

Technical Report

PREPARED BY TEAM 65

Contents

1 Experiments	2
1.1 LLM+P	2
1.2 ReAct	5
1.3 Platypus	8
1.4 PAL	10
1.5 Recursion of Thought	13
1.6 Tree of Thought	17
2 Final Approach	22
2.1 Retrievers	22
2.2 Composers	23
2.3 LLM's	24
3 Final Approach documentation	24
3.1 Introduction	24
3.2 Prerequisites	24
3.3 Example	24
3.4 Documentation	26
4 Sample Results	28
5 Bonus points formulation	32
6 Bonus points queries	33
7 Applet	35
8 Applet documentation - AI Agent 007	35
8.1 Overview	35
8.2 Installation	35
8.3 Application Composition	35
8.4 Usage	37
8.5 Application Workflow	37
8.6 Note	38
9 References	39

1 Experiments

1.1 LLM+P

1.1.1 Methodology

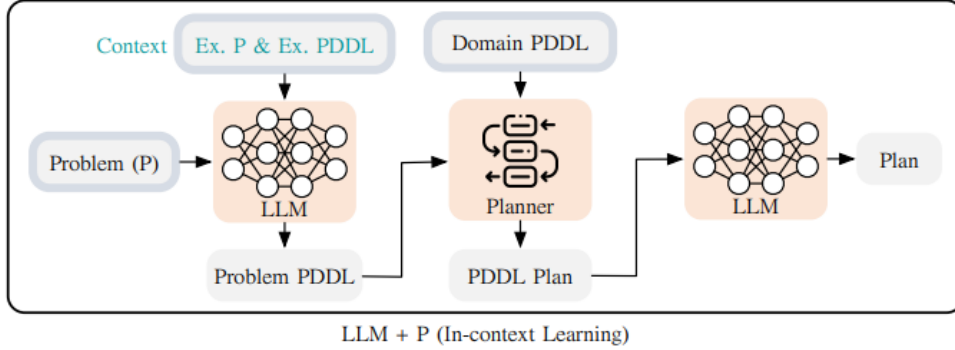


Figure 1: LLM+P (In-context Learning)

The planning domain definition language (PDDL) serves as a standardized encoding of classical planning problems. The PDDL representation of a planning problem P is separated into two files: a domain file and a problem file. The domain PDDL file provides a lifted representation of the underlying rules of the world. It includes a set of predicates that define the state space S and the actions (i.e., A) with their preconditions and effects (i.e., the transition function f). The problem PDDL file provides a list of objects to ground the domain, the problem’s initial state s_{init} and goal conditions S_G . There exists a rich set of symbolic planners that implement efficient search algorithms to solve planning problems formalized in PDDL. In this work, we aim to take a natural language prompt which describes the initial states $init$ and goal conditions S_G , formulate it in PDDL, and leverage symbolic planners to output correct plans. We assume the domain rules are available.

There are 3 parts in the methodology used -

1) Large Language Model as a PDDL Writer

LLMs are bad at planning (or long-horizon reasoning) but they are good at describing and translating textual inputs, including re-writing planning prompts in the PDDL format. The intuition is that we can view PDDL as a different language than English, so re-writing planning prompts in PDDL is essentially a “machine translation” task that LLMs are known to excel at. Here we provide an example of a PDDL problem file written by GPT-4 without any prompt engineering. As we see, the generated file appears to have the correct PDDL syntax but uses a made-up predicate (empty) and misses the initial condition that $b1$ is on the table.

2) In-Context Learning

LLMs are known to be capable of in-context learning without finetuning their parameters. By in-context learning, we mean LLMs’ ability to perform unseen downstream tasks by simply conditioning on a few input-label pairs (demonstrations). Here is a pair of problem descriptions in natural language

and in PDDL provided to GPT-4 as context. When the context is included with the prompt from the example above, the resulting PDDL problem file is directly solvable by the planner.

3) Large Language Model + Classical Planner (LLM+P)

Having introduced the LLM’s ability to encode problems in PDDL and in-context learning, we are ready to introduce the proposed LLM+P solution. In particular, we assume that for each problem domain, a human expert can provide a domain description (i.e. action preconditions and effects) that will be fixed for all problem instances that happen in that domain. While the problem of automatically generating the description is another valuable research question, in this proposed work, we assume that the description is available as a PDDL domain file. The LLM+P method is directly applicable as a natural language interface for giving tasks to robot systems. For instance, assume we want a robot to act as a bartender to make cocktails. It is reasonable to tell it what actions it can take, but leave itself to infer how to make new cocktails most efficiently given a set of ingredients to combine. Moreover, we assume the agent is provided with a minimal example that demonstrates what an example problem PDDL looks like for a simple problem inside that domain. Next, the agent is provided with a new (potentially quite complicated) problem (P). The LLM then uses the in-context learning to infer the problem PDDL file corresponding to P. Once the problem PDDL file is generated, we feed it into any classical planner, together with the provided domain PDDL file, to generate a PDDL plan. In the end, the LLM translates the PDDL plan back into the natural language to finish up the LLM+P pipeline.

1.1.2 Analysis

LLM+P works well for simpler cases of our problem statement, but fails to decipher complex logic, and also fails to decipher logical statements and conditional logic, in the case of bonus points.

LLM+P method also consumes a lot of tokens (Large language model api calls) for our problem statement.

But, LLM+P is surprisingly good at deciphering accurate logic and due to the PDDL language’s logical structure, it never gives inconsistent output.

