Inter IIT Tech Meet 12.0

DevRev's AI Agent 007: Tooling Up For Success

Endterm Report

Submitted by :
Team 80

Abstract

This documentation of the final evaluation presents an optimal pipeline for tool manipulation by LLMs and summarizes existing research work in this domain.

Contents

1. Introduction

Large Language Models have taken the world by storm in a variety of domains not just restricted to question answering and text generation. Researchers all over the world are constantly searching for new applications where they may prove to have natural talent. We now know however that LLMs struggle with some types of queries, particularly those that a) require some kind of reasoning or b) require new information that is out of the scope of the LLM.

To address this issue, one area of interest is leveraging external tools that can provide the model with additional capabilities, such as reasoning or fetching external data. In our context, we aim to develop a system that, given a conversational user query and a set of tools, identifies the subset of tools needed to answer the query, the arguments required for each tool, and the composition of these tools to obtain the answer. This report surveys the various methodologies and approaches that can be used to tackle this problem, and presents a solution for the problem statement based on the results of the survey, trying to be as accurate, extensible, and cost-effective as possible. Relevant research works and sources are cited throughout the document.

2. Implementation

We propose a multi-step Tool Manipulation pipeline based on a novel PIRO Prompting Technique, consisting of —

1. **Retrieval Step**: LLMs have restrictions on context length, and proprietary models often charge by the token; therefore, sending the complete list of tools and examples for each query is not prudent. We use an embedding model along with FAISS with the L2 metric for retrieving tools and examples relevant to the user query.

2. **Answer-Generation Step**: The LLM is now given the user query, retrieved tools, and examples using a prompting technique. We have selected GPT-3.5-Turbo combined with our PIRO prompting technique. We found this to be the most cost-effective solution for the accuracy achieved. Various other LLMs were tested, and their results are declared later in this study.

2.A. Retrieval

2.A.I. Parsing Queries

A significant drawback of nearest-neighbour searching is its computational cost, which can be minimized by employing approximate nearest-neighbor searching. Performing ANN search on high dimensional embeddings is not guaranteed to retrieve all relevant tools [5]. This problem is exacerbated when the query has too much information, for example `Summarize P0 tasks, add them to the current sprint, and delete the rest`. Directly embedding this query results in the tool retriever missing crucial tools required to answer it. Therefore, we introduced an intermediate step of parsing the query into clauses. We then retrieve tools relevant to each clause and send their union to the model. Because each clause (on average) corresponds to just one task, the tool retrieval is efficient, and we can do it with a lower distance threshold.

2.A.II. Generating Embedding

Models understand things not in words but in terms of vectorized representations called embeddings. We experimented with various embedding models to gauge their usefulness for our task. This is particularly important for retrieval, as bad embeddings cause suboptimal semantic search. We tested various embedding models on both vanilla FAISS indices equipped with the L2 norm and Langchain vectorstores with distance thresholding. `all-mpnet-v2` and `text-embedding-ada-002` came out on top consistently. We use the latter even though it is paid because the computational cost savings outweigh its price.

2.A.III. Searching Algorithm

Once the queries and sentences are converted to vector embedding, a robust semantic similarity-based searching algorithm is used to identify the correct context.

2.B. Answer Generation

Since we are not fine-tuning the model, our answer generation step involves the selection

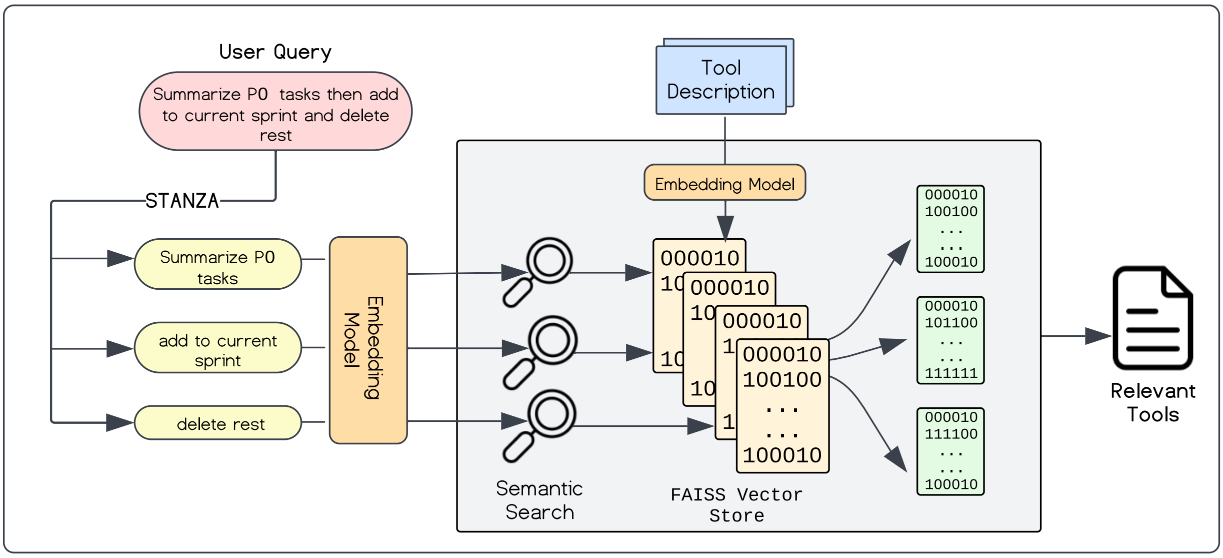

Figure 1: Retriever

Query	Base Retrieval	Retrieval on Clauses
"Add work items 'AB12-XYZ' and 'DEF-456' to sprint 'Sprint123' and retrieve the ID of the current sprint."	['get_sprint_id', 'add_work_items_to_sprint', 'works.list', 'works_list']	['add_work_items_to_sprint', 'works.list', 'works.export', 'get_sprint_id', 'who_am_i', 'works.get']
"Search for work items similar to 'WK-789' and add them to the current sprint, then summarize the list of added work items."	['add_work_items_to_sprint', 'works.list', 'get_similar_work_items', 'works_list']	['get_similar_work_items', 'works.list', 'works_list', 'works.delete', 'add_work_items_to_sprint', 'get_sprint_id', 'dev-users.list', 'works.export']

Table 1: Performing retrieval on sentence clauses fetches more relevant tools

of the right LLM, the correct settings for the LLM, and, finally, the right prompt to get the best answer possible.

2.B.I. LLM Selection

We have gone ahead with OpenAI's GPT 3.5 Turbo as the LLM for the task. This is a perfect compromise between cost and accuracy, as demonstrated in our experimentation. GPT 4 and GPT 4 Turbo are models that can perform slightly better at the task but also come with huge price-hikes, being the SOTA LLMs available to consumers today.

2.B.II. LLM Settings

The task does not require much creative ability, so we use very low temperatures with our LLM. We set the LLM to a temperature of 0.1 in the Planning phase and 0 for all steps after that, reflecting our expectation of creative ability that might be needed in the respective phases.

2.B.III. Prompting

We explored a variety of prompting techniques and proposed our novel PIRO prompting technique for the task. We present here the details of PIRO Prompting. It is an optimized prompt for `gpt-3.5-turbo` and can be adapted to `gpt-4` and `gpt-4-1106-preview` with minor changes. We have attached the prompts for GPT 4 models in Appendix A. PIRO is an acronym standing for **P**lanning, **I**mprovement, **R**eflection and **O**ptimization and describes the four prompts used to guide the model towards the correct answer.

