Inter IIT Tech Meet 12.0

AI Agent 007 : Tooling up for Success
End-Term Submission

Team ID: 31

December 2023

1 Introduction

Large Language Models (LLMs) demonstrate exceptional capabilities in handling language-based tasks. However, for LLMs to progress towards Artificial General Intelligence (AGI), they must also efficiently perform logical and mathematical operations, an area where they currently struggle [1]. Along similar lines, tool-augmented LLMs gain importance. This involves using LLMs to access external APIs for assistance with logical and mathematical challenges.

In our project, we have established two distinct tool-based pipelines. The first pipeline is designed to maximize efficiency, while the second is optimized for performance. Both systems are developed with specific objectives in mind. They are crafted to analyze the input query, select the appropriate tools, define the tool arguments, and sequence the deployment of these tools. Moreover, our aim is to create systems that are highly scalable and capable of processing queries from various domains. These systems are planned to maintain low operational costs and minimize the carbon footprint.

Furthermore, we have developed a novel method for high-quality data generation. This is particularly useful for fine-tuning a tool-augmented LLM. We fine-tuned multiple publicly available models like OpenChat and GPT 3.5.

2 Literature Review

In our mid-term submission we explored various papers on datasets like, APIBank [2], Tool Bench [3]. We also explored papers on various prompting techniques like ReACT [4], Step-back Prompting [5], Analogical Prompting [6]. We also explored papers with different model-based solutions, like, ART [7] TALM [8] etc.

Our literature review post mid-evaluation submission leads us to LLM Enforcers which solve a major issue with Language Models, especially in the context of Multi-Agent Systems (Section 3.2). When requiring a precise output format, LLMs do not always perform as instructed. Prompt engineering techniques are not always sufficient. Output enforcers filter the model's generated tokens at every time step. Possible Solutions found to control outputs. We leverage ***ToolDec*** [9] and ***LLM output enforcer*** [10] heavily in our Proposed solution shown as discussed in Section 8.2

The ***R**easoning-vi**a**-**P**lanning* ***(RAP)*** paper [11] tackles LLMs' struggle with complex reasoning tasks. They propose an internal *world model* to predict the *world state* and simulate long-term outcomes of actions. The LLM incrementally builds a reasoning tree under given reward metrics. A modified version of Monte Carlo Tree Search is then employed to obtain a high-reward reasoning path. We extensively incorporate their ideas in one of our proposed solutions, shown in Section 11.1.

Constraints The problem statement is subject to a set of specific constraints. The pipeline output is required to conform to a predefined JSON schema. Additionally, the database of tools must be designed for adaptability. This means that our solution should possess the capability to dynamically integrate new tools or omit existing ones.

Discussion on Latency and cost The primary metric of competition remains the sum of cost and inference time, expressed as

$$metric = \frac{\text{tokens}}{\text{cost}} + \text{inference\_time}$$

Our empirical observations revealed that fine-tuned GPT-3.5 Turbo exhibited considerably lower latency, achieving speeds upto 3 times faster. It can be primarily attributed to its reduced input/output tokens, at the expense of a threefold increase in inference cost than GPT 3.5. Despite this cost escalation, the performance enhancement is quite notable. We evaluate open-source models such as OpenChat [12] and Zephr-7B [13] to further mitigate costs on platforms like Replicate. Additionally, to improve the latency aspect, we explore advanced tactics such as *Paged Attention* [14] with the *vLLM* [15] library. It promises significant speed enhancements, potentially up to 24 times.

3 Learnings from Mid-Eval

3.1 Ideas on Data

There is a general agreement in the research field that creating tool-based language learning models starts with producing a high-quality dataset, which is then used to fine-tune an LLM like LLaMA [16] or Vicuña [17]. This approach is supported by multiple papers [3, 8, 18, 19]. Training with simulated data has been shown to effectively prepare models for specialized real-world scenarios involving the use of tools [18]. In the case of models like GPT 3.5, optimal performance is typically achieved by providing a range of 50-100 examples [20]. Our efforts are directed toward the automated creation of refined datasets, prioritizing the quality of data over its quantity. Further details of our methodology can be found in Section 3.2.

3.2 Data Generation

We aim to generate high-quality data and train our agents to make inferences. This involves the simulation of a software company, including the staff and users. Our contribution generated about 200 distinct tools across 1800 fields ranging from 'Data Analytics Tools' to 'Text Editing or Word Processing Software'. Our optimized code structure resulted in a cost of only about $3.5 using the GPT-3.5 Turbo API to create all the tools. We follow three methods for the generation of tools:

Multi-Agent Framework The use of multi-agent systems has shown promise in automatically generating large volumes of training data [21, 22, 23, 24, 25]. API Bank and ToolAlpaca have highlighted the importance of agent self-perception in improving task performance [2, 18]. In our implementation, we utilized a multi-agent framework for autonomous data generation. This involved a simulated group dialogue among agents with different areas of expertise, akin to a corporate hierarchy. A critique agent refined the outputs to maintain high-quality results while significantly reducing the cost of annotation by 98% compared to human annotation [2]. However, we encountered difficulties in standardizing the format of the model outputs, which hindered effective processing and agent interaction.

Conversation Framework Using the RAP and ControlLLM prompt techniques that are discussed later in the paper on GPT-4, we have generated an effective query generator. It converses with the solution generator and, with human feedback, produces high-quality data. We make our "golden" evaluation dataset with this.

Experiment A: Flow Based Generation Our first attempt at data generation was specified handcrafted domains and respective end-users. Domains may be Product management, Project management, Software Architecture etc. We used these domains and a custom prompt to generate *flows*. We define flows as probable scenarios with domain and end-user as guiding inputs. A flow is a JSON object with action items action descriptions, and dependencies between actions connecting based on the order/timeline of actions concerning their input-output correspondence.

