DevRev's AI Agent 007: Tooling up for Success

Team 15

Abstract

We aim to develop a Large Language Model (LLM) powered chatbot, augmented by a set of tools, each accompanied by its detailed description. The chatbot intelligently recommends a subset of these tools, specifying the arguments for their utilization, and providing guidance on how to combine them effectively to address user queries. Additionally, our solution incorporates features facilitating the seamless addition, deletion, and modification of tools and their arguments within our toolset.

In the following sections, we present our work which is backed by extensive literature review and experimentation.

1. Introduction

Our journey has led us to an in-depth exploration of advanced **Prompting** techniques, specifically concentrating on:

- **Few-Shot CoT (Chain of Thought)** Wei et al. (2023) involves prompting the LLM with a series of logical steps, encouraging it to "think aloud" and thereby produce more structured and understandable responses.
- **ReACT** Yao et al. (2023) is a novel approach that focuses on reactive and adaptive prompting to enhance the relevance and accuracy of the LLM's responses.
- **Decomposed Prompting**Khot et al. (2023) breaks down complex queries into simpler components. Each component is then solved one-by-one making it easier for the LLM to process and respond accurately. Afterward, the solutions for the components are merged to provide a final solution to the query.

Fine-tuning large language models is a way to exploit the reasoning capabilities of these models for our downstream tasks. Parameter Efficient Fine Tuning (**PEFT**) gives us a way of achieving this without adjusting the entire parameter set of the model. We explored open-source models like CodeGen by Nijkamp et al. (2023), Code Llama by Rozière et al. (2023), and NexusRaven on their abilities to adapt to tasks specific to our requirements.

One of the main aspects of fine-tuning open-source models for this task was **dataset generation**. We stuck to the synthetic dataset, based on a carefully curated manual seed set, to fine-tune our models, which were generated while keeping in mind the diversity of the queries concerning complexity and tools being used in the corresponding queries.

Additionally, we delved into advanced **Retrieval** techniques for our pipeline. This exploration aimed to enhance the chatbot's functionality by integrating new tools and demonstrating their usage examples in various conversational contexts. We maintain a database of examples as part of our Dynamic Retrieval In Context Learning methodology. We also maintain a semantic cache of past frequently asked queries to enable us to efficiently deal with them to reduce inference costs and latency by retrieving answers to semantically similar queries without running the entire pipeline.

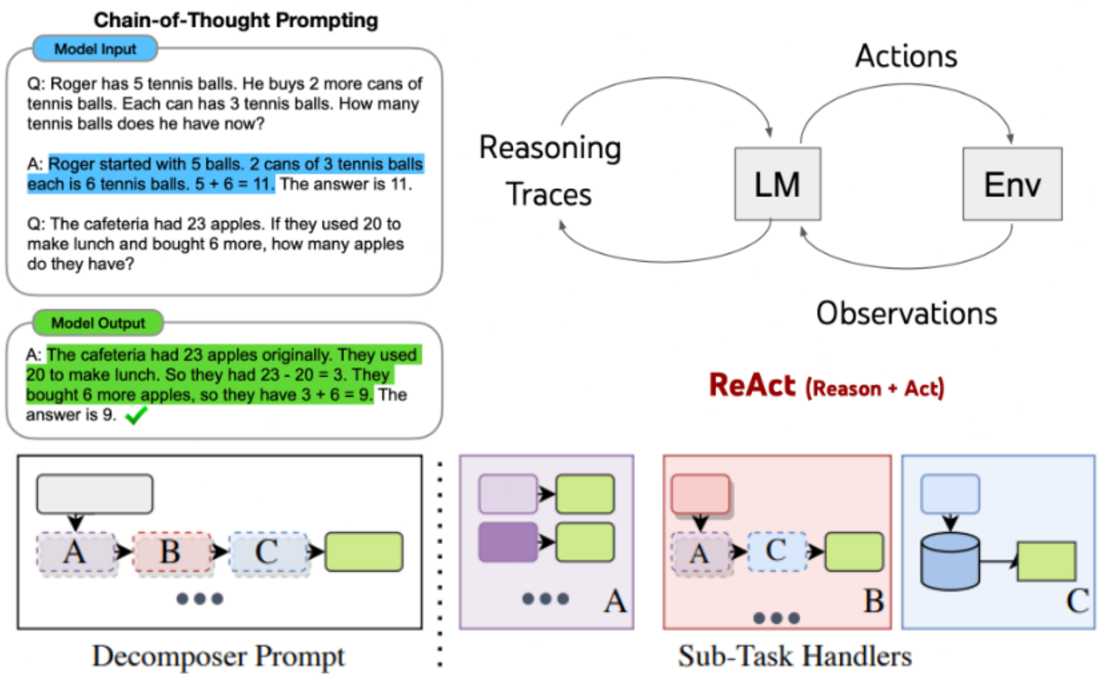
A key aspect of our work on the **inference pipeline** has been into various pre and post-processing techniques to enhance the ability of the LLM to effectively answer the user queries, reduce hallucinations, and ensure that we get the desired output structure.

We also run several **experiments** on our pipeline to gauge the inference costs and latency as a function of the query complexity. We ran these experiments on each part of our pipeline which enabled us to optimize each part to reduce inference costs and latency while maintaining performance.

Our post-mid-evaluation efforts have been centered on refining and experimenting with advanced prompting techniques, parameter-efficient fine-tuning methods, and several pre and post-processing techniques as part of our inference pipeline, aiming to significantly enhance the chatbot's interactive and context-aware capabilities.

2. Literature Review

In our comprehensive mid-evaluation report, we aimed to narrow down our focus to a specific section of the extensive literature on this domain. Here, we elaborate on the part of the domain that aligns with our interests and pertains to the particular task at hand.



Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Figure 1. Diagramatic overview of the **Advanced Prompting Techniques**, Few-Shot CoT, ReACT, and Decomposed prompting, which have been the focus of our attention as the main prompting techniques that we have experimented with for our final pipeline

2.1. API Interaction

Majority of the existing literature in this field revolves around actually utilizing the APIs or external tools Qin et al. (2023), Patil et al. (2023), Schick et al. (2023), Yang et al. (2023) and integrating the results from these interactions as observations into the context.

