Advanced Tool Learning and Selection System (ATLASS): A Closed-Loop Framework Using LLM

Mohd Ariful Haque[1], Justin Williams[2], Sunzida Siddique[3], Md. Hujaifa Islam[4],
Hasmot Ali[5], Kishor Datta Gupta[6], Roy George[7]
Dept. of Cyber Physical Systems, Clark Atlanta University, USA[1,2,6,7]
Dept. of Computer Science and Engineering[3,4,5], Daffodil International University, Bangladesh[3,5]
Ahsanullah University of Science and Technology, Bangladesh[4]
mohdariful.haque@students.cau.edu[1], justin.williams1@students.cau.edu[2],
sunzida15-9667@diu.edu.bd[3], hujaifaislam123@gmail.com[4],
hasmot15-9632@diu.edu.bd[5], kgupta@cau.edu[6], rgeorge@cau.edu[7]

***Abstract*—The combination of LLM agents with external tools enables models to solve complex tasks beyond their knowledge base. Human-designed tools are inflexible and restricted to solutions within the scope of pre-existing tools created by experts. To address this problem, we propose ATLASS, an advanced tool learning and selection system designed as a closed-loop framework. ATLASS facilitates the expansion of the LLM's capabilities by enabling it to generate tools on demand by incorporating external sources. In this framework, agents play a crucial role in orchestrating tool selection, execution, and refinement, ensuring adaptive problem-solving capabilities. The operation of ATLASS follows three phases: The first phase, Understanding Tool Requirements, involves the Agents determining whether tools are required and specifying their functionality; the second phase, Tool Retrieval/Generation, involves the Agents retrieving or generating tools based on their availability; and the third phase, Task Solving, involves combining all the component tools necessary to complete the initial task. The Tool Dataset stores the generated tools, ensuring reusability and minimizing inference cost. Current LLM-based tool generation systems have difficulty creating complex tools that need APIs or external packages. In ATLASS, we solve the problem by automatically setting up the environment, fetching relevant API documentation online, and using a Python interpreter to create a reliable, versatile tool that works in a wider range of situations. OpenAI GPT-4.0 is used as the underlying language model, with safety and ethical concerns addressed through human feedback prior to the execution of generated code. By overcoming the limitations of fixed toolsets and improving adaptability, ATLASS offers a practical solution that enables users to generate tools dynamically for complex problem-solving. The system demonstrates competitive performance when evaluated against baseline models and public datasets.**

***Index Terms*—LLM Agents, Automatic Tool Generation, API-Based Tools, Large Language Models**

I. Introduction

Large Language Models (LLMs) have capabilities in a wide range of tasks, from natural language processing to content generation and problem solving [18], [30], including support for user input from all domains and multimodal ability [35]. These attributes enable user demands to be met in different applications, including chat systems, intelligent virtual agents, automated content generation, and code synthesis [2], [37]. Although LLMs have achieved remarkable performance broadly, they still have inherent limitations, such as out-of-date information [5] and suffer from performance degradation [1]. Addressing these problems requires overcoming the limitations of predefined pipelines, which have restricted flexibility to calibrate incorrect actions. Additionally, it is challenging to adapt a general LLM-based agent to handle a wide range of specialized tasks. [26]. LLM agents are autonomous systems that combine reasoning, perception, and action to perform tasks. They bridge general-purpose LLMs and domain-specific needs by structuring complex problems into step-by-step reasoning. Researchers have introduced collaborative environments where multiple intelligent agent components, each with distinctive attributes and roles, work together to handle complex tasks more efficiently and effectively [28].

LLM agents often struggle with complex tasks that require external knowledge, computations, or real-world interactions beyond their pretrained capabilities. Tool learning, the ability of LLM agents to use external resources, is used to help LLM agents with various auxiliary resources, like search engines [19], [23] or calculators [8], [24] which empower them as tool-user agents and improve their ability to tackle complex tasks. Tool-based agents generally work by breaking down a task and planning a sequence of tools to complete it step by step. For each step, the agent executes the tools by passing arguments and continuously incorporating useful intermediates into the next action prediction. However, this approach has difficulty in adapting a single LLM-based agent to learn multiple specialized actions in solving a task. This limitation reflects a broader challenge: smaller models struggle with complex tasks, while specialized agents better handle tool selection and execution. To mitigate this problem, ConAgents coordinates three specialized agents for tool selection, tool execution, and action calibration separately [26].

In this work, we present the ATLASS framework to automate tool selection, generation, and execution. The framework uses the generated tools for a single inference and stores them in the tools database for further use. There are three key aspects to this contribution:

2642-6587/25/$31.00 ©2025 IEEE
DOI 10.1109/SOSE67019.2025.00012

- **Understanding Tool Requirements:** Analyzing user prompts, defining subtasks, defining the need for tools, and identifying the appropriate tools from the toolset.
- **Tool Retrieval or Generation:** Retrieving and registering tools from the toolset or generating tools and registering them with the agent.
- **Task Solving:** Solving user tasks by using tools.

This approach requires detailed task analysis, tool requirement understanding, and the integration of a comprehensive tool generation module. Simpler approaches, like the one used by Large Language Models as Tool Makers (LATM) [4], result in basic, task-specific tools that overlook the reuse of similar task requirements. These approaches also do not create generalized tools that permit this reusability. ATLASS addresses this limitation by analyzing user queries, breaking down the tasks, understanding tool requirements, and identifying that a single tool can efficiently handle similar queries. This approach creates reusable tools that may be applied to future tasks to reduce redundancy.

II. Literature Review

Recent advances in large language models (LLMs) have sparked growing interest in their ability to not only reason over language but also interact with external tools to solve complex tasks. Beyond traditional improvements in pre-training, fine-tuning, and evaluation, a significant body of research has emerged around enabling LLMs to generate, select, and use tools effectively. This includes work on program synthesis, API calling, code generation, and agent-based tool orchestration. As researchers continue to explore ways to enhance model performance, increasing attention is being placed on how LLMs can dynamically acquire capabilities, either by retrieving existing tools or constructing new ones, to extend their utility beyond static knowledge boundaries.