1.1.3 Sample Results

Technical Report

Query	Output
Summarize issues similar to don:core:dvr-v-us-1:devo/0:issue/1	['tool_name': 'search_object_by_name', 'arguments': ['argument_name': 'don:core:dvr-v-us-1:devo/0:issue/1', 'argument_value': ''], 'tool_name': 'get_similar_work_items', 'arguments': ['argument_name': '\$\$PREV[0]', 'argument_value': ''], 'tool_name': 'summarize_objects', 'arguments': ['argument_name': '\$\$PREV[1]', 'argument_value': '']]]
Summarize high severity tickets from the customer UltimateCustomer	['tool_name': 'works_list', 'arguments': ['argument_name': 'ticket.severity', 'argument_value': 'high', 'argument_name': 'ticket.rev_org', 'argument_value': 'ultimatecustomer', 'argument_name': 'type', 'argument_value': 'ticket'], 'tool_name': 'summarize_objects', 'arguments': ['argument_name': 'objects', 'argument_value': 'ultimatecustomer']]]
Given a customer meeting transcript T, create action items and add them to my current sprint	['tool_name': 'create_actionable_tasks_from_text', 'arguments': ['argument_name': 'text', 'argument_value': 'T'], '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]']]]

1.2 ReAct

1.2.1 Methodology

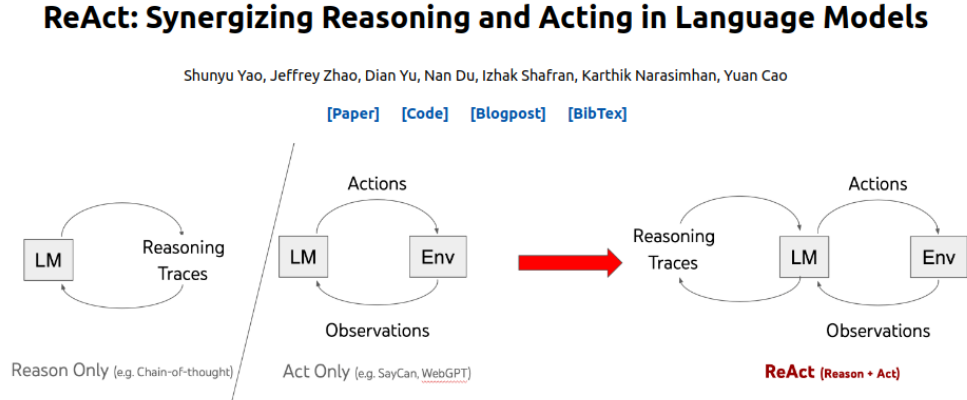


Figure 2: ReAct

ReAct is a prompt engineering method that uses few-shot learning to teach the model how to solve problems. CoT is supposed to imitate how humans think about problems, and ReAct also includes this reasoning element, but it goes further by allowing the agent text actions that let it interact with its environment as well. Humans use verbal reasoning (speaking or thinking) to help us strategize and remember things, but we can also take action to gain more information and achieve our goals. This is the foundation of ReAct. A ReAct prompt includes examples with actions, the observations gained by taking those actions, and the transcribed thoughts (reasoning strategies) of the human at various steps in the process. The LLM learns to emulate this approach of interleaved thinking and acting, making it an agent in its environment. Below is an illustration of how a ReAct agent operates, with a tragic example (in Thoughts -> Action -> Observation order) shown in monospaced font.

It is important to remember that the observations are not generated by the LLM but by the environment, which is a separate module that the LLM can interact with only through specific text actions. Therefore, in order to implement ReAct, you need:

An environment that takes a text action (out of a set of potential actions that can change based on the environment’s internal state) and returns a text observation. for our problem statement, we have made a custom environment consisting of Python functions and classes. An output parser framework that stops the agent from generating text once it has written a valid action, executes that action in the environment and returns the observation (appends it to the text generated so far and prompts the LLM with that). Here langchain is used for that. Human-generated examples of intermixed thoughts, actions, and observations in the environment to use for few-shot learning. Langchain is used for this.

1.2.2 Analysis

Since our problem statement only requires the logic from a query, we have to create a full virtual environment of tools, with which ReAct could act upon to take the next steps. This makes adding new functions and modularity difficult. Also, due to the incomplete and imprecise environment (Since how the tools work is not provided), ReAct hallucinates a lot, gets stuck in infinite loops, and gives wrong arguments. For full logic retrieval, ReAct methodology can only retrieve simpler logic. NOTE - If the original environment in which these functions exist would be given, then this methodology would be very accurate and precise.

1.2.3 Sample Results

Query	Output
Search for objects related to the customer CustomerABC and create action items from the search results	Entering new AgentExecutor chain... Question: What objects are you searching for related to the customer CustomerABC? Thought: I need to use the <code>search_object_by_name</code> tool to find the objects related to the customer. Action: "action": "search_object_by_name", "action_input": "query": "CustomerABC" searched objects. I have found the objects related to the customer CustomerABC. Now I need to create action items from the search results. Action: "action": "create_actionable_tasks_from_text", "action_input": "text": "searched objects" list of actionable tasks. I have created actionable tasks from the search results. Now I can provide the final answer. Final Answer: The actionable tasks created from the search results related to the customer CustomerABC are [list of actionable tasks]. Finished chain.
Search for objects related to ProjectX and summarize them.	Entering new AgentExecutor chain... Question: What objects are you looking for related to ProjectX? Thought: I need to search for objects related to ProjectX. Action "action": "search_object_by_name", "action_input": "query": "ProjectX" searched objects. I have searched for objects related to ProjectX. Action: "action": "summarize_objects", "action_input": "objects": ["object_id_1", "object_id_2", "object_id_3"] ['object_id_1', 'object_id_2', 'object_id_3'] summary of the objects ['summary of the objects'] I have summarized the objects related to ProjectX. Final Answer: The objects related to ProjectX are ['summary of the objects']. Finished chain.

