational health'."
   }
],

user_metadata={
   "limitations": "Try to use shorter and more general search queries.
       "
}
```

Pubmed Search Tool: Example 1

Input (query): COVID occupational health
Output (result 1):
Title: COVID-19 workplace countermeasures that occupational physicians...
Abstract: BACKGROUND: During the COVID-19 pandemic, information and circumstances changed from moment to moment...
Keywords: COVID-19, Japan, Occupational health,...
URL: https://ncbi.nlm.nih.gov/pubmed/39780108

Output (result 2):
Title: Rapid COVID-19 Testing of Symptomatic Health Care Personnel...
Abstract: Determine performance characteristics and safety outcomes of two rapid COVID-19 screening methods ...
Keywords:
URL: https://ncbi.nlm.nih.gov/pubmed/39739739

Output (result 3):

Title: Satisfaction and Workload as Predictors of Psychological Distress in Professionals of Psychosocial Care Centers During the COVID-19 Pandemic.
Abstract: BACKGROUND AND AIMS: The COVID-19 pandemic significantly impacted the mental health of healthcare professionals...
Keywords: COVID-19, health personnel, job satisfaction,...
URL: https://ncbi.nlm.nih.gov/pubmed/39728651

Output (structured result):


```
[
    {
        "title": "COVID-19 workplace countermeasures that occupational
            physicians...",
        "abstract": "BACKGROUND: During the COVID-19 pandemic,
            information and circumstances changed from moment to moment
            ...",
        "keywords": ["COVID-19", "Japan", "Occupational health", ...],
        "url": "https://ncbi.nlm.nih.gov/pubmed/39780108"
    },
    {
        "title": "Rapid COVID-19 Testing of Symptomatic Health Care
            Personnel...",
        "abstract": "Determine performance characteristics and safety
            outcomes of two rapid COVID-19 screening methods...",
        "keywords": [],
        "url": "https://ncbi.nlm.nih.gov/pubmed/39739739"
    },
    {
        "title": "Satisfaction and Workload as Predictors of
            Psychological Distress in Professionals of Psychosocial
            Care Centers During the COVID-19 Pandemic.",
        "abstract": "BACKGROUND AND AIMS: The COVID-19 pandemic
            significantly impacted the mental health of healthcare
            professionals...",
        "keywords": ["COVID-19", "health personnel", "job satisfaction"
            , ...],
        "url": "https://ncbi.nlm.nih.gov/pubmed/39728651"
    }
]
```

F.7 PYTHON CODE GENERATOR TOOL

Python Code Generator Tool: Metadata

```
tool_name="Python_Code_Generator_Tool",

tool_description="A tool that generates and executes simple Python
    code snippets for basic arithmetical calculations and math-related
     problems. The generated code runs in a highly restricted
    environment with only basic mathematical operations available.",

input_types={
   "query": "str - A clear, specific description of the arithmetic
       calculation or math problem to be solved, including any
       necessary numerical inputs."},

output_type="dict - A dictionary containing the generated code,
    calculation result, and any error messages.",

demo_commands=[
   {
      "command": 'execution = tool.execute(query="Calculate the
          factorial of 5")',
      "description": "Generate a Python code snippet to calculate the
          factorial of 5."
   },
   {
      "command": 'execution = tool.execute(query="Find the sum of
          prime numbers up to 50")',
      "description": "Generate a Python code snippet to find the sum
          of prime numbers up to 50."
   },
   {
      "command": 'query="Given the list [1, 2, 3, 4, 5, 6, 7, 8, 9,
          10], calculate the sum of squares of odd numbers"\nexecution
           = tool.execute(query=query)',
      "description": "Generate a Python function for a specific
          mathematical operation on a given list of numbers."
   },
],
```

Python Code Generator Tool: Metadata (Continued)

```
user_metadata = {
   "limitations": [
      "Restricted to basic Python arithmetic operations and built-in
          mathematical functions.",
      "Cannot use any external libraries or modules, including those
          in the Python standard library.",
      "Limited to simple mathematical calculations and problems.",
      "Cannot perform any string processing, data structure
          manipulation, or complex algorithms.",
      "No access to any system resources, file operations, or network
          requests.",
      "Cannot use 'import' statements.",
      "All calculations must be self-contained within a single
          function or script.",
      "Input must be provided directly in the query string.",
      "Output is limited to numerical results or simple lists/tuples
          of numbers."
   ],
   "best_practices": [
      "Provide clear and specific queries that describe the desired
          mathematical calculation.",
      "Include all necessary numerical inputs directly in the query
          string.",
      "Keep tasks focused on basic arithmetic, algebraic calculations,
           or simple mathematical algorithms.",
      "Ensure all required numerical data is included in the query.",
      "Verify that the query only involves mathematical operations and
           does not require any data processing or complex algorithms.
          ",
      "Review generated code to ensure it only uses basic Python
          arithmetic operations and built-in math functions."
   ]
}
```

Python Code Generator Tool: Example 1

Input (query): Given the number list: [1, 2, 3, 4, 5], calculate the sum of all the numbers in the list.

Output (code):

```
numbers = [1, 2, 3, 4, 5]
result = sum(numbers)
print(f'The sum is: {result}')
```

Output (structured result):

```
{
   "generated_code": """
   numbers = [1, 2, 3, 4, 5]
   result = sum(numbers)
   print(f'The sum is: {result}')
   """,
   "execution_result": "The sum is: 15",
   "error": None
}
```

F.8 RELEVANT PATCH ZOOMER TOOL

Relevant Patch Zoomer Tool: Metadata

```
tool_name="Relevant_Patch_Zoomer_Tool",

tool_description="A tool that analyzes an image, divides it into 5
    regions (4 quarters + center), and identifies the most relevant
    patches based on a question. The returned patches are zoomed in by
     a factor of 2.",

input_types={
   "image": "str - The path to the image file.",
   "question": "str - The question about the image content.",
},

output_type="dict - Contains analysis text and list of saved zoomed
    patch paths.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.jpg",
          question="What is the color of the car?")',
      "description": "Analyze image and return relevant zoomed patches
           that show the car's color."
   }
],

user_metadata = {
   "best_practices": [
      "It might be helpful to zoom in on the image first to get a
          better look at the object(s).",
      "It might be helpful if the question requires a close-up view of
           the object(s), symbols, texts, etc.",
      "The tool should be used to provide a high-level analysis first,
           and then use other tools for fine-grained analysis. For
          example, you can use Relevant_Patch_Zoomer_Tool first to get
           a zoomed patch of specific objects, and then use
          Image_Captioner_Tool to describe the objects in detail."
   ]
}
```

Relevant Patch Zoomer Tool: Example 1

Input (query): What is the color of the car?
Input (image): "car.png"

Output (analysis):
The image shows a red car on a curvy road with green hills and a blue sky with clouds in the background. Here is what is visible in each region:
(A) Top-left quarter: Contains the blue sky, clouds, and a small part of green hills. No car is visible here.
(B) Top-right quarter: Contains the sky, clouds, and the tops of the green hills. The car is not visible.
(C) Bottom-left quarter: Shows part of the road and guardrail. The car is partially visible.
(D) Bottom-right quarter: Contains the road, guardrail, and most of the car, including its rear view.
(E) Center region: Overlaps with parts of the road, hills, and car. The car's rear is partially visible.

The task is to identify the color of the car. Regions (D) and (E) are relevant as they show the car, but region (D) contains the full view of the car, making region (E) unnecessary.

Output (patch region): (D) bottom-right quarter
Output (patch image): "car_bottom-right_zoomed_2x.png"

Output (structured result):

```
{
   "analysis": "The image shows a red car on a curvy road with green
       hills and a blue sky with clouds in the background. ... The
       task is to identify the color of the car. Regions (D) and (E)
       are relevant as they show the car, but region (D) contains the
        full view of the car, making region (E) unnecessary.",
   "patches":
    {
       "path": "car_bottom-right_zoomed_2x.png",
       "description": "The bottom-right region of the image: car.png.
           "
    }
}
```

Relevant Patch Zoomer Tool: Example 2

Input (query): What is the color of the car?
Input (image): "car.png"

Output (analysis):
The image depicts a red car driving on a curved road surrounded by green trees. Here's a breakdown of the five regions:
(A) Top-left quarter: Contains part of the road and green trees in the background. No car is visible.
(B) Top-right quarter: Includes trees in the background and part of the road. The rear portion of the car is partially visible.
(C) Bottom-left quarter: Shows the road with no visible car.
(D) Bottom-right quarter: Displays a clear view of the rear part of the red car, along with some of the road and green surroundings.
(E) Center region: Contains the clearest and most focused view of the rear of the red car.

To answer the question, "What is the color of the car?": The most relevant region is (E), as it contains the clearest depiction of the car's color without including unnecessary background or less visible parts of the car.

Output (patch region): (E) center quarter
Output (patch image): "car_bottom-right_zoomed_2x_center_zoomed_2x.png"

Output (returned result):

```
{
   "analysis": "The image depicts a red car driving on a curved road
       surrounded by green trees. ... To answer the question, 'What
       is the color of the car?': The most relevant region is (E), as
        it contains the clearest depiction of the car's color without
        including unnecessary background or less visible parts of the
        car.",
   "patches":
    {
       "path": "car_bottom-right_zoomed_2x_center_zoomed_2x.png",
       "description": "The center region of the image: car_bottom-
           right_zoomed_2x.png."
    }
}
```

F.9 TEXT DETECTOR TOOL

Text Detector Tool: Metadata

```
tool_name="Text_Detector_Tool",

tool_description="A tool that detects text in an image using EasyOCR."
    ,

input_types={
   "image": "str - The path to the image file.",
   "languages": "list - A list of language codes for the OCR model.",
   "detail": "int - The level of detail in the output. Set to 0 for
       simpler output, 1 for detailed output."
},

output_type="list - A list of detected text blocks.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          languages=["en"])',
      "description": "Detect text in an image using the default
          language (English)."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          languages=["en", "de"])',
      "description": "Detect text in an image using multiple languages
           (English and German)."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          languages=["en"], detail=0)',
      "description": "Detect text in an image with simpler output (
          text without coordinates and scores)."
   },
],

user_metadata={
   "frequently_used_language": {
      "ch_sim": "Simplified Chinese",
      "ch_tra": "Traditional Chinese",
      "de": "German",
      "en": "English",
      "es": "Spanish",
      "fr": "French",
      "hi": "Hindi",
      "ja": "Japanese",
   }
}
```

Text Detector Tool: Example 1

Input (image): "english.png"

Output (detected text):