At the foundation of this evolution, Zhao et al. [38] provide a comprehensive review of LLM advancements, covering pre-training, tuning, and evaluation. Their work emphasizes key datasets such as Common Crawl, C4, and Wikipedia, and highlights tuning techniques like FLAN and RLHF. Notably, they demonstrate that instruction tuning significantly enhances the performance of LLaMA models, while Chinchilla scaling strategies improve parameter-to-data efficiency. Building on this, Marvin et al. [16] examine prompt engineering, another critical factor influencing LLM effectiveness. They explore few-shot learning, chain-of-thought prompting, and automatic instruction generation, ultimately showing that automated prompts outperform human-designed ones in 19 of 24 NLP tasks. These studies collectively highlight the synergy between scaling, tuning, and prompt design in driving LLM performance while also pointing out enduring challenges like hallucination, scalability, and bias.

Moving from foundational capabilities to alignment and customization, Wang et al. [30] propose the self-instruct framework based on GPT-3, demonstrating its superiority over publicly trained models and comparable performance to InstructGPT-001. In parallel, Liu et al. [13] investigate fine-tuning methods that allow for domain-specific adaptation, significantly boosting performance in specialized contexts. Addressing the efficiency of these methods, Hu et al. [11] introduce LoRA, a parameter-efficient approach that reduces the resource demands of model customization, thus democratizing access to LLMs. Expanding the conversation, Minaee et al. [18] explore LLM adaptability, showcasing their potential for real-world application across diverse industries. Together, these works emphasize the transition from general-purpose models to accessible, task-specific agents.

As LLM capabilities scale, research interest shifts toward models acting autonomously or in coordination. Duetting et al. [7] explore this space through the lens of combinatorial contract theory in multi-agent settings, presenting approximation algorithms for optimizing submodular rewards. Building on the concept of agent-based interaction, Guo et al. [9] survey LLM-based multi-agent systems, focusing on key aspects such as environment interaction, communication, and skill development. These works provide a theoretical and applied basis for understanding how LLMs can cooperate and specialize in increasingly complex systems.

At the reasoning level, Xu et al. [33] introduce LE-MCTS, which improves reasoning by optimizing process pathsyielding notable gains in mathematical problem-solving accuracy. Complementing this, Wu et al. [32] propose the AvaTaR framework to optimize agent tool use, achieving significant improvements on benchmarks like STARK and HotpotQA. These developments represent key milestones in enhancing the reasoning and decision-making capacities of LLM agents.

Further refining tool-use strategies, Shi et al. [27] propose a collaborative agent framework that relies on grounding, execution, and review agents communicating through adaptive protocols. Their SPAN technique distills action strategies from models like GPT-4 to open-source counterparts, boosting success rates on ToolBench and RestBench datasets. Similarly, Chen et al. [6] present AutoAgents, a system that creates task-specific agents through self- and collaborative refinement. By adding observers that track task execution, the framework achieves superior performance in open-ended question answering and writing tasks, outperforming even GPT-4.

Cai et al. [4] focus on tool creation through the LATM framework, which leverages LLMs to generate task-specific Python functions validated against user-defined datasets. While effective in structured scenarios, LATM lacks real-time external retrieval capabilities, a gap we aim to address in this work. By integrating SerpAPI [25], our system enhances adaptability through dynamic, web-based information extraction.

In the broader ecosystem of LLM orchestration, the MASS framework [39] enhances multi-agent collaboration through optimized prompts, communication protocols, and system instructions. Themis [12] aligns LLM outputs with human judgment via context-aware evaluations, while agentic

reasoning frameworks [31] integrate tools like web search and knowledge graphs to improve scientific reasoning. These frameworks underscore the growing sophistication and scalability of LLM-driven systems.

Despite this progress, challenges persist in real-world integration. Ni et al. [20] highlight issues like inconsistent API documentation and access control, proposing APILlama and a validation pipeline to streamline scientific tool generation. Similarly, Zhang et al. [36] present Auto-CoT, which automates chain-of-thought prompting using a clustering-based method. This reduces manual effort while achieving competitive results in symbolic and commonsense reasoning tasks, although reliability and computational overhead remain concerns.

Building upon these insights, our work introduces ATLASS, a framework designed to overcome the limitations LLM agents face when creating tools for complex tasks. ATLASS analyzes tasks, retrieves relevant tools, and dynamically constructs new ones when necessary. By incorporating real-time web search capabilities, the system adapts to evolving contexts, effectively bridging gaps in static toolchains. With robust task decomposition and adaptive tool generation, ATLASS advances the autonomy and domain generality of LLM-based agents.

III. ATLASS

ATLASS has a closed-loop architecture with multiple LLM agents, a multi-agent LLM, and a tool database. It can analyze user queries, break them down into subtasks, understand tool requirements, find existing tools and/or generate required tools, and solve user tasks using the generated or retrieved tools. Three key processes divide the ATLASS framework, each focusing on a distinct aspect of the overall workflow. Figure 1 shows the proposed ATLASS framework.

A. Tool Requirement Analysis

This stage of the framework processes the user's initial query and determines whether an external tool is necessary to complete the task. It includes agents that rely only on the LLM's internal knowledge base to generate an appropriate response, without utilizing any external tools.

Let the user query be denoted as q, and the encoded representation of the query using the LLM's internal encoder be $\vec{q} = \text{Enc}(q)$, where $\vec{q} \in \mathbb{R}^d$.

User Query: *Retrieve a list of the top 100 scientific books and organize them in ascending order.*

1) Task Analyzer: The Task Analyzer function $\mathcal{A}$ decomposes the query q into a set of subtasks $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$, where each s_i represents a smaller part of the task required to accomplish the overall task.

$$\mathcal{S} = \mathcal{A}(q)$$

Since the system follows a sequential agentic workflow, the output of each agent directly impacts the performance of downstream agents. Accurate decomposition of the users task into smaller subtasks is therefore critical. We define a subtask as the smallest unit of work that can be resolved through a single function call or atomic tool invocation. This coarseness promotes modularity, improves interpretability, and facilitates error isolation and recovery across the agent pipeline.