However, our unique problem statement necessitates working solely with API descriptions, excluding the actual execution of API calls. Notable works that share a similar approach include the Reverse Chain method proposed by Zhang et al. (2023b), the task planning step in HuggingGPT introduced by Shen et al. (2023) and REWOO by Xu et al. (2023a).

2.2. Singular vs Multiple Tool Integration

A lot of the early literature in the tool learning domain was based on mostly single tool usage like Toolformer by (Schick et al., 2023) and Gorilla by (Patil et al., 2023). However, our task involves queries that might require multiple API calls. The solution may require multiple iterations with a tool, and there can be multiple tools involved, each with its own set of parameters.

2.3. Open Source vs Closed Source LLMs

The research community suggests that there is a significant gap between open-source LLMs and closed-source LLMs when it comes to tool-use/ tool planning capabilities Qin et al. (2023), Xu et al. (2023c).

While closed-source LLMs may initially perform adequately "off the shelf" for this task, they lack the inherent advantages offered by open-source alternatives, such as cost-effectiveness, flexibility, security, and reduced dependency on external providers and therefore, it is crucial to engage in extensive experimentation with open-source LLMs as well.

We looked into several approaches that can help us align open-source LLMs to this task, possibly surpassing the capabilities of closed-source LLMs while giving us full control over the model.

We explored open-source models like CodeGen by Nijkamp et al. (2023), Code Llama by Rozière et al. (2023), and NexusRaven on their abilities to adapt to tasks specific to our requirements. We also explored parameter-efficient fine-tuning of these models.

2.4. Number of Trainable Parameters

Based on the count of trainable parameters, our methodologies can be broadly categorized into three groups: Hard

Prompting, which involves no trainable parameters; Parameter Efficient Fine Tuning (PEFT), where only a subset of parameters is tuned compared to the entire set; and Full Fine Tuning. A lot of our efforts went into extracting as much performance as possible from Hard Prompting, the most resource-friendly category among the three.

Concurrently, recognizing the constraints on resources, we actively engaged in two key initiatives. Firstly, we constructed an instruct dataset Wang et al. (2022), Zhuang et al. (2023) Qin et al. (2023) (Details discussed in the Dataset Generation section). Secondly, we explored various PEFT techniques Li and Liang (2021), Jain et al. (2023), Dettmers et al. (2023), Hao et al. (2023), Zhang et al. (2023a) to enhance the alignment of our model, particularly smaller open source Language Model Models (LLMs), with the designated task. The optimization of smaller LLMs for this task not only contributes to resource conservation during deployment but also facilitates quicker inference.

2.5. Domain Specific Question Answering

In cases where the toolset is extensive and the documentation exceeds the capacity of in-context learning methods, researchers have experimented with diverse retrieval techniques Qin et al. (2023), Yuan et al. (2023), Patil et al. (2023). These methods involve retrieving pertinent tools for user queries from a tool database and integrating them into the chatbot's context using Retrieval Augmented Generation (RAG) pipelines to enhance their chatbots. However, following our discussion with the company, it appears that, at present, there is no immediate necessity for the incorporation of retrieval techniques for the toolset. Most of the needed domain knowledge can be added in context.

However, we do incorporate retrieval in the form of Retrieval In Context learning to make sure that the example demonstrations are more representative of the user queries following similar work as seen in URIAL Lin et al. (2023). We also look into retrieval in the form of semantic caching of previously asked queries.

3. Dataset Generation

The creation of a dataset is a crucial aspect of addressing this problem statement, as well as for its evaluation and refinement processes. Initially, we experimented with various methods to automatically generate queries, employing zero-shot, few-shot, and other prompting strategies, particularly those involving an intermediate step of query generation. It soon became apparent, however, that there was a necessity for a compact dataset, curated either manually or semi-automatically. This initial dataset would serve as a foundation to ensure diversity and intricacy before expanding it further.

3.1. Self-Instruct

We tried the Self-Instruct approach, which automatically generates the instruction dataset based on a few provided seed tasks. We validated it for our domain-specific task but did not obtain convincing results, and as a result, we discarded it. We observed that Self-Instruct doesn't perform well in domain-specific tasks with GPT-3.5-turbo-instruct or Davinci. The framework consists of four main processes: Instruction generation, Instance Generation, Filtering, and Postprocessing. The first generates tasks or queries, and the second generates new instances (query-output pairs) based on the generated instructions. The third one is used to filter out similar instructions during the instruction generation process and also filters out some instances based on predefined conditions.

The context sizes for the best prompts of bootstrapping instructions and instance generation prompts were 3.1k and 1.7k, respectively. The generated instructions were mostly vague or unsolvable, making them unusable for our task without additional human annotation. After multiple experiments with different temperature values, the number of few-shot examples in the prompt, Rouge-L thresholds, and estimating cost-effectiveness, this approach was discarded.

3.2. Human Guided Query Generation

To enhance the dataset's diversity and complexity, we initially deployed a zero-shot approach using GPT 3.5 and ChatGPT 4, supplying it with a list of APIs. However, this method yielded overly simplistic queries, lacking the desired variety. Consequently, we shifted to a few-shot strategy, incorporating all examples from the Problem Statement into the prompt to stimulate the generation of analogous queries.

Further refinement involved selecting a handful of model-generated queries and manually augmenting them, thereby infusing additional diversity and complexity. These augmented queries were then reintegrated with the initial examples to facilitate the generation of more intricate queries. Additionally, providing specific instructions, such as which tool to use or the type of logic to apply, yielded more satisfactory and nuanced queries. This approach ensured a richer and more sophisticated dataset, better suited for our objectives.