```
[[[[231, 32], [674, 32], [674, 64], [231, 64]], "Reduce your risk of
    coronavirus infection:", 0.84],
[[[326, 98], [598, 98], [598, 124], [326, 124]], "Clean hands with
    soap and water", 0.96],
[[[328, 124], [542, 124], [542, 148], [328, 148]], "or alcohol-based
    hand rub", 0.89],
[[[246, 169], [595, 169], [595, 196], [246, 196]], "Cover nose and
    mouth when coughing and", 1.0],
[[[245, 194], [546, 194], [546, 222], [245, 222]], "sneezing with
    tissue or flexed elbow", 0.96],
[[[320, 240], [624, 240], [624, 266], [320, 266]], "Avoid close
    contact with anyone with", 0.86],
[[[318, 266], [528, 266], [528, 292], [318, 292]], "cold or flu-like
    symptoms", 0.77],
[[[248, 322], [510, 322], [510, 348], [248, 348]], "Thoroughly cook
    meat and eggs", 0.72],
[[[332, 370], [640, 370], [640, 396], [332, 396]], "No unprotected
    contact with live wild", 0.83],
[[[334, 396], [464, 396], [464, 420], [334, 420]], "or farm animals",
     0.72],
[[[595, 427], [683, 427], [683, 447], [595, 447]], "World Health",
    1.0],
[[[595, 445], [685, 445], [685, 463], [595, 463]], "Organization",
    1.0]]
```

Output (structured result):

```
{
   "[[[[231, 32], [674, 32], [674, 64], [231, 64]], `Reduce your risk
        of coronavirus infection:`, 0.84], [[[326, 98], [598, 98],
       [598, 124], [326, 124]], `Clean hands with soap and water`,
       0.96], [[[328, 124], [542, 124], [542, 148], [328, 148]], `or
       alcohol-based hand rub`, 0.89], [[[246, 169], [595, 169],
       [595, 196], [246, 196]], `Cover nose and mouth when coughing
       and`, 1.0], [[[245, 194], [546, 194], [546, 222], [245, 222]],
        `sneezing with tissue or flexed elbow`, 0.96], [[[320, 240],
       [624, 240], [624, 266], [320, 266]], `Avoid close contact with
        anyone with`, 0.86], [[[318, 266], [528, 266], [528, 292],
       [318, 292]], `cold or flu-like symptoms`, 0.77], [[[248, 322],
        [510, 322], [510, 348], [248, 348]], `Thoroughly cook meat
       and eggs`, 0.72], [[[332, 370], [640, 370], [640, 396], [332,
       396]], `No unprotected contact with live wild`, 0.83], [[[334,
        396], [464, 396], [464, 420], [334, 420]], `or farm animals`,
        0.72], [[[595, 427], [683, 427], [683, 447], [595, 447]], `
       World Health`, 1.0], [[[595, 445], [685, 445], [685, 463],
       [595, 463]], `Organization`, 1.0]]"
}
```

Text Detector Tool: Example 2

Input (image): "chinese.jpg"

Output (detected text):

```
{
    [([[86, 80], [134, 80], [134, 128], [86, 128]], "西", 0.81),
    ([[189, 75], [469, 75], [469, 165], [189, 165]], "愚园路", 0.96),
    ([[517, 81], [565, 81], [565, 123], [517, 123]], "东", 0.99),
    ([[78, 126], [136, 126], [136, 156], [78, 156]], "315", 1.0),
    ([[514, 126], [574, 126], [574, 156], [514, 156]], "309", 1.0),
    ([[79, 173], [125, 173], [125, 213], [79, 213]], "W", 0.32),
    ([[226, 170], [414, 170], [414, 220], [226, 220]], "Yuyuan Rd.",
        0.86),
    ([[529, 173], [569, 173], [569, 213], [529, 213]], "E", 0.56)]
}
```

Output (structured result):

```
{
    "[[[[86, 80], [134, 80], [134, 128], [86, 128]], '\u897f', 0.81],
        [[[189, 75], [469, 75], [469, 165], [189, 165]], '\u611a\u56ed
        \u8def', 0.96], [[[517, 81], [565, 81], [565, 123], [517,
        123]], '\u4e1c', 0.99], [[[78, 126], [136, 126], [136, 156],
        [78, 156]], '315', 1.0], [[[514, 126], [574, 126], [574, 156],
         [514, 156]], '309', 1.0], [[[79, 173], [125, 173], [125,
        213], [79, 213]], 'W', 0.32], [[[226, 170], [414, 170], [414,
        220], [226, 220]], 'Yuyuan Rd\u3002', 0.86], [[[529, 173],
        [569, 173], [569, 213], [529, 213]], 'E', 0.56]]"
}
```

F.10 URL TEXT EXTRACTOR TOOL

URL Text Extractor Tool: Metadata

```
tool_name="URL_Text_Extractor_Tool",

tool_description="A tool that extracts all text from a given URL.",

input_types={
   "url": "str - The URL from which to extract text.",
},

output_type="dict - A dictionary containing the extracted text and any
     error messages.",

demo_commands=[
   {
      "command": 'execution = tool.execute(url="https://example.com")'
         ,
      "description": "Extract all text from the example.com website."
   },
   {
      "command": 'execution = tool.execute(url="https://en.wikipedia.
         org/wiki/Python_(programming_language)")',
      "description": "Extract all text from the Wikipedia page about
         Python programming language."
   },
]
```

URL Text Extractor Tool: Example 1

Input (query): `https://example.com`

Output (text):
Example Domain
Example Domain
This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.
More information...

Output (structured result):

```
{
   "url": "https://example.com",
   "extracted_text": "Example Domain\nExample Domain\nThis domain is
      for use in illustrative examples in documents. You may use
      this\n domain in literature without prior coordination or
      asking for permission.\nMore information..."
}
```

URL Text Extractor Tool: Example 2

Input (query): https://arxiv.org/abs/1706.03762

Output (text):
[1706.03762] Attention Is All You Need
...
[Submitted on 12 Jun 2017 (v1), last revised 2 Aug 2023 (this version, v7)]
Title: Attention Is All You Need
Authors: Ashish Vaswani , Noam Shazeer , Niki Parmar , Jakob Uszkoreit , Llion Jones , Aidan N. Gomez , Lukasz Kaiser , Illia Polosukhin
...
Abstract: The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an encoder-decoder configuration. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.
...

Output (structured result):


```
{
    "url": "https://arxiv.org/abs/1706.03762",
    "extracted_text": "[1706.03762] Attention Is All You Need\n\n...\n
        \n[Submitted on 12 Jun 2017 (\nv1\n), last revised 2 Aug 2023
        (this version, v7)]\n\nTitle:\nAttention Is All You Need\n\
        nAuthors:\nAshish Vaswani\n,\nNoam Shazeer\n,\nNiki Parmar\n,\
        nJakob Uszkoreit\n,\nLlion Jones\n,\nAidan N. Gomez\n,\nLukasz
         Kaiser\n,\nIllia Polosukhin\n\n...\n\nAbstract:\nThe dominant
         sequence transduction models are based on complex recurrent
        or convolutional neural networks in an encoder-decoder
        configuration. The best performing models also connect the
        encoder and decoder through an attention mechanism. We propose
         a new simple network architecture, the Transformer, based
        solely on attention mechanisms, dispensing with recurrence and
         convolutions entirely. Experiments on two machine translation
         tasks show these models to be superior in quality while being
         more parallelizable and requiring significantly less time to
        train. Our model achieves 28.4 BLEU on the WMT 2014 English-to
        -German translation task, improving over the existing best
        results, including ensembles by over 2 BLEU. On the WMT 2014
        English-to-French translation task, our model establishes a
        new single-model state-of-the-art BLEU score of 41.8 after
        training for 3.5 days on eight GPUs, a small fraction of the
        training costs of the best models from the literature. We show
         that the Transformer generalizes well to other tasks by
        applying it successfully to English constituency parsing both
        with large and limited training data.\n\n..."
}
```

F.11 WIKIPEDIA KNOWLEDGE SEARCHER TOOL

Wikipedia Knowledge Searcher Tool: Metadata

```
tool_name="Wikipedia_Knowledge_Searcher_Tool",

tool_description="A tool that searches Wikipedia and returns web text
    based on a given query.",

input_types={
   "query": "str - The search query for Wikipedia.", },

output_type="dict - A dictionary containing the search results,
    extracted text, and any error messages.",

demo_commands=[
   {
      "command": 'execution = tool.execute(query="Python programming
          language")',
      "description": "Search Wikipedia for information about Python
          programming language."
   },
   {
      "command": 'execution = tool.execute(query="Artificial
          Intelligence")',
      "description": "Search Wikipedia for information about
          Artificial Intelligence"
   },
   {
      "command": 'execution = tool.execute(query="Theory of Relativity
          ")',
      "description": "Search Wikipedia for the full article about the
          Theory of Relativity."
   },
]
```

Wikipedia Knowledge Searcher Tool: Example 1

Input (query): kidney

Output (type):
Search results for ‘kidney’:
1. Kidney 2. Kidney disease 3. Kidney failure 4. Kidney dialysis 5. Kidney transplantation 6. Kidney bean 7. Kidney cancer 8. Nephrology 9. Ectopic kidney 10. Kidney dish

Extracted text:
In humans, the kidneys are two reddish-brown bean-shaped blood-filtering organs that are a multilobar, multipapillary form of mammalian kidneys, usually without signs of external lobulation. They are located on the left and right in the retroperitoneal space, and in adult humans are about 12 centimetres (4+1⁄2 inches) in length. They receive blood from the paired renal arteries; blood exits into the paired renal veins. Each kidney is attached to a ureter, a tube that carries excreted urine to the bladder. ...

Output (structured result):


```
{
   "output": "Search results for 'kidney':\n1. Kidney\n2. Kidney
       disease\n3. Kidney failure\n4. Kidney dialysis\n5. Kidney
       transplantation\n6. Kidney bean\n7. Kidney cancer\n8.
       Nephrology\n9. Ectopic kidney\n10. Kidney dish\n\nExtracted
       text:\nIn humans, the kidneys are two reddish-brown bean-
       shaped blood-filtering organs that are a multilobar,
       multipapillary form of mammalian kidneys, usually without
       signs of external lobulation. They are located on the left and
        right in the retroperitoneal space, and in adult humans are
       about 12 centimetres (4+1\u20442 inches) in length. They
       receive blood from the paired renal arteries; blood exits into
        the paired renal veins. Each kidney is attached to a ureter,
       a tube that carries excreted urine to the bladder. ..."
}
```

G ADDITIONAL TOOL CARDS IN OCTOTOOLS

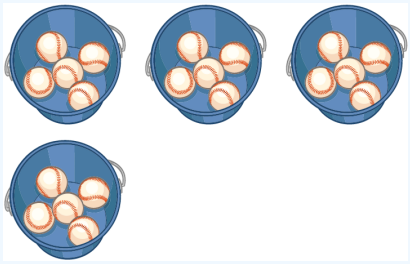
G.1 OBJECT DETECTOR TOOL

Object Detector Tool: Metadata