1.3 Platypus

1.3.1 Methodology

The Platypus models are a series of fine-tuned and merged variants based on the LLaMA and LLaMa-2 transformer architectures. Platypus takes advantage of LoRA and PEFT.

All base models are available via [HuggingFace/garage-bAInd](https://huggingface.co/garage-bAInd).

This Technique was used to fine-tune Llama-2-13b with 1k data entries, for our problem statement.

Our methodology employs Low-Rank Approximation (LoRA) for training efficiency, involving freezing pre-trained model weights and incorporating rank decomposition matrices. For dataset and generation techniques, refer to the data folder.

State-of-the-art Parameter-Efficient-Fine-Tuning (PEFT) is utilized for model merging, taking advantage of the PEFT library’s built-in capabilities. Fine-tuning initially focused on attention modules and expanded to other modules based on performance analysis. Hyperparameters were carefully chosen, and Alpaca’s prompt formatting template ensured consistency. The fine-tuning process utilized the Hugging Face transformers library for compatibility and efficiency.

1.3.2 Analysis

Table 1: Hyperparameters used to fine-tune Platypus.

Hyperparameter	Values
learning rate	4e-4
batch size	16
microbatch size	1
warmup steps	100
epochs	1, 5
weight decay	0.
lr scheduler	cosine
lora alpha	16
lora rank	16
lora dropout	0.05
lora target modules	gate_proj, up_proj, down_proj
cutoff length	4096
train on inputs	False
group by length	False
add eos token	False

Table 2: Hardware Specifications for training.

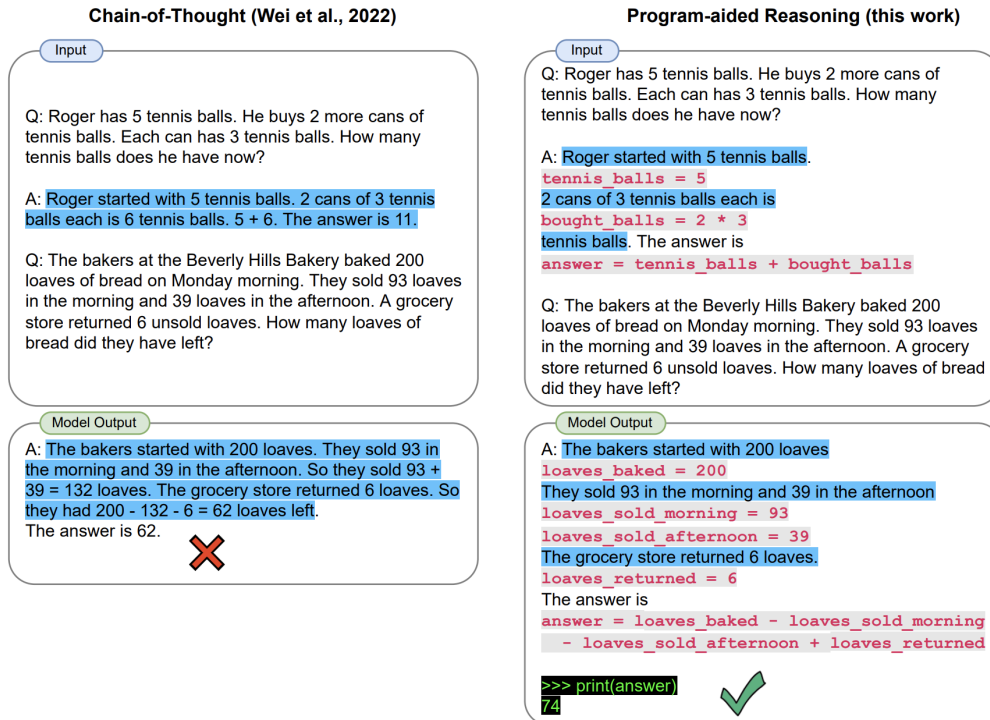
Specification	Values
GPU	A6000 (48 GB VRAM)
Number of GPUs	4
RAM	180 GB
Time taken for training	1 hour
Model size after training	49 GB

1.3.3 Sample Results

Query	Output
Search for objects related to CustomerABC and create action items from the search results.	<pre>['tool_name': 'search', 'arguments': ['argument_name': 'customer', 'argument_value': 'CustomerABC'], 'tool_name': 'action_item', 'arguments': ['argument_name': 'objects', 'argument_value': '\$\$PREV[0]']]</pre>
Find all work items in the testing stage that are related to parts PART-1 and PART-2 and summarize them.	<pre>['tool_name': 'search', 'arguments': ['argument_name': 'stage', 'argument_value': 'testing', 'argument_name': 'related_to', 'argument_value': '\$\$PREV[0]'], 'tool_name': 'summarize', 'arguments': ['argument_name': 'objects', 'argument_value': '\$\$PREV[1]']]</pre>

1.4 PAL

1.4.1 Methodology



PAL is an approach that combines the power of large language models (LLMs) with programmatic reasoning to solve complex tasks.

