Hierarchical Query-Tool Decomposition

In tackling the complexity of **multi-step queries**, a sophisticated hierarchical decomposition approach was adopted to ensure efficient tool selection and execution. Each query, often entailing intricate steps with multiple tool invocations, was broken down into sub-queries using a **heuristic-based query** generator. This synthetic query generation process ensured that the original intent of the query was preserved while breaking it into manageable subtasks. The transformation followed a structured decomposition format where queries were reformulated as:

1. **Object** – The entity being acted upon.
2. **Action** – The operation being executed.
3. **Result** – The expected outcome or response.

This restructuring was achieved by leveraging large language models (LLMs) to interpret the semantic meaning of the query while preserving the **key elements (e.g., keywords, verbs)** to ensure an accurate representation. Furthermore, sub-query generation followed a logical sequence based on task dependency rather than the order of words in the original query, ensuring a more accurate mapping to tools.

Intent to argument extractor

To maximize the efficiency of sub-query processing, it was essential to map each sub-task to the **most relevant argument before identifying the appropriate tool**. This step involved generating embeddings for the combined argument and tool descriptions. By comparing the cosine similarity scores between these embeddings and the query embedding, the system was able to determine the closest match.

This innovative combination of argument and tool descriptions in the embedding generation significantly improved the ability to **chain tools** effectively. Arguments, being the key functional components in tool execution, were prioritized in this mapping process, ensuring that the selected tool could process the given input accurately.

Argument value is extracted using the ecpected argument_name and its dtype, if it is within “ ‘ “

Why Cosine Similarity?
As part of the query conversion process, we assess the similarity between queries to ensure that the conversion logic aligns with the intended semantic meaning. Cosine similarity is chosen as the metric for this evaluation due to the following reasons:

Semantic Relevance: By evaluating the angle between vectors, cosine similarity provides a meaningful assessment of how closely related the queries are, thus ensuring that the conversion process maintains semantic integrity.

Sparsity Handling: Many embeddings are sparse, which makes cosine similarity a robust choice. It effectively evaluates similarities in high-dimensional spaces, enabling accurate matching of queries even when they contain different levels of detail.

We have tabulated the toolset table given into a dataframe to make it easier to find cosine similarity. Storing the info as a dataframe also makes it more accessible. It improved the accuracy of predicting tool names and argument names by a good amount

Weighted Scoring Mechanism

While cosine similarity provided a strong measure of semantic alignment between the query and tool descriptions, it was insufficient on its own. Recognizing the importance of query-specific keywords, a weighted scoring mechanism was introduced. This allowed for an improved selection process by:

- Assigning higher weights to **direct keyword matches** in argument names.
- Combining this weighted score with the cosine similarity score to ensure that both semantic and lexical relevance were considered.

This dual scoring approach ensured that arguments most closely matching both the meaning and literal content of the query were prioritized, leading to more accurate argument and tool selection.

Once the argument-tool pair was identified, the system generated a list of these pairs for further processing. When consecutive tools in the list were identical, the arguments associated with them were intelligently combined. This optimization allowed for a streamlined execution of tasks, **minimizing redundant tool invocations** and enhancing overall performance.

The final list, consisting of [argument, tool] pairs, was refined by merging tools with consecutive argument pairs, thereby maintaining the integrity of the query decomposition while improving execution efficiency. This sophisticated tool-argument combination mechanism ensured that complex, multi-step queries were processed efficiently and accurately.

Self-Consistency:

Identifying repeated and unique subqueries on running thrice, and only taking up the common ones, until the results are consistent.

Its a technique used to enhance the reliability and **robustness of generated subqueries** by encouraging multiple sampling and aggregation of responses. Instead of relying on a single output, the model generates a diverse set of completions for the same prompt, often through temperature sampling or beam search. These various outputs are then analyzed for consistency in their content. The rationale is that when a model produces similar answers across different samples, it indicates a stronger confidence in the correctness of the response. By aggregating these consistent outputs—such as through voting mechanisms or averaging—this method effectively reduces noise and enhances the quality of the final answer. This is particularly valuable in scenarios where precision is critical, as it helps mitigate the inherent variability and unpredictability associated with generative models, ultimately leading to more trustworthy and contextually accurate results.

Re-invoking for checks:

While we saw the need of some examples for LLMs the see the usage, **few-shot prompting with synthetic examples** is a good way as a fallbakc mechanism.

Ensemble with Langchain:

Implmented a **Multi Agentic** Framework for our usecase using the GPT 3.5 as our LLM agents fot tool matching and tool chaining logic
Tried wih pure logic and algorithms like FAISS(facebook AI Similarity Search) and Fuzzywuzzy approach for toolchaining
Researched upon and implemented langchain tool retrieval system and integrated it with GPT

Tool Selection Agent (GPT-3.5):
This agent leveraged the superior semantic capabilities of GPT-3.5 to understand user queries and map them to the most relevant tools.
Semantic Matching:
GPT-3.5's proficiency in understanding context enabled accurate mapping of both direct and abstract query parameters to tool arguments,even handling non-semantic tokens like "DEV123."

Chaining Coordinator Agent:
This agent orchestrated multiple tools when necessary, determining how to chain them by understanding the outputs of one tool as the inputs for the next.

Single-Agent GPT Approach:
We evaluated a simpler single-agent framework where GPT-3.5 handled the entire process from tool selection to chaining.While effective for small tasks, this increased the computational overhead as complex multi-tool queries were processed in a single GPT-3.5 call.