This strategy enables the subsequent agents to solve the problem step-by-step. The set $\mathcal{S}$ helps the Tool Master recognize which of the subtasks may require an external tool. Figure 2 demonstrates how the Task Analyzer works.

Task Analyzer:
1. A web crawler to fetch book data from a website.
2. Present the search result in ascending order.

2) Tool Master: An agent takes the breakdown of the sub-tasks from the Task Analyzer and determines whether external tools are required to solve the task or not. If external tools are required, this agent provides the 'name' and 'description' of all the required tools in a JSON format. If no tool is required, this agent simply responds appropriately.

The Tool Master agent $\mathcal{T}$ takes the subtasks $\mathcal{S}$ and decides whether any subtask $s_i \in \mathcal{S}$ requires an external tool. Formally, for each subtask s_i, a binary indicator function $f(s_i)$ is defined:

$$f(s_i) = \begin{cases} 1 & \text{if an external tool is required for } s_i \\ 0 & \text{otherwise} \end{cases}$$

The set of tools required is:

$$\mathcal{T}_{\text{required}} = \{t_i \mid f(s_i) = 1,\ t_i = \text{Name + Description}(s_i)\}$$

If $\mathcal{T}_{\text{required}} = \emptyset$, the framework proceeds directly to the solution generation using the LLM alone. Otherwise, the required tools are sent to the Tool Retrieval stage. Figure 3 shows the pipeline for the Tool Master agent.

The response of Tool Master Agent

```
[
   {
    "name": "Web_Scraper",
    "description": "A tool to extract
    specific information from web pages
    by crawling and parsing their content."
   }
]
```

B. Tool Retrieval/Generation

After understanding the tool requirements, the system determines whether a tool needs to be generated from scratch

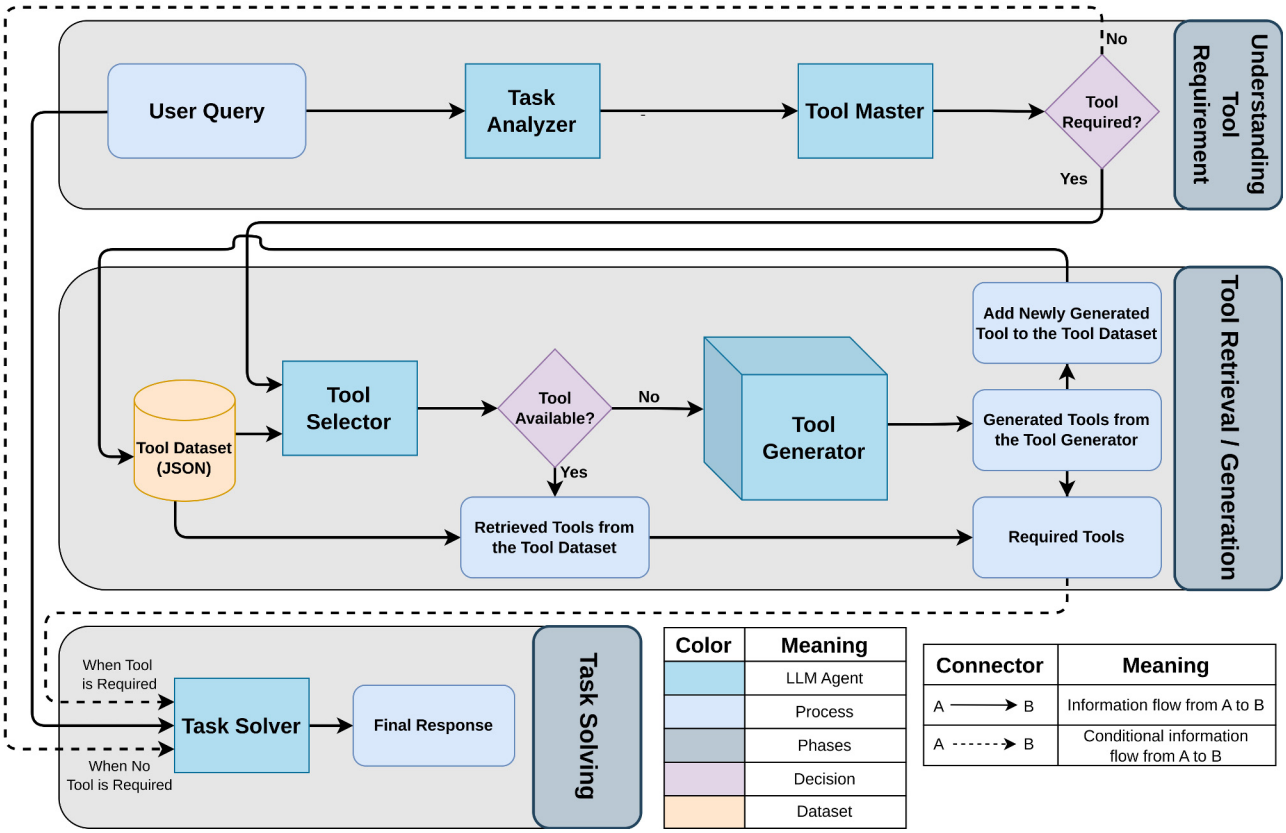

Fig. 1: Overview of ATLASS workflow, with Tool Requirements Analysis, Tool Retrieval/Generation, and Task Execution.