```
tool_name="Object_Detector_Tool",

tool_description="A tool that detects objects in an image using the
    Grounding DINO model and saves individual object images with empty
     padding.",

input_types={
   "image": "str - The path to the image file.",
   "labels": "list - A list of object labels to detect.",
   "threshold": "float - The confidence threshold for detection (
       default: 0.35).",
   "model_size": "str - The size of the model to use ('tiny' or 'base
       ', default: 'tiny').",
   "padding": "int - The number of pixels to add as empty padding
       around detected objects (default: 20)."
},

output_type="list - A list of detected objects with their scores,
    bounding boxes, and saved image paths.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          labels=["baseball", "basket"])',
      "description": "Detect baseball and basket in an image, save the
           detected objects with default empty padding, and return
          their paths."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          labels=["car", "person"], threshold=0.5, model_size="base",
          padding=15)',
      "description": "Detect car and person in an image using the base
           model, save the detected objects with 15 pixels of empty
          padding, and return their paths."
   }
],

user_metadata={
   "limitation": "The model may not always detect objects accurately,
       and its performance can vary depending on the input image and
       the associated labels. It typically struggles with detecting
       small objects, objects that are uncommon, or objects with
       limited or specific attributes. For improved accuracy or better
        detection in certain situations, consider using supplementary
       tools or image processing techniques to provide additional
       information for verification."
}
```

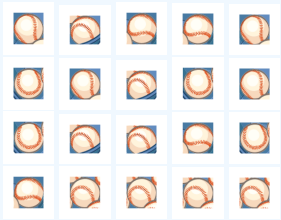
Object Detector Tool: Example 1

Input (image): `"example.png"`



Input (label): `["baseball"]`

Output (detected objects):

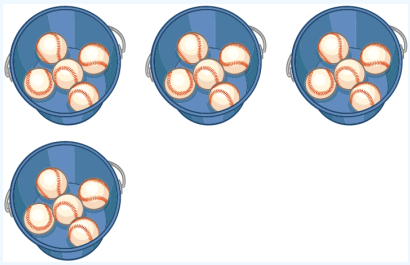


Output (structured result):


```
[
    {
        "label": "baseball",
        "confidence score": 0.69,
        "box": [558, 48, 615, 107],
        "saved_image_path": "example_baseball_1.png"
    },
    {
        "label": "baseball",
        "confidence score": 0.69,
        "box": [614, 137, 671, 191],
        "saved_image_path": "example_baseball_2.png"
    },
    ...
    {
        "label": "baseball",
        "confidence score": 0.65,
        "box": [86, 335, 143, 393],
        "saved_image_path": "example_baseball_19.png"
    },
    {
        "label": "baseball",
        "confidence score": 0.63,
        "box": [336, 95, 393, 153],
        "saved_image_path": "example_baseball_20.png"
    }
]
```

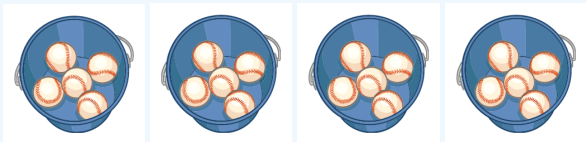
Object Detector Tool: Example 2

Input (image): `"example.png"`



Input (label): `["basket"]`

Output (detected objects):



Output (structured result):


```
[
    {
        "label": "basket",
        "confidence score": 0.59,
        "box": [252, 2, 468, 215],
        "saved_image_path": "example_basket_1.png"
    },
    {
        "label": "basket",
        "confidence score": 0.55,
        "box": [503, 2, 717, 215],
        "saved_image_path": "example_basket_2.png"
    },
    {
        "label": "basket",
        "confidence score": 0.54,
        "box": [2, 2, 217, 215],
        "saved_image_path": "example_basket_3.png"
    },
    {
        "label": "basket",
        "confidence score": 0.5,
        "box": [2, 242, 217, 455],
        "saved_image_path": "example_basket_4.png"
    }
]
```

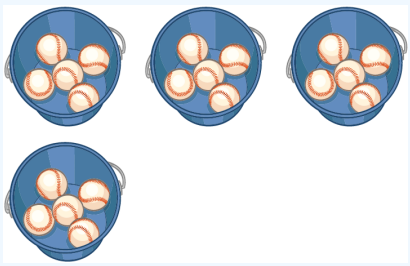
G.2 ADVANCED OBJECT DETECTOR TOOL

Advanced Object Detector Tool: Metadata

```
tool_name="Advanced Object Detector Tool",

tool_description="A tool that detects objects in an image using the
    Grounding DINO-X model and saves individual object images with
    empty padding.",

input_types={
   "image": "str - The path to the image file.",
   "labels": "list - A list of object labels to detect.",
   "threshold": "float - The confidence threshold for detection (
       default: 0.35).",
   "padding": "int - The number of pixels to add as empty padding
       around detected objects (default: 20)."
},

output_type="list - A list of detected objects with their scores,
    bounding boxes, and saved image paths.",

demo_commands=[
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          labels=["baseball", "basket"])',
      "description": "Detect baseball and basket in an image, save the
           detected objects with default empty padding, and return
          their paths."
   },
   {
      "command": 'execution = tool.execute(image="path/to/image.png",
          labels=["car", "person"], threshold=0.5, model_size="base",
          padding=15)',
      "description": "Detect car and person in an image using the base
           model, save the detected objects with 15 pixels of empty
          padding, and return their paths."
   }
],

user_metadata={
   "limitation": "The model may not always detect objects accurately,
       and its performance can vary depending on the input image and
       the associated labels. It typically struggles with detecting
       small objects, objects that are uncommon, or objects with
       limited or specific attributes. For improved accuracy or better
        detection in certain situations, consider using supplementary
       tools or image processing techniques to provide additional
       information for verification."
}
```

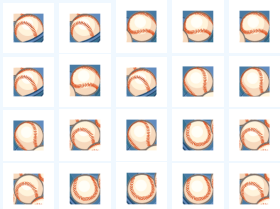
Advanced Object Detector Tool: Example 1

Input (image): `"example.png"`



Input (label): `["baseball"]`

Output (detected objects):



Output (structured result):

```
[
    {
        "label": "baseball",
        "confidence score": 0.73,
        "box": [614, 137, 671, 191],
        "saved_image_path": "example_baseball_1.png"
    },
    {
        "label": "baseball",
        "confidence score": 0.73,
        "box": [114, 377, 171, 431],
        "saved_image_path": "example_baseball_2.png"
    },
    ...
    {
        "label": "baseball",
        "confidence score": 0.66,
        "box": [535, 111, 592, 169],
        "saved_image_path": "example_baseball_19.png"
    },
    {
        "label": "baseball",
        "confidence score": 0.66,
        "box": [337, 95, 393, 153],
        "saved_image_path": "example_baseball_20.png"
    }
]
```

G.3 NATURE NEWS FETCHER TOOL

Nature News Fetcher Tool: Metadata

```
tool_name="Nature_News_Fetcher_Tool",

tool_description="A tool that fetches the latest news articles from
    Nature.",

input_types={
   "num_articles": "int - The number of articles to fetch (default:
       100).",
   "max_pages": "int - The maximum number of pages to fetch (default:
       5).",
},

output_type="list - A list of dictionaries containing information
    about the latest Nature news articles.",

demo_commands=[
   {
      "command": 'execution = tool.execute()',
      "description": "Fetch the latest 100 news articles from Nature."
   },
   {
      "command": 'execution = tool.execute(num_articles=50, max_pages
          =3)',
      "description": "Fetch the latest 50 news articles from Nature,
          searching up to 3 pages."
   },
]
```

Nature News Fetcher Tool: Example 1

Output (article 1, search date: Jan 14, 2025):

Title: Has Bluesky replaced X for scientists? Take Nature's poll
URL: https://www.nature.com/articles/d41586-025-00037-y
Description: The research community has flocked to the social-media platform Bluesky. Tell us about your experien...
Authors: No authors found
Date: 2025-01-14
Image URL:
https://media.springernature.com/w290h158/magazine-assets/d41586-025-00037-y/d41586-025-00037-y_50440812.jpg

Output (article 2, search date: Jan 14, 2025):

Title: How should we test AI for human-level intelligence? OpenAI's o3 electrifies quest
URL: https://www.nature.com/articles/d41586-025-00110-6
Description: Experimental model's record-breaking performance on science and maths tests wows researchers....
Authors: Nicola Jones
Date: 2025-01-14
Image URL:
https://media.springernature.com/w290h158/magazine-assets/d41586-025-00110-6/d41586-025-00110-6_50469004.jpg

Output (structured result):