3.3. Automated Generation Using Tuned Entities

A.G.U.T.E., or Automated Generation Using Tuned Entities, represents an innovative method we developed to enhance the diversity and complexity of generated queries. This method draws inspiration from Named Entity Recognition (NER) techniques. We identified 11 distinct types of named entities and initially assigned them a neutral weight of zero. Utilizing a randomization process, our system selects a set

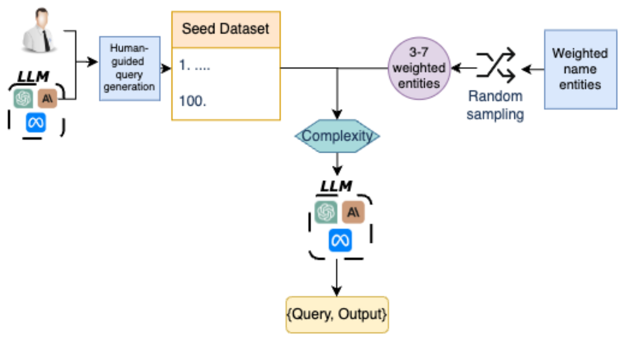


Figure 2. A.G.U.T.E: Automated Generation Using Tuned Entities

of 3 to 7 entities for each query. To leverage these weights effectively, we devised a complexity measurement function that aggregates the weights of the selected entities. This tailored approach allows us to adjust the weights dynamically, ensuring that each query achieves maximal complexity as determined by our function. The entities are then incorporated into a specially designed prompt, which is calibrated to optimize the complexity score. Compared to traditional few-shot prompt methods, queries generated through A.G.U.T.E. exhibit significantly greater diversity and complexity. This makes the method especially valuable for generating a seed dataset, a topic we explore in greater detail in the subsequent section.

Query Complexity	Avg Query Count	Avg API Count	Avg Arg Count	Avg Query Length
low	56	2	3	18
medium	160	3	6	25
high	121	5	8	29

Table 1. Data Statistics for AGUTE

3.4. Seed Dataset Generation

To construct our dataset, we initially formulated around 500 queries through a blend of two methodologies. Firstly, we crafted approximately 150 queries manually using the Human-guided Query Generation approach. This method ensures a high level of relevance and diversity in the queries. Subsequently, we employed the Automated Generation Using Tuned Entities (A.G.U.T.E.) technique to automatically generate the remaining queries. This process ensured a broad coverage of potential query types and contexts.

To further enrich our dataset, we employed Python scripts to augment the existing queries. This process involved systematically replacing entity names within the queries with other relevant entities, thereby creating new, yet contextually similar, queries. This method effectively doubled our dataset size to approximately 1000 queries, significantly enhancing its variety and scope.

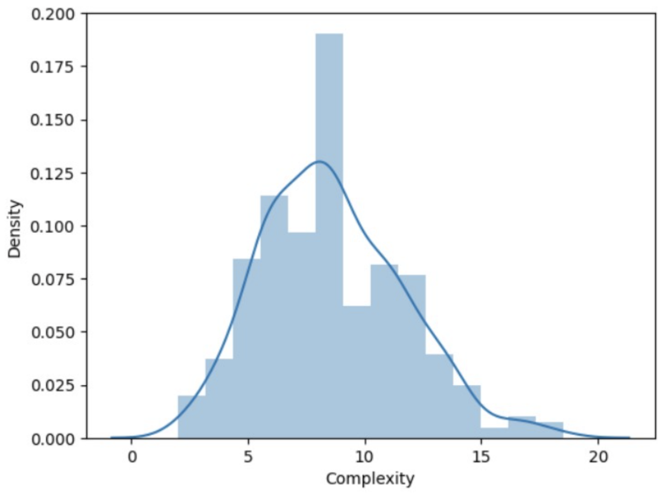


Figure 3. Query Complexity Distribution

3.5. Tool Generation

We have implemented a novel method for integrating and optimizing the use of various tools within our primary pipeline. This method involves adding, modifying, and deleting tools as needed, similar to the AGUTE approach but with a focus on APIs. To ensure the effective use of these tools, we've developed a system where each API is initially assigned a weight of zero. We then randomly select two APIs (excluding any recently modified or added) and assign them a weight of one. Using these weighted APIs, we generate queries with GPT-3.5, incorporating reasoning and JSON output. These queries are then added to our seed dataset, serving as practical examples to train the language model for enhanced tool utilization. This approach should greatly enhance the model's capacity to generate meaningful and relevant outputs when using these integrated tools.

4. Inference

We have developed and tested two distinct inference pipelines to meet varying deployment requirements and constraints. The second pipeline represents an advanced iteration of the first, incorporating the open-source large language model NexusRaven-V2 at its initial stage. For simpler queries, NexusRaven-V2 frequently yields accurate responses, thereby obviating the need for further processing along the pipeline. This integration results in expedited inference times and substantially reduced inference costs.

However, as the queries become more complex, the queries are processed through the entire pipeline. This renders the second pipeline more resource-intensive and time-consuming.