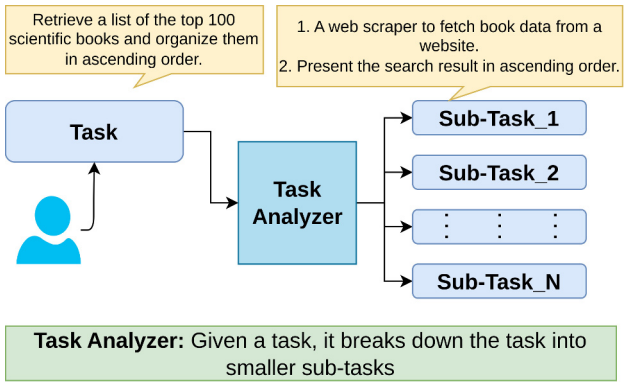

Fig. 2: Task Analyzer Pipeline

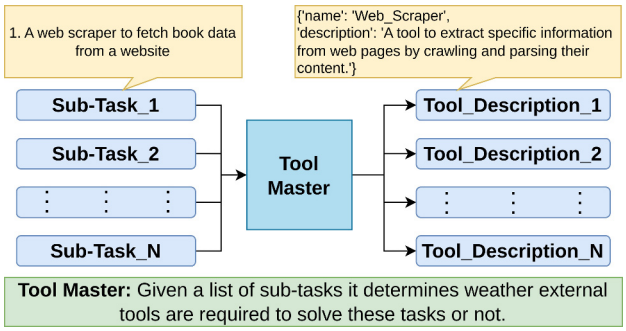

Fig. 3: Tool Master Pipeline

using the Tool Generator $\mathcal{G}$ or retrieved from the Tool Dataset $\mathcal{D}$. This stage utilizes a multi-agent setup along with the dataset $\mathcal{D}$ to execute the appropriate course of action.

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be the set of required tools extracted by the Tool Master. For each tool $t_i \in \mathcal{T}$, we define:

$$t_i = \{\text{name}_i, \text{desc}_i, \text{is\_available}_i\}$$

If $\text{is\_available}_i = \texttt{True}$, then t_i is retrieved from $\mathcal{D}$; otherwise, it is generated via $\mathcal{G}$. The Tool Generator $\mathcal{G}$ is composed of several specialized agents:

- `Code Writer` $\mathcal{W}$: Generates executable Python code.
- `Code Executor` $\mathcal{E}$: Executes and verifies code generated by $\mathcal{W}$.
- `Web Crawler` $\mathcal{S}$: Retrieves current API documentation if needed.

These agents (code writer, code executor, and web crawler) ensure the following:

- The tool is generated only when no similar tool is available in the dataset, and

- Ensure that the generated tool is functioning correctly.

1) Tool Database: ATLASS maintains a tool database $\mathcal{D}$, which is a JSON-based repository containing metadata of each tool in the format

$$t_i = \{\text{name}_i, \text{desc}_i, \text{func\_name}_i\}$$

The actual implementations are stored in corresponding Python scripts and func_name_i are used as a lookup key.

2) Tool Selector: This agent ascertains which necessary tools are already present in the system and which ones require creation using the Tool Dataset and Required Tools to tell us which tools need to be retrieved and which tools need to be generated. Given the required tools $\mathcal{T}$ and the dataset $\mathcal{D}$, the Tool Selector $\mathcal{S}_T$ performs the following classification:

$$\forall t_i \in \mathcal{T}, \quad \text{is\_available}_i = \begin{cases} \texttt{True}, & \text{if } t_i \in \mathcal{D} \\ \texttt{False}, & \text{otherwise} \end{cases}$$

Figure 4 shows how the Tool Selector process works.

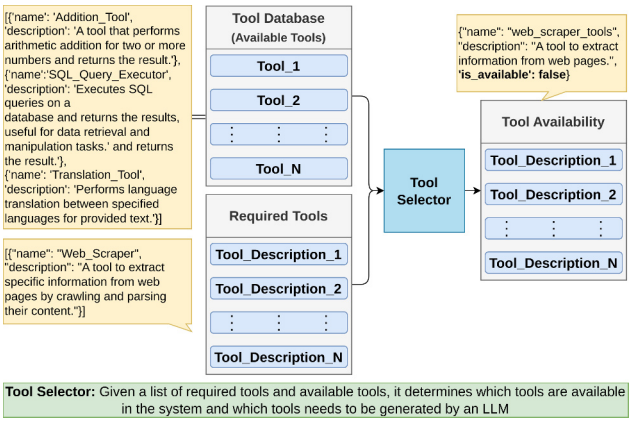

Fig. 4: Tool Selector Pipeline

The response of Tool Selector

```
[
    {
     "name": "web_scraper_tools",
     "description": "A tool to extract
     information from web pages.",
     "is_available": false
    }
]
```

3) Tool Generator: Non-API-Based: For the tools that are not available in the system and don't require any kind of API, we use the Tool Generator agent to generate Python code for those tools. The Tool Generator consists of two agents: a code writer and a code executor 5. For any tool t_i such that $\text{is\_available}_i = \texttt{False}$ and t_i does not require external APIs, the Tool Generator $\mathcal{G}_{\text{nonAPI}}$ proceeds as follows:

1) Pass $\{\text{name}_i, \text{desc}_i\}$ to $\mathcal{W}$ to generate:

$$c_i^{\text{install}} \quad \text{(dependency installation code)}$$

2) Execute c_i^{install} via $\mathcal{E}$ and return result r_i.
3) On success, generate function code c_i^{func} with annotations.
4) Execute c_i^{func} via $\mathcal{E}$.
5) If errors occur, send them back to $\mathcal{W}$; repeat until a working version is found.
6) Once functional, add t_i and c_i^{func} to $\mathcal{D}$.

The process operates in a loop, continuously executing until it locates a workable code base. Once it finds a working code base, it adds its information and the tool's information back to the Tool Dataset.