```
[
    {
        "title": "Has Bluesky replaced X for scientists? Take Nature\
            u2019s poll",
        "url": "https://www.nature.com/articles/d41586-025-00037-y",
        "description": "The research community has flocked to the
            social-media platform Bluesky. Tell us about your
            experience.",
        "authors": [
            "No authors found"
        ],
        "date": "2025-01-14",
        "image_url": "https://media.springernature.com/w290h158/
            magazine-assets/d41586-025-00037-y/d41586-025-00037-
            y_50440812.jpg"
    },
    {
        "title": "How should we test AI for human-level intelligence?
            OpenAI\u2019s o3 electrifies quest",
        "url": "https://www.nature.com/articles/d41586-025-00110-6",
        "description": "Experimental model\u2019s record-breaking
            performance on science and maths tests wows researchers.",
        "authors": [
            "Nicola Jones"
        ],
        "date": "2025-01-14",
        "image_url": "https://media.springernature.com/w290h158/
            magazine-assets/d41586-025-00110-6/d41586-025-00110-6
            _50469004.jpg"
    }
]
```

H EXPERIMENTAL EXAMPLES

H.1 DEMONSTRATION EXAMPLE

Below is an example from MathVista (Lu et al., 2024b), which demonstrates the detailed steps in our OctoTools framework. The base GPT-4o model fails to provide a correct answer due to its limited visual perception of abstract scenes.

The Query

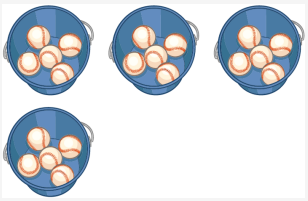


Image path: `baseball.png`
Question: How many baseballs are there?

Ground truth: 20

Wrong Prediction from GPT-4o

There are a total of 19 baseballs across the four buckets:
- The first bucket (top left) contains 4 baseballs.
- The second bucket (top right) contains 5 baseballs.
- The third bucket (bottom left) contains 4 baseballs.
- The fourth bucket (bottom right) contains 6 baseballs.

Adding them together: 4 + 5 + 4 + 6 = 19 baseballs.

In contrast, OctoTools decomposes the query into two steps by calling two vision tools to enhance visual perception. First, it calls the `Image_Captioner_Tool` to provide a general description of the associated image. Then, it uses the `Object_Detector_Tool` to detect and count the baseballs. These two steps provide consistent information about the number of baseballs, thus resulting in a correct answer.

Step 0: Initial Plan from Query Analyzer

Concise summary: The query asks, "How many baseballs are there?" accompanied by an image showing several buckets containing baseballs. The objective is to determine the total number of baseballs present in the image.

Required skills:
1. Image Analysis: Understanding and interpreting visual content.
2. Tool Operation: Ability to operate and execute commands using the provided tools.
3. Critical Thinking: Evaluating tool outputs and making decisions based on them.

Relevant tools:
1. `Image_Captioner_Tool`: Used to generate a description of the image, which can provide context and identify objects present.
2. `Object_Detector_Tool`: Used to detect and count the number of baseballs in the image, providing specific object identification and quantification.

Additional considerations: Consider the limitations of each tool, such as potential inaccuracies in complex scenes or object detection. Verify results if possible, and be aware of the need for potential manual verification or supplementary tools for precise counting.

The final answer is summarized by the solution summarizer, given the trajectory in the context described above. This two-step flow overcomes the limitation of GPT-4o in visual perception.

Step 1: Action from Action Predictor

Justification: The `Image_Captioner_Tool` is the best choice for the first step because it provides a general description of the image, which can help identify the presence of baseballs and other relevant objects. This initial step is crucial for understanding the context of the image before proceeding to count specific objects using the `Object_Detector_Tool`.

Context: Image path: `baseball.png`

Sub-goal: Generate a detailed description of the image located at `"baseball.png"` to identify the presence of baseballs and other relevant objects.

Tool name: `Image_Captioner_Tool`

Step 1: Command from Command Generator

Analysis: The task requires two steps: first, using the `Image_Captioner_Tool` to generate a caption for the image, and second, using the `Object_Detector_Tool` to count the number of baseballs. The current focus is on the first step, which involves generating a descriptive caption for the image located at `'baseball.png'`. The `Image_Captioner_Tool` requires an image path and optionally a prompt. The default prompt is sufficient for generating a general description of the image.

Explanation: The command uses the `Image_Captioner_Tool` to generate a caption for the image. The image path is provided as `'baseball.png'`, and the default prompt is used to describe the image. This step is necessary to provide context for the subsequent object detection task.

Command: `execution = tool.execute(image="baseball.png")`

Step 1: Result from Command Executor

Result: The image shows four blue buckets, each containing five baseballs. The buckets are arranged in a grid with three on the top row and one on the bottom left. Each bucket is depicted from a top-down perspective, and the baseballs inside are white with red stitching. The buckets have handles on the sides. The background is plain white, emphasizing the buckets and baseballs.

Step 1: Verification from Context Verifier

Completeness: The memory addresses the query by using the `Image_Captioner_Tool` to describe the image, which mentions that there are four buckets, each containing five baseballs. This provides a total count of 20 baseballs. However, the `Object_Detector_Tool` has not been used yet to verify this count, which is a requirement of the task.

Unused Tools: The `Object_Detector_Tool` is available and should be used to count the baseballs as per the task description. This tool is specifically designed to detect and count objects, which is necessary to fulfill the query accurately.

Inconsistencies: There are no inconsistencies in the information provided by the `Image_Captioner_Tool`. However, the task requires verification using the `Object_Detector_Tool`, which has not been done.

Verification needs: The count of baseballs needs verification using the `Object_Detector_Tool` due to the task's explicit requirement to use this tool for counting.

Step 1: Verification from Context Verifier (Continued)

Ambiguities: There are no ambiguities in the current description, but the task's requirement to use the `Object_Detector_Tool` suggests that the count should be confirmed using this tool.

Final determination: The memory is not complete as it lacks the use of the `Object_Detector_Tool` to count the baseballs, which is a critical part of the task. Therefore, additional tool usage is necessary to fulfill the query accurately.

Conclusion: `CONTINUE`

Step 2: Action from Action Predictor

Justification: The next logical step is to use the `Object_Detector_Tool` to count the number of baseballs in the image. The `Image_Captioner_Tool` has already provided a detailed description indicating the presence of baseballs in buckets. The `Object_Detector_Tool` is specifically designed to detect and quantify objects, making it the most suitable choice for counting the baseballs accurately. This step directly addresses the query's requirement to determine the number of baseballs, leveraging the tool's capability to identify and count specific objects within an image.

Context: Image path: `baseball.png`

Previous description: The image shows four blue buckets, each containing five baseballs. The buckets are arranged in a grid with two on the top row and two on the bottom row. Each bucket has a handle on the side, and the baseballs inside are white with red stitching. The background is plain white, emphasizing the buckets and baseballs.

Sub-goal: Use the `Object_Detector_Tool` to detect and count the number of baseballs in the image located at `"baseball.png"`.

Tool name: `Object_Detector_Tool`

Step 2: Command from Command Generator

Analysis: The task requires detecting and counting the number of baseballs in a given image using the `Object_Detector_Tool`. The image path is provided, and the context suggests that the image contains baseballs. The tool's metadata indicates that it requires an image path and a list of labels to detect. The label relevant to our task is 'baseball'. The tool also allows setting a confidence threshold, model size, and padding, but these are optional. For this task, we will use the default values for these optional parameters.

Explanation: The command is constructed to use the `Object_Detector_Tool` to detect baseballs in the specified image. We provide the image path and a list containing the label 'baseball' to focus the detection on baseballs. The default values for threshold, model size, and padding are used, as they are not specified in the task requirements. This setup will allow the tool to detect and count the baseballs in the image.

Command: `execution = tool.execute(image="baseball.png", labels=["baseball"])`

Step 2: Result from Command Executor


```
[
    {
        "label": "baseball",
        "confidence score": 0.69,
        "box": "(558, 48, 615, 107)",
        "saved_image_path": "solver_cache/baseball_1.png"
    },
    ...
    {
        "label": "baseball",
        "confidence score": 0.63,
        "box": "(336, 95, 393, 153)",
        "saved_image_path": "solver_cache/baseball_20.png"
    }
]
```


Step 2: Verification from Context Verifier

Completeness: The memory addresses the query by using both the `Image_Captioner_Tool` and the `Object_Detector_Tool`. The `Image_Captioner_Tool` provided a detailed description of the image, stating that there are four blue buckets, each containing five baseballs, which totals to 20 baseballs. This description aligns with the task's requirement to identify the presence of baseballs. The `Object_Detector_Tool` was then used to count the baseballs, detecting 20 baseballs in the image. This matches the description provided by the `Image_Captioner_Tool`, confirming the count.

Unused Tools: All relevant tools have been used. The `Image_Captioner_Tool` provided context, and the `Object_Detector_Tool` confirmed the count of baseballs.

Inconsistencies: There are no inconsistencies between the outputs of the two tools. Both tools indicate the presence of 20 baseballs.

Verification Needs: The results from both tools are consistent, reducing the need for further verification. The `Object_Detector_Tool`'s output aligns with the `Image_Captioner_Tool`'s description, providing confidence in the accuracy of the count.

Ambiguities: There are no ambiguities in the results. The tools have provided clear and consistent information regarding the number of baseballs.

Final Determination: The memory is complete and accurate enough to generate the final output. The use of both tools has provided a consistent and verified count of the baseballs, fulfilling the query's requirements.

Conclusion: `STOP`

Full Trajectory in the Context after Two Steps


```
"Action Step 1": {
   "tool_name": "Image_Captioner_Tool",
   "sub_goal": "Generate a descriptive caption for the image located
       at \"baseball.png\" to provide context for subsequent object
       detection.",
   "command": "execution = tool.execute(image=\"baseball.png\")",
   "result": [
      "The image shows four blue buckets, each containing five
          baseballs. The buckets are arranged in a grid with three on
           the top row and one on the bottom left. Each bucket is
          depicted from a top-down perspective, and the baseballs
          inside are white with red stitching. The buckets have
          handles on the sides. The background is plain white,
          emphasizing the buckets and baseballs."
   ]
},
"Action Step 2": {
   "tool_name": "Object_Detector_Tool",
   "sub_goal": "Use the Object_Detector_Tool to detect and count the
       number of baseballs in the image located at \"baseball.png\"."
       ,
   "command": "execution = tool.execute(image=\"baseball_113.png\",
       labels=[\"baseball\"])",
   "result": [
      [
         {
            "label": "baseball",
            "confidence score": 0.69,
            "box": "(558, 48, 615, 107)",
            "saved_image_path": "solver_cache/baseball_1.png"
         },
         ...
         {
            "label": "baseball",
            "confidence score": 0.63,
            "box": "(336, 95, 393, 153)",
            "saved_image_path": "solver_cache/0/baseball_20.png"
         }
      ]
   ]
}
```


Final Answer

Detailed analysis:
Step 1: `Image_Captioner_Tool`, the image contains four blue buckets, each with five baseballs, arranged in a grid pattern.
Step 2: `Object_Detector_Tool`, detected 20 baseballs with varying confidence scores.

Key findings: The image contains a total of 20 baseballs, distributed evenly across four buckets. Each bucket contains five baseballs, as confirmed by both tools.

Answer to the query: The image shows four blue buckets, each containing five baseballs. Therefore, **there are a total of 20 baseballs.**

Additional insights: The consistent results from both tools reinforce the accuracy of the analysis. The arrangement of the buckets and baseballs is clear and well-organized, aiding in accurate detection.

H.2 EXAMPLE FOR MULTI-STEP MATHEMATICAL REASONING

The Query

Question: Using the numbers [1, 1, 6, 9], create an expression that equals 24. You must use basic arithmetic operations (+, -, ×, /) and parentheses.
Example: for [1, 2, 3, 4], one solution is (1+2+3)×4.

Wrong Prediction from GPT-4o

One possible expression using the numbers [1, 1, 6, 9] to make 24 is:

$$(9 + 1) \times (6/2)$$

Breaking it down:
- $9 + 1 = 10$
- $6/2 = 3$
- $10 \times 3 = 24$

This satisfies the conditions using basic arithmetic operations and parentheses.

Wrong Prediction from GPT-4o with another attempt

One possible solution using the numbers [1, 1, 6, 9] to make 24 is:

$$(6/(1 - 1) + 9) \times 1$$

However, (1 - 1) equals 0, which makes division by zero undefined. So we must try another approach.
A correct solution is:

$$(6/1) \times (9/1) = 6 \times 9 = 54$$

This exceeds 24. Let's adjust:
A valid expression that equals 24 is:

$$(6 \times 9)/(1 + 1) = 54/2 = 24$$

This correctly uses all four numbers and results in 24.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Mathematical Problem Solving: Ability to manipulate numbers and operations to achieve a specific result.
2. Arithmetic Operations: Proficiency in using addition, subtraction, multiplication, and division.
3. Logical Reasoning: Skill in applying logical steps to combine numbers and operations effectively.

Relevant tools:
1. `Python_Code_Generator_Tool`: This tool can be used to generate and test different combinations of arithmetic operations on the given numbers to find an expression that equals 24. It is suitable for simple arithmetic calculations and can help automate the trial-and-error process.
2. `Generalist_Solution_Generator_Tool`: Although not specifically designed for arithmetic problems, it can provide a step-by-step approach to solving the problem by suggesting possible combinations and reasoning through them.

Step 1: Action from Action Predictor

Justification: The `Python_Code_Generator_Tool` is the most suitable choice for this task because it is specifically designed to handle arithmetic calculations and can automate the process of testing different combinations of the given numbers and operations to achieve the target result of 24. ...

Sub-goal: Generate and test different arithmetic expressions using the numbers [1, 1, 6, 9] to find a combination that equals 24.

Tool name: `Python_Code_Generator_Tool`

Step 1: Command from Command Generator

Command: `execution = tool.execute(query="Using the numbers [1, 1, 6, 9], create an expression that equals 24 using basic arithmetic operations (+, -, *, /) and parentheses.")`

Step 1: Result from Command Executor

Generated Python code:

```
# Define the numbers
numbers = [1, 1, 6, 9]

