DevRev's AI Agent 007
Tooling up for Success

Final Report

Team 22

Inter IIT Tech Meet 12.0

1 Introduction

The problem put forward deals with tool selection and generation of a tool call sequence conforming to a provided set of tools and rules. This report documents further experimentation, the final pipeline, testing, challenges faced, and future work possible with regards to the problem.

The solutions proposed in the Mid-Evaluation report have been implemented after performing thorough evaluations of outputs, with close consideration of bottlenecks and cases of failure. An independent implementation of ToolDec [1] was tested, and further evaluation of open-source models like the newly released NexusRaven[2] was carried out to augment tests made prior to the Mid-Evaluation. The QLoRA method (Quantized Low-Rank Adaptation)[3] was used to finetune these models to better follow query formats in general.

Even with finetuning, the smaller (7-13B parameter size) open-source models do not match the accuracy of closed-source models such as GPT-3.5/GPT-4-turbo/Claude. Therefore the GPT series of models is used via API for the final solution. To bring down the costs of model usage, GPT-3.5-turbo is used as a Tool Retriever, while GPT-4-turbo generates the final tool call sequence, providing cost-effective and accurate outputs.

2 Proposed Solution

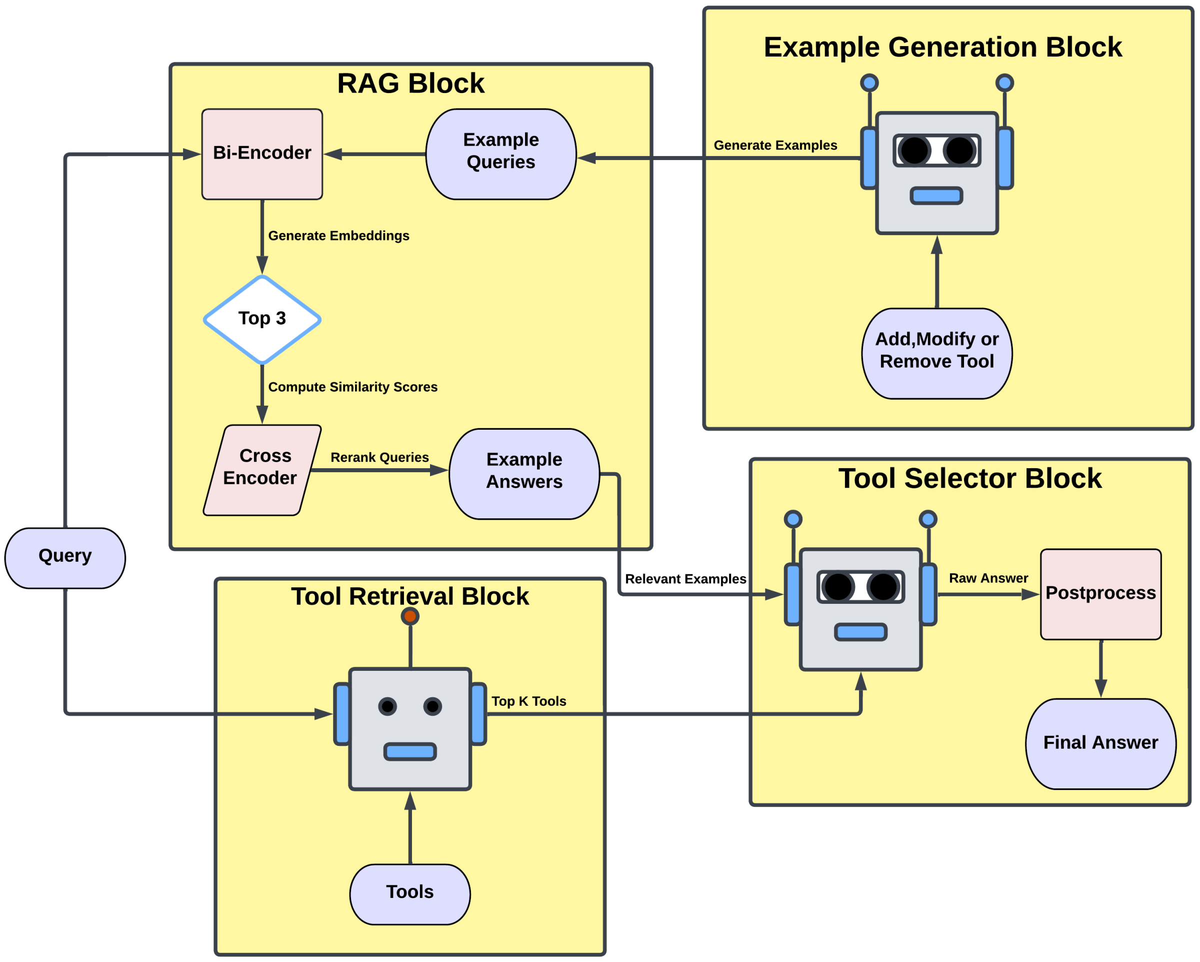

Figure 1: Our Proposed Pipeline composed of the RAG Block, Tool Retriever Block, Example Generation Block and Tool Selector Block

After experimenting with different approaches as mentioned in section 4, the best solution in terms of the inference time, cost, and accuracy is proposed in this section. The final pipeline that we propose consists of four primary blocks, an **Example Generation Block** for creating new examples on changing the tool dynamically, a **Tool Retriever Block** which reduces the tool pool and helps us narrow down the answer efficiently, a **Retrieval Augmented Generation Block** to retireve the relevant examples, and a **Tool Selector Block** to produce the final answer, which is filtered by a post-processing function.

2.1 Working

Assuming that we receive a toolset T, a few examples E, and a user query Q initially as input, we take the query Q and toolset T as inputs to GPT-3.5-Turbo, to narrow down the tool search pool and return the possible tools that may be used to solve the query Q. Let us call the reduced tool pool T_r. Now, this reduced tool set T_r is passed to the GPT-4-turbo prompt for producing the final answer. This is done to reduce the cost significantly by reducing the number of tokens in the prompt given to GPT-4-turbo. Meanwhile, along with the narrowed-down tool list T_r, we also provide GPT-4-turbo with top-k examples by performing RAG by computing the similarity between the user query and the example queries. Finally, we prompt GPT-4-turbo to produce the required tools to answer the given user query Q. Our solution is robust and flexible to the addition of new tools, deletion of old tools, and modification of the old tools dynamically. This is done by deleting/modifying or adding the tools in the tool list T and updating the examples E accordingly. We also have a mechanism that supports the creation of new examples on addition of new tools.