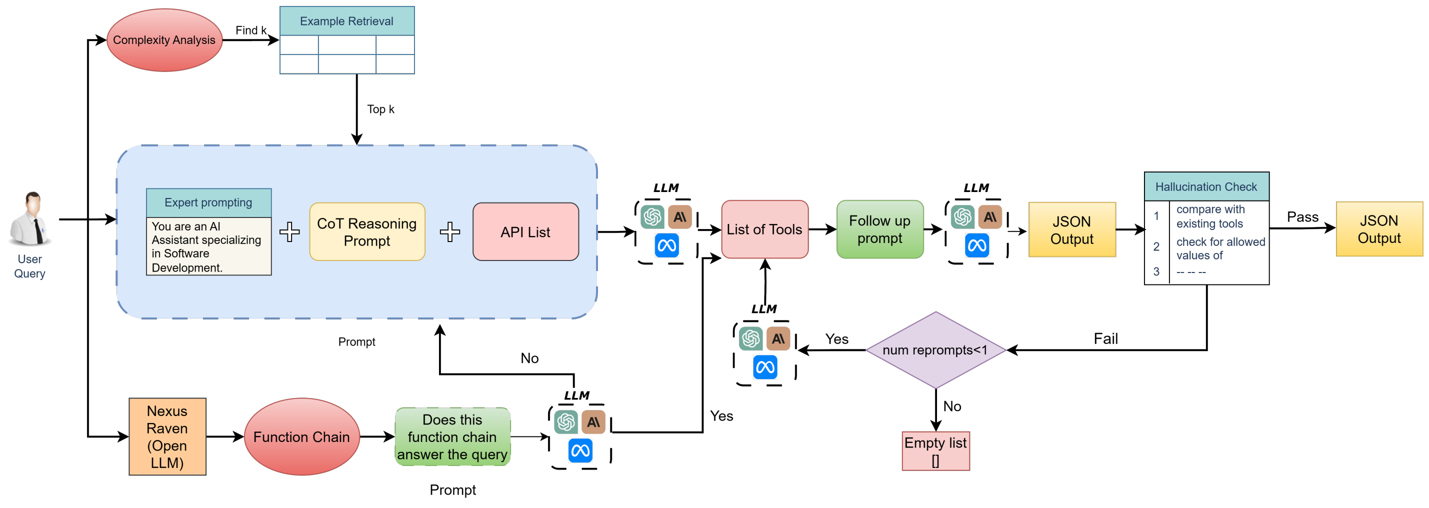


Figure 4. **Pipeline** A comprehensive depiction of our conversational AI system's operation. Starting with a Few-shot Chain of Thought prompt, high-quality examples are incorporated into the chat history. The system extracts JSON from the model's response, employing Python (Regex) initially. If unsuccessful, a secondary language model call is made, preserving the chat history. The core of the pipeline is the iterative hallucination check and correction loop, where JSON extraction is followed by error detection and subsequent correction attempts through model reprompting. This loop iterates up to 'n' times, defaulting to 1. If a hallucination-free output is not achieved, the system concludes with a blank list.

4.1. User Interface

The UI is built using the Streamlit framework. It offers three interfaces: one for the Chatbot, the second one for modifying the API toolset, and the final one for viewing the current API list and other utilities. The Chatbot page comprises a simple chat interface. The tool management page enables the user to: Add a new tool, Update or Delete a tool, Add an argument to a tool, Update or Delete an argument of a tool, and Delete multiple tools.

4.2. Pipeline

Our first pipeline begins with a Few-shot Chain of Thought chain, which includes the user query and a set of high-quality examples retrieved based on similarity to the query from a vector store. The output from the few shot-CoT prompt is kept in the chat history. We try to extract a JSON from the model's response using Python (Regex) code. If this extraction fails, we extract JSON from the response using another language model call, whilst also maintaining the chat history. The pipeline's core is the iterative hallucination check and the correction loop. We check for hallucinations in the extracted JSON and if hallucinations are detected as errors then the pipeline attempts to correct them. Hallucinations are corrected by reprompting the language model, giving it information about the errors it made, and instructing it to give a rectified response. We then again extract the JSON and the same procedure to check hallucinations and reprompting the model if hallucinations are present is repeated. This loop repeats up to n times, which by default is set to 1. If a hallucination-free output is still not obtained after n re-prompting attempts, the pipeline concludes that the query is unsolvable and outputs a blank list. Throughout this process, the chat history is managed according to the conversational context, being cleared or kept as needed.

4.2.1. Alternate Pipelines

We also utilized NexusRaven-V2 Huang et al. (2023), a 13-billion parameter open-sourced large language model (LLM), in our experimental setup. The primary reason for selecting NexusRaven-V2 (hereafter referred to as Raven) was its notable proficiency in function calling abilities, a feature where it demonstrably surpasses GPT-4 in zero-shot scenarios.

Our experimental observations indicated that Raven's performance was less than optimal in scenarios involving unsolvable queries or those requiring additional logical reasoning. To test Raven more thoroughly, we integrated it into our original pipeline with some modifications.

In our pipeline, initial queries are processed by Raven. For accuracy validation of Raven's responses, we utilized both GPT-3.5 and GPT-4. GPT-4 was slightly better at evaluating Raven's responses than GPT-3.5, but the difference was minimal. Therefore, for cost efficiency, we chose GPT-3.5. If Raven's response is confirmed as correct, we generate a JSON output via a GPT-3.5 call, using both the original query and Raven's response. If Raven's response is inadequate, the query is directed to our main pipeline.

We observed that Raven excels in handling simpler queries, particularly those less than 20 words in length. In such cases, both the average latency and token usage of this modified pipeline are reduced in comparison to our original setup.

In conclusion, while Raven shows promise in specific contexts, particularly in simpler, shorter queries and in its function-calling capabilities, its performance varies with the complexity and depth of the queries presented.

4.3. ExpertPrompting

ExpertPrompting Xu et al. (2023b) is an augmented strategy for instructing LLMs. For each specific instruction, ExpertPrompting first envisions a distinguished expert agent that is best suited for the instruction, and then asks the LLMs to answer the instruction conditioned on such expert identity.

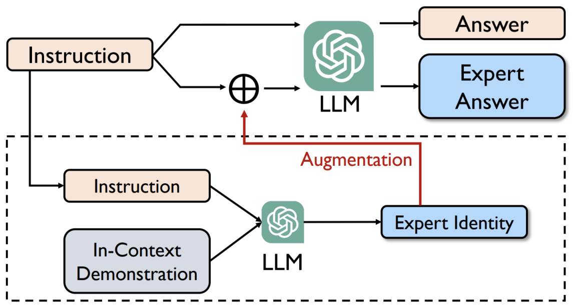


Figure 5. **Expert Prompting**, an augmented strategy for instructing LLMs. For each specific instruction, ExpertPrompting first envisions a distinguished expert agent that is best suited for the instruction, and then asks the LLMs to answer the instruction conditioned on such expert identity.

The expert identity is produced with In-Context Learning. Each expert identity is defined at very delicate granularity using a detailed and elaborate description. ExpertPrompting is also simple to implement, requiring no sophisticated crafting of prompt templates or iterative processes.

Our trials show that language models show improved performance with the inclusion of ExpertPrompting for our task of addressing user queries. It is also adaptable to new and different domains.

4.4. Retrieval ICL

In traditional In-Context Learning (ICL), a language model learns from a fixed set of examples that are provided to it in the prompt. These examples serve as a context for the model to understand and respond to new queries.

Retrieval ICL introduces a dynamic aspect to this learning process. Instead of using constant in-context examples, we retrieve examples that are most similar to the user query from a database of query, answer, and reasoning tuples related to the existing set of tools and arguments.