Further, we curate these flows using a custom tree-based approach to account for similar flows or redundant action items by backtracking and using the DFS (Depth First Search) algorithm. We utilized these flows further to generate well-documented step-by-step query descriptions. These serve as our guiding tools for generating solutions. Finally, we feed these flows along with query descriptions to GPT Models to generate the ideal solutions, which are nearly 100% accurate in utilizing the ControlLLM prompt. Also, we generate human-like concise queries from step-by-step descriptions to compile the final training data with queries and their JSON solutions. Some limitations we faced in this approach were hallucination in entity information and argument values information. Also, the generated examples lacked diversity in query-solution pairs, with instances of repetition and quite unrealistic queries.

Experiment B: Persona Based Generation Here, we explain how our final data generation pipeline tackles the limitations we faced in our previous approach. We start by hand-crafting multiple topics/domains which require API frameworks such as "Sales and Marketing", "Cybersecurity and Data Protection", "Inventory Management", etc. We define 18 such independent domains. Next, we prompt the GPT-3.5 turbo model to describe 5 possible personas working with the respective software for all the domains mentioned above. We extensively experimented with various models like Zephyr-7B, OpenChat-13B, Llama-13B and GPT-4-1106-preview and found GPT-3.5 turbo performed the best, with the right balance between performance and cost. We enable the model to generate additional relevant information about various personas, such as their age, profession, education, hobbies, etc., to better generalize on Out-of-Distribution data with increased diversity.

Next, we prompt GPT to generate all the relevant entities required in the respective domains and their properties. We then generate possible states based on the specific domain and associated entities, representing potential real-life scenarios in the software pipeline. Furthermore, we generate a 5-level task, distributed into five micro-actions based on the given state, each accompanied by a detailed, elaborate description.

3.3 Graph-of-Thoughts

We explored the Graph of Thoughts (GoT) framework for modelling thoughts [26]. In GoT, thoughts are represented as vertices in a graph, and dependencies between thoughts are represented as edges. This allows

for aggregating related thoughts by constructing vertices with multiple incoming edges. It is said GoT can extend the capabilities of existing frameworks, such as the Chain of Thoughts (CoT) and Tree of Thoughts (ToT), to accommodate more complex thought patterns. Our experiments, however, revealed that the excessive API calls resulted in numerous timeout errors by OpenAI, which is an area that should be addressed in future improvements.

3.4 Stepback Prompting

The RAP paper [11] highlights step-back prompting as a key element in achieving SOTA outcomes in mathematical reasoning tasks. We also noted the efficacy of stepback prompting during our mid-evaluation. When used alongside advanced models such as GPT-4, this technique enabled us to successfully resolve all queries presented by DevRev, albeit with a significant token expenditure associated with GPT-4's usage. Detailed results of this approach are presented in Listing 3 within Appendix B.

4 Experimentation

4.1 New Prompting Techniques

In the proposed pipeline discussed in Section 8, it is necessary to prompt an LLM such as OpenChat. We incorporate insights from prompting strategies we examined during the mid-submission phase. Specifically, to decompose a user query, our designed prompt instructs the model to formulate a thought considering the subsequent step. The few-shot prompting method is the most effective, balancing efficiency and performance. Additional details can be found in Section 8.

4.2 Graph-based Type Checking

While experimenting with different LLMs and prompts, we noticed that the LLM was hallucinating the substitution of argument values. For example, in place of `"$$PREV[i]"` the LLM may hallucinate `["$$PREV[i]"]` and vice-versa. We developed a Graph-based algorithm for type-checking these errors. The algorithm works as follows -

- **Initialisation:** Using the Tool database, we build a directed graph with nodes as tools and directed edge with weight 1 from tool 1 to tool 2, if the output of tool 1 can be directly fed into the input of tool 2. Similarly, a directed edge with weight 2 is drawn if the output of tool 1 is fed into the input of tool 2 within a list.
- **Type-Checking:** To check whether the `$$PREV[m]` is compatible tool N's arguments, the algorithm checks if there is a edge between tool N and tool m with the required arguments types. It also updates the arguments if the `$$PREV[m]` requires to be passed as an array

4.3 Reflexion

Reflexion represents a recent advancement in LLMs, mitigating hallucination in generative models. The architecture incorporates a feedback loop, resembling an LLM-in-the-loop rather than a human-in-the-loop approach. It converts binary/scalar feedback from the environment to text, acting as a 'semantic' gradient descent. Reflexion introduces a system with three models: the Actor, Evaluator, and Self-Reflection Model. The Actor generates output from the initial query, the Evaluator provides feedback akin to an environment responding to actions, and the Self-Reflection Model transforms this feedback into questions for the agent. The iterative process, resembling human learning, involves the agent generating responses based on self-evaluator questions, enhancing performance by learning from prior mistakes. In the ExpeL paper [27], the Self-Reflection agent generates cues during operation, which results in better performance than manually given instructions. An example cue is:

Listing 1: Example of cue generated by Self-Reflection Agent

```
The actor included unnecessary tools in
the solution, such as ' ' and ' '.
These tools are not required to
determine the priority of the
objects related to aorganizationon.
The actor did not use the ' ' tool,
which is not necessary for this query,
but it should have been used to ensure
the current user is properly identified.
Revised solution:
```

4.4 Output Enforcer for GPT!

Observing the success of our Proposed method shown in Section 8, we note how using an LLM Enforcer reaped great results. A simple OpenChat model and the LLM enforcer ensured unparalleled results. We propose a novel idea to extrapolate this idea to GPT 3.5. The LLM enforcer is only available for use on open-source models since the enforcer manipulates the token prediction probability of the model's final layer. We extend this to closed-source models. To ensure we reduce hallucinations on models such as GPT 3.5, we propose the following -

- Get output from a closed-source model like GPT 3.5.
- Pass this output as an input to an open-source model like OpenChat.
- Prompt the OpenChat model to exactly replicate the inputs as its inputs - similar to an identity function.
- Use LLM Enforcers on the OpenChat model. Since the OpenChat model is asked to give out the same output as its input from GPT, we

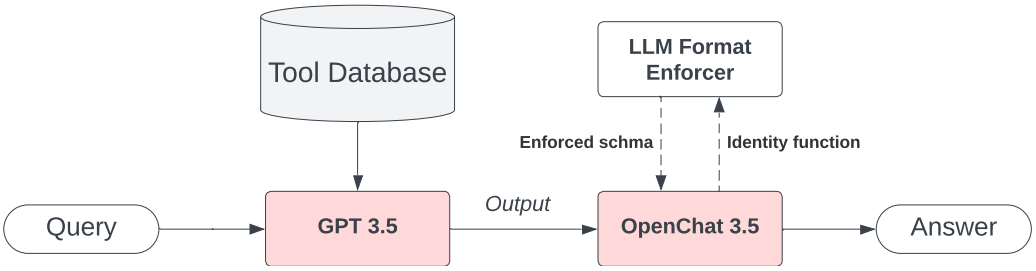

Figure 1: A novel pipeline to use LLM enforcers on closed-sourced models like GPT 3.5

are effectively enforcing the outputs of a closed-source GPT 3.5 model.