4) Tool Generator : API-Based: For the tools that do require APIs, we at first pass the tool's information to the code writer. When the code writer receives an API-based tool, it simply outputs "API KEY REQUIRED: Name of the API". The reason we don't try to generate API-based tools directly is: 1. An API key is required to run any API-based tool, which the agent currently doesn't have, and 2. API parameters change over time, and it is not necessary that the model knowledge base has the most current information. For these reasons, we use a web crawler agent, which uses a web searching tool (Serp API [25]) to get the current documentation's information on the API usages. The system also asks the user to provide the API key if required. The initial tool information (name and description), the latest documentation content from the internet, and the user-provided API key are combined to create a new prompt, which is then passed to the code generator to generate the tool.

Figure 5 shows how the Tool Generator generates the tool. Once the required tools are generated or retrieved, the system combines them and passes them to the final stage of the framework.

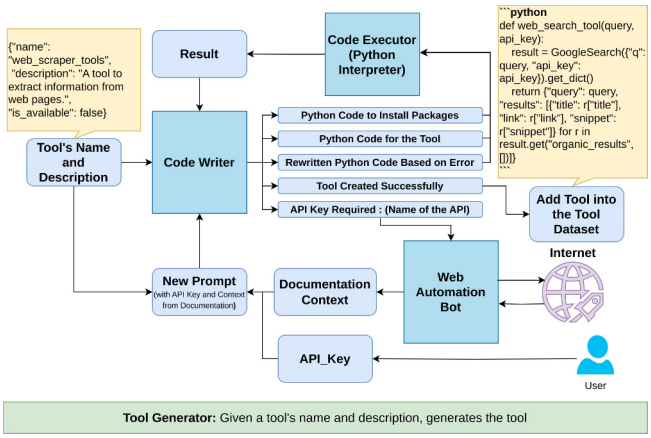

Fig. 5: Tool Generator Pipeline

The response of Tool Generator Agent

```
def web_search_tool(query, api_key):
    result = GoogleSearch("q": query,
      "api_key": api_key).get_dict()

    return "query": query, "results":
     ["title": r["title"], "link":
      r["link"], "snippet": r["snippet"]
      for r in result.get("organic_results",
      [])]
```

C. Task Solving

This final stage of the framework focuses on resolving the user's initial query q. The output from the Tool Master agent $\mathcal{T}_M$ includes both the structured representation $\mathcal{S}$ and the tool requirements $\mathcal{T}_{\text{required}}$.

Case 1: No Tools Required. If $\mathcal{T}_{\text{required}} = \emptyset$, the system bypasses the Tool Retrieval/Generation module, and the task is solved using the internal capabilities of the Task Solver agent $\mathcal{R}$. In this case, the final response a is computed as:

$$a = \mathcal{R}(q, \mathcal{S}), \quad \text{where } \mathcal{T}_{\text{required}} = \emptyset$$

Case 2: Tools Required. If $\mathcal{T}_{\text{required}} \neq \emptyset$, the Task Solver receives the set of tools, including:

$$\mathcal{T}_{\text{active}} = \mathcal{T}_{\text{retrieve}} \cup \mathcal{T}_{\text{generate}}$$

and uses them in conjunction with the structured information $\mathcal{S}$ to generate the final response:

$$a = \mathcal{R}(q, \mathcal{S}, \mathcal{T}_{\text{active}}), \quad \text{where } \mathcal{T}_{\text{required}} \neq \emptyset$$

Thus, the Task Solver $\mathcal{R}$ dynamically determines whether to depend solely on its internal knowledge base or to invoke external tools to generate a response.

1) Task Solver: The Task Solver agent $\mathcal{R}$ is responsible for producing the final answer a to the user's query q, either autonomously or by orchestrating tool usage. The agent employs a reasoning mechanism to:

- Use internal knowledge when $\mathcal{T}_{\text{required}} = \emptyset$.
- Invoke external tools in $\mathcal{T}_{\text{active}}$ as needed when $\mathcal{T}_{\text{required}} \neq \emptyset$.

Figure 6 shows the workflow of the Task Solver agent during the problem-solving process.

IV. Dataset Creation and Collection

While ATLASS functions without requiring specific datasets for task execution, we conducted evaluations using two distinct datasets containing 954 question-answer pairs. These datasets served as benchmarks for assessing system performance.

ATLASS Dataset: This is a newly created evaluation dataset curated and annotated by our research team. It comprises 492 question-answer pairs across diverse domains, including mathematical reasoning, data analysis, analytical thinking, problem-solving, information extraction, and data visualization. Each entry was manually constructed and validated by human experts to ensure quality, diversity, and domain coverage.

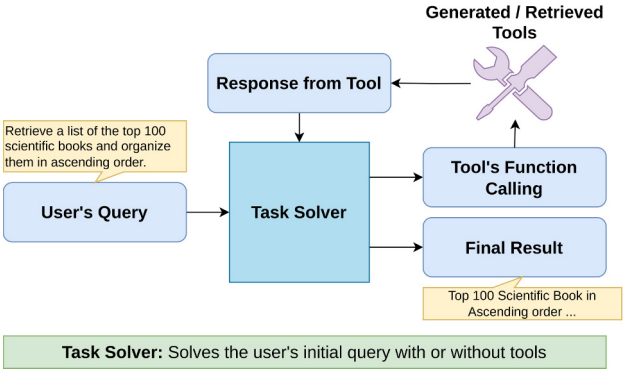

Fig. 6: Task Solver Pipeline

For each data point, we include the question, the reference answer, the tool name, a high-level tool description, and its corresponding function implementation. Tools are designed to be domain-agnostic and composable, favoring reusable functionality (e.g., plotting, filtering, summarization, or arithmetic operations) rather than task-specific heuristics. To ensure generalizability, tool definitions are reviewed across multiple examples and benchmarked for cross-task applicability.

The dataset is also designed to support longitudinal evaluation: tools undergo periodic refactoring and curation to eliminate redundancy, enhance abstraction, and maintain relevance with evolving task distributions. During evaluation, only the question is presented to the model, and the generated output is compared to the reference answer. This setup allows rigorous assessment of dynamic tool generation and selection capabilities, benchmarking ATLASS and baseline models in realistic agentic decision-making scenarios.