2.2 Components and Blocks

2.2.1 Tool Retriever Block

The Tool Retriever Block uses GPT-3.5-Turbo and plays a very important role in the entire pipeline as it narrows down the toolset T, significantly reducing the tokens in the prompt for the Tool Selector Block. This effectively halves the cost and time as GPT-3.5-Turbo is quicker and less costly than GPT-4-turbo. We are not losing on the accuracy as we refuse to get the final answer from this block but we ask to reduce the tool pool, which is a much simpler task. The prompt that we provide to GPT-3.5-Turbo consists of the list of tools (the toolset T), and the user query Q. Based on the user query Q, we ask the model to return the tools that can be possibly used to answer Q. After conducting experiments, we found the ground truth to be a subset of the T_r in most cases.

2.2.2 Example Generation Block

The Example Generation Block is responsible for the generation of relevant examples on change of the toolset T. When a new tool is added in the toolset T, the example generation block creates relevant examples incorporating the new tool. Similarly, when there is any modification in the existing tools, this block essentially modifies the respective examples. It also deletes the examples on the deletion of any tool. Moreover, this block also makes sure that the number of examples corresponding to each tool does not go down beyond a threshold number. When on deletion of a tool, the examples using that tool get deleted. Now if the number of examples goes below the threshold, the examples get replenished and the set E is then used to fetch the top-k examples using RAG. We apply RAG on the user query and the example queries. The top-k queries and their corresponding answers form the input of the Tool Selector Block. All the examples are generated using GPT-4-turbo for higher accuracy.

2.2.3 Retrieval Augmented Generation Block

The Retrieval Augmented Generation Block consists of the Biencoder and Cross Encoder Blocks which are used to compute the embeddings to retrieve the top-k example queries and rerank the retrieved queries based on the similarity with the user query. Then we get the answers to the example queries and pass it as examples to facilitate in-context learning.

2.2.4 Tool Selector Block

The Tool Selector Block consists of the GPT-4-turbo model with query Q, the reduced set of tools T_r, and the top-k examples as its input. It is prompted to accurately return the tools that would be able to solve the query Q. Since we are not passing the entire tool list into the prompt, this solution is extremely cost-efficient.

3 Literature Review

3.1 ToolTalk

ToolTalk [4] is a benchmark for complex user multi-step tool usage in conversation settings. It contains 28 tools and includes a simulated tool implementation allowing automated assistant evaluation relying on execution feedback. It not only includes tools that search for information but also changes the information affecting the external world. It evaluates GPT-3.5 and GPT-4-turbo on the benchmark and reports 26% and 50% success rates, respectively. The conversation dataset contains 28 easy (single tool call) and 50 hard (multi-tool call) human-made examples. It uses a two-phase method for evaluation, in the first phase it runs the assistant with all prefixes that end in a user utterance where it can predict a tool call or generate a response from calls and their results made earlier; if it predicts a tool call it executes the tool implementation and then provides the result to assistant. In the next phase, we compare the tool calls predicted for prefixes against the ground truth to compute tool invocation recall and incorrect action rate.

The authors noticed three reasons why GPT-3.5 and GPT-4-turbo fail on the benchmark:

1. The model predicts a tool call before the user gives all the details about it. This is mostly harmless for tools corresponding to 'GET' requests, but for 'POST', 'PUT', 'UPDATE', and 'DELETE' is highly dangerous, leading to tampering with real-world data.
2. The model exhibits poor tool use planning, resulting in omission and wrong tool use. This occurs mostly due to hallucinations.
3. It picks the correct tool but invokes the simulated implementation with wrong or missing arguments. This can happen from failing to understand documentation or understanding the output of previous tool invocations or weak mathematical skills.

The authors experimented with removing tool descriptions, leading to significant performance drops across all the benchmark metrics except the incorrect action rate. This could be possibly due to tools being harder to use without documentation, resulting in less tool usage overall.

3.2 ToolFormer

Most methods teach LLMs tool use via prompting or fine-tuning the LLM, which requires human annotation and supervision. However, ToolFormer [5] uses a self-supervised method, which requires nothing but a demonstration of some APIs for the LLM to learn tool usage. The ToolFormer method thus reduces the requirement for human annotations by building a finetuning dataset of its own and finetuning in such a manner that the LLM does not lose any generality or any of its language modelling abilities.

The method uses special '$\langle$API$\rangle$' and '$\langle$/API$\rangle$' tokens to demarcate API calls in the dataset. This dataset is generated by the LLM itself, by prompting the LLM with the details about the concerned APIs, with some demonstrations.

The methods used for the generation are: 1) Sampling API Calls, 2) Executing API Calls, 3) Filtering API Calls. The method uses its own novel loss function and uses thresholding to decide if the API call is usable.

Fine-tuning the LLM is done with the generated data. The authors use GPT-J (6.7B parameters) for all testing and benchmarking purposes. There are however multiple problems with the method. Firstly, the computational cost is very high. Secondly, several tools or APIs may be interlinked, i.e the outputs are dependent in some way. However the API calls are made independently.

3.3 TRICE

Tool learning with execution feedback (TRICE) [6] is a two-stage end-to-end framework that enables the model to continually learn through feedback derived from tool execution, thereby learning when and how to use tools effectively.

In the first stage [2], The Model is taught how to use the tools by sequence to sequence training on outputs generated by ChatGPT giving it preliminary functionality, this is called Behavior Cloning.