1. Natural Language Understanding:

The process starts with the LLM reading and understanding the natural language description of the task. This involves parsing the text, identifying key concepts and relationships, and extracting relevant information.

2. Program Generation:

Based on the understanding of the task, the LLM generates a series of interleaved natural language (NL) and programming language (PL) statements. These statements represent the reasoning steps and algorithms needed to solve the problem.

3. Program Execution:

The generated program is then passed to a runtime environment, such as a Python interpreter, for execution. This environment executes the program step-by-step, performing the required calculations and operations. In the context of the problem statement, the execution function does nothing as we require just the generated JSON schema.

4. Solution Output:

After program execution, the results are retrieved and translated back into natural language. This provides the final solution to the original task in a format that is understandable by the user.

Key Differences from Chain-of-Thought (CoT) Prompting:

PAL differs from CoT prompting in several key aspects

- **Intermediate Representation:** PAL uses a combination of NL and PL statements to represent the reasoning steps, while CoT relies solely on NL prompts.
- **Solution Computation:** PAL offloads the solution computation to a dedicated runtime, whereas CoT attempts to solve the entire problem within the LLM itself.
- **Transparency:** PAL's programmatic representation allows for easier debugging and analysis of the reasoning process, which can be difficult with CoT's black-box nature.

Benefits of PAL:

- **Improved Reasoning:** PAL can handle tasks that require complex reasoning and logic, which can be challenging for traditional LLMs.
- **Increased Efficiency:** By offloading computation to the runtime, PAL can solve tasks faster and with less memory usage.
- **Enhanced Accuracy:** The combination of LLM and programmatic reasoning can lead to more accurate and reliable solutions.
- **Better Explainability:** The explicit program representation provides a clear explanation of the reasoning process used to solve the task.
- **Overall,** PAL offers a promising approach for tackling complex NLP tasks by leveraging the strengths of both LLMs and programmatic reasoning.

1.4.2 Analysis

The combination of LLM and programmatic reasoning can lead to more accurate and reliable solutions, especially for tasks with complex logic. The PAL method also consumes a lot of tokens (Large language model API calls) for our problem statement. Although PAL is good for tasks involving complex logic, it's rigid in its methodology and can't be moulded well enough for our problem statement.

1.4.3 Sample Results

Question: Find all tasks related to customer ABC Inc. and summarize them.

Answer:

```
def solution():
    # Find all tasks related to customer ABC Inc. and summarize them.
    tasks = find_tasks(customer="ABC Inc.")
    summary = summarize_tasks(tasks)
    return summary

def find_tasks(customer):
    # code to find all tasks related to the given customer
    tasks = []
    for task in tasks_list:
        if task.customer == customer:
            tasks.append(task)
    return tasks

def summarize_tasks(tasks):
    # code to summarize the tasks
    summary = {}
    for task in tasks:
        if task.status in summary:
            summary[task.status] += 1
        else:
            summary[task.status] = 1
    return summary

# example usage
tasks_list = [
    Task("Task 1", "ABC Inc.", "completed"),
    Task("Task 2", "XYZ Corp.", "in progress"),
    Task("Task 3", "ABC Inc.", "completed"),
    Task("Task 4", "ABC Inc.", "pending"),
    Task("Task 5", "XYZ Corp.", "completed"),
]

print(solution())
```

Listing 1: Python code example

The `solution()` function is used to create our final solution.

1.5 Recursion of Thought

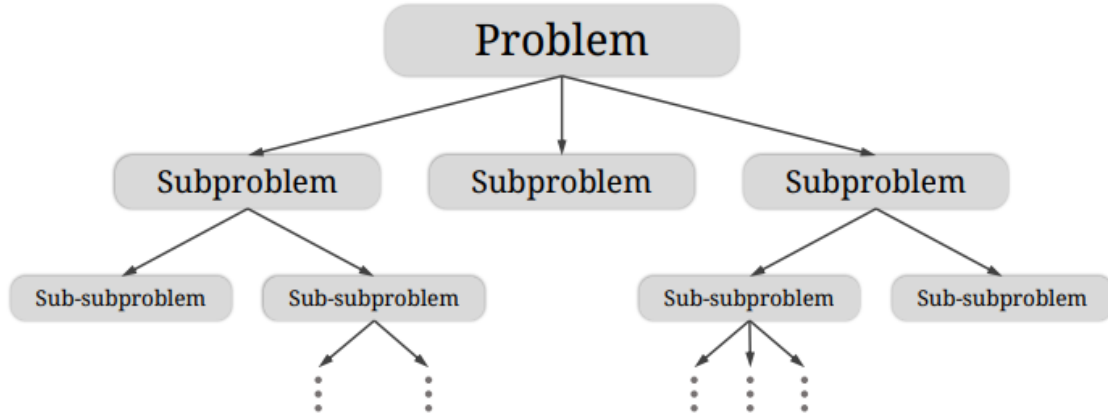


Figure 3: RoT

1.5.1 Methodology

Recursion of Thought is an approach in problem-solving that involves breaking down complex tasks into smaller subproblems and recursively solving them to derive the final solution. There are 4 parts in the methodology used:

1. Problem Definition and Context: The methodology begins with defining the problem or query that requires a solution. Each problem is formulated to include the main question (Q) and the expected answer sequence (A).