2.B.III.1. Planning

In the planning phase, we give the set of tools received from the retriever to the model and ask it to generate a solution for the user query. The prompt used to provide this information is —

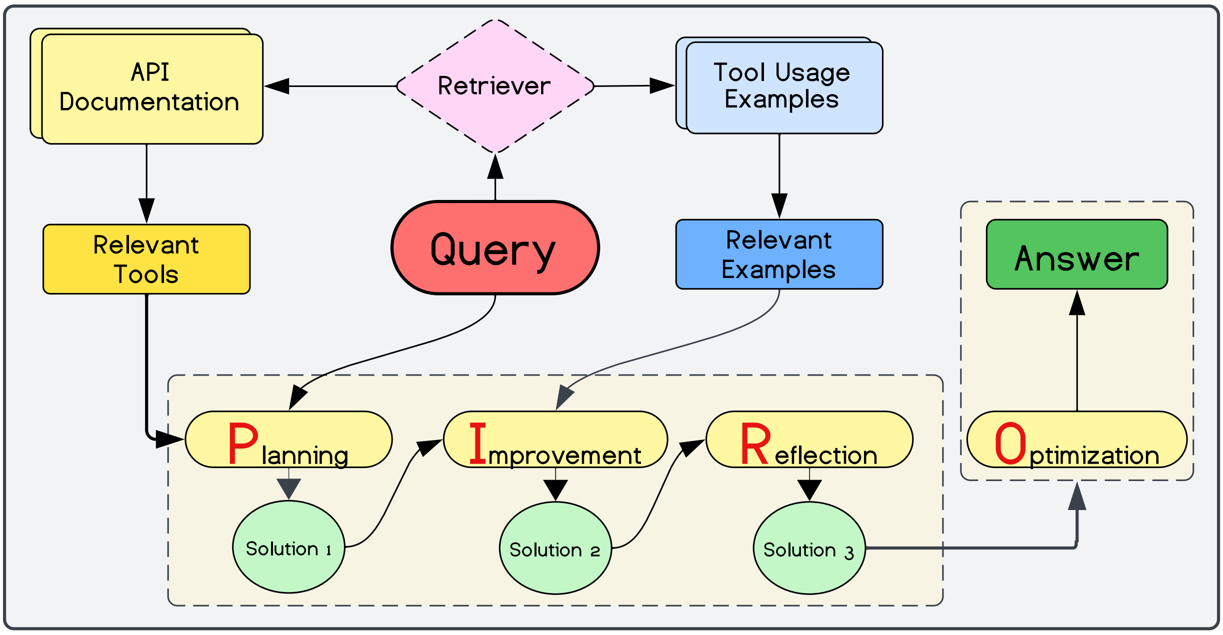

Figure 2: PIRO Prompting

```
Given a query and a list of
APIs with their arguments, find
whether the APIs can solve the

query or not. If not, return
empty list. Else, extract values
or variables from the query that
correspond to each argument
of the tools that needs to be
called. If some argument value
isn't known, use tools that can
retrieve those argument values.
Return a JSON file containing
the tools in the order they need
to be called. Identify cases
where an API argument depends
on the output of a previous API
call. Replace such arguments
with the notation $$PREV[i] where
'i' represents the order of the
previous API call in the chain.
Do NOT explain yourself, only
return the JSON.
Query: < Your Query >
Tools : < Relevant Tools >
Answer:
```

The answer obtained by the model is named Solution1 and passed to the following prompt.

2.B.III.2. Improvement

In the improvement phase, the model is given relevant examples fetched by the retrievers and asked to improve on Solution1. The prompt used for this task is —

```
Given a list of examples, each
containing a query and the
corresponding APIs to be called
to solve the query, find whether
the JSON present in current
solution solves the current
query. If yes, modify its format
to match with that of solutions
in examples and return the
modified JSON. If not, modify
the current JSON solution to
include necessary tools from
list of available tools. If no
solution is available, return
a blank list. Remember to use
double quotes instead of single
quotes in JSON output.
Examples : < Relevant Examples >
Current Query : < Your Query >
Current Solution : < Solution1 >
Answer:
```

The solution obtained at this stage is named Solution2 and sent to the next prompt.

2.B.III.3. Reflection

Researchers have found that asking an LLM to check its answer makes the LLM produce better answers. This is highlighted in works like Check Your Facts and Try Again[10] and Reflexion[11]. Inspired by this, when working with `gpt-3.5-turbo`, we give it the previous prompt again, asking it to improve the answer further now that it has access to all the information relevant to the task. This tends to improve the answer and capture nuances missed earlier on tricky queries. We found no need to do this with `gpt-4-turbo`, which usually makes no changes when fed this prompt. This prompt acts adversarially with open-LLMs, who are far less sure of their answer and get trapped between two answers.

```
Given a list of examples, each
containing a query and the
corresponding APIs to be called
to solve the query, find whether
the JSON present in current
solution solves the current
query. If yes, modify its format
to match with that of solutions
in examples and return the
modified JSON. If not, modify
the current JSON solution to
include necessary tools from
list of available tools. If no
solution is available, return
a blank list. Remember to use
double quotes instead of single
quotes in JSON output.
Examples : < Relevant Examples >
Current Query : < Your Query >
Current Solution : < Solution2 >
Answer:
```

The output received here is named Solution3 and sent ahead.

2.B.III.4. Optimization

This prompt focuses on reducing redundant API calls in the answer since this might be costly. The LLM is given Solution3 and the relevant tools from the retriever and is asked to remove any excess API calls. The prompt used to guide it here is —

```
Given a query, solution and
available APIs, optimize the
number of APIs calls in the
solution. Do not use API calls
unless absolutely necessary.
Look for redundancy and check
for mistakes in the combination
of API calls. Return current
solution if no changes are
necessary.
Current Query : < Your Query >
Current Solution : < Solution3 >
Available Tools : < Relevant
Tools >
Final Solution:
```

Thus, we get the final solution from the model.

2.C. Evaluation

We tested the results of our pipeline on both the set of tools provided in the PS and on a more extensive, augmented set. We generated queries and their solutions using an RLHF-based approach (see Appendix D). For ease of use and presentation, we created a web application that allows the end user to test our pipeline on any set of tools and examples.

2.C.I. Metrics Used

Deciding the right metric for a task is quite essential for making progress on it. Unfortunately, due to the relative infancy of this domain, there are no well-established metrics that can be used directly. Nevertheless, we use a subset of the metrics we have encountered during our Literature Review to check the performance of our models —

ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a textual similarity score that measures how much a machine-generated summary matches a human-written summary.

Exact Match

We use a function that constructs a dependency graph from the JSON and uses it to identify unbound arguments within API call sequences and to validate output accuracy against ground truth values by intelligently comparing the graphs in $O(N^2)$ time.

Human Evaluation

While the earlier two methods require strict matching in the format, occasionally the outputs were syntactically correct yet varied from the expected result. We conducted manual evaluations comparing these cases to the ground truth.

3. Experimentation

3.A. Retrieval

There are two main techniques to perform retrieval in the context of tool manipulation. The first one involves fine-tuning the model on many examples of tools, user queries and solutions so that the model learns information about all the tools in the API collection. Now, when faced with user queries at inference times, it can fetch the tools it needs in a zero-shot manner. However, this is costly - due to the computational resources required for fine-tuning, and additionally, and is not at all robust when it comes to adding or removing tools, since the model needs to be fine-tuned again from scratch every time you make a change. In order to optimize the retrieval step, we tried different model-threshold combinations using the FAISS retriever. We did not test competing approximate nearest neighbour search libraries because they are known to perform worse than FAISS on high dimensional data.

3.A.I. Clause Parsing

Humans understand complex queries by breaking them down into constituent clauses. While LLMs can do that, our pipeline does not use LLMs for tool retrieval because of context length restrictions and the high monetary cost associated. In order to improve our retriever's efficiency, we tried to parse the query into its clauses as a preprocessing step. We initially tried using Spacy, but the results were not encouraging. We ended up using `stanza` and some custom code to parse queries. This significantly improved the performance of lower-end models and `gpt-3.5-turbo` but did not considerably affect the performance of `gpt-4`. That is because `gpt-4` is capable of extracting the correct tool name from the examples provided and does not depend significantly on the tools provided.

3.A.II. Embedding Model

Langchain supports a huge number of embedding models out of the box. We compared seven models, including OpenAI's SOTA `text-embedding-ada-002`. It came out in the top 3 according to both metrics (top-k and threshold), missing 0.5 tools on average.

3.A.III. Threshold

There are two ways to use a FAISS index: by retrieving top-k values or by retrieving values that cross a certain threshold (retrieving all vectors that are less than a certain distance away). We tried both and found that going with FAISS' default threshold gives the best results. Using distance thresholds is also preferable because it helps reduce noise in the pipeline and puts a lower bound on the relevance of retrieved tools. The graphed results of our experimentation on retrievers is attached at Appendix E

3.B. Prompt Engineering

Large Language Models are great at understanding natural language, which makes them powerful when used for everyday tasks, however constructing a detailed prompt can significantly improve an LLM's capabilities for any task. A good prompt provides context, input and examples that could be useful for the LLM in constructing its answer. Thus, we explored a variety of prompting techniques, some of which are summarised here.