The database of examples is continuously updated with new information. This ensures that the model has access to the latest and most relevant examples, allowing it to provide up-to-date responses.

4.4.1. Dynamic Top K

The value of K, where K is the number of retrieved examples, is determined based on Query Complexity. More complex queries are augmented with a greater number of examples, while simpler queries are supplemented with fewer examples.

The complexity of a query is assessed using a simple yet effective heuristic: the number of words in the query. Other metrics to measure complexity like the Gunning Fog Index or SMOG (Simple Measure of Gobbledygook) grade may also be used.

4.4.2. Sentence Transformer Embeddings

We store embeddings of "user query - solution" example pairs. Embeddings capture the semantic meaning of the text, which allows for quickly and efficiently finding similar examples. We use sentence transformer embeddings from HuggingFace as they offer state-of-the-art performance, whilst also being free.

4.4.3. Vector Database

Initially, the embeddings of 80 carefully manually created examples are stored in a vector database. GPT-4 is then used to automate the process of updating our database so it is always up to date. The use of GPT-4 ensures that the database only contains high-quality examples. Our process is cost-effective as the database is only infrequently modified when the list of tools is changed.

4.4.4. FAISS

We use FAISS (Facebook AI Similarity Search) as our vector store. FAISS allows you to quickly search for similar examples. It includes nearest-neighbor search implementation that optimizes the memory-speed-accuracy tradeoff.

4.4.5. MMR Search

For extraction of examples for Retrieval-ICL, we use the Maximum marginal relevance(MMR) search. MMR is chosen as it optimizes for similarity to query and diversity among selected examples. Maintaining diversity is important as similar examples can limit the model's ability to generalize to diverse scenarios making it overfit to the characteristics of a narrow subset of examples.

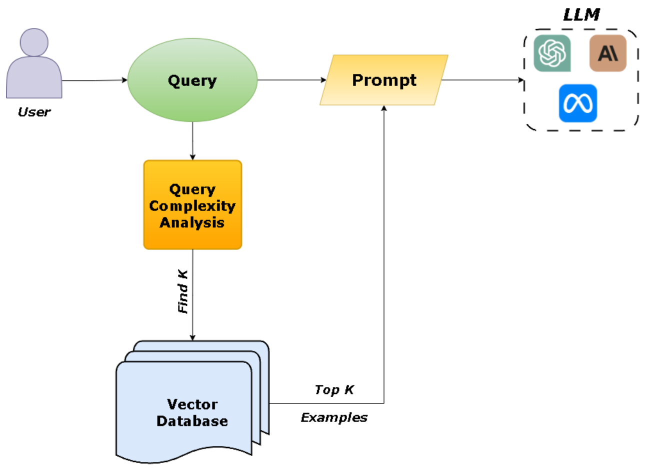


Figure 6. **Retrieval ICL with Dynamic Top K**, Instead of using constant in-context examples, retrieve K examples that are most similar to the user query from a database of query, answer, and reasoning tuples related to the existing set of tools and arguments. The value of K is determined based on Query Complexity.

4.5. Error Handling

We have designed a system to validate the JSON responses from the language model against a predefined set of tools and argument specifications. This system is used to ensure that the JSON responses are coherent and adhere to certain constraints.

4.5.1. Detecting Hallucinations

Here, we check for inconsistencies or errors in the tool names, argument names, and argument values provided in the JSON response. For this purpose, we maintain the following:

- `allowed_args_dict`

  A dictionary of allowed argument values.

- `available_tools`

  A list of valid tools.

- `available_arguments`

  A list of valid argument names.

- `args_in_list_dict`

  A dictionary indicating which arguments should be in list format.

Tool Name Validation: We validate each tool name in the JSON response against the list of available tools, identifying any hallucinated (invalid) tools.

Argument Name Validation: We check each argument's name in the JSON response to ensure it exists in the list of available arguments, identifying any hallucinated arguments.

Argument Value Formatting: We ensure that arguments that should be in a list format are indeed lists.

Argument Value Validation We validate argument values against the allowed values defined in allowed_args_dict (for those arguments for which this is applicable).

4.5.2. Corrective Reprompting

Next, we generate a correction prompt based on the output of our hallucination detection logs. A message is constructed to highlight these errors.

4.5.3. Placeholder Handling

For checking Placeholder tags, we use our custom-made function. Our function is designed to inspect the final JSON output and determine whether it contains placeholders in the argument values. The presence of placeholders is indicative of hallucinations.

Using a regular expression (re.search), the function checks if the argument value contains any placeholders enclosed within angle brackets ("<" and ">"). If a placeholder is found, the system is prompted to re-evaluate the response.

4.6. Corrective Reprompting

The primary objective of incorporating corrective reprompting is to iteratively refine and correct the output generated by the language model, particularly in scenarios where hallucinations or errors are detected in the JSON responses, by introducing a mechanism that allows for multiple attempts at correction. The key components of Corrective Reprompting are:

1. **JSON Extraction and Analysis Loop** We have implemented a loop that involves the extraction and analysis of JSON responses generated by the language model. This iterative process allows us to detect and address hallucinations.

2. **Iterative Correction Attempts** The system is designed to re-prompt the language model for correction if hallucinations are identified in the JSON responses. This iterative correction process is repeated up to a predefined number of attempts ('n times,' with the default set to 2).

3. **Unsolvable Query Conclusion** In cases where a hallucination-free output is not achieved after the specified number of correction attempts, the system concludes that the query is unsolvable.