5 Evaluation Metrics

We adopt a multi-faceted evaluation framework inspired by ControlLLM [28], which considers tool selection, argumentassignment, and overall solution adequacy. Below, we outline the same

5.1 Tool Selection Metrics

- *Irrelevant Tool Inclusion Rate (IR)*: Ratio of irrelevant tools to predicted tools.
- *Necessary Tool Inclusion Rate (NR)*: Ratio of necessary tools to predicted tools.
- *Missing Tool Rate (MR)*: Ratio of missing tools to necessary tools.

5.2 Argument Assignment Metrics

- *Resource Hallucination Rate (HR)*: Identifies the prevalence of nonexistent resources in the tool arguments provided by the method. Lower HR values suggest a reduced tendency towards creating hallucinated arguments.

5.3 Solution Evaluation Metrics

- *BLEU Score*: Assess similarity between generated outputs and actual target.
- *Rouge-L F1 Score*: measures the longest common subsequence (LCS) between the system-generated solution and the reference solution. The F1 score computes the harmonic mean of precision and recall, where precision is the ratio of the length of the LCS to the length of the system solution, and recall is the ratio of the length of the LCS to the length of the target solution.

5.4 Efficiency Evaluation Metrics

- *API Call and Token Count*: We tally the quantity of API calls and token generation to approximate the cost implications.
- *Correct Path Rate:* Model can generate a solution path which may contain the most optimum solution as a subsequence , such solutions these paths are used to calculate the Correct Path Rate of the model.

The metrics discussed find precedence in several key works [28, 29, 30], and have informed our selection of Irrelevant Tool Inclusion Rate (IR), Missing Tool Rate (MR), and Resource Hallucination Rate (HR).

5.5 Meta-Programming

Defining tools with JSON formatting results in a high token redundancy, with keys such as `tool_name` repeated for each tool, significantly increasing the token count. For example, multiple instances of the same key can unnecessarily consume our token budget.

JSON to Python/Typescript We attempt using python functions to describe tools, since parameter names describe `argument_name`. We also try conversion to TypeScript, shrinking the length to roughly *one-fourth* of the token count.

Back to JSON Upon generating the Python or TypeScript output from the model Upon generating the Python or TypeScript output from the model, we use a simple implementation to convert it into the required JSON format. Using *metaprogramming*, we have implemented all the tools as Python functions in the runtime. When these functions are called, they append a call object to a global list, initially empty. The function returns an object of the output class. The response from GPT-3.5 is executed, and necessary functions are called to populate the array with the actual response data. The output class has overridden all arithmetic operators, enabling seamless handling of arithmetic operations. When such operations are utilized, the overridden operators facilitate their proper execution. However, while the technique seems promising, the models we trained worked better with JSON; hence, we returned to JSON. Besides, with JSON mode, we could utilize techniques like JSON-former to generate only the value tokens, reducing the number of tokens generated.

6 Benchmarking

6.1 ControlLLM

ControlLLM [28] proposes a Thoughts-on-Graph based method for tool selection, aiming to enhance scalability and prevent hallucination. In our implementation, we adopt the task decomposition aspect of the model utilizing their prompting technique to break down user queries into sub-tasks. This implementation with GPT-4 achieves 100% accuracy, as detailed in the provided prompt found in Appendix C. However, a limitation is observed due to the high cost associated with this approach, the large number of tokens required and the expense of GPT-4.

6.2 RAP

Like ControlLLM, we observe that the RAP paper, when used in conjunction with GPT-4, gives a near-perfect performance. Hence, we leverage RAP for benchmarking our model's outputs. Besides this, we also found that using a refined version of RAP instructions helped GPT-3.5 turbo generate very good results without any training. This forms our final pipeline.

6.3 Retrievers

In our methodology, we harness the capabilities of two distinct retrievers, first is the OpenAI's text embedding 'openai/text-embedding-ada-002' and the second is 'Toolbench IR_bert_based_uncased' (ToolBench's API Retriever)[3], which serve as crucial components within our tool recommendation system. The first retriever, 'openai/text-embedding-ada-002,' retriever is used in our 'Main solution. On the other hand, the second retriever, based on BERT and employed in our Alternate Solution, is a sentence transformer model. It maps sentences and paragraphs to a 768 dimensional dense vector space and is useful for tasks like clustering or semantic search. We evaluate the retrievers for their efficiency in Appendix Table 2

7 Main Solution

Our proposed solution utilizes an effective tool and example retrieval system coupled with a novel prompting technique to effectively solve the query using just a single LLM API call. The process involves tool retrieval, task decomposition and step-by-step reasoning. This is the best solution, taking deployment cost and inference times into consideration.

7.1 Salient Features of our proposal

Retriever chooses the relevant tool to solve the problem. Then, examples retriever chooses relevant examples which use the tools. This speeds up the reasoning process and reduces input tokens. A novel prompting technique utilizes just a single API call with relatively small number of input tokens. This minimizes cost drastically and makes reduces latency significantly. Examples from our "golden" dataset are augmented to increase accuracy. Users can freely add new tools or add relevant examples. This approach is extremely robust and generalizable and works exceedingly well on a one-shot setup.

7.2 Retrieval

Both tools and examples superset is embedded in Ada embeddings.