2. Special Tokens and Context Division: RoT introduces special tokens, namely GO, STOP, and THINK, to guide the LLM in generating recursive intermediate steps. The GO and STOP tokens mark the start and end of a problem sequence, while THINK initiates a recursion procedure. These tokens facilitate the division of the problem into multiple contexts, allowing the model to handle problems exceeding the maximum context size.

- RoT introduces special tokens that a model can output to control its context:

GO Start of a problem **STOP** End of a problem **THINK** Initiate recursive thinking

- RoT's context structure:



- RoT inference:

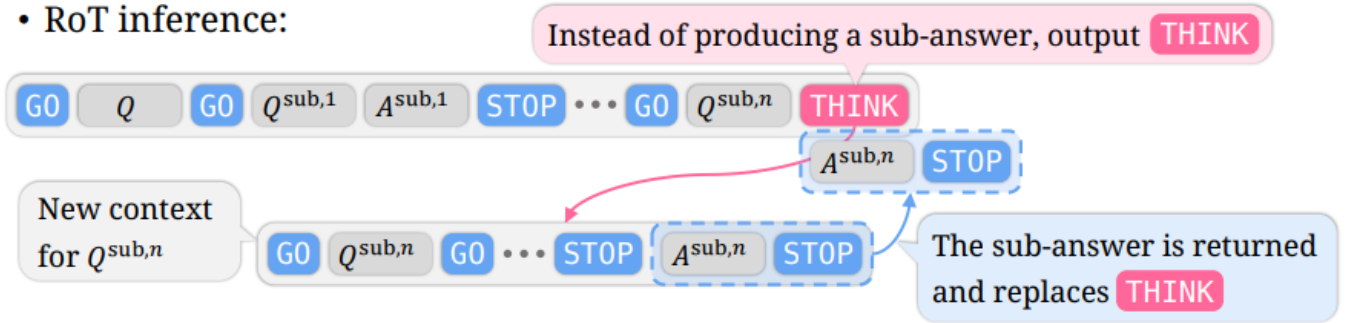


Figure 4: RoT

3. Inference and Context Generation: The LLM is tasked with inferring the probability of the next token given a sequence ($p(x_{i+1} \mid x_{1:i})$). The inference process involves generating contexts (X) based on the concatenation of the main question (Q), subproblems ($Q_{sub,i}$), and their corresponding answers ($A_{sub,i}$). The recursive context control allows the model to create multiple contexts by producing special tokens, effectively dividing the problem into manageable segments.

4. Recursive Problem-Solving: During the inference process, the LLM solves multiple subproblems by generating sub-questions ($Q_{sub,i}$) and their respective answers ($A_{sub,i}$). The key innovation lies in the use of the **THINK** token, which triggers a recursive process, separating the sub-question into a new context. If the new context is a base case, the answer is produced directly; otherwise, the model recursively solves more subproblems.

1.5.2 Analysis

The lengths of the Chain of Thought (CoT) can rapidly expand with the complexity of a problem, often surpassing the maximum context size. Recursion of Thought (RoT), can address this challenge. RoT introduces several special tokens that the models can output to trigger context-related operations. This allows the model to break down a problem into different contexts, offering a more versatile approach to handling complex tasks. RoT demonstrates a remarkable enhancement in LLMs' reasoning capabilities. By introducing special tokens and recursive context control, the technique enables models to break down complex problems into manageable segments. This results

in improved accuracy in solving problems with reasoning steps extending beyond the traditional context limits.

However, the introduction of recursive processes and multiple contexts may incur computational overhead. While RoT demonstrates effectiveness in solving complex problems, it is crucial to consider the potential impact on computational resources, especially when dealing with large-scale reasoning tasks.

1.5.3 Sample Results

1. Divide the problem into subproblems:

- Subproblem 1: Retrieve work items associated with the Rev organisation 'Rev-789'.
- Subproblem 2: Filter work items from Subproblem 1 to only include those that need a response.
- Subproblem 3: Create a summary of the work items obtained from Subproblem 2.

2. Generate GO and STOP tokens:

GO

... Subproblem 1 ...

... Subproblem 2 ...

... Subproblem 3 ...

STOP

3. Solve each subproblem recursively:

1.
 - For Subproblem 1, use the "works_list" tool with the argument "ticket.rev_org" set to "Rev-789."
 - For Subproblem 2, use the "works_list" tool with the argument "ticket.needs_response" set to true and "objects" set to the output of Subproblem 1.
 - Subproblem 3 does not require further division and can be solved directly using the "summarize_objects" tool with the argument "objects" set to the output of Subproblem 2.

4. Use appropriate tools and combine the outputs of Subproblems 1, 2, and 3. Replace the corresponding THINK tokens with the actual results obtained from each subproblem.

The final answer is the summarized list of work items associated with 'REV-789' that need a response.