3.B.I. Chain-of-Thoughts

Chain-of-Thought (CoT) Prompting [13] is a general-purpose prompting technique where intermediate reasoning steps are provided to the model using few-shot prompting techniques. This encourages the model to generate a reasoning for each step it takes to solve the query. This method improves the LLM's ability to reason even when the steps provided are invalid or flawed, as the LLM still tries to generate a valid rationale to answer the query. CoT prompting is frequently adapted on a task-by-task basis to improve the results of an LLM on any task that needs reasoning. In the simplest case, it can be used as zero-shot by simply adding, "Let's think step by step" to the query.

CoT prompting cannot be directly used for the task as it makes the model much more chatty with its solution, making it harder to parse out the JSON solution from the model's response. However, the ideology behind it is quite powerful and was instrumental in inspiring PIRO Prompting.

3.B.II. ReAct

ReAct [14] is a framework to generate reasoning traces and task-specific actions from the LLM. The reasoning traces allow the model to induce, track and update action plans. The action step allows the model to interface with and gather information from external sources such as knowledge bases or environments. It can be extremely powerful when used in conjunction with techniques like CoT, however the structural constraint of interleaving reasoning and action also means that this technique is less flexible and can get caught up on a single incorrect step, unable to recover. This renders it unsuitable for our task, as we want to minimize the amount of human feedback that might be needed.

3.B.III. Reverse Chain

Based on the concept of in-context learning, Reverse Chain [16] is a prompting technique designed specifically for multi-API planning problems. As implied by its name, Reverse Chain operates in a retrograde manner. It first finds the final API needed to solve the query and works its way up.

1. API Selection Step - The LLM selects a proper API to solve the task of interest.
2. Argument Completion Step - The LLM completes the arguments of the selected API using keywords from the user query or with other APIs.

The authors claim that since the model tries to work from the end-up, the chance of hallucinations leading the model into diverging roadblocks is significantly reduced, which leads to better performance than ReAct or CoT. The prompts used by Reverse Chain for both steps are provided in Appendix C. However, we were unable to replicate their results, and the answers generated tended to be in inconsistent formats, which proved challenging to parse into JSON. Additionally, if the solution required multiple calls of the same API, it would result in the model getting confused. Furthermore, `gpt-3.5-turbo` was unable to work with this prompting technique, and we only achieved partial success even with `gpt-4-1106-preview`. Hence, we opted to start from scratch and engineer a new prompt for our specific use-case.

3.B.IV. Reflexion

Self-reflection induced by human feedback to an LLM has been shown to improve its ability to perform many different kinds of tasks. Reflexion [11] is a framework specifically for providing this feedback in natural language. The authors take this concept to the very limit to test how much additional performance can be squeezed out of the LLM using this technique. This encouraged us to check the model's ability to improve its own answer when presented with earlier versions of its answer.

3.B.V. Chain-of-Verification

During inference time, we would like our model to not rely on any human feedback. We also look at Chain-of-Verification [1], which suggests that LLMs can come up with good verification questions to fact-check a claim made by an LLM. In this study, we improve the LLM's base answer by presenting a verification question and answer along with its initial answer and asking it to revise its answer based on this information. This methodology has shown remarkable success on a variety of tasks. Hence, we adapt it for our use-case by presenting the LLM with its solution and additional information from a retriever and asking it to revise its answers if needed.

3.C. LLM Selection

The race to make the best LLM is well underway in this day and age, and every large corporation is constantly trying to make better and better LLMs. We thus explored both open and closed LLMs to check which would be more suited for our task. Closed-LLMs are, on average, larger and better models, hosted on an external server farm and hence needing little computation power at the point of inference but come at a cost that rapidly increases with the number of tokens - which is quite significant in

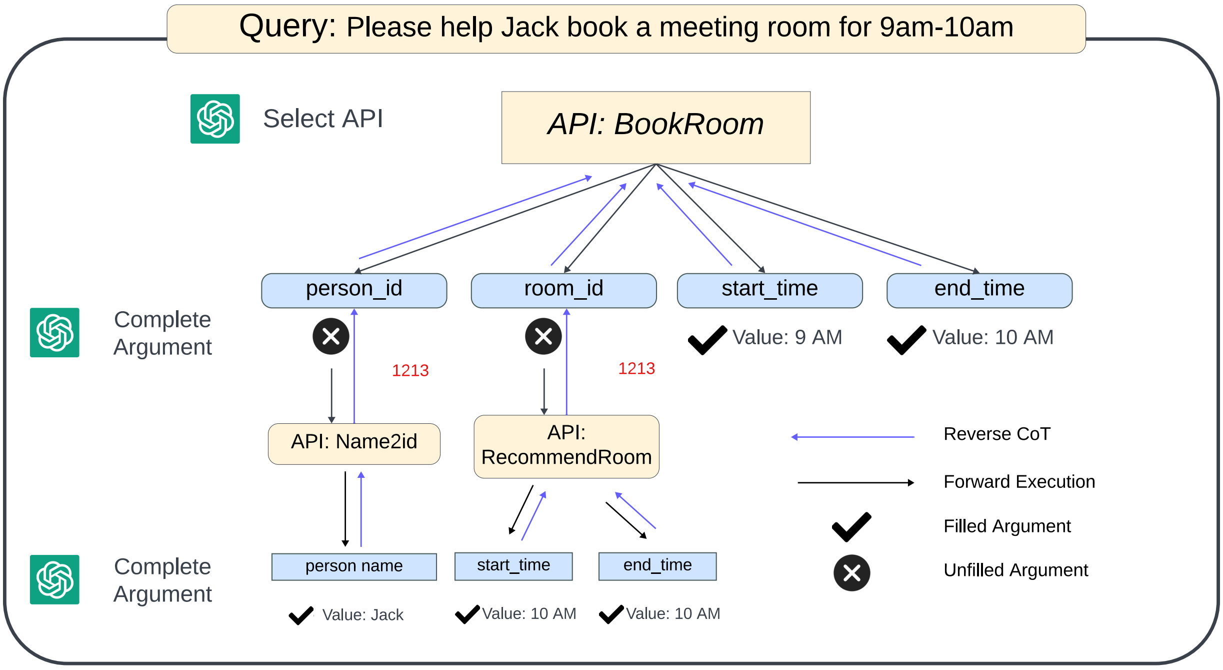

Figure 3: Reverse Chain

our use case depending on how well documented the API collection is. Open-LLMs are usually smaller and worse, and need to be hosted at the site of inference itself, but are entirely free to use otherwise.

3.C.I. Open LLMs

Our first point of investigation was to check whether any existing open LLMs could handle the task. We tested open-source models using the Ollama framework. It allows us to run a large number of open-source LLMs locally. While there are indeed a large number of models available in its library, we focused our efforts on the original LLaMA2-13B and `wizard-vicuna-uncensored-7b`, based on initial soft-testing across a diverse set of prompts.

3.C.I.1. Wizard-Vicuna

`wizard-vicuna-uncensored-7b` is a finetuned version of a LLaMA2 model trained on an uncensored dataset by Eric Hartford. The specific issues we faced while working with this model include —

1. The model has a tendency to hallucinate tool names, generating incorrect or even non-existent tools at times. During a test case, the solution generated included a **work_export** tool which did not exist in the API collection.

2. The model does not generate solutions in any specific format. This makes it impossible to parse the solution into JSON unless we manually handle each and every case. It also usually picks up a language of choice like JavaScript and writes a one line API call.

3.C.I.2. LLaMA2

LLaMA2 [12] is one of the most successful open-LLMs at the time of writing this report. However, the output generated still had several short-comings. Furthermore, none of the solutions were anywhere close to the ground-truth.

1. The model takes very long to answer the query. Average times ranged from around **77s** on a V100 GPU to over **4 minutes** on a T4 GPU.

2. Despite clear instructions to stick to the format, the output format deviates from the ideal format by a great extent in an inconsistent manner.

3. There were no mentions of the **argument description** in the format during the prompting, however the model generates argument descriptions trying to explain the query instead of solving it.