7.3 Novel Prompting Techniques

From the paper on Reasoning via Planning (RAP) [11], we learnt a few critical things. Firstly, we introduce the concepts of "state" and "action". This helps in task decomposition step-by-step by asking sub-questions about which tool(action) to use to get the next "state". Many "actions" can be taken at any state. The LLM evaluates each one of them to decide the next action, which ensures it covers a larger part of the reasoning space. Further, the LLM maintains a "world model" that maintains context concerning the task itself and how any new "action" may affect the overall picture. This greatly eliminates hallucinations.

From the Expel paper [27], we noticed that the best way to decide what the next state should be is a series of "insights". These provide the LLM with a set of guidelines that allows it to choose the correct action with minimal hallucination. We carefully curate these insights by analyzing how the sub-questions were developed and answered by RAP-prompted GPT-4.

Our prompt utilizes the best ideas of both papers. The effective task decomposition of RAP combined with good quality "insights" to make decisions at the task level makes this model highly accurate. The prompt can convey such a large amount of information and maintains a relatively low number of input tokens(2900 tokens for 17 tools and two large examples). Further, it is highly generalizable and works well on new tools without adding relevant examples.

7.4 Deployment

We make a pip-installable library and upload it to Pypi. This ensures our pipeline is easily accessible and can be run by anyone with an OpenAI key. We also deployed a react web app showcasing our pipeline.

7.5 Justifying use of OpenAI model and embeddings

Firstly, GPT-3.5-turbo performed fairly poorly on most tasks on its own and in implementations of many recent papers. The model is cheaper to deploy cost-wise than the much more powerful GPT-4. On the other hand, deploying "open-source" models on platforms like Replicate is costlier in this application, as it bills time-wise. Hence, implementing our pipeline on GPT-3.5-turbo makes perfect sense as it is very cheap

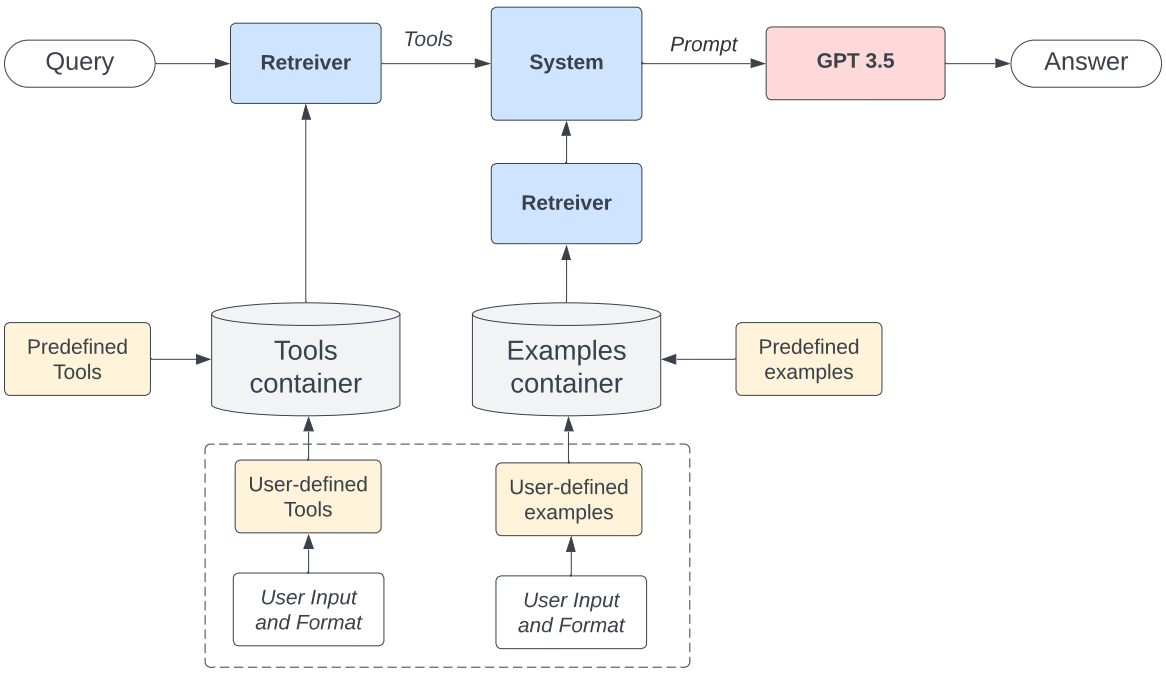

Figure 2: Pipeline for the main proposed solution. The dotted lines represent the optional component of the process.

and has fast inference times. On average, inference of large complex queries from out "golden" dataset takes less than Rs 0.4.

The Ada embeddings are very cheap and effective and are better than hosting open-source retrievers on a cost basis. Per query, the costs are almost negligible.

8 Proposed Alternate Solution

This model leverages the usage of ToolBench's API Retriever to optimize tool retrieval [3]. The system is structured around a tailored three-stage process comprising Tool Retrieval, Decomposition, and Recomposition, designed with a focus on maximizing efficiency in terms of both time and cost metrics.

8.1 Salient Features of our Proposal

Noteworthy features of our pipeline include full open-source implementation the complete elimination of tool and argument hallucinations through the use of the Language Model Format Enforcer (LM Format Enforcer). Integrating the open-source OpenChat into our chat model is a pivotal decision; its cost-efficient deployment in Replicate adds operational efficiency, optimising resource management. Even with a 4-bit quantised model, we accomplish high-quality task decomposition. The entire model can be run with less than 6 GB GPU. Due to the enforcement of tool and argument names, we get a negligible hallucination rate, showcasing the benefits of controlled generation.

8.2 Discussion on LLM Hallucination

Implementing the Language Model Format Enforcer (LMFE) is crucial in mitigating tool name hallucination. LMFE systematically reduces the probabilities assigned to disallowed tokens in the output space to zero. Extending beyond simple JSON format enforcement, complex JSON schemas can be enforced. This compels the system to prioritize and utilize only permissible tokens, hence eliminating hallucinations.

8.3 The Pipeline:

Step 1: Tool Retrieval We use ToolBench's API retriever (ToolBench_IR_bert_based_uncased) to convert the query and tools from the text format to Embeddings. Then, we take the cosine similarity between the tool embeddings and the query to get an array with similarity scores for each tool concerning the query. From this, we pick the top-k (10, in our case) tools.