Technical Report

```
[
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "ticket.rev_org",
        "argument_value": "Rev-789"
      }
    ]
  },
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[0]"
      },
      {
        "argument_name": "ticket.needs_response",
        "argument_value": "true"
      }
    ]
  },
  {
    "tool_name": "summarize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[1]"
      }
    ]
  }
]
```

1.6 Tree of Thought

1.6.1 Methodology

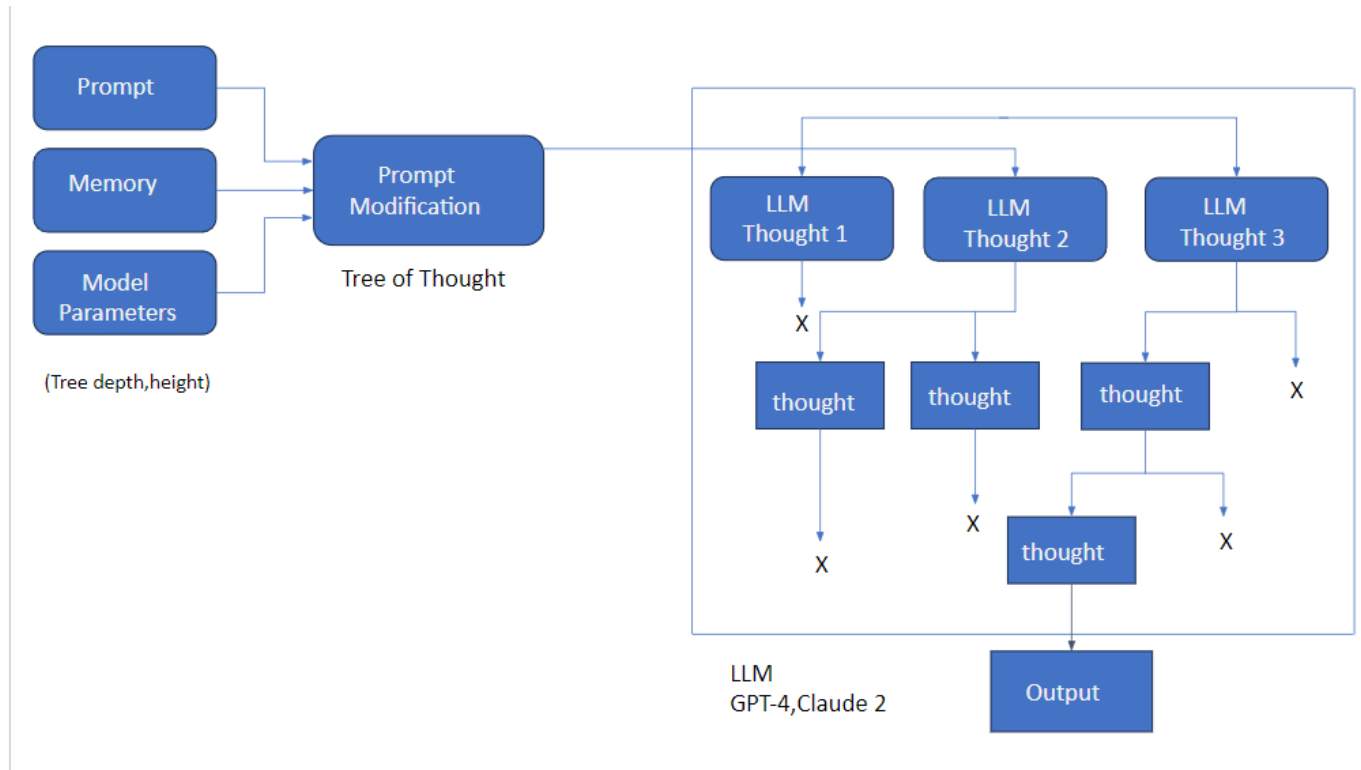


Figure 5: ToT

ToT maintains a tree of thoughts, where thoughts represent coherent language sequences that serve as intermediate steps toward solving a problem. This approach enables an LM to self-evaluate the progress intermediate thoughts make towards solving a problem through a deliberate reasoning process. The LM’s ability to generate and evaluate thoughts is then combined with search algorithms (e.g., breadth-first search and depth-first search) to enable systematic exploration of thoughts with lookahead and backtracking. It follows the following steps -

1. **Thought decomposition** Unlike CoT prompting, ToT explicitly decomposes a problem into intermediate steps or thoughts, which are combined to form a solution to the underlying problem. Depending on the problem, this decomposition can take a variety of different forms, such as outputting a few words or a single line of an equation.

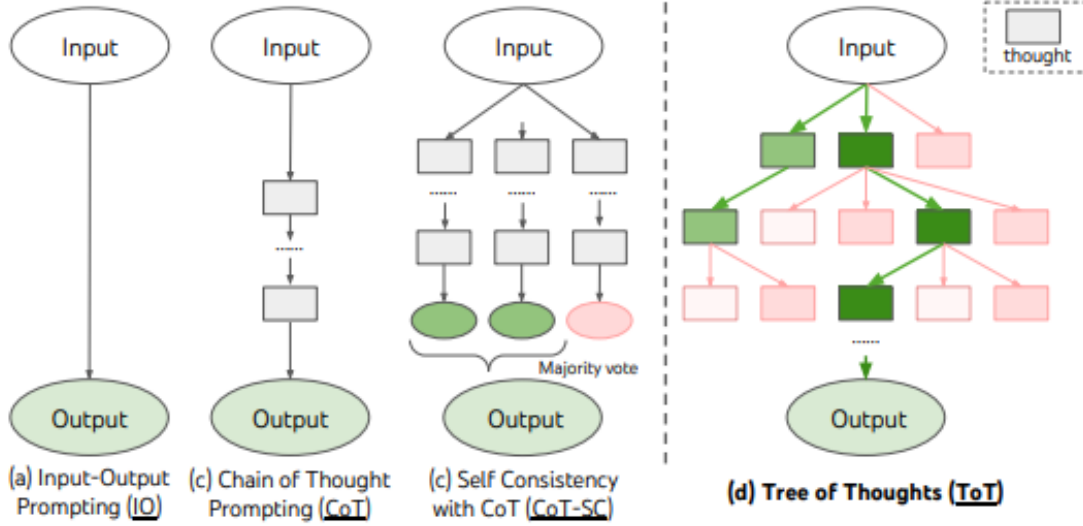


Figure 6: ToT

2. Thought generation Once we have decided what will constitute a thought, we need to determine how thoughts should be generated during ToT prompting. Two basic techniques for thought generation are proposed:

- a) Sampling: generating several thoughts independently with the same prompt
- b) Proposing: generating several thoughts sequentially with a “propose prompt

The sampling approach works best when the thought space is rich, as several independently-generated thoughts are unlikely to be duplicates. If the thought space is more constrained, then the proposing technique can be used to generate several thoughts while avoiding duplicates.