4. The responses generated were not reproducible, and the answer generated to the same query varied greatly between runs.

5. The model generated a coded solution instead of JSON formatted output.

3.C.I.3. Gorilla-OpenFunctions

In contrast to the others, Gorilla-OpenFunctions is a fine-tuned version of LLaMA specifically for function calling abilities hosted by UC Berkeley. It overcomes Gorilla's [9] limitation of a constrained API collection, as you can provide your own list of functions to this model. However our experimentation showed that the model was designed specifically for single API calls, and simply cannot handle queries where APIs need to be chained together. Even for single API calls, the model is extremely susceptible to hallucinating argument names and completely disregarding any types or formats needed by the functions. We also faced difficulties connecting to this model and felt that the server uptime was erratic, making it hard to conduct any thorough benchmarking for this model.

3.C.II. Finetuning LLaMA

We also tried fine-tuning LLaMA2 on our dataset of generated example queries described in DevRev_Large with the hypothesis that this might solve the problem of output formatting and improve results on the task, however this did not work out because of computational constraints and paucity of data which would resulted in poor generalization capabilities.

To summarize, open LLMs tend to provide very generic and vague answers, and even though they may at times seem to understand the logic that might be needed to solve the query, they are unable to apply that to the specific toolset provided to them to give a precise answer. They are also unable to obey rules regarding output format, and fail to understand any restrictions placed on them.

3.C.III. Closed LLMs

We primarily investigated OpenAI's suite of LLMs for the task including `gpt-3.5-turbo`, `gpt-4-1106-preview` and `gpt-4`. We felt this provided a balanced array of models across different price points and performance.
Other closed LLMs like Anthropic's Claude 2, and Google's Lambda are also options that could be considered, however access to Claude 2 is restricted, and Lambda performs worse than ChatGPT on average. Google's new Gemini-Ultra announced just a few days before the submission of this report appears promising, but is likely to be even more expensive than GPT4, and definitely will not be available for consumer use anytime soon, and is yet to be tested by any external reviewers as well.
Finetuning closed LLMs is much harder than open LLMs due to their large size and closed-source nature. Companies do tend to provide means to do this anyways like OpenAI, however the cost associated with it is huge, and you would need a lot of human data to actually see significant improvements with it. Based on OpenAI's pricing estimates, for every 15 human written examples, you can expect to pay around 2.4 USD. This was judged to be unsustainable in terms of both cost and manpower for the given time constraints, and hence we have avoided this option.

Model Name	Num. Tokens[1]	Cost/ Query[2]
GPT 3.5 Turbo	6500	$ 0.007
GPT 4	5000	$ 0.165
GPT 4 Turbo	5000	$ 0.055

Table 2: Pricing Estimates

4. Dataset Generation

4.A. Collection

In order to verify that our pipeline worked on a large number of tools not in the PS, we obtained the OpenAPI specifications of multiple REST APIs and brought them to a common format. We chose APIs of products operating in a similar space as DevRev to ensure that our solution works for tasks not mentioned in the problem statement but still integral to consumer management. Then, using

[1] On average, the 4 PIRO Prompts are approximately 6000 tokens long in total, and the model output is approximately 500 tokens long. GPT 4 Models only need to use the 3 PRO Prompts, which brings their input and output token count down to 75% as compared to GPT 3.5.
[2] Average Cost Per Query based on queries from the DevRev_small dataset

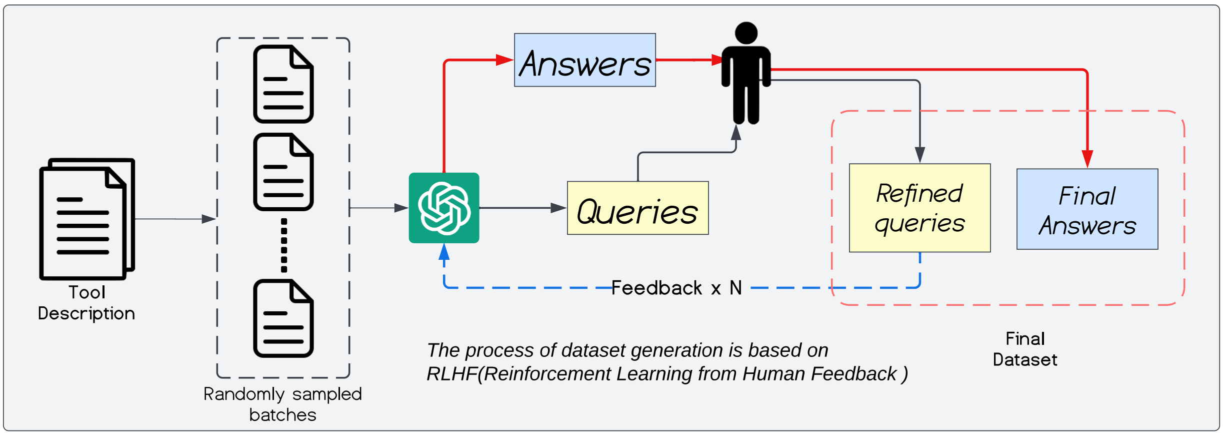

Figure 4: Dataset Generation

the techniques mentioned below, we generated a dataset to test our pipeline.

4.B. Generation

4.B.I. MultiAgent Generation

APIBank [7] describes a method to automatically generate queries and their solutions using multiple large language models. We tried to implement that using open-source models, but quickly realised that open models were not cut out for the task. Getting them to generate just the queries without answers was extremely difficult. The whole process required considerable human intervention, and it became clear that using proprietary LLMs would be easier. We realised that we could obtain better quality results by interacting with the model rather than letting multiple models interact with each other.

4.B.II. RLHF-Inspired Generation

Prompting LLMs to modulate their responses formed the bedrock of our dataset generation pipeline. By exploiting the short term memory of LLMs and providing feedback at each stage, we can steer LLMs into generating complex queries that use a diverse set of tools. By carefully choosing prompts, we were able to get GPT 3.5 to generate queries of varying difficulties. Similarly, using prompts, we were able to get the LLMs to outline a solution to these queries, which we later refined manually.

5. Results

We tested our pipeline on both the DevRev_Small and DevRev_Large dataset, with different prompting techniques, embedding models, and LLMs. Open source LLMs had a very disappointing run and were impossible to evaluate, so we have excluded their results from the table. Overall, GPT 3.5 Turbo with PIRO prompting and query parsing achieved the best performance. It was as good as the SOTA GPT 4 Turbo model with PRO prompting. A qualitative comparison of the models on prompts across queries of different difficulty levels are presented in Appendix B and the quantitative results are presented in Table 3.

GPT 3.5 Turbo also had the lowest average inference time, which can be attributed to model size and also to OpenAI's server load since GPT 4 and GPT 4 Turbo are SOTA models available as previews.

6. Conclusion

The report provides a comprehensive research into the task of tool manipulation by LLMs. It includes a comparative study of the performance of various pipeline components supported by experimental results. According to the results, we propose the usage of OpenAI's GPT 3.5 Turbo, in conjugation with the PIRO prompting technique. This offers the best results on our evaluation datasets while being extremely cost efficient at only 0.007 USD per query on average. The following section presents some future work that can significantly improve this task.

Model Name	D1 Score[1]	D2 Score[2]	AIT[3]	AIC[4]
GPT 3.5 Turbo + PRO	40%	16%	17s	$ 0.006
GPT 3.5 Turbo + PIRO	55%	18%	19s	$ 0.006
GPT 3.5 Turbo + PIRO w. Stanza	**65%**	24%	**19s**	$ 0.007
GPT 4 Turbo + PRO	**65%**	24%	39s	$ 0.04
GPT 4 Turbo + PRO w. Stanza	60%	**26%**	39s	$ 0.06
Human Performance	80%	46%	76s	N.A

Table 3: Pipeline Performance

7. Future Work

7.A. Confucius

If in the future, the model is required to work with more complicated tools, we could look at Confucius [2] to create better test sets based on the tools that the model struggles with. It describes a novel method of training models to master complicated tools. The paper describes an ingenious method of dataset modification to achieve this task called Iterative Tool Learning from Introspection Feedback. It starts by sampling a subset $\mathcal{T}$ from the available toolset and prompts a proprietary LM to generate instruction sets (queries and responses). It then measures the **perplexity** score of the response (inversely proportional to the likelihood of the LLM predicting that response), and instruction sets with a high perplexity score are used to generate more instruction sets to aid in the mastery of the Tool. The benchmarks indicate that a model trained on their dataset equals and even outperforms GPT-4 in most tests.