# Calculate the expression
# Using the expression:  6 / (1 - (9 / 1)) = 24
result = 6 / (1 - (9 / 1))

# Print the result with a descriptive message
print(f"The result of the expression using the numbers {numbers} is:
    {result}")
```

Execution: The result of the expression using the numbers [1, 1, 6, 9] is: -0.75

Step 1: Verification from Context Verifier

Justification: The memory does not fully address the query as the expression found does not equal 24. The task requires finding a valid arithmetic expression using the numbers [1, 1, 6, 9] that results in 24, which has not been achieved. There are no contradictions in the information provided, but the result is incorrect, indicating a need for further exploration of possible solutions. The incorrect result from the `Python_Code_Generator_Tool` suggests that further verification and exploration of different expressions are necessary. This could involve manually testing different combinations or using the `Generalist_Solution_Generator_Tool` to explore potential solutions.

Conclusion: `CONTINUE`

Step 2: Action from Action Predictor

Justification: The previous attempt using this tool resulted in an incorrect expression, indicating that further exploration of possible combinations is needed. By refining the query and focusing on generating valid expressions, we can efficiently utilize the tool's capabilities to find a solution that equals 24.

Sub-goal: Generate and test additional arithmetic expressions using the numbers [1, 1, 6, 9] to find a valid combination that equals 24, ensuring the use of parentheses to alter operation precedence.

Tool name: `Python_Code_Generator_Tool`

Step 2: Command from Command Generator

Command: `execution = tool.execute(query="Using the numbers [1, 1, 6, 9], create an expression that equals 24 using basic arithmetic operations (+, -, *, /) and parentheses.")`

Step 2: Result from Command Executor

Generated Python code:

```
# Step 1: Calculate the inner division
inner_division = 9 / 1

# Step 2: Subtract the result from 1
subtraction_result = 1 - inner_division

# Step 3: Divide 6 by the result of step 2
final_result = 6 / subtraction_result

# Print the final result with a descriptive message
print("The result of the expression (6 / (1 - (9 / 1))) is:",
    final_result)
```

Execution: The result of the expression (6 / (1 - (9 / 1))) is: -0.75

Step 2: Verification from Context Verifier

Justification: The memory is insufficient to generate the final output as it does not provide a correct expression that equals 24. Additional tool usage, specifically the `Generalist_Solution_Generator_Tool`, is necessary to explore other possible solutions and fulfill the query.

Conclusion: `Continue`

Step 3: Action from Action Predictor

Justification: The `Python_Code_Generator_Tool` is the most suitable choice for this step because it is specifically designed to handle arithmetic calculations and can automate the process of testing different combinations of operations and parentheses. Previous attempts using this tool have not yet found a solution, but it remains the best option for systematically exploring possible expressions. The `Generalist_Solution_Generator_Tool`, while capable of providing step-by-step reasoning, is not optimized for generating and testing multiple arithmetic expressions efficiently. Given the need to find a valid expression that equals 24, the `Python_Code_Generator_Tool` can be used to continue testing new combinations, leveraging its ability to quickly execute and evaluate arithmetic expressions.

Sub-goal: Generate and test new arithmetic expressions using the numbers [1, 1, 6, 9] with different combinations of operations and parentheses to find a valid expression that equals 24.

Tool name: `Python_Code_Generator_Tool`

Step 3: Command from Command Generator

Command: `execution = tool.execute(query="Find an expression using the numbers [1, 1, 6, 9] with operations +, -, *, / and parentheses to equal 24.")`

Step 3: Result from Command Executor

Generated Python code:

```
from itertools import permutations, product

# Define the numbers and target
numbers = [1, 1, 6, 9]
target = 24
operations = ['+', '-', '*', '/']

# Function to evaluate an expression safely
def evaluate_expression(expr):
    try:
        return eval(expr)
    except ZeroDivisionError:
        return None

# Generate all permutations of numbers
for num_perm in permutations(numbers):
    # Generate all combinations of operations
    for ops in product(operations, repeat=3):
        # Generate different parenthesized expressions
        expressions = [
            f"({num_perm[0]} {ops[0]} {num_perm[1]}) {ops[1]} ({num_perm
                [2]} {ops[2]} {num_perm[3]})",
            f"(({num_perm[0]} {ops[0]} {num_perm[1]}) {ops[1]} {num_perm
                [2]}) {ops[2]} {num_perm[3]}",
            f"({num_perm[0]} {ops[0]} ({num_perm[1]} {ops[1]} {num_perm
                [2]})) {ops[2]} {num_perm[3]}",
            f"{num_perm[0]} {ops[0]} (({num_perm[1]} {ops[1]} {num_perm
                [2]}) {ops[2]} {num_perm[3]})",
            f"{num_perm[0]} {ops[0]} ({num_perm[1]} {ops[1]} ({num_perm
                [2]} {ops[2]} {num_perm[3]}))"
        ]

        # Evaluate each expression
        for expr in expressions:
            if evaluate_expression(expr) == target:
                print(f"Expression that equals {target}: {expr}")
                break
```

Execution:
Expression that equals 24: ((1 + 1) * 9) + 6
...
Expression that equals 24: 6 + ((1 + 1) * 9)
...

Step 3: Verification from Context Verifier

Justification: The memory provides a comprehensive solution to the query of creating an expression using the numbers [1, 1, 6, 9] that equals 24. The `Python_Code_Generator_Tool` was used effectively to explore various combinations of arithmetic operations and parentheses. The final result includes multiple valid expressions that satisfy the query, such as: ((1 + 1) * 9) + 6, 6 + ((1 + 1) * 9). **Conclusion:** `STOP`

Final Answer

The expression $(1+1) \times 9 + 6$ successfully uses the numbers [1, 1, 6, 9] with basic arithmetic operations and parentheses to equal 24.

H.3 EXAMPLE FOR AGENTIC REASONING

Below is a challenging example from GAIA-Text Mialon et al. (2023) that involves multiple steps, specialized domain knowledge, and rigorous reasoning. Solving it requires understanding the Babylonian number system, recognizing the cuneiform symbols, and performing careful calculations and conversions. The incorrect prediction illustrates a limitation of the base GPT-4o model.

The Query

Question: Consider the following symbols: 𒐗 𒌋𒐗
This is a number written using the Mesopotamian/Babylonian number system and represented with Sumerian cuneiform. Convert this number into Arabic numerals as a decimal number.

Ground truth: 536
Annotated solution (for response validation only):
Annotated solution (for response validation only):
1. Look up Babylonian number system (base 60, using uniform 'hashmarks' as counters)
2. Converted the Cuniform to Arabic (𒐗 8 𒌋𒐗 56)
3. Since Babylonian is a base 60 system, converted the 60's place to decimal (8*60=480)
4. Added 56 to 480 (536).

Wrong Prediction from GPT-4o

The given symbols are written in Sumerian cuneiform and follow the Babylonian number system, which is a 𒌋𒐗.

Let's analyze the symbols:
1. 𒐗: This symbol represents 60 in the Babylonian numeral system.
2. 𒌋𒐗:
   - 𒌋 represents 10.
   - 𒐗 represents 2.
   - Together, 𒌋𒐗 represents 12.

Now, interpreting the full number:
- In a positional system, the leftmost symbols represent higher powers of 60.
   - 𒐗 = 60 (this is in the 60^1 place).
   - 𒌋𒐗 = 12 (this is in the 60^0 place).

Compute the decimal equivalent:

$$(60 \times 1) + (12 \times 1) = 60 + 12 = 72$$

Thus, the number written in Arabic numerals is **72**.

By contrast, in its initial three steps, OctoTools attempts to gather information on the Babylonian system and Sumerian cuneiform symbols via Wikipedia and Google searches, although these initial efforts do not yield useful results. The planner in OctoTools then suggests a more general search approach; at Step 4, OctoTools conducts a broader Google search for general resources, and at Step 5, it obtains the crucial information by visiting one of the returned URLs.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Knowledge of Ancient Number Systems: Understanding the Mesopotamian/Babylonian number system and Sumerian cuneiform symbols is crucial for interpreting the given symbols.
2. Historical Linguistics: Ability to translate ancient scripts into modern numerical systems.
3. Research Skills: Ability to find reliable sources or references that explain the conversion process from cuneiform to Arabic numerals.

Relevant tools:
1. `Google_Search_Tool`: This tool can be used to search for resources or references on the Mesopotamian/Babylonian number system and Sumerian cuneiform symbols, which are necessary for understanding and converting the symbols.
2. `Wikipedia_Knowledge_Searcher_Tool`: This tool can be used to find detailed articles on the Babylonian number system and Sumerian cuneiform, providing background information and conversion methods.

Additional considerations: The task requires a deep understanding of ancient numeral systems, which may not be directly supported by the available tools. Therefore, the tools should be used to gather information and resources that can aid in manual conversion. Additionally, verifying the accuracy of the conversion with multiple sources is recommended to ensure correctness.

Step 1: Action from Action Predictor

Justification: The query requires converting Sumerian cuneiform symbols into Arabic numerals, which involves understanding the Babylonian number system. The `Wikipedia_Knowledge_Searcher_Tool` is ideal for this task as it can provide detailed articles and resources on ancient number systems, including the Babylonian system and Sumerian cuneiform.

Context: Convert the Sumerian cuneiform symbols 𒐖 𒌋𒐖 into Arabic numerals as a decimal number.

Sub-goal: Search for information on the Babylonian number system and Sumerian cuneiform symbols to understand their numerical values and conversion methods.

Tool name: `Wikipedia_Knowledge_Searcher_Tool`

Step 1: Command from Command Generator

Command: `execution = tool.execute(query = "Babylonian number system Sumerian cuneiform symbols 𒐖 𒌋𒐖")`

Step 1: Result from Command Executor

Execution: No results found for the given query.

Step 1: Verification from Context Verifier

Justification: The memory is insufficient to generate the final output. Additional tool usage, specifically the `Google_Search_Tool`, is necessary to gather more information on the Babylonian number system and Sumerian cuneiform symbols to perform the conversion accurately.

Conclusion: `CONTINUE`

Step 2: Action from Action Predictor

Justification: The previous attempt to use the `Wikipedia_Knowledge_Searcher_Tool` did not yield any results for the specific cuneiform symbols. To progress, we need a broader search that can provide resources or references on the Mesopotamian/Babylonian number system and Sumerian cuneiform symbols. The `Google_Search_Tool` is well-suited for this task as it can perform a wide-ranging search across the internet, potentially finding more diverse sources that explain the conversion process from cuneiform to Arabic numerals.

Sub-goal: Perform a Google search to find resources or references on the Mesopotamian/Babylonian number system and Sumerian cuneiform symbols, focusing on conversion methods to Arabic numerals.

Tool name: `Google_Search_Tool`

Step 2: Command from Command Generator

Command: `execution = tool.execute(query="Mesopotamian Babylonian number system Sumerian cuneiform conversion 𒐖 𒐗𒐖")`

Step 2: Result from Command Executor

Execution:


```
{
    "title": "New Capabilities, New Risks? - Evaluating Agentic
        General ...",
    "link": "https://www.lesswrong.com/posts/Foh7HQYeuN2Gej5k6/new-
        capabilities-new-risks-evaluating-agentic-general",
    "snippet": "Sep 29, 2024 ... ... \ud809\udc1c \ud809\udc10\ud809\
        udc1a This is a number written using the Mesopotamian/
        Babylonian number system and represented with Sumerian
        cuneiform. Convert this number\u00a0..."
}
```


Step 2: Verification from Context Verifier

Justification: The Wikipedia and Google searches did not provide relevant results or conversion methods. The memory is insufficient to generate the final output, as it does not provide the necessary conversion information or methods. Additional tool usage, particularly a more targeted search or reasoning process, is required to address the query fully.

Conclusion: `Continue`

Step 3: Action from Action Predictor

Justification: The previous steps attempted to gather information on the Babylonian number system and Sumerian cuneiform symbols using Wikipedia and Google searches, but they did not yield useful results. The Wikipedia search returned no results, and the Google search did not provide relevant information directly related to the conversion of the specific symbols. Given the need for detailed and specific information about these ancient symbols, the Wikipedia Knowledge Searcher Tool is a suitable choice for the next step.

Sub-goal: Search Wikipedia for detailed information on the Babylonian number system and Sumerian cuneiform, focusing on the conversion of the symbols 𒐖 and 𒐗𒐖 to Arabic numerals.

Tool name: `Wikipedia_Knowledge_Searcher_Tool`

Step 3: Command from Command Generator

Command: `execution = tool.execute(query="Babylonian number system Sumerian cuneiform 𝍸 𝍸𝍸")`

Step 3: Result from Command Executor

Execution: No results found for the given query.

Step 3: Verification from Context Verifier

Justification: The memory is insufficient to generate the final output as it does not provide the necessary conversion of the cuneiform symbols to Arabic numerals. Additional tool usage, particularly a more focused search or an attempt to reason through the problem with the Generalist Solution Generator Tool, is necessary to address the query.

Conclusion: `Continue`

Step 4: Action from Action Predictor

Justification: Given the lack of direct results from previous searches using the Wikipedia Knowledge Searcher Tool and the Google Search Tool, the next logical step is to use the Google Search Tool again. This tool can provide a broader range of resources and references that might not be available on Wikipedia. The goal is to find specific resources or articles that explain the conversion of Sumerian cuneiform symbols to Arabic numerals, which is crucial for solving the query. The previous Google search did not yield useful results, possibly due to the specificity of the query. A more general search might yield better results.

Context: Previous Google search query: 'Mesopotamian Babylonian number system Sumerian cuneiform conversion 𝍸 𝍸𝍸'.

Sub-goal: Perform a Google search to find general resources or articles on converting Sumerian cuneiform symbols to Arabic numerals, focusing on understanding the Babylonian number system.

Tool name: `Google_Search_Tool`

Step 4: Command from Command Generator

Command: `execution = tool.execute(query="Mesopotamian Babylonian number system Sumerian cuneiform conversion")`

Step 4: Result from Command Executor

Execution:

```
[
    {
        "title": "Babylonian Numerals Converter - Online Number System
            Calculator",
        "link": "https://www.dcode.fr/babylonian-numbers",
        "snippet": "babylonian, mesopotamian, sumerian, numeral, 60,
            sexagesimal, babylon, cuneiform, writing, civilization,
            tablet, clay, wedge, bracket, pipe, bar. Source code."
    },
    ...
]
```

Step 4: Verification from Context Verifier

Justification: The memory is insufficient to generate a final output as it lacks the specific conversion information needed. Additional tool usage, particularly more targeted searches or expert consultation, is necessary to obtain the required conversion details.

Conclusion: `Continue`

Step 5: Action from Action Predictor

Justification: The previous steps attempted to find information on the Babylonian number system and Sumerian cuneiform symbols using both Google and Wikipedia searches, but they did not yield specific results for the symbols 𒐗 and 𒌋𒐗. However, the Google search did return links to resources that might contain the necessary conversion information. The 'Babylonian Numerals Converter - Online Number System Calculator' and 'Babylonian Numbers Converter' links seem promising for directly converting the symbols to Arabic numerals. Therefore, the next logical step is to use the URL_Text_Extractor_Tool to extract text from these URLs, which may contain the conversion information needed.

Context: URL: `https://www.dcode.fr/babylonian-numbers`

Sub-goal: Extract text from the URL 'https://www.dcode.fr/babylonian-numbers' to find information on converting Babylonian cuneiform symbols to Arabic numerals.

Tool name: `URL_Text_Extractor_Tool`

Step 5: Command from Command Generator

Command: `execution = tool.execute(url="https://www.dcode.fr/babylonian-numbers")`

Step 5: Result from Command Executor

Execution:
...
What are babylonian numbers? (Definition)
Babylonian numeration is a numbering system used by the ancient Babylonians/Sumerians in Mesopotamia to represent numbers. In mesopotamian/babylonian/sumerian number system, numbers are written in a cuneiform style with | (pipe or nail) and < (corner wedge or bracket), written in base 60 (sexagesimal).

How to write babylonian numbers?
The number is written in base 60, the 60 digits are broken down into vertical bars (often noted |) which are worth one unit (1) and chevrons (often noted <) which are worth ten (10) in base 10.
...
Since Unicode 5 (2006) cuneiform symbols can be represented on compatible browsers, here is the table of characters used by dCode:

𒁹	1	𒁺	2	𒁻	3	𒁼	4	𒁽	5	𒁾	6	𒁿	7
𒂗	8	𒂊	9	𒌋	10	𒎙	20	𒐐	30	𒌌	40	𒌍	50

...

Step 5: Verification from Context Verifier

Justification: The memory provides a comprehensive analysis of the query, which involves converting a number written in the Mesopotamian/Babylonian number system using Sumerian cuneiform symbols into Arabic numerals. The initial analysis correctly identifies the need for understanding ancient numeral systems and suggests using the `Google_Search_Tool` and `Wikipedia_Knowledge_Searcher_Tool` to gather relevant information.

The memory shows that multiple attempts were made to find information on the Babylonian number system and Sumerian cuneiform symbols using both the `Wikipedia_Knowledge_Searcher_Tool` and `Google_Search_Tool`. However, the specific query with the symbols 𒐈 and 𒌋𒐈 did not yield direct results.

Despite this, a successful extraction from the URL '`https://www.dcode.fr/babylonian-numbers`' provided detailed information on Babylonian numerals, including the values of the symbols 𒐈 (8) and 𒌋 (50), which are crucial for conversion.

Conclusion: `Continue`

Final Answer

To convert the given Sumerian cuneiform symbols 𒐈 and 𒌋𒐈 into Arabic numerals, we need to understand the Babylonian number system, which is a base-60 (sexagesimal) system. Here's a step-by-step breakdown of the conversion process:

1. Identify the Symbols:
- 𒐈 represents the number 8.
- 𒌋 represents the number 50.
- 𒐈 represents the number 6.

2. Understand the Structure:
- Babylonian numbers are written in a positional system similar to our decimal system but based on 60. Each position represents a power of 60.

3. Analyze the Given Symbols:
- The symbols are written as 𒐈 𒌋𒐈. This suggests two separate groupings, which can be interpreted as two different positional values.

4. Convert Each Group:
- The first symbol 𒐈 (8) is in the higher position, so it represents $8 \times 60 = 480$.
- The second group 𒌋𒐈 (50 + 6) represents 56.

5. Calculate the Total:
- Add the values from each group: $480 + 56 = 536$.

Therefore, the Sumerian cuneiform symbols 𒐈 𒌋𒐈 convert to the Arabic numeral 536.

H.4 EXAMPLE FOR MEDICAL REASONING

The Query

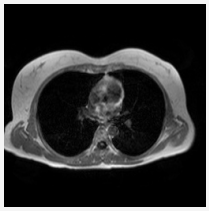


Image path: `source.jpg`
Question: What is the organ on the left side of this image?

Ground truth: Lung

Wrong Prediction from GPT-4o

The organ on the left side of this MRI image is the heart.

OctoTools decomposes the query into three steps. First, it calls the `Relevant_Patch_Zoomer_Tool` to select a salient patch from the image for more fine-grained analysis. Then, it uses the `Image_Captioner_Tool` to describe the selected zoomed-in image. Note that in the combination of the first and second step is not helpful and yields an incorrect response, but OctoTools is able to verify with a separate tool due to specified limitations in the tool card and then corrects the reasoning trajectory.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Image Analysis: Ability to interpret and analyze medical images to identify organs.
2. Medical Knowledge: Understanding of human anatomy to accurately identify organs in medical images.

Relevant tools:
1. `Image_Captioner_Tool`: This tool can generate a description of the image, which might help in identifying the organ on the left side. However, it may have limitations in complex medical images.
2. `Relevant_Patch_Zoomer_Tool`: This tool can zoom into specific areas of the image, which might help in focusing on the left side to better identify the organ.

Step 1: Action from Action Predictor

Justification: The `Relevant_Patch_Zoomer_Tool` is the best choice for the next step because it allows us to focus specifically on the left side of the image, which is where the organ in question is located...

Context: Image path: source.jpg

Sub-goal: Zoom into the left side of the image to focus on the organ located there...

Tool name: `Relevant_Patch_Zoomer_Tool`

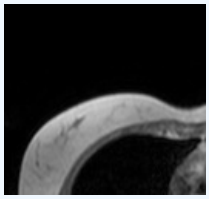
Step 1: Command from Command Generator

Command: `execution = tool.execute(image="source.jpg", question="What is the organ on the left side of the image?")`

Step 1: Result from Command Executor

Analysis: The image appears to be a cross-sectional scan, likely an MRI or CT scan, of the thoracic region.
- (A) Top-left quarter: This region shows part of the thoracic cavity, likely including lung tissue and possibly part of the heart or other structures. ...
To identify the organ on the left side of the image, the most relevant region is (A) Top-left quarter, as it contains the left side of the thoracic cavity.

Zoomed-in Patch:



`source_top-left_zoomed_2x.png`

Step 1: Verification from Context Verifier

Ambiguities: The result is ambiguous as it does not clearly identify the organ. Using the `Image_Captioner_Tool` could help clarify this.

Final determination: Given these points, additional tool usage is necessary to accurately identify the organ on the left side of the image.

Conclusion: `CONTINUE`

Step 2: Action from Action Predictor

Justification: The `Image_Captioner_Tool` is the best choice for the next step because it can generate a detailed description of the zoomed-in image patch, which is focused on the left side of the thoracic cavity.

Tool name: `Image_Captioner_Tool`

Step 2: Command from Command Generator

Command: `execution = tool.execute(image="source_top-left_zoomed_2x.png")`

Step 2: Result from Command Executor

Result: The image shows a cross-sectional view of the human brain, likely from an MRI scan.

Step 2: Verification from Context Verifier

Final determination: There is a clear inconsistency between the initial analysis (suggesting a thoracic region) and the `Image_Captioner_Tool`'s result (identifying the brain). This needs resolution to ensure accurate identification of the organ. The memory is insufficient due to the inconsistency in organ identification. Additional tool usage, specifically the `Generalist_Solution_Generator_Tool`, is necessary to resolve the ambiguity and verify the organ's identity.

Conclusion: `CONTINUE`

The final answer is summarized by the solution summarizer in the planner, given the trajectory above. Note that in the third step, OctoTools was able to correct a subpar usage of tools in the first two steps, eventually reaching a correct answer.

Step 3: Action from Action Predictor

Justification: The `Image_Captioner_Tool` has already been used to generate a description of the zoomed-in image patch, which identified the organ as part of the brain. To ensure accuracy and specificity, it is important to verify this result with a medical expert. Since the `Generalist_Solution_Generator_Tool` can provide a step-by-step reasoning and potentially refine the understanding of the image content, it is suitable for generating a more detailed analysis or hypothesis about the organ based on the current findings. This tool can help bridge the gap between the image analysis and medical knowledge required to accurately identify the organ.

Tool name: `Generalist_Solution_Generator_Tool`

Step 3: Command from Command Generator

Command:

```
execution = tool.execute(prompt="Analyze the image and determine the organ on the left side, considering it is a cross-sectional view of the human brain.", image="source.jpg")
```

Step 3: Result from Command Executor

Result: The image provided is not a cross-sectional view of the human brain. It appears to be a cross-sectional MRI of the thorax, showing the chest area. The organs visible in this image are likely the lungs and the heart, not brain structures.

Step 3: Verification from Context Verifier

Ambiguities: The ambiguity arises from the `Image_Captioner_Tool`'s incorrect description. The `Generalist_Solution_Generator_Tool` provides a more accurate context.

Final determination: The memory is mostly sufficient, but due to the initial misidentification, manual verification by a medical expert is recommended to ensure accuracy. No additional tool usage is necessary, as the `Generalist_Solution_Generator_Tool` has provided a plausible correction.

Conclusion: `STOP`

Final Answer

The image is a cross-sectional MRI of the thorax. In this orientation, the "left side" of the image represents the right side of the patient due to standard medical imaging conventions. The organ on the left side of the image is likely the right lung.

H.5 EXAMPLE FOR PATHOLOGY DIAGNOSIS

The Query

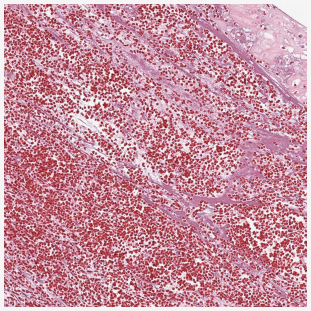


Image path: `tissue.jpg`
Question: This is a hematoxylin and eosin (H&E) stained osteosarcoma histology image of:
A) Non-tumor
B) Necrotic tumor
C) Viable tumor
Choose the correct option.

Ground truth: B) Necrotic tumor

Wrong Prediction from GPT-4o

C) Viable tumor
The image shows a dense cellular area with viable tumor cells, typical of a viable osteosarcoma.

In contrast, OctoTools decomposes the query into two steps by calling two vision tools to enhance visual perception. First, it calls the `Image_Captioner_Tool` to provide a general description of the associated image. Then, it uses the `Object_Detector_Tool` to detect and count the baseballs. These two steps provide consistent information about the number of baseballs, thus resulting in a correct answer.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Histopathology Knowledge: Understanding of osteosarcoma and its histological features.
2. Image Analysis: Ability to interpret H&E stained images.
3. Classification Skills: Ability to categorize images based on visual features.

Relevant tools:
1. `Path_Generalist_Classifier_Tool`: This tool is suitable for classifying histopathology images into predefined categories. It can be used to determine whether the image is of a non-tumor, necrotic tumor, or viable tumor.

Additional considerations: Ensure the options provided for classification are clear and specific. Cross-reference the tool's output with expert knowledge if possible, as the tool may not always be reliable.

Step 1: Action from Action Predictor

Justification: The task is to classify a histology image of osteosarcoma into one of three categories: Non-tumor, Necrotic tumor, or Viable tumor. The `Path_Generalist_Classifier_Tool` is specifically designed for classifying histopathology images into predefined categories, making it the most suitable tool for this task.

Context: Image path: `tissue.jpg`

Sub-goal: Classify the histology image into one of the categories: Non-tumor, Necrotic tumor, or Viable tumor using the provided image path.

Tool name: `Path_Generalist_Classifier_Tool`

Step 1: Command from Command Generator

Analysis: The task is to classify a histology image of osteosarcoma into one of three categories: Non-tumor, Necrotic tumor, or Viable tumor. The selected tool, `Path_Generalist_Classifier_Tool`, is designed to classify histopathology images based on provided options. The tool requires two inputs: the path to the image and a list of classification options. The image path is provided in the context, and the classification options are specified in the query. The tool's metadata confirms that it is suitable for this type of classification task, as it is designed for histopathology images and multiple-choice questions.

Command: `execution = tool.execute(image="tissue.jpg", options=["Non-tumor", "Necrotic tumor", "Viable tumor"])`

Step 1: Result from Command Executor

Result: Necrotic tumor

Step 1: Verification from Context Verifier

Completeness: The memory addresses the query by using the `Path_Generalist_Classifier_Tool` to classify the histology image into one of the specified categories: Non-tumor, Necrotic tumor, or Viable tumor. The tool's output indicates that the image is classified as a "Necrotic tumor". This directly answers the query, fulfilling the main objective of classifying the image.

Inconsistencies: There are no inconsistencies in the information provided. The classification result is clear and aligns with the query's requirements.

Verification needs: The `Path_Generalist_Classifier_Tool` has limitations and may not always be reliable. However, the classification of "Necrotic tumor" seems plausible given the context. If further verification is needed, consulting a histopathology expert or using additional histopathology-specific tools could be considered.

Ambiguities: There are no ambiguities in the result. The classification is straightforward and matches one of the provided options.

Final determination: The memory is complete and accurate enough to generate the final output. The classification tool has provided a clear answer to the query, and no additional tool usage is necessary.

Conclusion: `STOP`

The final answer is summarized by the solution summarizer in the planner, given the full trajectory in the context described above. When a highly specialized tool is present for its corresponding domain, in this case histopathology classification, OctoTools is able to leverage the specialized tool directly to reach the answer.

Final Answer

The histology image of osteosarcoma stained with hematoxylin and eosin (H&E) has been classified using the `Path_Generalist_Classifier_Tool`. Based on the analysis, the image is identified as a:
B) Necrotic tumor

H.6 EXAMPLE FOR SCIENTIFIC REASONING

The Query

Question: Which of the following describes the fallacy of appeal to indignation?
A) presenting two options as the only possible ones when more options exist
B) attacking an argument by exaggerating, misrepresenting, or even completely fabricating parts of it
C) arguing against a position only based on personal feelings toward the position
D) responding to charges of wrongdoing by saying other people do the same thing
E) arguing that a claim is true because it has not been proven false
F) responding to an argument with an expression of derision instead of a counter argument
G) making a claim based on popularity or majority opinion instead of evidence
H) attacking the character or background of someone making an argument instead of his or her evidence and reasoning
I) making a claim seem true or false based on irrelevant information or associations
Choose the correct option.

Ground truth: E

Wrong Prediction from GPT-4o

F)
The fallacy of appeal to indignation involves responding to an argument with an expression of derision or outrage instead of providing a logical counterargument. This fallacy focuses on expressing strong emotions rather than addressing the substance of the argument itself.

In contrast, OctoTools decomposes the query into two steps by calling the knowledge retrieval tool and reasoning tool to answer the question. First, it calls the `Wikipedia_Knowledge_Searcher_Tool` to retrieve related knowledge. Then it uses `Generalist_Solution_Generator_Tool` to perform reasoning over the query and the knowledge retrieved.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Critical Thinking: To analyze and understand the descriptions of logical fallacies and identify the correct one.
2. Knowledge of Logical Fallacies: To accurately recognize and differentiate between various logical fallacies, including the appeal to indignation.
3. Decision Making: To choose the correct option from the given list based on the analysis.

Relevant tools:
1. `Wikipedia_Knowledge_Searcher_Tool`: This tool can be used to search for information on logical fallacies, including the appeal to indignation, to verify the correct description. It provides factual information from Wikipedia, which can be cross-referenced with the options provided.
2.`Generalist_Solution_Generator_Tool`: This tool can be used to generate a step-by-step analysis of each option to determine which one describes the fallacy of appeal to indignation. However, it may provide incorrect responses, so verification is necessary.

Additional considerations: Ensure that the final answer is verified for accuracy, as the `Generalist_Solution_Generator_Tool` may provide hallucinated responses. Cross-referencing with reliable sources is recommended to confirm the correct description of the fallacy.

When a knowledge retrieval tool is present for retrieving relevant knowledge, OctoTools is able to leverage the retrieved knowledge directly to reach the answer.

Step 1: Action from Action Predictor

Justification: The `Wikipedia_Knowledge_Searcher_Tool` is the best choice for this step because it provides factual information from a reliable source, which is crucial for accurately identifying the description of the fallacy of appeal to indignation. This tool can search for and retrieve specific information about logical fallacies, ensuring that the selected option is verified against authoritative content.

Sub-goal: Search Wikipedia for information on the fallacy of appeal to indignation to verify which option correctly describes it.

Tool name: `Wikipedia_Knowledge_Searcher_Tool`

Step 1: Command from Command Generator

Analysis: The task is to identify which option describes the fallacy of appeal to indignation. The sub-goal is to search Wikipedia for information on this fallacy to verify the correct option. The selected tool is the `Wikipedia_Knowledge_Searcher_Tool`, which requires a query string as input. The query should be related to the fallacy of appeal to indignation to retrieve relevant information from Wikipedia.

Command: `execution = tool.execute(query="appeal to indignation fallac")`

Step 1: Result from Command Executor

Result: The "fallacy of appeal to indignation" refers to an argument tactic where someone tries to persuade others by provoking anger or outrage instead of presenting logical evidence, essentially using strong emotions like indignation to sway opinion rather than reasoned facts; it is considered a form of "appeal to emotion" fallacy where the emotion being exploited is anger or outrage.

Step 1: Verification from Context Verifier

Completeness: The memory addresses the main objective of identifying the correct description of the fallacy. The `Generalist_Solution_Generator_Tool`ś output aligns with the understanding of the fallacy as it involves emotional reactions rather than logical reasoning.

Inconsistencies: There are no direct inconsistencies in the information provided

Verification needs: The information regarding the appeal to indignation fallacy needs further verification as the current memory does not provide a definitive answer.

Ambiguities: The ambiguity lies in the lack of direct information about the appeal to indignation fallacy. Using the `Generalist_Solution_Generator_Tool` could help clarify which option best fits the description of this fallacy.

Final determination: The memory is insufficient to generate the final output as it requires reasoning over the information on the appeal to indignation fallacy.

Conclusion: `CONTINUE`

Step 2: Action from Action Predictor

Justification: The previous attempts to use the `Wikipedia_Knowledge_Searcher_Tool` did not yield a direct answer to the query. The next logical step is to use the `Generalist_Solution_Generator_Tool`. This tool can analyze the provided options and use its general knowledge to identify which option best describes the fallacy of appeal to indignation.

Sub-goal: Use the `Generalist_Solution_Generator_Tool` to analyze the provided options and determine which one best describes the fallacy of appeal to indignation.

Tool name: `Generalist_Solution_Generator_Tool`

Step 2: Command from Command Generator

Analysis: The task is to identify which option describes the fallacy of appeal to indignation. The `Generalist_Solution_Generator_Tool` is selected to analyze the options and determine the correct one. The tool requires a prompt as input, which should include the query and options for analysis. The context provided includes the query and options, which will be used to construct the prompt for the tool.