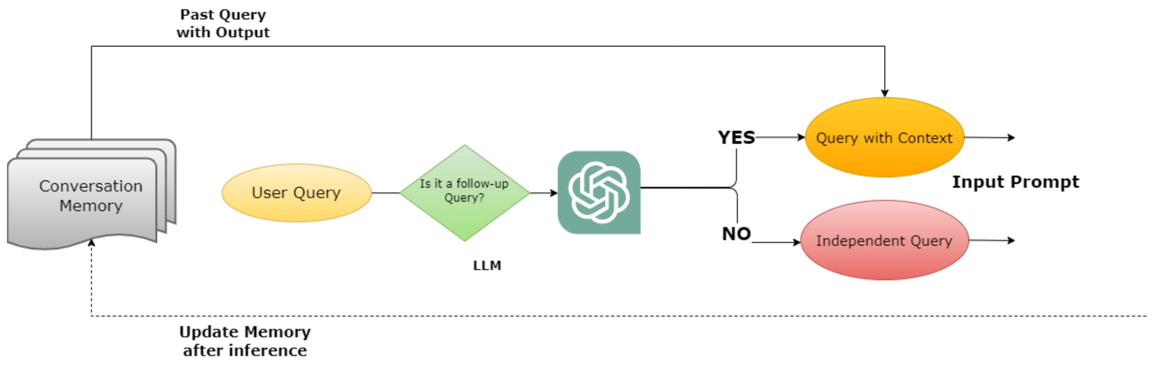


Figure 7. **Conversational Memory Handling**, One of the chains in inference for deciding whether a query is a follow-up to the previous one. Based on the decision, context/information is passed to the prompt.

4. **Chat History Management** The chat history is managed dynamically based on the conversational context.

5. **Correction Prompt Generation** Upon detection of errors, a correction prompt is generated based on the output of hallucination detection logs. This prompt highlights the specific errors found during the analysis and guides correction.

4.7. Output Structuring & Constraints

For converting complex outputs into structured JSON format, a variety of approaches can be employed, each with its own set of advantages and challenges. The key objective is to efficiently and accurately translate varied outputs into a standardized, easily interpretable format. In exploring this, we came across three different ways:

1. **Manual Method with Regex**

   This method involved processing pseudo-code outputs where the tool name appeared as function calls and argument names and values served as parameters. A Python script using regular expressions (regex) was employed to extract tools and arguments in the order they were called, bypassing the need for an additional language model (LLM) pass. This approach reduced token consumption and latency by eliminating extra LLM processing. However, this approach struggled with scalability due to the variability in pseudo-code structures.

2. **Language Model Query Language (LMQL)**

   This approach utilized LMQL to constrain the model's token output. Initially, the model was constrained to output an ordered list of tools. Subsequently, the model was further constrained to fill in argument values without reproducing the entire JSON structure. This approach significantly saved tokens by focusing only on the necessary output components and facilitated the generation of outputs in a more organized manner. However, the model did not always adhere to the constraints, especially in newer versions like GPT-3.5 Turbo and GPT-4.

3. **LLM Call with Intermediate Output**

   This approach Involved basic prompting to the LLM, followed by providing it with intermediate output. The LLM was tasked with generating JSON from this intermediate output. GPT-3.5 demonstrated strong capabilities in processing and converting intermediate outputs to JSON even on being given simple prompts.

We also looked into various guardrailing toolkits to add additional output constraints. NeMo Guardrails enables developers building LLM-based applications to easily add programmable guardrails between the application code and the LLM. Key benefits of adding programmable guardrails include building Trustworthy, Safe, and Secure LLM-based Applications, connecting models, chains, and other services securely and controllable dialog along with various other output constraints

4.8. Solvability Classification

Classifying a user's query as either solvable or not is a complex task that demands careful consideration and reasoning. Large Language Models (LLMs) have shown a tendency to hallucinate on unsolvable queries, often incorrectly categorizing them as solvable. To mitigate this issue we introduced a simple yet effective method by injecting a prompt in the original prompting method. To address this issue, we propose an intuitive method that improves upon the original prompting technique. After each "thought", our model is explicitly instructed to assess the solvability of the query. The pipeline proceeds based on the answer to this question by the model. Though not completely preventing the hallucinations, this makes the model less prone to hallucinating, ensuring that subsequent steps are guided by the model's assessment of the query's solvability.

4.9. Caching

From a production standpoint, as this application grows in popularity and encounters higher traffic levels, the expenses related to LLM API calls can become substantial. Additionally, LLM services might exhibit slow response times, especially when dealing with a significant number of requests.

To tackle this challenge, we can use semantic caching for storing LLM responses. Essentially, this allows us to store a database of previously asked and answered questions.

4.10. Conversational Memory Management

In the domain of chatbots, the concept of continuous memory is crucial for maintaining a conversation. Common methods employed include:

- Conversation Buffer: This involves retaining the complete context of the last 'N' messages exchanged.
- Summaries: Providing condensed versions of the conversation.
- Context Retrieval: Involves storing conversations in a vector database, which can be intelligently retrieved, especially beneficial for lengthy dialogues.

While these methods are prevalent, they fall short of addressing the requirements of managing continuous queries in our specific use case. The task of handling user queries in a conversation includes answering queries that may refer to some information mentioned in a past dialogue. The crux of the challenge involves:

- Identifying when to retrieve past dialogue.
- Extracting Relevant Entities.

These tasks, though seemingly straightforward, demand a high level of sophisticated reasoning, planning, and decision-making. To enhance our outcomes from the LLM, we use a method that models the memory contents fed into our AI system.

The large language model assesses, based on the context, whether to incorporate information from a previous conversation. It asks itself:

- "Is the current query a follow-up to the previous one?"
  - If the answer is yes, the model then identifies which specific information from the preceding dialogue is necessary to address the query effectively.
  - Conversely, if the query is unrelated, it is treated as an independent inquiry, thus requiring a fresh response devoid of past context.

Being domain-specific queries, it is highly unlikely the user might refer to some information mentioned long before the present conversation. So, we store the memory as a string as part of the chain instead of using a vector database. The memory is updated after every successful inference. This simple augmentation in prompting and the bot's memory management significantly enhances the flexibility and quality of response to contiguous chain queries in our system.

5. Toolset Updation

We seamlessly handle the process of toolset updation in our backend, taking all possible cases of toolset updation into account. Whenever a tool is modified/deleted or whenever a new tool is added, the necessary changes are made in the stored API list. The additional data structures that we maintain for handling hallucinations are also appropriately modified.

Whenever any tool or argument is deleted, the examples using the corresponding tools or arguments stored in our vector database are searched for and removed from the database.

In the case of tool or argument addition, new queries, explanations, and solutions are generated using GPT-4. The model is provided with few-shot examples from our database and is prompted to generate queries that use the specific new tool or argument. The model is also instructed to use other specific already existing tools. Which other tools the model should use and the few shot examples provided are randomized to promote diversities in the generated queries?