7.B. CRAFT

Most existing approaches to augment LLMs with tools are constrained by general-purpose APIs and lack the flexibility to tailor them to specific tasks. This paper [15] proposes a method to create toolsets specifically curated for specific tasks and equips LLMs with a component that retrieves tools from these sets. This method offers a plug & play approach to adapt LLMs to unseen domains without fine-tuning. They collect code solutions for each task by prompting GPT-4 to solve training examples and abstract those solutions to code snippets to enhance reusability. At inference time, the model retrieves snippets from the toolsets and executes them to generate the output. The authors experimented on three domains, i.e., vision-language, tabular processing & mathematical reasoning, and showed promising results on these three distinct tasks.

7.C. ToolChain*

As we increase the number of tools available at the LLMs' disposal, the available action space grows significantly, ToolChain*: Efficient Action Space Navigation in Large Language Models with A* Search [17] is a promising tree search-based planning approach that utilizes the A* search algorithm to navigate a tree representation of possible API calls. Unlike loop systems, tree search-based algorithms navigate all possibilities comprehensively and avoid any snowball effect from a single error step. ToolChain* is also more efficient than other tree search-based methods like DFS and MCTS since it uses both the cost of the current path and the estimated future cost of the plan completion to decide which paths are worth exploring. While we have not incorporated this at present since the size of the API collection is quite small, and a retriever is able to comfortably handle it, it could be looked into at a later stage if this requirement changes.

7.D. The Power of Scale for Parameter-Efficient Prompt Tuning

While prompting frozen language models is surprisingly effective at modulating their behavior and allows us to share a frozen model across a range of diverse tasks, it has

[1] D1 Score: Percentage of completely correct answers on the DevRev_Small dataset
[2] D2 Score: Percentage of completely correct answers on the DevRev_Large dataset
[3] AIT: Average Inference Time for a query in seconds. Note that this may fluctuate due to differences in server load, internet latency etc.
[4] AIC: Average Inference Cost for a query in USD.

several drawbacks. Its effectiveness is limited by the model's context size, resulting in subpar performance when compared with fine tuned models. This paper [6] proposes prompt tuning and shows that it becomes competitive with scale. In prompting, we seek to find a prompt P that for a given input X maximizes the likelihood of the model generating the correct output Y. Usually, the tokens of P come from the model's vocabulary and are parameterized by the model (frozen) parameters, requiring manual search to find the optimal prompt. Prompt tuning removes this restriction and endows prompt tokens with their own set of parameters θ_p. Optimal prompt can now be found by maximizing the likelihood of Y via back-propagation, applying gradient updates only to θ_p. The Prompt tuning involves the freezing of the core language model parameters, preventing direct modifications to the model's overall comprehension of language. Instead, it utilizes prompt representations to indirectly influence how the input is represented. This approach curtails the model's tendency to over-fit by memorizing specific words or unrelated correlations within a dataset.

8. Literature Review

We went through various publications in this research area and shortlisted a few that we thought were relevant to our task. The following section contains brief descriptions of the publications.

8.A. Reflexion: Language Agents with Verbal Reinforcement Learning

LLMs struggle to learn quickly and efficiently from trial and error as traditional reinforcement learning methods require extensive training samples and expensive model fine-tuning. Reflexion [11] is a framework to reinforce language agents through linguistic feedback instead of updating weights. The paper empirically shows that self-reflection is extremely useful to learn complex tasks over a few trials. Concretely, Reflexion agents verbally reflect on task feedback signals, then maintain their own reflective text in an episodic memory buffer to induce better decision-making in subsequent trials. In tasks that require reasoning, Reflexion beats strong benchmarks. For instance, in the HotPotQA benchmark - where the model is required to go through significant contexts to piece together an answer, it beats strong baselines by 20%. In our testing, while we could not drastically improve open LLMs to make them useful using this method, we saw a significant jump in the quality of answers produced.

8.B. LLaMA: Open and Efficient Foundation Language Models

Meta AI's research work titled LLaMA: Open and Efficient Foundation Language Models [12] outlines the structure of a Large Language Model and presents LLaMA, one of the most popular open LLMs. The paper focuses on training language models to achieve the best possible performance at various inference budgets. LLaMA models range from 7B to 65B parameters with competitive performance compared to top-flight LLMs. One more highlight of LLaMA is that it is only trained on publicly available data. The open source community has massively adopted LLaMA models and is conducive to further improvements on a budget.

8.C. The False Promise of Imitating Proprietary LLMs

It is no secret that in the present day, closed LLMs are considerably better than any available open LLMs. To help close this gap, one could look into training these open LLMs based on the outputs of a superior closed LLM. This imitation approach was thought of as a way to cheaply reach the performance of a proprietary LLM. However, the authors of this [3] work argue that while this method would help the trainee model replicate the style of the proprietary model, it would not learn anything else of significance. This dichotomy is evident when testing the results of the method on human testers who report great success, while targeted automated evaluation reveals the gaps in content. Imitation is thus a false promise, and if we start with a far better open LLM or have ludicrous amounts of imitation data, there is hope of catching up to a proprietary LLM. We will thus not focus our effort on closing the gap between open and closed LLMs for this task and try to keep our solution as LLM agnostic as possible to ensure that it can be deployed on either type of LLM based on the requirement.

8.D. Check Your Facts and Try Again

Although LLMs are exceptionally good in generating fluent, coherent and informative natural language texts, they tend to hallucinate when deployed to do mission-crictical tasks. Also, due to exponentially large sizes of models, they can never fully encode all the information needed for many applications. There have been many advances in equipping LLMs with external knowledges but mostly all of them need to fine tune the parameters which is computationally expensive. This paper [10] presents the LLM-Augmenter to improve LLMs with external knowledge and automated feedback using PnP modules. The LLM-Augmenter retrieves evidence from external knowledge, then queries a fixed LLM using a prompt that contains the consolidated evidence to generate a candidate response from the fixed LLM and verifies the response for possible hallucination, if such an instance is found, it generates a feedback which is used to revise the prompt for the fixed LLM, this is done iteratively until the optimal response in generated. However for this task we don't use any information from external sources and use the information only provided in the description of the tools in their documentation.

8.E. Chain-of-Verification Reduces Hallucination in Large Language Models

Hallucination, defined as generating plausible yet incorrect factual information, is one of the most significant unsolved issues in large language models. It is tough to fact-check an LLM without extensive domain knowledge on the subject, and this prevents the usage of LLMs in many mission-critical domains where such mistakes are not tolerable. Chain-of-Verification [1] suggests that LLMs can check on their hallucination tendencies by independently planning and answering fact-checking questions and using this information to improve on its answer.

8.F. RLHF: Understanding the Effects of RLHF on LLM Generalisation and Diversity

Every day, the LLMs become more and more complex and capable, but so do the tasks we want them to perform. Providing and evaluating the training data for such complex tasks is challenging. In such cases, humans may find evaluating or ranking model outputs easier and faster than providing full demonstrations. This paper [4] introduces a pipeline consisting of three stages. A Supervised delicate tuning stage, where the model is fine-tuned with desired demonstrations. Reward modeling, where the pre-trained model is fine-tuned to predict human preferences between pairs of output for a given input, and lastly, reinforcement learning (RL), where the reward model is used to fine-tune the model produced by the SFT stage using an on-policy RL algorithm. We use a similar approach to generate examples; however, instead of teaching the model to predict human responses, we manually evaluated and tuned a few of its responses to produce optimal outputs.

8.G. Prefix-Tuning: Optimizing Continuous Prompts for Generation

Fine-tuning can be prohibitively expensive for large language models, so a natural approach is freezing most of the pre-trained parameters and tuning only a smaller set. At the limit, LMs can be deployed using in-context learning without modifying any parameters through prompting, but this restricts the capabilities of the LM given the restrictive context length limits. This paper [8] proposes prefix tuning that prepends a sequence of *task specific* vectors to the input. The prefix can then be tuned using standard gradient descent. Consequently, we can store one large language model and a set of learned task-specific prefixes.