3. State evaluation Once we have defined our thoughts and chosen how they will be generated, we need to define a heuristic for evaluating the quality of certain chains of thought. Otherwise, there is no way to know whether we are making progress towards a final solution. Given several thoughts that have been generated, an LLM is used to reason about the quality of each thought. In particular, two different strategies are followed:

- a) Value: independently assign a scalar value (i.e., rating from 1-10) or classification (i.e., sure, likely, or impossible to reach a solution) to each state.
- b) Vote: compare different solutions and select the most promising one.

Although both approaches can work well, voting is best when a successful solution to a problem is hard to directly value (e.g., creative writing tasks). In both cases, the LLM can be prompted multiple times like self-consistency to achieve more reliable evaluations of each state.

4. Search algorithm The final component of ToT prompting is the search algorithm that is used to explore the solution space.

1.6.2 Specialized Python Classes for our PS

Class: TreeofThoughts This class represents the base class for the Tree of Thoughts search algorithm. It contains the following methods:

- `__init__(self, model)`: Initializes the TreeofThoughts object with the given model.
- `save_tree_to_json(self, file_name)`: Saves the tree to a JSON file with the given file name.
- `logNewState(self, state, evaluation)`: Logs a new state and its evaluation to the tree.
- `adjust_pruning_threshold_precentile(self, evaluated_thoughts, percentile)`: Adjusts the pruning threshold based on the percentile of evaluated thoughts.
- `adjust_pruning_threshold_moving_average(self, evaluated_thoughts, window_size)`: Adjusts the pruning threshold based on the moving average of evaluated thoughts.

Class: TreeofThoughtsBFS This class represents the Breadth-First Search (BFS) variant of the Tree of Thoughts search algorithm. It inherits from the TreeofThoughts class and contains the following method:

`solve(self, initial_prompt, num_thoughts, max_steps, max_states, value_threshold, pruning_threshold=0.5)`

Class: TreeofThoughtsDFS This class represents the Depth-First Search (DFS) variant of the Tree of Thoughts search algorithm. It inherits from the TreeofThoughts class and contains the following method:

`solve(self, initial_prompt, num_thoughts, max_steps, value_threshold, pruning_threshold=0.5)` Solves the problem using DFS with the given parameters.

Class: TreeofThoughtsBEST This class represents the Best-First Search variant of the Tree of Thoughts search algorithm. It contains the following methods:

- `__init__(self, model)`: Initializes the TreeofThoughtsBEST object with the given model.
- `save_tree_to_json(self, file_name)`: Saves the tree to a JSON file with the given file name.
- `log_new_state(self, state, evaluation)`: Logs a new state and its evaluation to the tree.
- `solve(self, initial_prompt, num_thoughts, max_steps, pruning_threshold)`: Solves the problem using Best-First Search with the given parameters.

Class: TreeofThoughtsASearch This class represents the A* Search variant of the Tree of Thoughts search algorithm. It contains the following methods:

- `__init__(self, model)`: Initializes the TreeofThoughtsASearch object with the given model.
- `solve(self, initial_prompt, num_thoughts=5, max_steps=30, pruning_threshold=0.4)`: Solves the problem using A* Search with the given parameters.
- `is_goal(self, state, score)`: Determines if the given state is a goal state based on its score.
- `reconstruct_path(self, came_from, current_state, initial_prompt)`: Reconstructs the path from the initial state to the current state using the `came_from` dictionary.

Class: MonteCarloTreeofThoughts This class represents the Monte Carlo Tree Search variant of the Tree of Thoughts search algorithm. It inherits from the TreeofThoughts class and contains the following methods:

- `__init__(self, model, objective="balance")`: Initializes the MonteCarloTreeofThoughts object with the given model and objective.

- `optimize_params(self, num_thoughts, max_steps, max_states)`: Optimizes the search parameters based on the objective.

- `solve(self, initial_prompt, num_thoughts, max_steps, max_states, pruning_threshold)` Solves the problem using Monte Carlo Tree Search with the given parameters:

- `monte_carlo_search(self, initial_prompt, num_thoughts, max_steps, max_states, pruning_threshold)` Performs the Monte Carlo Tree Search with the given parameters.

Solves the problem using an optimized search algorithm with the given parameters.

For detailed documentation on how to use these classes to get the results, refer to the README.md of docs under the `Tree_of_Thought_Experiment` folder

1.6.3 Analysis

Tree-of-Thought (ToT) Prompting is an innovative technique that builds upon the principles of the Tree-of-Thoughts framework and expands the capabilities of the well-known Chain-of-Thought prompting concept. Adopting this approach, empowers Large Language Models, such as ChatGPT, and Bard to demonstrate advanced reasoning abilities. The Tree-of-Thought Prompting technique enables these models to autonomously rectify errors and continuously accumulate knowledge, resulting in enhanced performance and improved decision-making. Tree of thought gives the best accuracy of all the methods and even solves queries that involve some additional logic like combining the outputs of various functions, like mathematical operations, iterations, conditional logic etc and solves the bonus task.