If an already existing tool is modified, we first remove all the examples using the corresponding tool from the database, followed by the generation of new examples for the modified tool.

The number of examples stored in the vector database at a time will be around 50-100, so infrequent checking of the database will not be labor expensive. We encourage checking the examples stored after many tool changes have taken place to check for any low-quality examples that may have been generated by the language model.

6. Latency and Cost Analaysis

Besides the correctness of the response, we measure inference latency and token cost usage for both pipelines. Both of these factors should be taken into account since there is a clear trade-off between these two factors and the correctness of their outputs. The cost is measured for both the pipelines on queries divided into three categories based on the complexities of the query. We categorize queries into three complexity levels and measure the inference latency and token cost for both pipelines. Complex queries demand more resources but require accurate responses.

Complexity	Few-Shot CoT chain	Reprompt chain	Overall	Avg Token Usage
low	4.425	5.02	17.745	8725.95
medium	7.87	4.31	19.445	9648.815
high	7.72	6.97	21.06	10,725

Table 2. Avg Latency and Cost Analysis for ChatGPT

Complexity	Raven chain	Follow-up chain	Overall Latency	Avg Token Usage
low	3.8	2.09	5.89	2461
medium	4.42	2.08	6.5	2483
high	5.71	3.05	8.76	2520

Table 3. Avg Latency and Cost Analysis for Raven

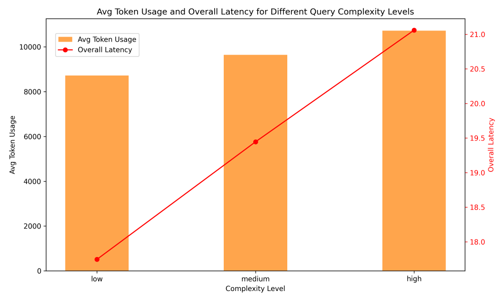


Figure 8. **Latency and cost vs complexity**, for ChatGPT

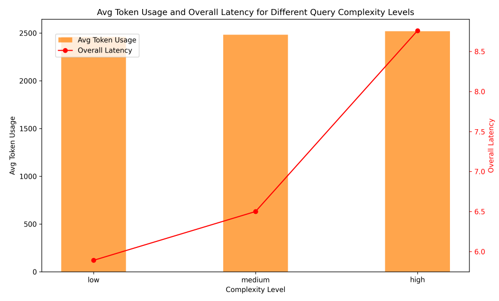


Figure 9. **Latency and cost vs complexity**, for Raven

The difference between latency and token usage between the two models is clear and the reason behind it is obvious. Though Chatgpt takes about 4x the cost of the Raven pipeline, ChatGPT proves to be more effective in handling complex queries, justifying its higher resource consumption. We explore ways to explore optimizations to reduce token cost without compromising accuracy, ensuring efficient use of resources in LLM pipelines.

7. Parameter Efficient Fine-Tuning

Some open-source models have shown really good performance in terms of reasoning as well as coding. We explored the idea of PEFT to be able to utilize these abilities by open-source LLMs for tasks as specific as ours.

For fine-tuning, we relied on synthetically generated data, which included queries from complex and conditional developer queries to simple compositional queries to make the model robust. We applied Qlora(Quantized LoRA) on two open-source models, Code Llama(7b) and CodeGen(7b). These models were chosen due to their capabilities in the domain and affordability in terms of training and inference costs.

The data it was fine-tuned on had Query and corresponding JSON output. The testing on the models revealed, that although the structuring of output was not exactly captured, the models were still able to recognize the relation between some entities and tools from the API list.

It was expected that these models wouldn't perform up to the mark just on fine-tuning, but this opens the possibility of using these models on even more downstream tasks. PEFT shows a lot of potential in improving the pipelines, though we were not able to test that quality due to a lack of appropriate datasets.

8. Additional Experimentation

8.1. NER

We experimented with Named Entity Recognition (NER). Our prompt provided the model with the definition of an entity and a few shot examples for NER, guiding it to identify and extract pertinent entities from user queries (Ashok and Lipton, 2023). The model would then request user verification to confirm the identified entities and their respective types. This approach stemmed from our observations that the model occasionally confounded similar entity types, particularly confusing IDs with names. Additionally, we anticipated that this entity extraction process would enhance the model's effectiveness by ensuring no critical information necessary for resolving the query was overlooked.

Our implementation of this approach did not yield the expected level of improvement. As the trials progressed, it became evident that the marginal increase in performance did not justify the additional latency and computational demands incurred.

8.2. Chain of Verification

We implemented the Chain-of-Verification (COVE) (Dhuliawala et al., 2023) method whereby the model first (i) drafts an initial response; then (ii) plans verification questions to fact-check its draft; (iii) answers those questions independently so the answers are not biased by other responses; and (iv) generates its final verified response. We also tried providing pre-defined verification questions to the model but neither of the two approaches worked as we observed a decrease in performance.

8.3. Emotional Prompting

Emotional Prompting (Li et al., 2023) integrates emotional stimuli into prompts to enhance the efficacy of Large Language Models. In our experiments, we incorporated emotional prompting for our task by adding phrases like "Please be sure, as it is very important" to our prompt but we realized that this inclusion did not enhance the performance for the task at hand.

8.4. Language Model Query Language

Language Model Query Language or LMQL by Beurer-Kellner et al. (2023) provides a novel way of combining prompts, constraints, and scripting on LLM outputs by interleaving traditional programming with the ability to call LLMs in the code.
Our problem involves three major tasks, that are, minimizing token usage, minimizing inference latency, and output structuring to JSON, all while maintaining the correctness of the output. We tested LMQL to achieve exactly this on OPENAI models of gpt-3.5-turbo-instruct, chatgpt, gpt-4, and text-davinci-003.
For gpt-3.5-turbo-instruct and text-davinci-003, LMQL effectively enforced token constraints and structured outputs, aligning with our efficiency goals. However, with gpt-4 and chatgpt, we encountered limitations. The OpenAI API Completions and Chat API currently lack robust support for token masking and detailed token distribution analysis, which hindered similar successes in efficiency with these models.
A comparative analysis revealed that, despite advanced prompting and forced output constraints, the base performance of these models was not optimal at all, even on simpler queries. The low accuracy significantly diminished the benefits of time and token cost savings, suggesting a need for further refinement in LMQL's integration Future work might explore alternative approaches to integrating LMQL with these models or propose modifications to the OpenAI API that enable more control over token usage and output formatting. "

8.5. Intermediate Pseudo Code Representation

Traditional JSON structures are generally not well-suited for representing complex logical constructs. To address this, we considered generating intermediate pseudo-code representations. Pseudo-code, with its flexibility and closeness to actual programming logic, can easily represent conditions, loops, and other control structures.