The second stage, RLEF (Reinforcement Learning with Execution Feedback), enables the model to selectively utilize

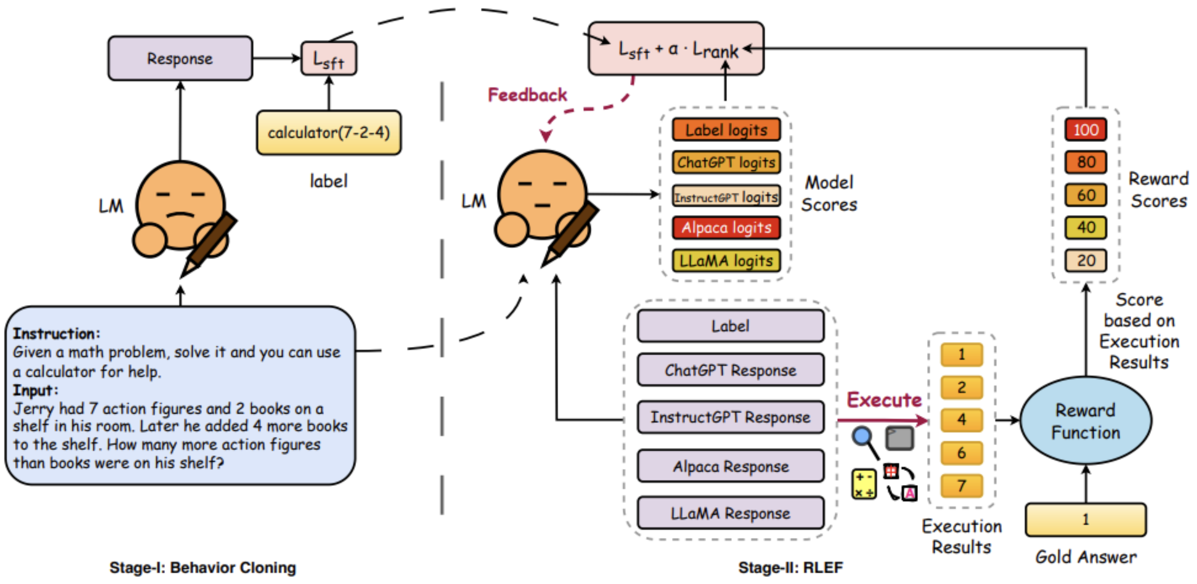
Response
L_{sft}
calculator(7-2-4)
label
LM
Instruction:
Given a math problem, solve it and you can use a calculator for help.
Input:
Jerry had 7 action figures and 2 books on a shelf in his room. Later he added 4 more books to the shelf. How many more action figures than books were on his shelf?
Stage-I: Behavior Cloning
$L_{sft} + a \cdot L_{rank}$
Feedback
LM
Label logits
ChatGPT logits
InstructGPT logits
Alpaca logits
LLaMA logits
Model Scores
100
80
60
40
20
Reward Scores
Score based on Execution Results
Label
ChatGPT Response
InstructGPT Response
Alpaca Response
LLaMA Response
Execute
1
2
4
6
7
Execution Results
Reward Function
1
Gold Answer
Stage-II: RLEF

Figure 2: The overview of our proposed framework TRICE. In stage-I (Behavior Cloning), We conduct supervised fine-tuning on the dataset to let the model imitate the tool-using behavior. In stage-II (RLEF), We further reinforce the model based on RRHF with tool execution feedback, guiding the model to selectively use tools.

tools and increases the accuracy of the model by enhancing its capability to selectively utilize tools and improving the accuracy of decision-making regarding tool types and corresponding inputs. Here the outputs are generated by LLMs like ChatGPT, LLaMA, or even provided by human experts. These outputs are then scored using a reward function using the gold answer. Along with this the loss function also contains a Supervised Fine-tuning Loss term so that the model doesn't deviate too much from the goal.

The authors use mathematical reasoning datasets to train and evaluate the model and find it to be better than Toolformer. We reason that we can apply the same methodology for our toolset in the future.

4 Experimentation

4.1 Setup

To standardize the process we formatted the toolset in JSON in the specified format.

```
[{
 "tool_name":<NAME OF THE TOOL>,
 "tool_description":<DESCRIPTION OF THE TOOL>,
 "return_type":<RETURN TYPE OF THE TOOL>,
 "argument_list":[{
 "argument_name":<NAME OF THE FIRST ARGUMENT>,
 "argument_description":<DESCRIPTION OF THE FIRST ARGUMENT>,
 "argument_type":<TYPE OF THE FIRST ARGUMENT>,
 "examples":[<EXAMPLE 1>, <EXAMPLE 2>, ...]
 },{
 "argument_name":<NAME OF THE SECOND ARGUMENT>,
 "argument_description":<DESCRIPTION OF THE SECOND ARGUMENT>,
 "argument_type":<TYPE OF THE SECOND ARGUMENT>,
 "examples":[<EXAMPLE 1>, <EXAMPLE 2>, ...]
 },
]}]
```

4.2 Retrieval Augmented Generation (RAG)

LLMs, in seeking to model the probability distribution of their training data, tend to make up ”facts” that are simply not true. While fine-tuning an LLM on some corpus of information may help incorporate the knowledge enclosed within the corpus in the LLM’s parameters, it is costly to repeat this process every time the corpus is updated. Given the task at hand, it is undesirable for the LLM to generate responses that use tools that do not exist at all and are made up or have changed in some way due to an update. Retrieval augmented generation provides an external knowledge base for the LLM to draw upon during generation and hence is appropriate for tasks where an extensive corpus of knowledge is essential for good outputs and where the knowledge base may change with time. Therefore, we found it essential to use the concept of retrieval augmented generation, providing the tools to the LLM for grounded generation of tool call sequences. This enables easy switching of tool sets, as these are simply provided as information with the prompt.

4.2.1 Single Tool Example

First we tried to generate examples of the tools such that each example contained one single tool, the main idea being that the model would be able to comprehend the usage of the tools provided the examples. This approach would be very cost efficient since each example only contains one tool call but there is one major drawback which we found during our experimentation that the model was not able to properly compose tools to solve a complex query.

4.2.2 Multi Tool Example

After our single tool experiment we proceeded to generate examples containing multiple tools and used those examples in our RAG pipeline. This boosted our accuracy by a minimum of 20% as shown in table 2.

We also conducted experiments where we mixed the number of single and multi-tool examples and found that using 3 multiple tool examples performed the best.