Example of ATLASS data

```
[
   {
    "question": "Find the Roman numeral
    for 1987",
    "answer": "MCMLXXXVII"
   },
   {
    "question": "Find the quarter of
    2023-09-18",
    "answer": "3"
   },
   ......
]
```

CRAFT (Math + Table): This dataset contains 462 question-answer pairs sourced from the CRAFT repository [34], covering two specific domains: mathematics and tabular data reasoning. The mathematics subset, derived

from the algebra portion of the MATH dataset [10], features challenging, competition-level algebra problems along with their solutions. The tabular subset is based on the TabMWP dataset [15], which includes natural language questions paired with tables. This subset is designed to assess the ability of language models to interpret and reason over structured data.

To demonstrate the ability to generate API-based tools, we have created a collection of 10 questions that require extensive API calls, such as `SERP_API`, `OpenWeatherMap_API`, `AlphaVantage_API`, `Tavily_AI_API`, `News_API`, `YFinance_API`, and `CoinGecko_API` to get the required data to answer user queries. Also, some of them require a personalized API key to access the data.

V. Result Analysis

The performance of the ATLASS framework is influenced by the efficiency of multiple interconnected modules, including the Task Analyzer, Tool Master, Tool Selector, and Tool Generator. As a sequential conversational system, the output of each module directly impacts the subsequent modules, creating a flow throughout the system. We assess multiple criteria to evaluate the overall system performance, taking into account the individual contributions and interdependencies of each module. We evaluate our approach on different domains, including mathematical operations, data analysis, data visualization, forecasting, NLP tasks, and API-based information retrieval.

A. Tool Selection Performance

Given a user prompt, after task and tool requirement analysis, the Tool Selector module determines whether the necessary tool exists in the current tool database. For example, consider the user prompt: *Generate a bar chart with the last five days stock price of Apple Inc.* The Tool Selector identifies that this task requires two distinct tools: *Stock Price Checker* and *Data Visualizer*. If the database includes a tool named *Bar Chart Generator*, the Tool Selector demonstrates the ability to map *Data Visualizer* to this functionally equivalent tool. This mapping ability enhances tool reusability and generalization while minimizing redundant tool generation.

To evaluate this mechanism, we conducted a comprehensive analysis over the entire dataset of 954 question-answer pairs. This full-scale evaluation provides stronger empirical grounding for assessing the accuracy and generalizability of the tool selection process. Table I presents a detailed breakdown of Tool Selector performance across all examples, measuring exact match accuracy, semantic equivalence, and failure cases.

This illustrates that ATLASSs Tool Selector generalizes well across semantically similar prompts, reducing redundancy and improving reusability.

B. Tool Creation Performance Analysis

1) Performance of Non-API-Based Tool Generation: We focus on two crucial criteria to assess the effectiveness of various tool generation pipelines: **Correctness** and **Executability**. These evaluate the semantic correctness and the practical usability of the tools generated. Our comparison is based on three baseline tool generation models: CRAFT [34], CREATOR [22], and LATM [3].

Tools Name:	*Word Frequency Counter*
Description:	*A tool that can split a sentence into individual words count the frequency of each word, sort them according to their frequency, and select the most common words.*
Origin Prompt:	*Find 10 most common words in the sentence.*
Alternative Prompts:	***1.*** *Rank the following keywords in order of relevance in this document.* ***2.*** *Can you find the unique words in this sentence and tell me how rare they are?* ***3.*** *Find out how similar these two texts are based on their most commonly used words.*

TABLE I: The tool *Word Frequency Counter* generated by the prompt *"Find 10 most common words in the sentence."* is reusable by the list of Alternative Prompts

a) Correctness (G-Eval): Correctness is evaluated using the G-Eval [14] metric, which employs an LLM-as-a-judge approach, enhanced by Chain-of-Thought (CoT) prompting. G-Eval constructs evaluation prompts based on predefined criteria and test case parameters (*LLMTestCase* as mentioned in the DeepEval evaluation framework). The LLM then assigns a score between 1 and 5, reflecting the alignment of the tool output with the expected result. These individual scores are aggregated and normalized using a weighted summation of output token probabilities to ensure consistent evaluation across different tool generations. We consider a tool output to be correct if its normalized correctness probability score exceeds a threshold of 0.5.

b) Executability: In addition to correctness checking, we verify the *executability* of the generated tools. Executability is defined as the percentage of tools that run without errors and produce correct outputs. Formally, let T be the total number of generated tools, and let T_{exec} denote the number of tools that execute successfully and produce correct outputs. Then, the executability score E is given by:

$$E = \frac{T_{\text{exec}}}{T} \times 100\%$$

A high executability score ($E \approx 100\%$) indicates that the tool generation process is stable and viable, whereas lower scores ($E \ll 100\%$) suggest challenges in ensuring stable functionality.

Table II encapsulates non-API-based tool generation correctness and executability scores. As shown, ATLASS scores the highest on the Correctness metric with a strong score of 83.87%. In terms of executability, ATLASS nearly matches the top score with 99.56%. CREATOR, on the other hand, achieves the highest executability score of 99.78%. This indicates that ATLASS consistently generates accurate and operational tools. CREATOR also demonstrates solid

performance, with a relatively high executability score of 97.85%, but it falls behind ATLASS in terms of correctness (61.07%). By contrast, ATLASS excels across both metrics, demonstrating impressive correctness and executability scores of 83.87% and 99.56%, respectively. These results highlight ATLASSs ability to consistently generate both accurate and highly functional tools, showcasing the strength and reliability of its generation pipeline.

Metric	Dataset	ATLASS	CRAFT	CREATOR	LATM
Correctness	ATLASS dataset	**83.87%**	80.43%	61.07%	52.04%
	CRAFT (Math + Table)	**89.5%**	89.2%	47.5%	66.7%
Executability	ATLASS dataset	**99.78%**	83.44%	97.85%	71.40%
	CRAFT (Math + Table)	99.56%	89.85%	**99.78%**	83.89%

TABLE II: Evaluation of Correctness and Executability across Tool Generation Models

These results affirm that ATLASS achieves a balanced and superior performance in terms of both correctness and executability compared to baseline models.