Step 2: Task Decomposition We few-shot prompt an open-chat model [31] to decompose the given query into multiple sub-tasks, each associated with one tool for resolution. A custom structure for these sub-tasks has been developed, and its adherence is enforced by the LMFE. Additionally, the LMFE eliminates tool name hallucinations. Prompt is given in Appendix D

Step 3: Task Re-composition We again use a one-shot prompt with open-chat [31] model to recompose the given sub-tasks into one cohesive JSON file. We use LMFE to enforce the JSON schema. The LMFE also eliminates the tool name hallucinations

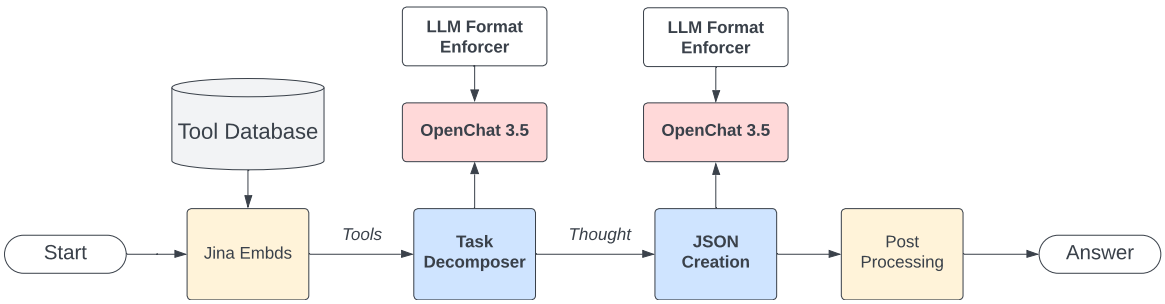

Figure 3: Pipeline of Proposed Architecture

and argument name hallucination. We do some post-processing on the output from the open-chat, to remove common mistakes.

9 Evaluation

We evaluate our experimented models on the metrics defined in Section 5. We show benchmarking results of expensive-yet-strong models like GPT4 with Control LLM and GPT4 with RAP in Table 1. We observe that RAP prompt with 'gpt-4-1106-preview' model gives the most accurate results. We then compare the results of our RAP-based GPT3.5 model, with other RAP-based models. This is shown in Table 2, which highlights the efficiency of our proposed solution in Section 7. Comparing the ControlLLM prompt with the Main Solution(RAP), we see that the Main solution based on the RAP prompt gives a much lower IR score and a much higher NR and BLEU score.

10 Bonus Task

Our Main Solution can effectively works first time on a new set of tools with minimal hallucination. This is because of the highly generalizable and robust prompt coupled with the retrievers that always get relevant examples and tools for good context. Further, the tool generates largely accurate solutions for queries that can only be partly solved correctly.

11 Future Work

11.1 Reasoning Tree

In our approach, we implemented a graph-based method focusing on domain classification. The system includes specific domains, such as an exclusive authentication domain and a termination domain marked by an end_tool. We utilised the Language Model (LLM) to generate sub-questions for each task, which the retriever then used. The graph structure is established by connecting the initial state to sub-states generated by tools within the authentication domain. Subsequent states are determined by the LLM, considering the context of the current tool and all tools leading to it. The LLM selects the domain for further states. The retriever identifies top-n tools within the chosen domain based on sub-questions generated for that domain. These selected tools become the children nodes in the graph. The process involves assigning probabilities to paths based on the retriever's output, and the desired output data type constrains the LLM. For backtracking, we applied Dijkstra's algorithm, considering the probability of each path. The graph, when generated appropriately, supports a depth-first-search approach where the LLM makes tool choices at each stage, ensuring a systematic exploration of possible solutions. A Monte-Carlo tree method to exploit nodes of the tree will make this method highly optimised.

11.2 Finetuning LLMs

Our motivation is to achieve a system that is faster, capable of processing logical tasks, and cost-efficient. This is why we chose to utilize Python over JSON. All tools and their corresponding solutions were converted to Python format to reduce token sizes. We also experimented by converting them to TypeScript to reduce them further. Admittedly, the conversion process from one language to another does involve some degree of complexity. We leverage meta-programming tactics for the same. Being dynamic instead of being based on sub-processes, it works significantly faster and is much more cost-effective. In other words, code is written during run-time. Converting the Python output to our desired JSON format takes only 120 ms on Google Colab.

We tried fine-tuning many open-source models, including Mistral, Vicuna, OpenChat, and GPT 3.5. We observed that Finetuning open source models led to generalised models. It can be inferred that such large language models only learn to pick up the output template of the given samples. There is no real learning by the model. This can be intuitively understood as a large model with billions of parameters, a size in gigabytes, being asked to fine-tune on only a few bytes of data!

However, in a surprising achievement, fine-tuning GPT 3.5 led to a very strong model. Finetuning GPT-3.5 enabled a significant reduction in response times. Our fine-tuned version improved performance, delivering responses in approximately 818 ms. By adopting the GPT-3.5-turbo pricing model, we cut response time by about 15% and costs by nearly 37%. We experimented with a Python toolset to generate JSON

Implementation	Model Name	IR ↓	NR ↑	HR ↓	MR ↓	BLEU Score ↑	ROUGE-L-F1 Score ↑
ControlLLM prompt	gpt-4-1106-preview	0	1	0	0.15	0.953	0.620
ControlLLM prompt	gpt-3.5-turbo-1106	0.098	0.902	0.00	0.067	0.872	0.457
ControlLLM prompt	openchat_3.5	0.250	0.750	0.250	0.367	0.483	0.304
RAP prompt	gpt-4-1106-preview	0	1	1	0	1	0.798
Main Solution	gpt-3.5-turbo-1106	0.152	0.956	0	0.11	0.959	0.675

Table 1: Comparing benchmark results from the evaluation of different implementations on DevRev queries like ControlLLM and RAP. Metrics like IR, HR and MR have to be lower, while NR, BLEU score and ROUGE score have to be higher**(JSON-to-JSON approach)**