Model	**Number of Multi-tool examples**	**Number of Single-tool examples**	**Accuracy**
GPT-4-turbo	0	0	76.19%
	1	0	90.48%
	2	0	90.48%
	3	0	95.24%
	4	0	90.48%
	1	3	85.71%
	0	5	66.66%
GPT-4-turbo (with return types)	0	0	61.90%
	1	0	85.71%
	3	0	100.00%
	1	3	85.71%
	0	5	57.14%
Claude-2.1	0	0	38.10%
	1	0	66.67%
	3	0	80.95%
	1	3	71.43%
	0	5	28.57%

Table 1: Results from the RAG experiments on closed-source models

4.3 Prompting

We conducted experiments to evaluate and compare various open-source models. The goal was to measure their accuracy and overall performance. We used various prompting strategies during the benchmarking process to accomplish this. During benchmarking, several prompting techniques were investigated. Four distinct prompting strategies were explored. Simple prompting, the chain of thoughts (CoT), the contrastive chain of thoughts (CCoT), and the tree of thoughts with a single prompt (SPToT) were the prompting techniques that were evaluated.

4.3.1 Simple Prompting Technique

The simple prompting technique was very straightforward for the model to initiate the response. We used a single prompt to get the result from the model. In this single prompt, we included the tool list and 3 examples of the tool use. The tools list was provided in a proper and valid JSON format. The examples chosen for the prompt were such that they cover most of the tool use so that the model could understand the tool usage properly. This technique aims to assess the model's ability to maintain context, coherence, and logical flow of information across a series of interconnected prompts

4.3.2 Chain of Thoughts (CoT)

CoT technique involves presenting a sequence of related prompts or questions to the model, creating a narrative or progression of ideas. In this technique, we provide the model with the prompt with has the complete tool list and the 3 examples, along with the reasoning of why this tools were used. The example chosen were such that they cover the maximum variation of tool usage.

4.3.3 Contrastive Chain of Thoughts (CCoT)

CCoT involves method of contrast and/or opposition. The prompt that we used for the CCoT technique gives the model the complete tool list in proper format. Also 3 examples were provided in the prompt. Each of these examples includes the query, its correct tool use, the correct reasoning for the tool use and a reasoning that leads to mistakes for the tool used. Using this method the model learns more about tool usage.

4.3.4 Tree Of Thoughts With Simple Prompting (spToT)

The spToT technique involves branching of different thought from the main hub. In this technique we provided the model with a tool list and three examples. Along with these examples, a tree like structure of thoughts was provided, leading to a correct answer. At each step, upto three and at least two options were provided for selection. The model was thus encouraged to look at a wider set of possibilities before deciding upon the final solution.

4.3.5 Results

All the four techniques were tried out on various open source model to evaluate their overall performance. We found out that simple prompting outperforms CoT, CCoT, and spToT on most of the open source models. This might be due to model hallucination because of the additional information provided in the examples. These prompting techniques seem to work much better for larger LLMs (i.e., greater than 100B parameters). Smaller models are found to produce illogical chains of thought, thus degrading performance compared to standard prompting.

4.4 Retriever Fine-Tuning

As discussed in the proposed solution section 2, we utilize an instance of GPT-3.5-turbo to first retrieve tools relevant to a given query, utilizing the names and descriptions of the tool. Our initial experiments proved to be fruitful, achieving an F1 score of around 0.80 out of 1 with a simple prompt containing the tool names and the tool descriptions. To further improve our tool retrieval capabilities, we attempted to fine-tune GPT 3.5. We constructed a dataset containing around 120 query-answer pairs made by prompting Claude-2.1 and GPT-4-turbo with an expanded tools list of around 25 tools. We then expand our dataset for the purpose of retrieval by removing certain unused tools for the tools list appended to each prompt, while maintaining the answer of retrieved tools.

	Prompting Technique			
Model	Simple prompting	COT	CCoT	SPToT
Zephyr	3	2	3	3
Neural Hermes	4	0	3	2
Starling	1	0	1	2
Tora	0	0	0	0
Orca	0	1	0	0

Table 2: Open-source model results on different prompting techniques

We fine-tuned the GPT-3.5-turbo-0613 model on this dataset for 2 epochs. The training and validation loss both decrease and converge. On the holdout test dataset, we observe the finetuned model performs not much better than the base GPT-3.5-turbo-0613 model. Both models tended to produce similar F1 scores, though the finetuned model was ahead by 1 or 2 percent.

4.5 Adversarial Feedback Mechanism

The adversarial feedback mechanism involves something similar to reinforcement learning where there are two agents, a predictor and a corrector. The predictor proceeds to predict the answer in the required format. The corrector validates the response of the predictor and gives critical feedback to the predictor to improve on its response to reach closer to the correct answer. We realize that this may go on in an infinite loop. Therefore, we tried to perform two experiments which are as follows:

4.5.1 Finite Trials-Predictor and Corrector Mechanism

The response of the predictor is fed into the corrector and feedback on the response is produced. This conversation is stopped after three cycles. We found that this solution though intuitively very smart, does not produce accurate results. It tends to give repetitive feedback and the predictor is unable to reach the correct solution. We tried to experiment with this on GPT-3.5-turbo. The results were not great. Another disadvantage of this is that this solution is very expensive because of multiple agents and a greater cost on multiple API calls.

4.5.2 Query Validation Method

In this method, we use the predictor agent to predict the tool calls given a query and list of tools, while the corrector analyses the answer produced by the predictor agent and tries to go backward by producing a query based on the tool calls. Now the response of the corrector is checked against the real query. If the response of the corrector matches the real query, then the answer is mostly supposed to be right. In case, the query produced by the corrector does not match the real query, feedback is given to the predictor to improve on its answer. This also does not work. This proceeds to give repetitive responses. Also, in some cases, it gave a matching query but the answer was not remotely related to the ground truth. It also has a lot of disadvantages in terms of accuracy, time, and cost because of multiple API calls.

4.6 ToolDec

We implemented ToolDec [1] which is an FSM-based decoding strategy for LLMs as we proposed in our Mid Evaluation. We create a finite-state machine with states as per Table **??**.