2) Performance of API-Based Tool Generation: Most of the existing pipelines, including CRAFT, CREATOR, and LATM, fall short in generating API-based tools, which is a significant limitation, given that the majority of real-world LLM tools are external API-based. The primary challenges are the need for secure API key access and the constantly evolving nature of API implementations. These pipelines either ignore these challenges altogether or generate nonfunctional, legacy code. ATLASS overcomes these limitations through two key innovations:

- **Human-in-the-Loop for API Key Access:** ATLASS is built on the LangGraph and leverages its built-in human-in-the-loop. Whenever an API key is necessary, the execution will be paused and the user will be prompted for secure key input. The generated tools can then run correctly without having the sensitive credentials hardcoded.
- **Web Retrieval for Up-to-Date Implementations:** ATLASS integrates a web retrieval system (SerpAPI) to query Internet sources regarding the current Python versions of requested APIs. Samples retrieved are made available as background to the code generation framework so that the system can create tools reflecting the latest usage and syntax of modern APIs.

To evaluate this capability, we tested ATLASS using 10 queries involving various free APIs. In the initial experiments, ATLASS successfully generated functional tools for APIs such as `SERP_API`, `OpenWeatherMap_API`, `AlphaVantage_API`, `Tavily_AI_API`, `News_API`, `YFinance_API`, and `CoinGecko_API`. These results show promising potential, though further research is needed to handle more complex or less-documented APIs. Nevertheless, ATLASS demonstrates a marked advancement in API-based tool generation over prior methods.

3) Feature-based Comparison: Table III provides a high-level comparison of functional features supported by the evaluated tool generation pipelines. Among them, **ATLASS** demonstrates the most comprehensive feature set, supporting advanced capabilities such as package installation, API-based tool creation, web data retrieval, and interactive user-driven tool generation. These features collectively distinguish ATLASS in terms of flexibility, automation depth, and practical usability.

In contrast, other pipelines like CRAFT, CREATOR, and LATM offer partial support across various dimensions. While all pipelines can handle code debugging and task breakdown, only ATLASS supports all advanced features, such as generating multiple tools concurrently, retrieving/storing tools, and integrating external APIs or online content. This broader functionality positions ATLASS as a more versatile and production-ready pipeline.

Features	ATLASS	CRAFT	CREATOR	LATM
Install Packages	✓	✗	✗	✗
Code Debugging	✓	✓	✓	✓
API based Tools	✓	✗	✓	✗
Web Retrieval	✓	✗	✗	✗
Generation from Query	✓	✗	✓	✗
Task Breakdown	✓	✓	✓	✓
Multiple Tools at once	✓	✓	✓	✗
Tool Retrieval	✓	✓	✗	✓
Tool Storage	✓	✓	✗	✓

TABLE III: Evaluation of Features across Tool Generation Model

C. Efficiency of Inference

Because LLM-based agents have a high inference cost, we look into how efficiently ATLASS can solve tasks by generating required tools. At the time of the experiment, we had used the *"gpt-4-0613"* model for all the tasks. On average, we have consumed 2895 tokens at an average cost of 0.1008 USD per prompt when the tool is not available; on the other hand, the consumed tokens are 1920, and the cost is 0.0624 USD per prompt when the tool is available. Table IV shows the token consumption and cost for each prompt. The cost calculation is based on the OpenAI pricing page [21].

Task	With no tools in DB		With tools in DB	
	Token	Cost (USD)	Token	Cost (USD)
Sorting	3161	0.1127	2337	0.0781
Reversing	3195	0.1078	1678	0.0536
Cleaning	2627	0.0918	1901	0.0620
Extraction	2620	0.0891	1930	0.0617
Graph Generation	3881	0.1363	2118	0.0703
Stock Exchange	2495	0.0875	1740	0.0558
Sentiment	2588	0.0909	1822	0.0586
SerpAPI	2596	0.0909	1838	0.0591

TABLE IV: Cost Analysis for end-to-end framework

As an end-to-end framework, every query undergoes task analysis, retrieval or generation, and task execution, ensuring

the system can handle any given task. The overall cost can be minimized by utilizing smaller model variations, such as GPT-3.5.

VI. Conclusion

The experimental evaluations validate the effectiveness and robustness of the ATLASS framework across key dimensions of automated tool generation and selection. ATLASS consistently outperforms strong baselines such as CRAFT, CREATOR, and LATM, particularly in terms of correctness, executability, and adaptability in both non-API and API-based tool settings. Notably, ATLASS fills a critical gap in prior work by enabling secure, real-time generation of API-integrated toolsan essential capability for dynamic, real-world applications that has been largely underexplored in previous systems. Beyond accuracy, ATLASS incorporates a rich set of features designed for practical deployment, including dynamic package installation, web data retrieval, multi-tool chaining, and human-in-the-loop API access. These capabilities collectively position ATLASS as a production-ready framework, capable of scaling across diverse domains and evolving task distributions. Overall, ATLASS represents a significant step toward the development of autonomous, trustworthy, and extensible agentic systems for tool generation. Future research will focus on enhancing its cross-model generalizability, minimizing human intervention through risk-aware automation, and strengthening interoperability across heterogeneous LLM environments.

VII. Limitations and Future Work

While ATLASS demonstrates superior tool-generation capabilities compared to previous works, certain limitations remain. The current implementation relies on a single language model (OpenAI GPT-4.0) for all agentic tasks, which may restrict the systems generalizability and robustness across different deployment environments. Moreover, challenges persist in reliably generating highly abstract, API-integrated, or multi-step tools, which can affect the applicability of ATLASS in more complex domains.