Implementation	Model Name	IR ↓	NR ↑	HR ↓	MR ↓	BLEU Score ↑	ROUGE-L-F1 Score ↑
ControlLLM prompt	gpt-4-1106-preview	0.038	0.955	0.103	0.152	0.953	0.838
RAP prompt	openchat-3.5	0.438	0.563	0.875	0.875	0.278	0.125
RAP prompt	llama-70b	0.596	0.754	0.254	0.456	0.456	0.332
RAP prompt	gpt-4-1106-preview	0.019	0.971	0.082	0.057	0.968	0.895
Main Solution	gpt-3.5-finetuned	0.098	0.902	0.171	0.108	0.841	0.832
Main Solution	gpt-3.5	0.036	0.964	0.164	0.195	0.923	0.774
Alternate Solution	openchat-3.5	0.305	0.695	0.01	0.345	0.629	0.552

Table 2: Comparing benchmark results from the evaluation of different implementations on our **"golden dataset"**. It must be noted the remarkable improvement of Our Method. **(JSON-to-JSON approach)**

outputs to reduce the token cost. However, it was seen that the model performed well on JSON input and outputs compared to using Python for any stage. Our model performs almost perfectly on all available DevRev queries. However, we do not keep the solution as our proposed solution. Comparative experiments with OpenChat [12], while yielding slightly slower processing times, still demonstrated the utility of our TypeScript approach. This innovative methodology conserves token usage and accelerates response times. However, we observed that the fine-tuned OpenChat model does not perform comparably to the fine-tuned GPT3.5 model.

11.3 Domain Classification

Taking inspiration from the software architecture, we propose automatically generating the objects from the tool descriptions. Then we try to predict the attributes of these entities. Parallely, we can automatically organize the given set of tools into static tools and methods grouped by entity. This way we rephrase the given tool manipulation problem into a software architecture-based problem. In our experiments showed that this approach ensures tools like "who_am_i" are easily recognized and utilized. This is something that other approaches generally struggle with.

12 Conclusion

In conclusion, our project focuses on developing efficient and accurate AI agents for tool-augmented language models. We have proposed two pipelines, the main and alternate solutions, aiming to maximize efficiency and minimize cost while ensuring accurate results.

Our main solution incorporates a tool, an example retrieval system, and novel prompting techniques to solve queries using a single LLM API call. This pipeline has been designed to be highly scalable and cost-effective, making it suitable for various domains.

On the other hand, our alternate solution leverages a state-of-the-art retriever for tool retrieval and implements task decomposition and recomposition techniques to generate accurate solutions. This pipeline also prioritizes efficiency and cost-effectiveness.

We have conducted extensive experimentation and evaluation throughout our project to fine-tune our models and ensure their accuracy. We have also explored various literature on data generation techniques, prompting strategies, and control LLMs to enhance our understanding of the field.

We plan to refine our models by incorporating additional techniques such as reasoning trees and domain classification. We also aim to explore more advanced prompting methods and continue benchmarking our models against existing solutions.

Overall, our project represents a significant step towards developing efficient AI agents that can effectively handle language-based tasks with the assistance of external tools.

References

[1] Jie Huang and Kevin Chen-Chuan Chang. *Towards Reasoning in Large Language Models: A Survey*. 2023. arXiv: 2212.10403 [cs.CL].

[2] Minghao Li et al. *API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs*. 2023. arXiv: 2304.08244 [cs.CL].

[3] Qiantong Xu et al. *On the Tool Manipulation Capability of Open-source Large Language Models*. 2023. arXiv: 2305.16504 [cs.CL].

[4] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL].

[5] Huaixiu Steven Zheng et al. *Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models*. 2023. arXiv: 2310.06117 [cs.LG].

[6] Michihiro Yasunaga et al. *Large Language Models as Analogical Reasoners*. 2023. arXiv: 2310.01714 [cs.LG].

[7] Bhargavi Paranjape et al. *ART: Automatic multi-step reasoning and tool-use for large language models*. 2023. arXiv: 2303.09014 [cs.CL].

[8] Aaron Parisi, Yao Zhao, and Noah Fiedel. *TALM: Tool Augmented Language Models*. 2022. arXiv: 2205.12255 [cs.CL].

[9] Anonymous. "ToolDec: Syntax Error-Free and Generalizable Tool Use for LLMs via Finite-State Decoding". In: *Submitted to The Twelfth International Conference on Learning Representations*. under review. 2023. URL: https://openreview.net/forum?id=27YiINkhw3.

[10] Noam Gat and Benedikt Fuchs. *lm-format-enforcer*. 2023. URL: https://github.com/noamgat/lm-format-enforcer.

[11] Shibo Hao et al. *Reasoning with Language Model is Planning with World Model*. 2023. arXiv: 2305.14992 [cs.CL].

[12] Guan Wang et al. *OpenChat: Advancing Open-source Language Models with Mixed-Quality Data*. 2023. arXiv: 2309.11235 [cs.CL].

[13] Lewis Tunstall et al. *Zephyr: Direct Distillation of LM Alignment*. 2023. arXiv: 2310.16944 [cs.LG].

[14] Woosuk Kwon et al. *Efficient Memory Management for Large Language Model Serving with PagedAttention*. 2023. arXiv: 2309.06180 [cs.LG].

[15] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[16] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[17] Lianmin Zheng et al. *Judging LLM-as-a-judge with MT-Bench and Chatbot Arena*. 2023. arXiv: 2306.05685 [cs.CL].

[18] Qiaoyu Tang et al. *ToolAlpaca: Generalized Tool Learning for Language Models with 3000 Simulated Cases*. 2023. arXiv: 2306.05301 [cs.CL].

[19] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL].

[20] Conor Kelly. *OpenAI Fine-tuning: GPT-3.5-Turbo*. 2023. URL: https://humanloop.com/blog/fine-tuning-gpt-3-5.

[21] Benfeng Xu et al. *ExpertPrompting: Instructing Large Language Models to be Distinguished Experts*. 2023. arXiv: 2305.14688 [cs.CL].

[22] Yashar Talebirad and Amirhossein Nadiri. *Multi-Agent Collaboration: Harnessing the Power of Intelligent LLM Agents*. 2023. arXiv: 2306.03314 [cs.AI].