Different tokens are added to the input according to the state which the FSM is present in, and the state changes depending on the output provided by the LLM. This process continues until END state is reached, which denotes the completion of the answer. The next state is decided depending on the output provided by the LLM and we instruct the LLM to generate an output from the list of allowable outputs that can be used to continue the answer. We then change the state accordingly and add the output generated by the LLM in the next state.

The above table provides a brief overview of all the 13 different states we used in our implementation. The code our FSM is provided in the Appendix. We however observe that it takes a lot of time (3 to 5 minutes) for execution, having

State Name	Token Added	Next States	Description
START	'['	NT, END	The Starting State
END	']'	-	The Ending State
NT	'{"tool_name":"'	CT	This state starts adding new tool
CT	next_tool_token	SAL, CT	This state continue adding the tool name to the answer
SAL	',"arguments":['	EAL, CAN	This state starts the argument list in the respective tool
EAL	']}'	END, CTL	This state ends the argument list
CTL	','	NT	This state continue adding new tools to the existing tool list
CAN	'"{argument_name":"'	AV, CA	This state start creating the argument name
CA	next_argument_token	AV, CA	This state continue creating the argument name
AV	',"argument_value":'	CAV	This state starts adding the argument value to the answer
CAV	next_argument_value_token	EAV	This state continue adding the argument value to the answer
EAV	' "} '	CAL, EAL	This state ends the argument value
CAL	','	CAN	New arguments are added to the argument list

Table 3: ToolDec states, description, and the relation with tokens & next states for the Finite-State Machine (FSM)

to execute a forward pass through the LLM at each deciding token which can change the answer. The output of the LLM is however not perfect. We did try different methods like changing the prompt and using RAG. However the end results were inconclusive due to time constraints.

4.7 Example Generation

We have tried two techniques for generating examples: zero-shot and few-shot using GPT-4-turbo, GPT-3.5-turbo, Claude-2.1, and Claude-instant-1.2. While Claude-instant-1.2 generated the fastest, it mostly gives examples that can be answered using one tool only and misses on a few tools like search_object_by_name, who_am_i. GPT-3.5-turbo hallucinates by generating examples with previous tool references like `$$PREV[0].result`. Claude-2.1 works great on generating accurate query-answer pairs, but it generates examples similar to one another and fairly easier queries. GPT-4-turbo, by far, generates the best examples as the queries are complex, and it generates a correct answer in accordance with the return type and argument data type.

4.8 Open Source Models

Open Source Models have improved a lot in recent times, even beating GPT-4-turbo on Zero shot Tool Use [2]. We selected the top models using the OpenLLM Leaderboard [7]. It should be noted that all the models were quantized and loaded in 4 bits using the BitsandBytes library.

- Q-bert/Optimus-7B: Fine-tuned On mistralai/Mistral-7B-v0.1 with meta-math/MetaMathQA
- fblgit/una-cybertron-7b-v2-bf16: Trained on SFT, DPO and UNA (Unified Neural Alignment) on Open-Orca/SlimOrca-Dedup, allenai/ultrafeedback_binarized_cleaned and fblgit/tree-of-knowledge
- HuggingFaceH4/zephyr-7b-beta : A fine-tuned version of mistralai/Mistral-7B-v0.1 that was trained on on a mix of publicly available, synthetic datasets such as HuggingFaceH4/ultrachat_200k using Direct Preference Optimization (DPO).
- mistralai/Mistral-7B-v0.1: A 7 billion parameters pretrained Large Language Model(LLM).
- Intel/neural-chat-7b-v3-2: fine-tuned model based on mistralai/Mistral-7B-v0.1 on the open source dataset Open-Orca/SlimOrca.
- NurtureAI/una-cybertron-11b-v1-fp16: It is a 7B MistralAI based model, this model trained on SFT, DPO and Unified Neural Alignment(UNA).
- ajibawa-2023/SlimOrca-13B: It is trained on 517981 set of conversations and is very good in various types of general purpose content generation such as Q&A.
- 01-ai/Yi-6B-200k: The first open source model with 200k context length capabilities..

- mlabonne/NeuralHermes-2.5-Mistral-7B: NeuralHermes is an teknium/OpenHermes-2.5-Mistral-7B model that has been further fine-tuned with Direct Preference Optimization (DPO) using the mlabonne/chatml_dpo_)pairs dataset.
- Q-bert/MetaMath-Cybertron-Starling: it is a merge of Q-bert/MetaMath-Cybertron and berkeley-nest/Starling-LM-7B-alpha using slerp merge.
- berkeley-nest/Starling-LM-7B-alpha: This model trained by Reinforcement Learning from AI Feedback (RLAIF).

4.8.1 Simple Prompting

We used these open-sourced models and worked on different methods. Our first experiment was simple prompting, in which we prompted the LLM with all the instructions and tool data provided in the problem statement and gave 3 hard-coded examples that were provided in the problem statement. That is, we used 3-shot prompting. We also tried 5-shot prompting but providing 5 examples increases the prompt length nearly upto the context length limit, increases the cost and due to the usage of hard-coded examples, the model does not generalise too well to new tools.

4.8.2 Retrieval Augmented Generation

Then we used RAG (Retrieval Augmented Generation), aiming to provide the top 3 query-answer pairs whose queries have the most similarity score with the user query to the LLM. To evaluate these open-source LLMs we used GPT-3.5 to generate some examples (11 examples) and evaluated the RAG-enhanced pipeline on these examples. Zephyr and Mistral-Instruct proved to be the best in this round.

4.8.3 Benchmarking Dataset

To further evaluate the abilities of these open-source models, we used Anthropic's Claude to generate query-answer pairs, with the aim of using these for evaluation. This dataset was significantly more complex, having queries that made use of list slicing, Python lambda functions (expressed as a string), and had a larger set of tools.

The Open-source LLMs did not perform very well when benchmarked on this dataset. Results of the evaluation are provided in the above table 4.We aimed to improve our results by changing the prompt and using standard prompting techniques, and increasing the diversity of the example pool from which the retrieval model selects examples, as more diverse examples would lead to wide range of contextual understanding.

4.8.4 Fine-tuning

To further improve performance, we also fine-tuned the best performing model UNA Cybertron 7B on a custom dataset which consisted of 21 queries which were equally complex and similar to those generated by the Claude model mentioned above. The dataset was a question and answer pair of queries and their required JSON output. We used the SFT(Supervised fine-tuning) Trainer for this purpose of fine-tuning using the QLoRA (Quantized Low Rank Adapters) approach.