We recognize that transitioning ATLASS to support other LLMs (e.g., ChatGPT-3.5 or open-source models like Mistral or LLaMA) introduces a number of non-trivial challenges. These include: (i) variation in tool generation accuracy due to differences in instruction-following behavior, (ii) inconsistent handling of structured tool specifications or API schemas, and (iii) interoperability issues arising from discrepancies in tool-calling formats and execution contexts. Addressing these will require the design of abstraction layers or adapter modules that enable consistent tool invocation across heterogeneous models.

To address these limitations and guide future improvements, we propose the following directions for research:

- Enhance the Tool Generators capacity to handle more complex, multi-step, and API-based tools.
- Strengthen framework security by addressing risks associated with tool execution, including secure management of user-provided API keys.
- Conduct a comprehensive, quantitative evaluation of ATLASSs tool generation and selection performance compared to other agentic tool-use pipelines.
- Extend ATLASS to support multiple LLMs, including ChatGPT-3.5 and open-source alternatives, by modularizing components such as Task Analyzer and Tool Selector. Evaluate the trade-offs in accuracy, reasoning capability, and tool interoperability.

VIII. Safety and Ethics

Automating tool generation introduces important safety, ethical, and security considerations, including the risk of executing harmful code and the exposure of sensitive API credentials. To mitigate these risks, the current implementation incorporates the following safeguards:

1) **Human Feedback** Generated code may pose security threats if executed without inspection [29]. Therefore, the framework enforces a human-in-the-loop mechanism, requiring human verification before any generated Python code is executed. This allows users to assess intent, detect malicious logic, and ensure safe behavior.
   However, we acknowledge that this approach, while effective for safety, may not scale efficiently in high-volume or real-time environments. To address this, future iterations of the framework will incorporate risk-aware automation strategies. These include automated static code analysis, execution sandboxing, and trust scoring of code segments, enabling selective human intervention only when risk thresholds are exceeded.
2) **Security of API Keys** API keys represent a critical security vulnerability when passed directly to language models [17]. In our current design, API keys are never exposed to the model directly. Instead, we use regular expressions to dynamically insert the keys into execution contexts after code generation, keeping them outside the model's prompt space.
   To further improve this, future versions will adopt secret management best practices (e.g., environment-based vault injection) and integrate policy-based access control mechanisms to restrict tool-level access.

References

[1] Toufique Ahmed, Christian Bird, Premkumar Devanbu, and Saikat Chakraborty. Studying llm performance on closed- and open-source data, 2024.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

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

[4] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2024.

[5] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models, March 2024.

[6] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*, 2023.

[7] Paul Duetting, Tomer Ezra, Michal Feldman, and Thomas Kesselheim. Multi-agent combinatorial contracts. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1857–1891. SIAM, 2025.

[8] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models, 2023.

[9] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[10] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

[11] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[12] Renjun Hu, Yi Cheng, Libin Meng, Jiaxin Xia, Yi Zong, Xing Shi, and Wei Lin. Training an llm-as-a-judge model: Pipeline, insights, and practical lessons, 02 2025.

[13] Alisa Liu, Xiaochuang Han, Yizhong Wang, Yulia Tsvetkov, Yejin Choi, and Noah A. Smith. Tuning language models by proxy. In *First Conference on Language Modeling*, 2024.

[14] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*, 2023.

[15] Pan Lu, Liang Qiu, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, Tanmay Rajpurohit, Peter Clark, and Ashwin Kalyan. Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning. In *ICLR*, 2023.

[16] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*, pages 387–402. Springer, 2023.

[17] Trend Micro. Security vulnerabilities of chatgpt-generated code, 2023. Accessed: 2025-02-07.

[18] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2024.

[19] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[20] Xinyi Ni, Qiuyang Wang, Yukun Zhang, and Pengyu Hong. Toolfactory: Automating tool generation by leveraging llm to understand rest api documentations. *arXiv preprint arXiv:2501.16945*, 2025.

[21] OpenAI. Pricing - OpenAI API. https://platform.openai.com/docs/pricing. [Accessed 28-01-2025].

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

[23] Yujia Qin, Zihan Cai, Dian Jin, Lan Yan, Shihao Liang, Kunlun Zhu, Yankai Lin, Xu Han, Ning Ding, Huadong Wang, Ruobing Xie, Fanchao Qi, Zhiyuan Liu, Maosong Sun, and Jie Zhou. WebCPM: Interactive web search for Chinese long-form question answering. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8968–8988, Toronto, Canada, July 2023. Association for Computational Linguistics.

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dess, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

[25] SerpAPI. SerpApi: Google Search API. https://serpapi.com/. [Accessed 03-02-2025].

[26] Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. Learning to use tools via cooperative and interactive agents, 2024.

[27] Zhengliang Shi, Shen Gao, Xiuyi Chen, Yue Feng, Lingyong Yan, Haibo Shi, Dawei Yin, Pengjie Ren, Suzan Verberne, and Zhaochun Ren. Learning to use tools via cooperative and interactive agents. *arXiv preprint arXiv:2403.03031*, 2024.

[28] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents, 2023.

[29] Natalie Tischler. The risks of automated code generation and the necessity of ai-powered remediation, 2024. Accessed: 2025-02-07.

[30] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023. Association for Computational Linguistics.

[31] Junde Wu, Jiayuan Zhu, and Yuyuan Liu. Agentic reasoning: Reasoning llms with tools for the deep research, 02 2025.

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

[33] Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, Chenyang Shao, Yuwei Yan, Qinglong Yang, Yiwen Song, Sijian Ren, Xinyuan Hu, Yu Li, Jie Feng, Chen Gao, and Yong Li. Towards large reasoning models: A survey on scaling llm reasoning capabilities, 2025.

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

[35] Duzhen Zhang, Yahan Yu, Jiahua Dong, Chenxing Li, Dan Su, Chenhui Chu, and Dong Yu. MM-LLMs: Recent advances in MultiModal large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12401–12430, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[36] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*, 2022.

[37] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2024.

[38] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[39] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vuli, Anna Korhonen, and Sercan Ark. Multi-agent design: Optimizing agents with better prompts and topologies, 02 2025.