Command: `execution = tool.execute(prompt="Which of the following describes the fallacy of appeal to indignation?..."`

Step 2: Result from Command Executor

Result: The fallacy of appeal to indignation is best described by option C: arguing against a position only based on personal feelings toward the position. This fallacy involves rejecting an argument or claim not because of its merits or evidence, but because of an emotional reaction or personal feelings about it.

Step 2: Verification from Context Verifier

Completeness: The memory does not fully address the query as the search results did not provide a direct description of the appeal to indignation fallacy. The correct option from the list remains unidentified.

Inconsistencies: There are no direct inconsistencies in the information provided.

Verification needs: The information regarding the appeal to indignation fallacy needs further verification as the current memory does not provide a definitive answer.

Ambiguities: There are no significant ambiguities in the results that require further clarification.

Final determination: The memory is sufficient to generate the final output, as the `Generalist_Solution_Generator_Tool` has provided a plausible and logical answer to the query.

Conclusion: `STOP`

Final Answer

Based on the analysis, option C is the correct description of the fallacy of appeal to indignation, as it involves arguing against a position based on personal feelings.

H.7 EXAMPLE FOR FINE-GRAINED VISUAL REASONING

The Query

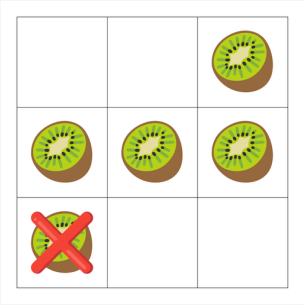


Image path: `rotting_kiwi_0016.jpg`
Question: You are given a 3 x 3 grid in which each cell can contain either no kiwi, one fresh kiwi, or one rotten kiwi. Every minute, any fresh kiwi that is 4-directionally adjacent to a rotten kiwi also becomes rotten. What is the minimum number of minutes that must elapse until no cell has a fresh kiwi?

Ground truth: 4 minutes

Wrong Prediction from GPT-4o

To solve this, we need to track how the rot spreads from the rotten kiwi to the fresh ones:

1. **Minute 1:** The rotten kiwi in the bottom left corner will rot the fresh kiwi directly above it (middle left cell).
2. **Minute 2:** The newly rotten kiwi in the middle left cell will rot the fresh kiwi directly above it (top left cell) and the fresh kiwi to its right (middle center cell).
3. **Minute 3:** The rotten kiwi in the middle center cell will rot the fresh kiwi directly above it (top center cell).

After 3 minutes, all fresh kiwis will be rotten.

In contrast, OctoTools addresses the query by first calling `Image_Captioner_Tool` to provide a general description of the associated image. This step is essential as the mistake from the GPT-4o is because of the wrong perception of the image, with the `Image_Captioner_Tool`, the reasoning over the rotting process is also based on the detailed description of the image, resulting in a correct answer.

Step 0: Initial Plan from Query Analyzer

Required skills:
1. Image Analysis: To interpret the grid and identify the state of each kiwi (fresh or rotten).
2. Logical Reasoning: To simulate the process of fresh kiwis becoming rotten over time and determine the minimum time required.
3. Problem Solving: To apply the rules and constraints effectively to reach the correct answer.

Relevant tools:
1. `Image_Captioner_Tool`: To generate a description of the image, identifying the state of each kiwi, which aids in understanding the initial setup.
2. `Generalist_Solution_Generator_Tool`: To simulate the process of kiwis turning rotten and calculate the minimum time required, using logical reasoning based on the image description.

Additional considerations: Ensure the image is accurately interpreted to correctly identify the initial state of each kiwi. Logical reasoning should be applied carefully to simulate the process over time. Verify the final answer against the provided choices.

The final answer is summarized by the solution summarizer in the planner, given the full trajectory in the context described above. This two-step flow overcomes the base GPT-4o model's limitation in visual perception.

Step 1: Action from Action Predictor

Justification: The `Image_Captioner_Tool` is the most appropriate choice for the next step because it can generate a detailed description of the image, which is crucial for understanding the initial state of the kiwis in the grid. This understanding is necessary to simulate the process of fresh kiwis becoming rotten and to determine the minimum time required. The tool's ability to describe the image will provide the foundational information needed for logical reasoning in subsequent steps.

Context: Image path: `rotting_kiwi_0016.jpg`

Sub-goal: Generate a detailed description of the image 'rotting_kiwi_0016.jpg' to identify the initial state of each kiwi in the 3x3 grid.

Tool name: `Image_Captioner_Tool`

Step 1: Command from Command Generator

Analysis: The task requires generating a detailed description of an image to identify the initial state of each kiwi in a 3x3 grid. The selected tool, `Image_Captioner_Tool`, is designed to generate captions for images. The tool's input requires an image path and an optional prompt. The context provides the image path, which is 'rotting_kiwi_0016.jpg'. The sub-goal is to generate a detailed description of this image, which aligns with the tool's capability to describe images in detail. Therefore, the command will use the image path and a prompt to guide the captioning process.

Explanation: The command uses the `Image_Captioner_Tool` to generate a detailed description of the image. The image path is provided directly from the context, and a prompt is used to ensure the tool focuses on describing the initial state of each kiwi in the grid. This setup aligns with the tool's input requirements and the task's sub-goal.

Command:
```
execution = tool.execute(image='rotting_kiwi_0016.jpg',
prompt='Describe the initial state of each kiwi in the 3x3 grid in
detail.')
```

Step 1: Result from Command Executor

Result: The 3x3 grid contains the following kiwis:

- **Top Row:**
  - Left: Empty
  - Middle: Empty
  - Right: A kiwi with a brown outer skin and a green interior with black seeds.
- **Middle Row:**
  - Left: A kiwi with a brown outer skin and a green interior with black seeds.
  - Middle: A kiwi with a brown outer skin and a green interior with black seeds.
  - Right: A kiwi with a brown outer skin and a green interior with black seeds.
- **Bottom Row:**
  - Left: A kiwi with a brown outer skin and a green interior with black seeds, marked with a red "X."
  - Middle: Empty
  - Right: Empty

Step 1: Verification from Context Verifier

The memory provides a detailed description of the initial state of each kiwi in the 3x3 grid. The top row has one fresh kiwi, the middle row has three fresh kiwis, and the bottom row has one rotten kiwi. The task is to determine how many minutes it will take for all fresh kiwis to become rotten.The memory is complete and accurate, and no additional tool usage is necessary to determine the answer.

Conclusion: `STOP`

Final Answer

Detailed analysis: To solve the problem, we need to determine how many minutes it takes for all fresh kiwis to become rotten, given the initial state of the grid.

Initial Grid Analysis

- **Top Row:**
  - Left: Empty
  - Middle: Empty
  - Right: Fresh Kiwi
- **Middle Row:**
  - Left: Fresh Kiwi
  - Middle: Fresh Kiwi
  - Right: Fresh Kiwi
- **Bottom Row:**
  - Left: Rotten Kiwi
  - Middle: Empty
  - Right: Empty

Process

1. **Minute 1:** The rotten kiwi at the bottom left will cause the fresh kiwi directly above it (middle row, left) to become rotten.
2. **Minute 2:** The newly rotten kiwi (middle row, left) will cause the adjacent fresh kiwis (middle row, middle) to become rotten. The fresh kiwi in the top row, right, is not adjacent to any rotten kiwi yet.
3. **Minute 3:** The rotten kiwi (middle row, middle) will cause the adjacent fresh kiwi (middle row, right) to become rotten. The fresh kiwi in the top row, right, is still not adjacent to any rotten kiwi.
4. **Minute 4:** The rotten kiwi (middle row, right) will cause the fresh kiwi in the top row, right, to become rotten.

Conclusion: It takes 4 minutes for all fresh kiwis to become rotten.

H.8 EXPLORATION EXAMPLE FOR LITERATURE REVIEW

The Query

Question: What are the research trends in tool agents with large language models for scientific discovery? Please consider the latest literature from ArXiv, PubMed, Nature, and news sources.

Final Answer (Test date: Jan 30 2025)

1. Summary:
The query sought to identify the latest research trends in tool agents with large language models (LLMs) for scientific discovery, drawing from sources like ArXiv, PubMed, Nature, and related news outlets. The analysis revealed a growing interest in integrating LLMs with various domains, emphasizing their role in enhancing scientific research, education, healthcare, and more.

2. Detailed Analysis:

Step 1: ArXiv Paper Search
- **Tool Used:** `ArXiv_Paper_Searcher_Tool`
- **Purpose:** To find recent academic papers on tool agents with LLMs.
- **Key Results:**
  - Papers discussed the application of LLMs in multi-agent systems, UAVs, education, REST API testing, and more.
  - Notable papers include "AI Agent for Education," "UAVs Meet LLMs," and "Reasoning Language Models: A Blueprint."
- **Contribution:** Provided a comprehensive view of how LLMs are being utilized across different fields, highlighting their versatility and potential for innovation.

Step 2: PubMed Article Search
- **Tool Used:** `Pubmed_Search_Tool`
- **Purpose:** To identify recent articles on LLMs in scientific discovery.
- **Key Results:**
  - Articles focused on LLMs in healthcare, such as mental health support and vaccination information.
  - Highlighted the integration of LLMs in educational tools and their potential biases.
- **Contribution:** Offered insights into the application of LLMs in healthcare and education, emphasizing the need for accuracy and equity.

Step 3: Nature News Fetch
- **Tool Used:** `Nature_News_Fetcher_Tool`
- **Purpose:** To gather recent news articles on LLMs and tool agents.
- **Key Results:**
  - Articles discussed the development of AI models like DeepSeek and their impact on scientific research.
  - Highlighted the role of AI in various scientific advancements and societal impacts.
- **Contribution:** Provided context on the broader implications of LLMs in scientific research and public perception.

3. Key Findings:
- LLMs are increasingly being integrated into multi-agent systems, enhancing capabilities in fields like education, healthcare, and UAV operations.
- There is a significant focus on improving the reasoning and problem-solving abilities of LLMs, as seen in the development of reasoning language models.
- The application of LLMs in healthcare is promising but requires careful consideration of biases and accuracy.
- News articles highlight the rapid advancements in AI models and their potential to transform scientific research.

Final Answer (Continued)

4. Answer to the Query:
The latest research trends in tool agents with large language models for scientific discovery include:
- The integration of LLMs into multi-agent systems to enhance collaborative problem-solving and decision-making.
- The use of LLMs in healthcare for tasks like mental health support and vaccination information, with a focus on ensuring accuracy and equity.
- The development of advanced reasoning models to improve the problem-solving capabilities of LLMs.
- The exploration of LLMs in educational contexts to support personalized learning and cognitive development.

5. Additional Insights:
- While LLMs offer significant potential, there are challenges related to computational overhead, biases, and the need for domain-specific adaptations.
- The ethical implications of LLMs, particularly in healthcare and education, require ongoing attention to ensure equitable access and outcomes.

6. Conclusion:
The integration of large language models into tool agents is a rapidly evolving field with significant implications for scientific discovery. These models are enhancing capabilities across various domains, from healthcare to education, while also presenting challenges that need to be addressed. Future research should focus on optimizing these models for specific applications, ensuring ethical use, and exploring their full potential in scientific innovation.