References

[1] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason Weston. Chain-of-verification reduces hallucination in large language models, 2023.

[2] Shen Gao, Zhengliang Shi, Minghang Zhu, Bowen Fang, Xin Xin, Pengjie Ren, Zhumin Chen, and Jun Ma. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum, 2023.

[3] Arnav Gudibande, Eric Wallace, Charlie Snell, Xinyang Geng, Hao Liu, Pieter Abbeel, Sergey Levine, and Dawn Song. The false promise of imitating proprietary llms, 2023.

[4] Robert Kirk, Ishita Mediratta, Christoforos Nalmpantis, Jelena Luketina, Eric Hambro, Edward Grefenstette, and Roberta Raileanu. Understanding the effects of rlhf on llm generalisation and diversity, 2023.

[5] Nikolaos Kouiroukidis and Georgios Evangelidis. The effects of dimensionality curse in high dimensional knn search. In *2011 15th Panhellenic Conference on Informatics*, 2011.

[6] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. pages 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[7] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms, 2023.

[8] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

[9] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.

[10] Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, and Jianfeng Gao. Check your facts and try again: Improving large language models with external knowledge and automated feedback, 2023.

[11] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[12] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[13] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[14] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

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

[16] Yinger Zhang, Hui Cai, Yicheng Chen, Rui Sun, and Jing Zheng. Reverse chain: A generic-rule for llms to master multi-api planning, 2023.

[17] Yuchen Zhuang, Xiang Chen, Tong Yu, Saayan Mitra, Victor Bursztyn, Ryan A. Rossi, Somdeb Sarkhel, and Chao Zhang. Toolchain*: Efficient action space navigation in large language models with a* search, 2023.

Appendix

A. Engineered Prompts for GPT 4

GPT4 only requires the PRO prompts as it provides optimal results without the additional Improvement prompt. GPT4 also needs an explicit instruction to not explain itself. Thus, the prompts needed for GPT4 are -

Planning

```
Given a query and a list of APIs with their arguments, find whether the
APIs can solve the query or not. If not, return empty list. Else, extract
values or variables from the query that correspond to each argument of
the tools that needs to be called. If some argument value isn't known,
use tools that can retrieve those argument values. Return a JSON file
containing the tools in the order they need to be called. Identify cases
where an API argument depends on the output of a previous API call.
Replace such arguments with the notation $$PREV[i] where 'i' represents
the order of the previous API call in the chain. Do NOT explain yourself,
only return the JSON.
Query: < Your Query >
Tools : < Relevant Tools >
Answer:
```

Reflection

```
Given a list of examples, each containing a query and the corresponding
APIs to be called to solve the query, find whether the JSON present in
current solution solves the current query. If yes, modify its format to
match with that of solutions in examples and return the modified JSON.
If not, modify the current JSON solution to include necessary tools from
list of available tools. If no solution is available, return a blank list.
Remember to use double quotes instead of single quotes in JSON output. Do
NOT explain yourself, only return the JSON.
Examples : < Relevant Examples >
Current Query : < Your Query >
Current Solution : < Solution1 >
Answer:
```

Optimization

```
Given a query, solution and available APIs, optimize the number of APIs
calls in the solution. Do not use API calls unless absolutely necessary.
Look for redundancy and check for mistakes in the combination of API
calls. Return current solution if no changes are necessary. Do NOT explain
yourself, only return the JSON.
Current Query : < Your Query >
Current Solution : < Solution3 >
Available Tools : < Relevant Tools >
Final Solution:
```

B. Qualitative Comparison between GPT 3.5 Turbo, GPT 4 Turbo and LLaMA

Query 1 - Easy

Add a new task with ID ‘NEW-001’ to the sprint named ‘Sprint456’.

Q1) Ground Truth

```
[
    {
        ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
        ‘‘arguments’’: [
            {
                ‘‘argument_name’’: ‘‘work_ids’’,
                ‘‘argument_value’’: [‘‘NEW-001’’]
            },
            {
                ‘‘argument_name’’: ‘‘sprint_id’’,
                ‘‘argument_value’’: ‘‘Sprint456’’
            }
        ]
    }
]
```

Q1) GPT 3.5 Turbo

```
[
  {
    ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
    ‘‘arguments’’: [
      { ‘‘argument_name’’: ‘‘work_ids’’, ‘‘argument_value’’: ‘‘NEW-001’’ },
      { ‘‘argument_name’’: ‘‘sprint_id’’, ‘‘argument_value’’: ‘‘Sprint456’’ }
    ]
  }
]
```

Q1) GPT 4 Turbo

```
[
    {
        ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
        ‘‘arguments’’: [
            {
                ‘‘argument_name’’: ‘‘work_ids’’,
                ‘‘argument_value’’: [‘‘NEW-001’’]
            },
            {
                ‘‘argument_name’’: ‘‘sprint_id’’,
                ‘‘argument_value’’: ‘‘Sprint456’’
            }
        ]
    }
]
```

Q1) LLaMA2

```
Thank you for providing the query, available APIs, and the current solution.
Based on the information provided, I can suggest a more optimized solution that reduces the
number of API calls:

Current Solution:
[
```

```
    {
        ‘tool_name’: ‘get_sprint_id’,
        ‘arguments’: []
    },
    {
        ‘tool_name’:‘’add_work_items_to_sprint’,
        ‘arguments’: [
            {
                ‘argument_name’: ‘sprint_id’,
                ‘argument_value’: ‘$$PREV[0]’
            },
            {
                ‘argument_name’: ‘user_id’,
                ‘argument_value’: ‘$$PREV[1]’
            }
        ]
    },
    {
        ‘tool_name’: ‘who_am_i’,
        ‘arguments’: []
    }
]
Optimized Solution:
[
    {
        ‘tool_name’: ‘get_sprint_id’,
        ‘arguments’: []
    },
    {
        ‘tool_name’: ‘add_work_items_to_sprint’,
        ‘arguments’: [
            {
                ‘argument_name’: ‘sprint_id’,
                ‘argument_value’: ‘$$PREV[0]’
            },
            {
                ‘argument_name’: ‘user_id’,
                ‘argument_value’: ‘$$PREV[1]’
            }
        ]
    },
    {
        ‘tool_name’: ‘who_am_i’,
        ‘arguments’: [
            {
                ‘argument_name’: ‘user_id’,
                ‘argument_value’: ‘$$PREV[1]’
            }
        ]
    }
]

Explanation:

* Remove the unnecessary ‘who_am_i’ API call in the current solution. Since we are already
getting the ID of the current user from the previous API call, we don’t need to make an
additional API call for this information.
* Modify the ‘add_work_items_to_sprint’ API call to pass only the required arguments.
Instead of passing the ID of the sprint and the ID of the current user as separate arguments,
pass them as a single argument ‘sprint_id’ along with the ‘user_id’. This reduces the
number of API calls and makes the solution more efficient.

The optimized solution reduces the number of API calls from 3 to 2, which can improve
performance and reduce latency in the system.
```

Query 2 - Medium

Find work items in ‘CAPL-123’ owned by ‘DEVU-123’. Get the current sprint ID.