[23] Various authors. *AutoGPT*. 2023. URL: https://github.com/Significant-Gravitas/AutoGPT.

[24] Qingyun Wu et al. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework". In: 2023. arXiv: 2308.08155 [cs.AI].

[25] Lilian Weng. "LLM-powered Autonomous Agents". In: *lilianweng.github.io* (June 2023). URL: https://lilianweng.github.io/posts/2023-06-23-agent/.

[26] Maciej Besta et al. *Graph of Thoughts: Solving Elaborate Problems with Large Language Models*. 2023. arXiv: 2308.09687 [cs.CL].

[27] Andrew Zhao et al. *ExpeL: LLM Agents Are Experiential Learners*. 2023. arXiv: 2308.10144 [cs.LG].

[28] Zhaoyang Liu et al. "ControlLLM: Augment Language Models with Tools by Searching on Graphs". In: *arXiv preprint arXiv:2305.10601* (2023).

[29] Yujia Qin et al. *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. 2023. arXiv: 2307.16789 [cs.AI].

[30] Yifan Song et al. *RestGPT: Connecting Large Language Models with Real-World RESTful APIs*. 2023. arXiv: 2306.06624 [cs.CL].

[31] Guan Wang et al. *OpenChat: Advancing Open-source Language Models with Mixed-Quality Data*. 2023. arXiv: 2309.11235 [cs.CL].

Appendix

In the appendix section, we provide detailed information on the following aspects of our study

A An example of our generated dataset

A.1 An Example Tool

```
{
        "tool_name": "merge_work_items",
        "tool_description": "Combines multiple work items into a single item.",
        "args": [
            {
                "argument_name": "source_work_ids",
                "argument_type": "str",
                "is_array": true,
                "is_required": true,
                "argument_description": "The IDs of the source work items to merge.",
                "example": [
                    "TASK-123",
                    "ISSUE-456"
                ]
            },
            {
                "argument_name": "target_work_id",
                "argument_type": "str",
                "is_required": true,
                "argument_description": "The ID of the target work item to merge into.",
                "example": "TASK-789"
            }
        ],
        "output": {
            "argument_type": "object",
            "is_array": false,
            "is_required": true
        }
    }
```

Listing 1: An example of a tool generated by us

A.2 An Example datapoint in our generated dataset

```
{
    "query": "Summarize the works created by user DEVU-123 that are currently 'In Progress'",
    "solution": [
        {
            "tool_name": "works_list",
            "arguments": [
                {
                    "argument_name": "created_by",
                    "argument_value": [
                        "DEVU-123"
                    ]
                }
            ]
        },
        {
            "tool_name": "filter_by_status",
            "arguments": [
                {
                    "argument_name": "work_ids",
                    "argument_value": "$$PREV[0]"
                },
                {
                    "argument_name": "status",
                    "argument_value": [
                        "In Progress"
                    ]
                }
            ]
        },
        {
            "tool_name": "summarize_objects",
            "arguments": [
                {
                    "argument_name": "objects",
                    "argument_value": "$$PREV[1]"
                }
            ]
        }
    ]
}
```

Listing 2: An example of query json pair with retrieved generated by us on the tool dataset given in the problem statement and generated by us

B StepBack Prompting

```
Given the set of tools provided in the JSON format, here is an example scenario where we want to identify work items created by the current user that are high priority and add selected items to the current sprint:

Question: How can we find high priority work items created by the current user and add them to the current sprint?

Sub-Question 1: Which tool to select now to identify the current user?
Answer 1: We will use the tool "who_am_i" to get the id of the current user. The answer is "who_am_i".

Sub-Question 2: Which tool to select now to get the current sprint id?
Answer 2: After identifying the current user, we should use the tool "get_sprint_id" to get the ID of the current sprint. The answer is "get_sprint_id".

Sub-Question 3: Which tool to select now to list the work items created by the current user?
Answer 3: With the current user ID, we can utilize the "works_list" tool to retrieve a list of work items matching the request. We need to specify the "created_by" argument, which requires the user ID. The answer is "works_list".

Sub-Question 4: Which tool to select now to prioritize the listed work items?
Answer 4: Given the list of work items, we need to use the "prioritize_objects" tool to sort these items by priority. The answer is "prioritize_objects".

Sub-Question 5: Which tool to select now to summarize the high-priority work items?
Answer 5: The output from "prioritize_objects" should be fed into the "summarize_objects" tool to get a summary of the high-priority work items. The answer is "summarize_objects".

Sub-Question 6: Which tool to select now to add the high-priority work items to the current sprint?
Answer 6: We will use "add_work_items_to_sprint" to add the given high-priority work items to the sprint. This requires the work item IDs and the sprint ID. The answer is "add_work_items_to_sprint".

Sub-Question 7: Which tool to select now to validate that the work items are added?
Answer 7: No tool is needed to validate the addition of work items directly. Instead, we can use "works_list" again to confirm if the items are part of the sprint by using appropriate arguments such as "sprint_id" and "owned_by". The answer is "works_list".

Since we have completed the sequence of actions needed to reach a logical conclusion, we will add an end tool.

Sub-Question 8: Now we can answer the question: Which tool do we use to end the process?
Answer 8: The "end_tool" indicates that the process is complete. The answer is "end_tool".
```