- **Dataset:** The model was fine-tuned on 21 queries generated by the Anthropic's Claude model. The dataset used for fine-tuning were question and answer pairs of queries and their corresponding answer. The queries were generated on the common errors that the open source LLMs were committing and we aimed to reduce those errors using fine-tuning. We ensured that the queries generated were equally complex and captured the most probable mistakes committed by the open source models to get the best result from the fine-tuning.
- **Trainer:** We used the SFT trainer to fine-tune the base model. SFT trainer is essentially a wrapper around the transformer module which enabled us to effectively train our base model on our custom dataset. We fine-tuned our base model for 10 steps using the Adam optimizer, with the parameters set as follows: learning rate was set to 5e-4, and weight decay was set to 0.5.
- **QLoRA:** QLoRA (Quantized Low Rank Adapters) is an efficient, low memory usage method which we used for fine-tuning our model. This enabled us to selectively train the weights of the base model. This method was very time and memory efficient which helped us to fine-tune the model on T4 GPU on Google Colaboratory in nominal time relative to fine-tuning the entire model weights.

Model	Simple Prompt + RAG
Optimus 7B	25%
UNA Cybertron 7B	50%
Zephyr 7B	35%
Mistral 7B	10%
Neural Chat 7B	30%
Neural Hermes 7B	35%
Starling 7B	40%
Cybertron Starling Merge 7B	35%
Yi 6B	0%
Orca Slim 13B	20%

Table 4: Results of RAG on open-source models

- **Merging the Adapter weights:** The QLoRA essentially provides us with adapters consisting the selectively trained weight values after fine-tuning which need to be merged with the base model to obtain the fine-tuned model. This was achieved by using the PEFT (Parameter-Efficient Fine-Tuning) module.
- **Evaluation:** The fine-tuned model was evaluated on the original Claude generated queries which were used to evaluate the base model using simple prompting along with RAG implementation.

4.9 Bonus Section: Non-Standard Composition

As stated in the problem statement, it is very possible that the query might require some mathematical or logical operations to be performed in addition to the tool calls. To solve this problem we propose two solutions:

4.9.1 Lambda Tools

We introduce a special Lambda tool, that denotes a Python lambda function to be executed. We provide a format for the tool below.

```
  {
  "tool_description": "Given the outputs from previous tools, process relevant
      outputs, combining them using mathematical operations, iterations, conditional
      logic etc and returns output matching the request",
  "tool_name": "lambda",
  "argument_list": [
     {
        "argument_name": "expression",
        "argument_description": "Operation to be performed",
        "argument_type": "lambda statements",
        "example": "['lambda $$PREV[3], $$PREV[5] : $$PREV[3] + $$PREV[5]','lambda
            $$PREV[0]: len($$PREV[0])']"
     }
  ]
  "return_type": "Any"
}
```

When this tool is called, a lambda function is created and evaluated upon the specified arguments (upon `$$PREV[3]` and `$$PREV[5]`). Standard methods like Python's `eval()` can be used for this, or a custom parser can be written with ease. The idea behind this is that a lambda function can only be used to perform a certain range of tasks, hence it limits the leeway the model is allowed to have (thus reducing hallucination) while allowing greater expressiveness.

The final pipeline utilises the lambda tools method to handle non-standard queries.

4.9.2 Python Function Writing

Another probable solution we came up with involved using GPT-4-turbo to write a Python function, acting as an editor of the output of the pipeline. This instance of GPT-4-turbo would be prompted to check if the output answer matched the query, otherwise, it would be obliged to write a Python function, which would then be represented as a tool call, and used during evaluation of the tool call sequence. The name for this Python function and the tool call in the json will be kept the same, and the argument names for the function will match the argument names of the tool call. Below is an example representing a tool call and its Python function, which provides the internal execution logic.

```python
def compare_and_set(current,expected,newValue):
   orig_val=current
   if (current==expected):
      current=newValue
   return orig_val
```

Above is a python function to implement the compare_and_swap operation.

```
   {"tool_description": "Performs the Compare and Swap operation",
   "tool_name": "compare_and_swap",
   "argument_list": [
      {
   "argument_name" : "current",
   "argument_description" : "Current value for compare and swap operation.",
   "argument_type": "boolean",
   "argument_example" : true
   },

   {
   "argument_name" : "expected",
   "argument_description" : "Value to compare against for swap operation.",
   "argument_type": "boolean",
   "argument_example" : false
   },
   {
   "argument_name" : "newValue",
   "argument_description" : "New value to set after swap.",
   "argument_type": "boolean",
   "argument_example" : false
   }
   ],
   "return_type": "boolean"
   }
```

Above is the definition of the tool in the standard json format provided.

```
   {
   "tool_name": "compare_and_swap",
   "arguments": [
      {
   "argument_name" : "current",
   "argument_value" : "$$PREV[0]"
   },

   {
   "argument_name" : "expected",
   "argument_value" : "$$PREV[1]"
   },
   {
   "argument_name" : "newValue",
   "argument_value" : "$$PREV[2]"
   }
   ]
   }
```

The above demonstrates how a tool call would be made, as per the above. The Python function generation is ultimately more flexible, as it allows for a much wider range of operations, including recursion.

We were inspired to come up with these solutions due to the rapidly increasing power of LLMs, particularly in the field of code-generation. GPT-4-turbo proves to be very good at writing short snippets of code, and thus both approaches are validated.

5 Challenges

5.1 Errors and Common mistakes in Generation

During our experimentation with prompting of all kinds and with retrieval augmented generation, we found several errors that the LLMs were prone to.

5.1.1 String to JSON decode errors

Sometimes, the LLM will return an output with all double quotes inside the json being single quotes instead. At other times, boolean values like `true` and `false` inside the json are replaced by their Python equivalents `True` and `False`, leading to errors.