Q2) Ground Truth

```
[
    {
        ``tool_name'': ``add_work_items_to_sprint'',
        ``arguments'': [
            {
                ``argument_name'': ``work_ids'',
                ``argument_value'': [``NEW-001'']
            },
            {
                ``argument_name'': ``sprint_id'',
                ``argument_value'': ``Sprint456''
            }
        ]
    }
]
```

Q2) GPT 3.5 Turbo

```
[
  {
    ``tool_name'': ``works_list'',
    ``arguments'': [
      {
        ``argument_name'': ``applies_to_part'',
        ``argument_value'': [``CAPL-123'']
      },
      {
        ``argument_name'': ``owned_by'',
        ``argument_value'': [``DEVU-123'']
      }
    ]
  },
  {
    ``tool_name'': ``get_sprint_id'',
    ``arguments'': []
  }
]
```

Q2) GPT 4 Turbo

```
[
  {
    ``tool_name'': ``works_list'',
    ``arguments'': [
      {
        ``argument_name'': ``applies_to_part'',
        ``argument_value'': [``CAPL-123'']
      },
      {
        ``argument_name'': ``owned_by'',
        ``argument_value'': [``DEVU-123'']
      }
    ]
  },
  {
    ``tool_name'': ``get_sprint_id'',
    ``arguments'': []
  }
]
```

Q2) LLaMA2

```
Thank you for providing the current query, solution, and available APIs. Based on the
information provided, I can suggest a modified solution that optimizes the number of API
calls.
Currently, the solution consists of four tools: ‘get_sprint_id’, ‘add_work_items_to_sprint’,
‘works_list’, and ‘get_similar_work_items’. Let’s analyze each tool and suggest improvements:

1. ‘get_sprint_id’: This tool is not used in the current query, so it can be removed.
2. ‘add_work_items_to_sprint’: This tool is used to add work items to a sprint. Since the
query only includes a single work item (‘work_id’), we can eliminate the need for this
tool and use the ‘works_list’ tool instead.
3. ‘works_list’: This tool is used to retrieve a list of work items that match the request
criteria. However, since the query includes multiple filters (e.g., ‘applies_to_part’,
‘created_by’, ‘issue.priority’, ‘issue.rev_orgs’, ‘owned_by’, ‘stage.name’,
‘ticket.needs_response’, ‘ticket.rev_org’, ‘ticket.severity’, and ‘type’), we need to make
multiple API calls to retrieve the required data.

To optimize the number of API calls, we can use the ‘combine_filters’ tool to combine
the filters into a single API call. This tool allows us to create a filter chain by
combining multiple filters using logical operators (‘AND’, ‘OR’, and ‘NOT’). By using this
tool, we can reduce the number of API calls required to retrieve the data.

Here’s an example of how we can modify the current solution to use the ‘combine_filters’
tool:
{
  "query": {
    "work_id": "WMS-123",
    "filters": [
      {
        "name": "applies_to_part",
        "values": ["FEAT-123"]
      },
      {
        "name": "created_by",
        "values": ["DEVU-123"]
      },
      {
        "name": "issue.priority",
        "values": ["p0"]
      },
      {
        "name": "issue.rev_orgs",
        "values": ["REV-123"]
      },
      {
        "name": "owned_by",
        "values": ["DEVU-123"]
      },
      {
        "name": "stage.name",
        "values": ["PROD-123"]
      },
      {
        "name": "ticket.needs_response",
        "values": ["true"]
      },
      {
        "name": "ticket.rev_org",
        "values": ["REV-123"]
      },
      {
        "name": "ticket.severity",
        "values": ["medium"]
      }
    ]
  },
  "tools": [
    {
```

```
      "name": "combine_filters",
      "description": "Combines multiple filters into a single API call"
    }
  ]
}

By using the ‘combine_filters’ tool, we can reduce the number of API calls required to
retrieve the data. The final solution will consist of a single API call to retrieve all
the required data.

Therefore, the modified solution will be:

{
  "query": {
    "work_id": "WMS-123",
    "filters": [
      {
        "name": "combine_filters",
        "values": [
            "applies_to_part=FEAT-123,
            created_by=DEVU-123,
            issue.priority=p0,
            issue.rev_orgs=REV-123,
            owned_by=DEVU-123,
            stage.name=PROD-123,
            ticket.needs_response=true,
            ticket.rev_org=REV-123,
            ticket.severity=medium"]
      }
    ]
  },
  "tools": [
    {
      "name": "combine_filters",
      "description": "Combines multiple filters into a single API call"
    }
  ]
}

By using the ‘combine_filters’ tool, we can reduce the number of API calls required to
retrieve the data while still providing the necessary information to generate the report.
```

Query 3 - Hard

Search for work items similar to ‘WK-789’ and add them to the current sprint, then summarize the list of added work items.

Q3) Ground Truth

```
[
    {
        ‘‘tool_name’’: ‘‘get_similar_work_items’’,
        ‘‘arguments’’: [
            {
                ‘‘argument_name’’: ‘‘work_id’’,
                ‘‘argument_value’’: ‘‘WK-789’’
            }
        ]
    },
    {   ‘‘tool_name’’: ‘‘get_sprint_id’’,
        ‘‘arguments’’: [ ]
    },
    {
        ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
        ‘‘arguments’’: [
```

```
            {
                ‘‘argument_name’’: ‘‘work_ids’’,
                ‘‘argument_value’’: [‘‘$$PREV[0]’’]
            },
            {
                ‘‘argument_name’’: ‘‘sprint_id’’,
                ‘‘argument_value’’: [‘‘$$PREV[1]’’]
            }
        ]
    },
    {
        ‘‘tool_name’’: ‘‘summarize_objects’’,
        ‘‘arguments’’: [
            {
                ‘‘argument_name’’: ‘‘objects’’,
                ‘‘argument_value’’: [‘‘$$PREV[2]’’]
            }
        ]
    }
  ]
```

Q3) GPT 3.5 Turbo

```
[
  {
    ‘‘tool_name’’: ‘‘get_similar_work_items’’,
    ‘‘arguments’’: [
    { ‘‘argument_name’’: ‘‘work_id’’,
    ‘‘argument_value’’: ‘‘WK-789’’ }
    ]
  },
  {
    ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
    ‘‘arguments’’: [
      { ‘‘argument_name’’: ‘‘work_ids’’,
      ‘‘argument_value’’: ‘‘$$PREV[0]’’ },
      { ‘‘argument_name’’: ‘‘sprint_id’’, ‘‘argument_value’’: ‘‘$$PREV[1]’’ }
    ]
  },
  {
    ‘‘tool_name’’: ‘‘works_list’’,
    ‘‘arguments’’: [
      { ‘‘argument_name’’: ‘‘issue.priority’’, ‘‘argument_value’’: ‘‘p0’’ },
      { ‘‘argument_name’’: ‘‘owned_by’’,
      ‘‘argument_value’’: ‘‘$$PREV[0]’’ }
    ]
  },
  {
    ‘‘tool_name’’: ‘‘summarize_objects’’,
    ‘‘arguments’’: [
    { ‘‘argument_name’’: ‘‘objects’’,
    ‘‘argument_value’’: ‘‘$$PREV[2]’’ }]
  }
]
```

Q3) GPT 4 Turbo

```
[
  {
    ‘‘tool_name’’: ‘‘get_similar_work_items’’,
    ‘‘arguments’’: [
      {
        ‘‘argument_name’’: ‘‘work_id’’,
        ‘‘argument_value’’: ‘‘WK-789’’
      }
    ]
```

```
  },
  {
    ‘‘tool_name’’: ‘‘get_sprint_id’’,
    ‘‘arguments’’: []
  },
  {
    ‘‘tool_name’’: ‘‘add_work_items_to_sprint’’,
    ‘‘arguments’’: [
      {
        ‘‘argument_name’’: ‘‘work_ids’’,
        ‘‘argument_value’’: ‘‘$$PREV[0]’’
      },
      {
        ‘‘argument_name’’: ‘‘sprint_id’’,
        ‘‘argument_value’’: ‘‘$$PREV[1]’’
      }
    ]
  },
  {
    ‘‘tool_name’’: ‘‘summarize_objects’’,
    ‘‘arguments’’: [
      {
        ‘‘argument_name’’: ‘‘objects’’,
        ‘‘argument_value’’: ‘‘$$PREV[0]’’
      }
    ]
  }
]
```