Multi-Agent Approach:
Our modular multi-agent design offered flexibility, **reducing costs** by utilizing GPT-3.5 only for specific tasks such as chaining and advanced tool selection, while other tasks were handled by lightweight algorithms like fuzzy matching.

Planning on combining he output of both the embeddings and langchain methods and taking the OR/AND operation, havent applied ensemble on code.

Autoregressive Planning (trial)

The placeholder condition to check if query requirements are met can be replaced with a more robust validation mechanism based on your specific criteria.
The cluster_hypotheses function uses KMeans clustering to group the generated hypotheses based on their embeddings. This helps in identifying similar hypotheses that can be refined together- this is to check the correct subqueries.
It enhances the performance of query generation through a structured process of hypothesis formulation, clustering, and refinement. Initially, it generates a set of hypotheses from the input query, which are then transformed into dense vector embeddings using advanced models such as Sentence Transformers. These embeddings undergo cluster analysis, typically via algorithms like KMeans, to identify and group semantically similar hypotheses, thus facilitating the discovery of distinct sub-queries. From these clusters, representative hypotheses are selected for further refinement, promoting the exploration of diverse perspectives on the original query. This iterative loop continues until the requirements of the initial query are fully satisfied, ensuring a comprehensive and targeted exploration of the problem space. The approach not only enhances the specificity and relevance of the generated queries but also leverages the self-consistency of outputs, allowing for more coherent and contextually aware responses.

Possible Extensions-

a) Adding a previous Episodic Memory learning- Reinforcement learning in the background
b) Dependency graph learning for tools-With example sets elaborately, develop a graph of dependency for the tools, and do a Depth First Search on the graph, after finding the most relevant tool.
c) neural symbolic-
   1. Input Processing (Neural Network)
   The neural network processes unstructured data (e.g., text, images, or speech).
   In NLP, this could involve extracting semantic meaning from a query, generating embeddings, or summarizing information.

Example: In a query like "Fetch 'medium' severity work items in 'QA Review' or 'Testing'," a neural network model (e.g., GPT or T5)
would be used to extract key terms like 'medium', 'QA Review', and 'Testing'.
2. Symbolic Reasoning Layer
After the neural network extracts the relevant information, the symbolic reasoning system applies formal rules, logical structures,
or knowledge representations to interpret the results or make decisions.
Example: Once the key terms are extracted, symbolic reasoning could ensure that the selected tool and its arguments follow a logical sequence, such as enforcing constraints about which tools can be invoked together or ensuring that arguments adhere to specific types.

d) Graph neural networks
(GNNs) can be used to model the relationships between tools, arguments, and their dependencies.GNNs are designed to operate on graph-structured data. A graph is composed of:
Nodes (Vertices): Represent individual entities, such as tools or arguments.
Edges: Represent relationships or dependencies between these entities.
In GNNs:
Each node has an embedding (a vector that encodes information about that node).
Each edge represents the relationship between two nodes, which can also have associated features.
GNNs aggregate information from neighboring nodes and edges to update the embeddings of each node,
allowing the model to learn both local and global patterns in the graph.

Overview of Gemini 1.5 Flash

For generating embeddings, we employ Gemini 1.5 Flash, a cutting-edge language model specifically designed for fast and efficient text representation. Unlike traditional models like BERT, which produce embeddings based on a word-level approach, Gemini focuses on contextual embeddings that capture the semantic meaning of the entire query. This allows for a richer representation of text, which is essential for accurately converting regular queries into structured JSON objects.

Key Features of Gemini 1.5 Flash:

Contextual Understanding: Captures the nuances of language by providing embeddings that reflect the meaning of queries in context.
Efficiency: Optimized for quick processing, enabling real-time conversion of multiple queries into JSON format.
Versatility: Suitable for various NLP applications, including query transformation, semantic search, and text classification.

Tried attempts-

Toolset Pre-processing/Augmenting:

Tried relying on the LLM (Llama 3.2-1b) to generate a more elaborate tool description, such as: generating a set of keywords that could possibly be related to the tool name + description – and then running a similarity1 search. Identified that Llama 3.2-1b was probably not complex enough for the task (note: Llama 3.2-3b was too big to run on google collab or my pc); moved to the fully cloud-based Gemini 1.5-flash (we are aware that Gemini 1.5 pro is also available free of charge, but its rate limits are far more restrictive) – This move seemed to greatly improve our data pre-processing efforts. Also tried generating a full ultimate tool description of the tool (”tool astrology”), (by prompting the LLM with the tool name, tool description, and all tool argument names), instead of a list of keywords, with the hopes that our LLMs could work with those better. The LLM was unable to capture all the arguments in the output, and hence there was a need to pass the tool name + description + argument name one by one to generate an argument synopsis for each argument, and then merge these synopses for a more well-covered result.
1To run similarity scores, we tried using bare-basic keyword searches, several embeddings such as all-MiniLM-L12, all-mpnet-bas, paraphrase-MiniLM-L6-v2 and FAISS. All scores were more or less comparable, but paraphrase-MiniLM-L6-v2 seemed to perform somewhat more reliably on the example set. However, (perhaps owing to the fact that many tools/arguments were somewhat similar), our similarity scores of the query to the tool-astrologies were all too close to one another, effectively drowning the need for any preprocessing/augmenting.

Chain of Thought prompting-

Asking the LLM in a loop to keep checking if the first logical element is obtained, and exiting the loop when no tool is obtained.