5.1.2 Incorrectly Handled Type (Normal Arguments)

Here, normal arguments refer to the arguments that are passed along in the query, rather than generated by the tool calls and referenced as `$$PREV[i]`. Several times, the LLM replaces what should be a boolean value (`true` in a json, for instance) with a string (”true” or ”True” here). At other times, arguments meant to be passed as a list are passed as strings, and at others, the list is represented as a string (for example, a list with strings ”A” and ”B” may be represented as `'["A","B"]'` instead of `["A","B"]`.

5.1.3 Incorrectly Handled Type (**`$$PREV`** Type)

Upon close examination of the examples provided in the problem statement, we observe that the return type of arguments passed by `"$$PREV[i]"` is also considered. For example, if the *i*th tool call returns a string and some tool is referencing this output value, then `"$$PREV[i]"` may have to be passed to this tool as `["$$PREV[i]"]`, if the argument type is a list. The reverse may also happen, though in this case there is ambiguity in which element of the list to pass.

5.1.4 Incorrect Function Calls

We observe that at times, the LLM attempts to use *tools* that take no arguments at all, as an input argument to another tool. In particular, the format the LLM uses these functions is `$$function_name`, where `function_name` is the `tool_name` attribute of the tool in question. This is quite prevalent across most models, including GPT-3.5, GPT-4-turbo, Claude, and all tested open-source models.

5.2 Solutions

To solve the above problems, we developed a post-processing function that uses heuristics gleaned from extensive comparative observation of the LLM outputs and ideal answers on several generated queries. We elucidate what our function does to solve each problem below.

5.2.1 String to JSON decode

Firstly, we seek to minimize JSON decode errors by prompting the model appropriately to generate output strings delimiting the json in question with ` ```json ` and ` ``` `. In case the LLM does not return the json delimited with this, we search for the first instance of ’[’ from the left and the first instance of ’]’ from the right, and try to decode this section of the text as a JSON. We also handle the single quotes problem by replacing all single quotes inside the string with double quotes, if the JSON does not decode with this problem. Further, we also look for instances of the mis-written boolean True, and convert it to true as is appropriate for the JSON.

5.2.2 Incorrectly Handled Type(Normal Arguments)

To solve this problem, we check the expected input type of the argument, and the type of the provided argument value, and try to force the types to be the same, either via conversion of a string or an integer to a boolean, wrapping a string within a list, unpacking a list's elements, etc. We also handle the ”list passed as string” error by removing '[' and ']' if they occur both at the start and at the end.

5.2.3 Incorrectly Handled Type (`$$PREV` Type)

To solve this problem, we iterate over the tool sequence and the arguments of each tool, checking for arguments provided as `$$PREV[i]`. Then, we check the return type of the *i*th tool called in the sequence, and compare this against the expected argument type. If they do not match, then we enforce the type similar to the above case.

5.2.4 Incorrect Function Calls

This problem proves to be slightly more complex than the others. First, we iterate over the tool call sequence, checking the argument values for `$$function_name` style argument values. Once one such argument value is found, we perform the following steps:

1. Insert the tool with `tool_name function_name` directly above the tool with argument `$$function_name`, and increment i.
2. For every tool at or after the *i*th tool in the sequence with an argument `$$PREV[j]`, where $j >= i-1$, set the argument to `$$PREV[j+1]`.
3. Set the `$$function_name` argument to `$$PREV[i-1]`.

This procedure handles such errors in a correct manner. An example of such a tool call corrected step-by-step is below. The corresponding query for this tool sequence is ”Find all work items that belong to me, and add them to the current sprint”.

```
[   {"tool_name":"get_sprint_id",
     "arguments":[]
    },
    {"tool_name":"works_list",
     "arguments":[
        "owned_by" : "$$WHO_AM_I"
     ]
    }
    {"tool_name":"add_work_items_to_sprint",
     "arguments":[
        "work_ids" : "$$PREV[1]",
        "sprint_id" : "$$PREV[0]"
     ]
    }
]
```

We notice that an erroneous argument value is provided at tool position *i*=1. We execute step 1 and add in the insert the appropriate tool. We then set *i* to *i+1* = 2.

```
[   {"tool_name":"get_sprint_id",
     "arguments":[]
    },
    {"tool_name":"who_am_i",
     "arguments":[]
    },
    {"tool_name":"works_list",
     "arguments":[
        "owned_by" : "$$WHO_AM_I"
     ]
    }
    {"tool_name":"add_work_items_to_sprint",
     "arguments":[
        "work_ids" : "$$PREV[1]",
        "sprint_id" : "$$PREV[0]"
     ]
    }
]
```

Then, we add go over the tools starting from position *i* (so from position 2 here) in the modified list, and increment when appropriate, as per step 2. So, `$$PREV[1]` is set to `$$PREV[2]` while `$$PREV[0]` remains as is.

```
[   {"tool_name":"get_sprint_id",
     "arguments":[]
    },
    {"tool_name":"who_am_i",
     "arguments":[]
    },
    {"tool_name":"works_list",
     "arguments":[
        "owned_by" : "$$WHO_AM_I"
     ]
    }
    {"tool_name":"add_work_items_to_sprint",
     "arguments":[
        "work_ids" : "$$PREV[2]",
        "sprint_id" : "$$PREV[0]"
     ]
    }
]
```

Finally we set the erroneous argument as per step 3, to `$$PREV[i-1]` (so `$$PREV[1]` here).

```
[   {"tool_name":"get_sprint_id",
     "arguments":[]
    },
    {"tool_name":"who_am_i",
     "arguments":[]
    },
    {"tool_name":"works_list",
     "arguments":[
        "owned_by" : "$$PREV[1]"
     ]
    }
    {"tool_name":"add_work_items_to_sprint",
     "arguments":[
        "work_ids" : "$$PREV[2]",
        "sprint_id" : "$$PREV[0]"
     ]
    }
]
```

Thus, the provided algorithm corrects such errors. We choose to insert a new tool every time such an error is encountered, because we anticipate that such tools depend on some internal state which may be changed by commands in between, hence referring to a previous tool call of the same type may not provide a correct answer.

In addition, we strictly enforce the validity of the tool call sequence. In case some made-up tool is created by the LLM and it does not have any equivalent in the tool set, the entire answer is rejected and an empty list is returned.