We treated tool names as function calls, with their corresponding arguments and values represented as parameters and parameter values in the pseudo-code. This enabled a more expressive and versatile representation of complex queries.

In addition, we also attempted to make the pseudo-code more deterministic by following a fixed format. Unique identifiers were introduced before every function call and argument names to facilitate manual extraction using Python scripts.

However significant challenge that we faced was the variability in pseudo-code outputs. Despite efforts to standardize the format, the inherent flexibility and diversity in pseudo-code representation made it difficult to have a fixed format.

8.6. Query Refinement for Multi-Turn Conversations

We tried updating the user queries in multi-turn conversations using an LLM call to take into account the previous conversation so that we could use the queries without passing the entire conversational history.

The issue we faced here was in classifying which query needs to be modified based on the conversational history and which should be left as it is but we were not able to find a satisfactory method that allowed us to accomplish this classification.

9. Resources

Our initial testing of prompting techniques with various language models was conducted using several conversational AI platforms including claude.ai, the free version of ChatGPT, the LMSYS chat interface, and Forefront.ai. We also leveraged Langchain to implement various complex conversational chains and analyze the performance of multiple pre-trained language models. In addition, we utilized the OpenAI API to access their language models for evaluation.

Forefront.ai hosts the GPT-3.5 and Claude Instant 1.2 models with free public access. A key benefit is the ability to adjust the response temperature setting, improving the reproducibility of test results. Specialized assistants are available on this platform to assist with unique domains, helping maximize the benefits of ExpertPrompting Xu et al. (2023b). Leveraging the software engineer assistant allows for further refinement in handling user queries.

In addition to using OpenAI's API, ChatGPT was also used to test GPT-3.5. Further, Claude.ai was used to facilitate the testing of Claude 2. The LMSYS Chat interface also served as a valuable tool in our trials as it offers access to a diverse range of language models.

After our initial prompt tests, we maintained the use of these platforms for testing each component of our inference pipeline, while continuing our prompt tests in parallel. Furthermore, we secured access to ChatGPT Plus, enabling us to extend our evaluation on GPT-4.

Further, the fine-tuning process was conducted on Kaggle using their T4 GPUs, while the inference scripts were executed on Google Colab.

References

D. Ashok and Z. C. Lipton. Promptner: Prompting for named entity recognition, 2023.

L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969, June 2023. ISSN 2475-1421. doi: 10.1145/3591300. URL http://dx.doi.org/10.1145/3591300.

T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

S. Dhuliawala, M. Komeili, J. Xu, R. Raileanu, X. Li, A. Celikyilmaz, and J. Weston. Chain-of-verification reduces hallucination in large language models, 2023.

S. Hao, T. Liu, Z. Wang, and Z. Hu. Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings, 2023.

J. Huang, W. Ping, P. Xu, M. Shoeybi, K. C.-C. Chang, and B. Catanzaro. Raven: In-context learning with retrieval augmented encoder-decoder language models, 2023.

N. Jain, P. yeh Chiang, Y. Wen, J. Kirchenbauer, H.-M. Chu, G. Somepalli, B. R. Bartoldson, B. Kailkhura, A. Schwarzschild, A. Saha, M. Goldblum, J. Geiping, and T. Goldstein. Neftune: Noisy embeddings improve instruction finetuning, 2023.

T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal. Decomposed prompting: A modular approach for solving complex tasks, 2023.

C. Li, J. Wang, Y. Zhang, K. Zhu, W. Hou, J. Lian, F. Luo, Q. Yang, and X. Xie. Large language models understand and can be enhanced by emotional stimuli, 2023.

X. L. Li and P. Liang. Prefix-tuning: Optimizing continuous prompts for generation, 2021.

B. Y. Lin, A. Ravichander, X. Lu, N. Dziri, M. Sclar, K. Chandu, C. Bhagavatula, and Y. Choi. The unlocking spell on base llms: Rethinking alignment via in-context learning. *ArXiv preprint*, 2023.

E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.

Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, R. Tian, R. Xie, J. Zhou, M. Gerstein, D. Li, Z. Liu, and M. Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code, 2023.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.

Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi. Self-instruct: Aligning language model with self generated instructions, 2022.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

B. Xu, Z. Peng, B. Lei, S. Mukherjee, Y. Liu, and D. Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models, 2023a.

B. Xu, A. Yang, J. Lin, Q. Wang, C. Zhou, Y. Zhang, and Z. Mao. Expertprompting: Instructing large language models to be distinguished experts, 2023b.

Q. Xu, F. Hong, B. Li, C. Hu, Z. Chen, and J. Zhang. On the tool manipulation capability of open-source large language models, 2023c.

R. Yang, L. Song, Y. Li, S. Zhao, Y. Ge, X. Li, and Y. Shan. Gpt4tools: Teaching large language model to use tools via self-instruction, 2023.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models, 2023.

L. Yuan, Y. Chen, X. Wang, Y. R. Fung, H. Peng, and H. Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets, 2023.

R. Zhang, J. Han, C. Liu, P. Gao, A. Zhou, X. Hu, S. Yan, P. Lu, H. Li, and Y. Qiao. Llama-adapter: Efficient fine-tuning of language models with zero-init attention, 2023a.

Y. Zhang, H. Cai, Y. Chen, R. Sun, and J. Zheng. Reverse chain: A generic-rule for llms to master multi-api planning, 2023b.

Y. Zhuang, Y. Yu, K. Wang, H. Sun, and C. Zhang. Toolqa: A dataset for llm question answering with external tools, 2023.