Q3) LLaMA2

```
Thank you for providing the modified JSON file and the available APIs. Based on the current
query and the available APIs, I can suggest a solution to optimize the number of API calls:

Instead of calling ‘get_similar_work_items’ twice, once for each ID in ‘WK-789’, we can call
it once with a range query to retrieve all the similar work items in one API call. Here’s
an updated solution:

Current Query: Search for work items similar to ‘WK-789’ and add them to the current
sprint, then summarize the list of added work items.

Current Solution: To solve the query, we need to call the following APIs in the specified
order:

1. ‘add_work_items_to_sprint’ - This API adds the work items with IDs ‘WK-789‘ to the
current sprint.
2. ‘get_similar_work_items’ - This API retrieves a list of similar work items for the given
work item ID, which will be used to add the work items to the sprint.
3. ‘summarize_added_work_items’ - This API summarizes the list of added work items.

Here’s the updated solution in the same format as before:

[
    {
        "name": "add_work_items_to_sprint",
        "description": "Adds work items with IDs in WK-789 to the current sprint",
        "arguments": [
            {
                "name": "",
                "description": "",
                "argument_type": "",
                "examples": []
            }
        ],
        "tool": "add_work_items_to_sprint"
    },
    {
```

```
        "name": "get_similar_work_items",
        "description": "Retrieves a list of similar work items for the given work item ID",
        "arguments": [
            {
                "name": "work_item_id",
                "description": "The ID of the work item for which you want to find similar
                                items",
                "argument_type": "string",
                "examples": ["WK-789"]
            }
        ],
        "tool": "get_similar_work_items"
    },
    {
        "name": "summarize_added_work_items",
        "description": "Summarizes the list of added work items",
        "arguments": [
            {
                "name": "",
                "description": "",
                "argument_type": "",
                "examples": []
            }
        ],
        "tool": "summarize_added_work_items"
    }
]

By calling 'get_similar_work_items' once with a range query to retrieve all the similar work
items for 'WK-789', we can reduce the number of API calls from 2 to 1.
```

C. Prompts Used in Reverse Chain

Prompt used in Reverse chain for API selection: `We have N APIs < your tools >. If someone is saying "< your query >". Which final API should we use for this instruction? Only return API code. Only return one word!`

Prompt for Argument Completion: `You are an Argument extractor you need to find whether you can extract the argument from the user input directly or you need an API to get the value. Output must be json format, key is the argument, and value is the value of argument or the API Name, return None if you cannot get value or API Name."`

D. Prompts Used for Dataset Generation

Read the given examples of tool usage :

```
    {
        "query": "Retrieve the current announcement banner configuration including required
        permissions for access, and then get details of all application roles in Jira.",
        "solutions": [
            {
                "tool": "announcement_banner",
                "arguments": [
                    {
                        "argument_name": "include_permissions",
                        "argument_value": "true"
                    }
                ]
            },
            {
                "tool": "application_roles",
                "arguments": [
```

```
                {
                    "argument_name": "applicationrole",
                    "argument_value": "Jira"
                }
            ]
        }
    ]
},
{
    "query": "Retrieve all filters, allowing to get, create, update,
    or delete filters, and then get details of all default sharing
    settings for new filters and dashboards for user with user_id qu_bit.",
    "solutions": [
        {
            "tool": "filters",
            "arguments": []
        },
        {
            "tool": "filter_sharing",
            "arguments": [
                {
                    "argument_name": "user",
                    "argument_value": "qu_bit"
                }
            ]
        }
    ]
}
```

Now use combinations of these tools to generate queries that are like real-life application commands used to do tasks. Include specific details like names or projects, companies and id etc. Make names very random. Generate queries like before using combinations of 2-3 tools however, make them like the earlier time, instead of making it cryptic or creative.
The correct format is supposed to be

```
    {
    "query": "Retrieve work items owned by 'DEVU-123' in 'FEAT-123' and 'PROD-123'.
    Get similar work items for 'ENH-123'.",
    "solutions": [
        {
            "tool": "works_list",
            "arguments": [
                {
                    "argument_name":"applies_to_part",
                     "argument_value": ["FEAT-123", "PROD-123"],
                     "argument_name":"owned_by",
                     "argument_value" : ["DEVU-123"]
                }
            ]
        },
        {
            "tool": "get_similar_work_items",
            "arguments": [
                {
                    "argument_name":"work_id",
                     "argument_value": "ENH-123"
                }
            ]
        }
    ]
}
```

Make sure all your responses adhere to this format.

E. Retriever Benchmarking Graphs

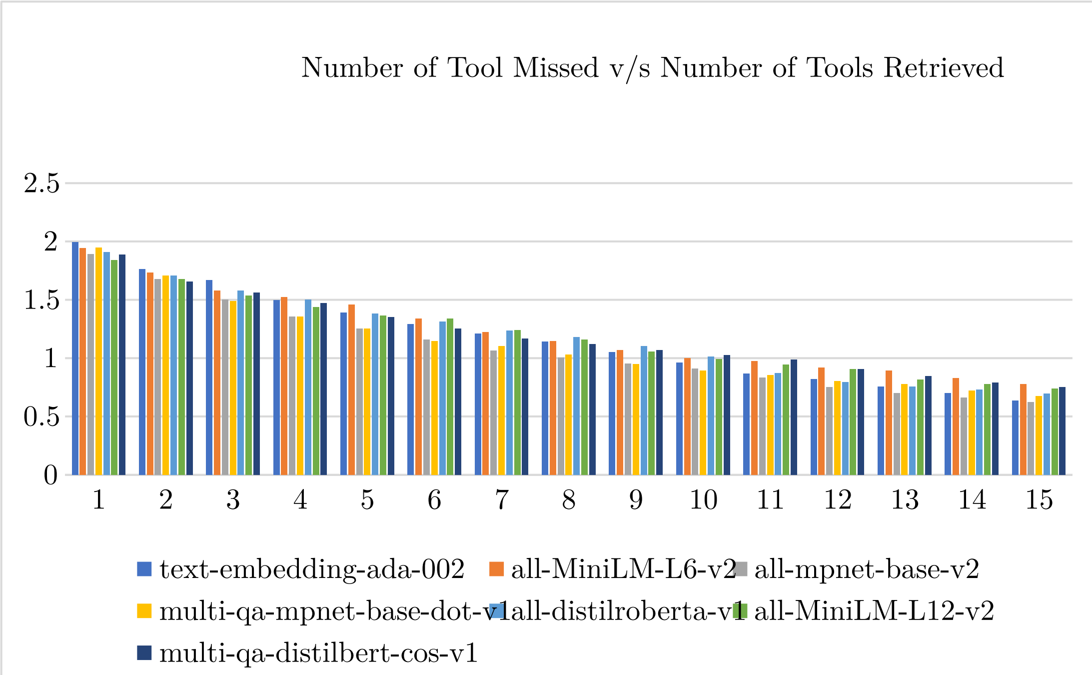

Figure 5: Top-K Retrieval Scores

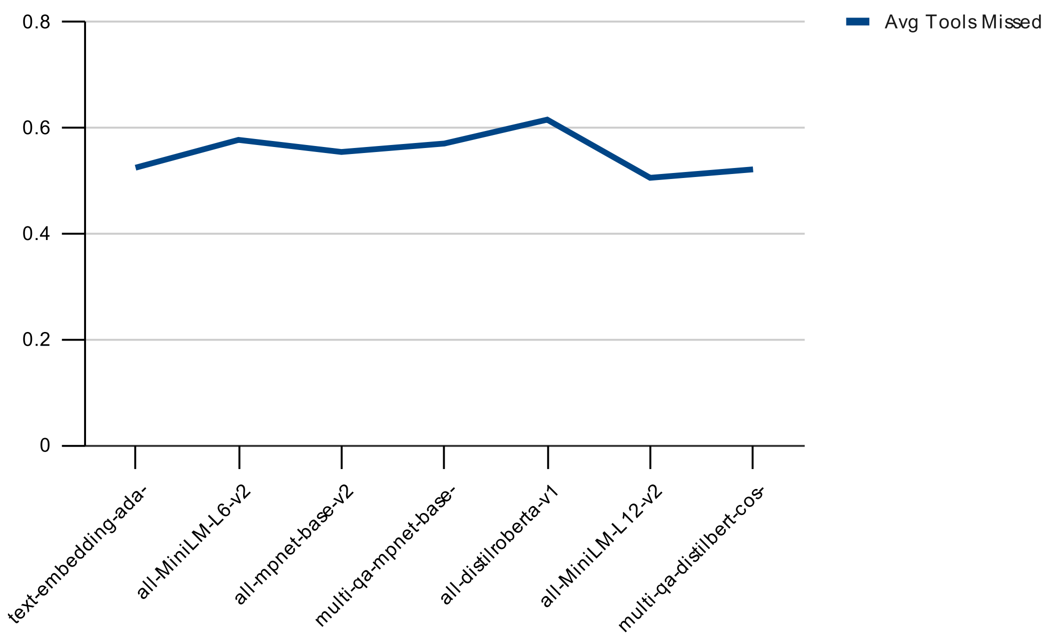

Figure 6: Threshold Based Retrieval Scores