Listing 3: StepBack Prompting with 100% results using GPT-4

C ControlLLM Prompting

```
The following is a friendly conversation between a human and an AI. The AI is professional and parses user input to several tasks with lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know. The AI assistant can parse user input to several tasks with JSON format as follows:<Solution> ["description": task description, "tool_name": tool name, "id": task_id, "dep": dependency_task_id, "arguments": ["argument_name": name of the argument the tool expects, "argument_value": value of the specific argument or $$PREV[dependency_task_id]]]</Solution>. The "description" should describe the task in detail, and AI assistant can add some details to improve the user's request without changing the user's original intention. The special tag "dependency_task_id" refers to the one in the dependency task (Please consider whether the dependency task generates arguments required by this tool.) and "dependency_task_id" must be in "dep" list. The "dep" list denotes the ids of the previous prerequisite tasks, which generate a new resource that the current task relies on. The special tag "task_id" must be integers, which refers to the order in which the tasks should be implemented.  The "arguments" field denotes the input resources this tool expects to execute the task. The "tool_name" MUST be selected from the following options: "works_list", "summarize_objects", "prioritize_objects", "add_work_items_to_sprint", "get_sprint_id", "get_similar_work_items", "search_object_by_name", "create_actionable_tasks_from_text", "who_am_i", nothing else. Think step by step about all the tasks that can resolve the user's request. Parse out as few tasks as possible while ensuring that the user request can be resolved. Pay attention to the dependencies and order among tasks. If some inputs of tools are not found, you cannot assume that they already exist. You can think a new task to generate those args that do not exist or ask for the user's help. If the user request can't be parsed, you need to reply empty JSON []. You should always respond in the following format:
<Solution><YOUR_SOLUTION></Solution>.
<YOUR_SOLUTION> should be strict with JSON format described above.
Your knowledge base consists of tool descriptions and argument descriptions as explained below:
[
  {
    "tool_name": "who_am_i",
    "tool_description": "Returns the id of the current user",
    "args": [],
    "output": {
      "arg_type": "string",
      "is_array": false,
      "is_required": true
    }
  }
]
```

Listing 4: ControlLLM Prompting with 100% results using GPT-4

D Task Decomposition Prompting

```
You are a helpful assistant. Your job is to decompose a user query into tasks that can be solved with one tool each.
        Analyse the list of tools and split the query into tasks. Solve the problem by thinking step by step.
        In each thought, think about what the next step should be in order to solve the problem, based on the available tools. Explain why the tool is required.
        Find the required tool that should be called (make sure it is related to the thought), and construct a task to be completed using it, based on the thought. If no tool is required, use "no_tool".
        Ensure that each task contains the necessary information to call the tool associated with it. Do not create unnecessary steps, if they haven't been mentioned in the query. Keep the minimum number of required tools. Be short and concise.
        The output of the ith task can be referenced using "$$PREV[i]" (starts from 0). DO NOT call tools or write any code in the arguments, and do not make up arguments that don't exist.


        If the query cannot be solved with the given tools, just return an empty list []

        Note that this is a product management system, and we call our customers revs. Objects are things like customers, parts, and users.
        {formatted_tools(retrieved_tools)}

        Example:
        Query: Summarize high severity tickets from the customer (rev) UltimateCustomer and add them to the current sprint.
        Solution:
        [
            {
                "thought": "First, we need to get the id of the customer UltimateCustomer",
                "tool_name": "search_object_by_name",
                "task": "Use the search_object_by_name tool with the argument 'query' = UltimateCustomer"
            },
            {
                "thought": "Next, we need to get the high severity tickets for the customer",
                "tool_name": "works_list",
                "task": "Use the works_list tool with the arguments: 'issue.rev_orgs' = "$$PREV[0]", 'ticket_severity' = 'high', and 'type' = 'ticket'."
            },
            {
                "thought": "Now, we need to summarize the high severity tickets obtained from the previous task",
                "tool_name": "summarize_objects",
                "task": "Use the summarize_objects tool with the output of the second task, argument 'objects' = "$$PREV[1]""
            },
            {
                "thought": "In order to add the tool to the current sprint, we need to get the current sprint ID",
                "tool_name": "get_sprint_id",
                "task": "Use the get_sprint_id tool to get the current sprint ID"
            },
            {
                "thought": "Finally, we need to add the work items obtained using works_list in the second task to the current sprint, whose ID was obtained in the previous task.",
                "tool_name": "add_work_items_to_sprint",
                "task": "Use the add_work_items_to_sprint tool with arguments 'work_ids'='$$PREV[1]' and 'sprint_id'='$$PREV[3]'
            }
        ]
```

Listing 5: Task Decomposition Prompt for OpenChat with retrieved tools

E Tool JSON Formation Prompting

```
You are a helpful assistant. Your job is to output a json file which can be used to call the tool given below, based on the task given to you.
        These tasks are required in order to solve a given query. In case any arguments are missing in the task, you can still add them to the json.
        The output of the ith task can be referenced using "$$PREV[i]" (starts from 0). This is important, since many queries require a composition of tools.
        You can't reference tools that haven't been called yet.

        Tools must be explicitly called and cannot be called inside the arguments.


        Example:
        Query : Obtain work items from the customer support channel, summarize the ones related to part 'FEAT-345' part and prioritize them.

        Completed tasks and thought process:
        Task 0:
        Thought: First, retrieve all work items from the customer support channel related to 'FEAT-345'
        Tool_name: works_list
        Task: "Use the 'works_list' tool with the arguments 'ticket.source_channel'= ['customer support'] and 'applies_to_part'= ["FEAT-345"]

        Task 1:
        Thought: Next, summarize the work items related to 'FEAT-345' for clarity.
        Tool_name: summarize_objects
        Task: Use the 'summarize_objects' tool with the 'objects' argument being the output from the 'works_list' tool, 'objects'='$$PREV[0]'

        Your Task:
        Task 2:
        Thought: Finally, prioritize the issues from the customer support channel that are urgent.
        Tool_name: "prioritize_objects
        Task: "Use the 'prioritize_objects' tool with the 'objects' argument being the output from the 'works_list' tool, argument 'objects' = '$$PREV[1]'

        Answer:
        {
            "tool_name": "prioritize_objects",
            "arguments": [
                {
                    "argument_name": "objects",
                    "argument_value": "$$PREV[1]"
                }
            ]
        }
```

Listing 6: Tool JSON Formation Prompt for OpenChat with retrieved tools

F Evaluating Retrievers

	OpenAI	ToolBench Retriever
Top 5	0.7625	0.7325
Top 7	0.8562	0.8367
Top 9	0.9479	0.9362

Table 3: Benchmarking the two major Dense retrievers we use - OpenAI (`openai/text-embedding-ada-002`) and Jina Retrievers `ToolBench Retriever`. The Top 'N' score indicates the average percentage of tools needed to solve the query that are in the list of top 'N' fetched tools.