6 Conclusion

After extensive testing on several open-source models on generated datasets, we find Cybertron-UNA seems to be the best open-source model for this task. However, it still falls massively short of the closed-source models, hence we opt to utilize the GPT family of models sourced via the OpenAI API.

To reduce hallucination and improve accuracy, the technique of grounded generation using the provided tool set is employed. To fetch this list of tools, a GPT-3.5-turbo instance is used. This instance is given a shortened description of the tools, and is told to select all the required tools, and this helps shorten the prompt for the Tool-Selector, GPT-4-turbo. This brings down the cost.

To augment the generation capabilities, every time a new tool is added a new tool into the tool database, new examples of tool call sequences that utilise this new tool are generated. These are provided with the tool list to make it a few-shot prompt and to increase knowledge of tool usage. Upon deletion of a tool, the associated examples are also removed from the example database.

To strictly enforce type and to correct minor mistakes in generation, a post-processing script is utilised. The final pipeline, consisting of a GPT-3.5-turbo based Tool Retriever block, a GPT-4-turbo based Tool-Selector block, and an on-the-fly Example Generation Block performs admirably on several different kinds of queries. Coupled with robust type-checking and post-processing of LLM outputs which reduce error rates due to hallucination, the pipeline proves to be even more performant and reliable.

References

[1] Kexun Zhang, Hongqiao Chen, Lei Li, and William Wang. Syntax error-free and generalizable tool use for llms via finite-state decoding, 2023.

[2] Nexusflow.ai team. Nexusraven: Surpassing the state-of-the-art in open-source function calling llms, 2023. URL `http://nexusflow.ai/blog`.

[3] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.

[4] Nicholas Farn and Richard Shin. Tooltalk: Evaluating tool-usage in a conversational setting, 2023.

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

[6] Khaled Ardah, Sepideh Gherekhloo, André L. F. de Almeida, and Martin Haardt. Trice: A channel estimation framework for ris-aided millimeter-wave mimo systems, 2021.

[7] Edward Beeching, Clémentine Fourrier, Nathan Habib, Sheon Han, Nathan Lambert, Nazneen Rajani, Omar Sanseviero, Lewis Tunstall, and Thomas Wolf. Open llm leaderboard. `https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard`, 2023.

7 Appendix

7.1 Query Answering Prompts

7.1.1 RAG Prompt

System: The following is a friendly conversation between a human and an AI. The AI is professional and parses user input to several tasks. If the AI does not know the answer to a question, it truthfully says it does not know. The AI will be provided with a set of tools their descriptions and the argument in them. Here is the list of tools:

⟨TOOLS⟩

Provide the answer in the exact format as given in the following examples.

⟨EXAMPLES⟩

User: Use the above tools to learn how to use the tool on any query. Analyse how to parse the query and extract the correct information and place in the argument name and value. Use all the required tools and arguments in correct order of its calling based on the query and your learning from all the examples. Do not assume any value, you can take the value from query or the previous called tool as shown in the examples. Also focus on the allowed values argument present in tool definition. Now its your task to respond to the user queries in the same format as that in the above examples which is json. Query: ⟨QUERY⟩

System: Generate the answer in a json format only. Enclose the strings in double quotes

7.1.2 Zero Shot Prompt

System You are a intelligent AI agent specialized in giving the tool responses given a dictionary of tools. Here is the dictionary of tools:

⟨TOOLS⟩

User

Now its your task to respond to the user queries in the format given below

FORMAT:

```
[
{"tool_name": "...",
"arguments": [
{"argument_name": "...",
"argument_value": ... (depending on the argument_type)}, ...]
},
...]
```

To reference the value of the ith tool in the chain, use $$PREV[i] as argument value. i = 0, 1, .. j-1; j = current tool's index in the array If the query could not be answered with the given set of tools, output an empty list instead. Output in the JSON format. Query: ⟨Query⟩

7.1.3 Example Generation Prompts

System:

You are an intelligent AI Agent specialized in modifying the old data and generating the new relevant data.

User:

Given a list of old tools : ⟨Tools⟩ Let us say that I modified the tool 'MODIFIED_TOOL_NAME' to be ⟨MODIFIED_TOOL⟩ Now your task is to modify the following examples where this tool was used according to its new definition keeping in mind the new schema of JSON mentioned above.

RELEVANT_EXAMPLES

7.1.4 Tool Retrieval Prompt

System:

You are an intelligent assistant. Please help the user below.

User: You are given the following set of tools:

TOOLS

Can you please figure out which tools the query ⟨QUERY⟩ will require to solve, out of these tools? Please return only the tool names inside []. If it does not need any tool, return an empty list

7.1.5 Bonus Section Prompts

Lambda Tools **System**: The following is a friendly conversation between a human and an AI. The AI is professional and parses user input to several tasks. If the AI does not know the answer to a question, it truthfully says it does not know. The AI will be provided with a set of tools their descriptions and the argument in them.
Here is the list of tools: + ⟨TOOL⟩ + Provide the answer in the exact format as given in the following examples.

⟨EXAMPLES⟩

User: Use the above tools to learn how to use the tool on any query. Analyse how to parse the query and extract the correct information and place in the argument name and value. Use all the required tools and arguments in correct order of its calling based on the query and your learning from all the examples. Do not assume any value, you can take the value from query or the previous called tool as shown in the examples. Also focus on the allowed values argument present in tool definition.After producing the list of tools, analyze the query and figure out whether it requires the combination of tool outputs via mathematical operations, iterations, conditional logic etc. or not. In case it does, use the lambda function to produce the required results. Examples of such queries are given below:

⟨QUERY⟩

Assistant:

⟨ANSWER⟩

User: Now its your task to respond to the user queries in the same format as that in the above examples which is json. Use the lambda function only when necessary.

⟨QUERY⟩

System: Generate the answer in a json format only. Enclose the strings in double quotes

Python Functions You specialize in writing python function that involve combination of tools via mathematical operations, iterations, conditional logic etc and their respective answers that are correct, simple, and concise. In this environment you are given few example query and respective answers.

⟨EXAMPLE⟩

Now generate the python function such that it amswers the above query completely by taking composition of available tool, and might need some additional logic around combining the outputs of those tools, like mathematical operations, iterations, conditional logic.