The only downside is it makes a lot of requests and consumes a high number of tokens, which makes it unfeasible for our problem.

1.6.4 Sample Results

Query	Output
Search for objects related to ProductABC, retrieve work items related to the search results, and then add them to the current sprint.	<pre>['tool_name': 'search_object_by_name', 'arguments': ['argument_name': 'query', 'argument_value': 'ProductABC'], 'tool_name': 'works_list', 'arguments': ['argument_name': 'applies_to_part', 'argument_value': ['\$\$PREV[0]']], 'tool_name': 'get_sprint_id', 'arguments': [], 'tool_name': 'add_work_items_to_sprint', 'arguments': ['argument_name': 'work_ids', 'argument_value': ['\$\$PREV[1]'], 'argument_name': 'sprint_id', 'argument_value': '\$\$PREV[2]']]</pre>
Retrieve all issue work items that need a response and are associated with the Rev organization REV-789	<pre>['tool_name': 'works_list', 'arguments': ['argument_name': 'ticket.needs_response', 'argument_value': true, 'argument_name': 'ticket.rev_org', 'argument_value': 'REV-789', 'argument_name': 'type', 'argument_value': 'issue']]</pre>

2 Final Approach

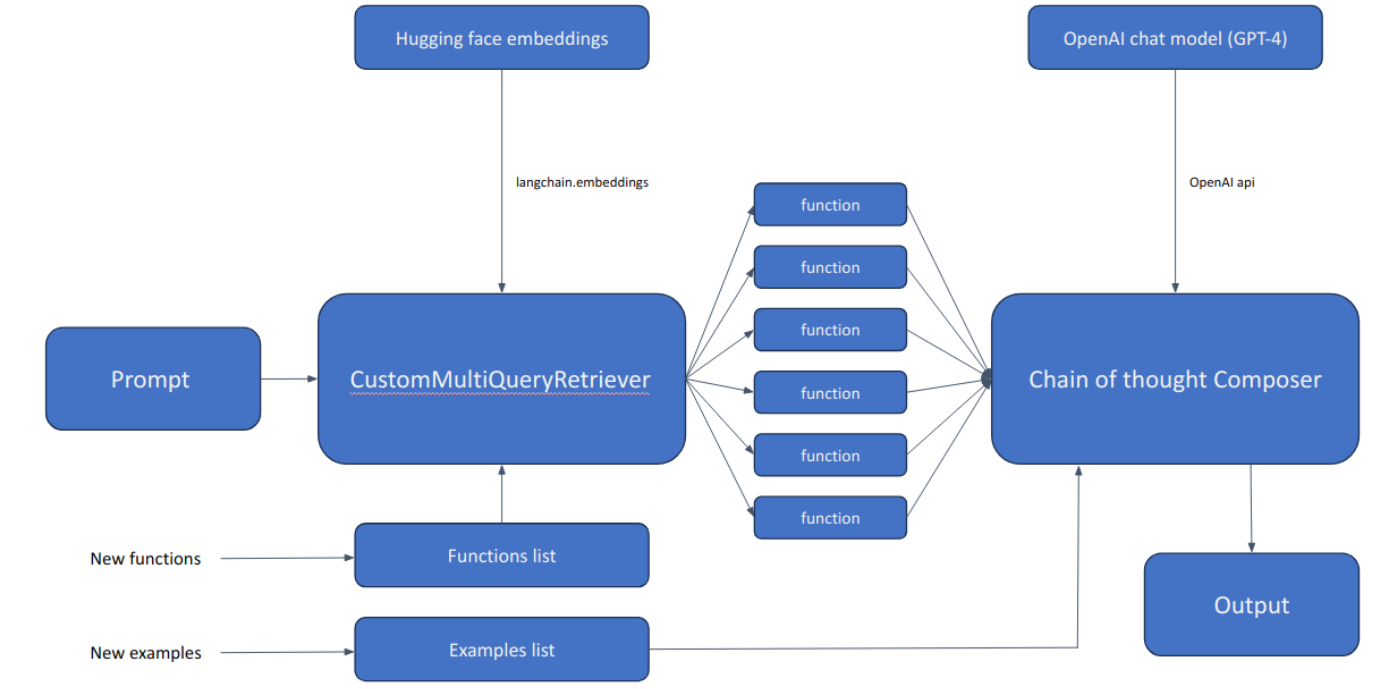


Figure 7: Final Approach

For our final approach, we used a retriever-composer approach. Multiple retrievers and composers were tested, and the best combination for our problem statement is our own Custom Multi Query retriever, and a Chain of Thought composer.

2.1 Retrievers

Multiple retrievers based on different LLMs and methods were tested. some of them are -

2.1.1 Vector Store Based Retriever

A vector store retriever is a retriever that uses a vector store to retrieve documents. It uses the search methods implemented by a vector store, like similarity search and MMR, to query the texts in the vector store.

2.1.2 Custom Contextual Compression Retriever

This retriever is a variation of the contextual Compression Retriever, which works on this principle - instead of immediately returning retrieved documents as-is, you can compress them using the context of the given query so that only the relevant information is returned. “Compressing” here refers to both compressing the contents of an individual document and filtering out documents wholesale.

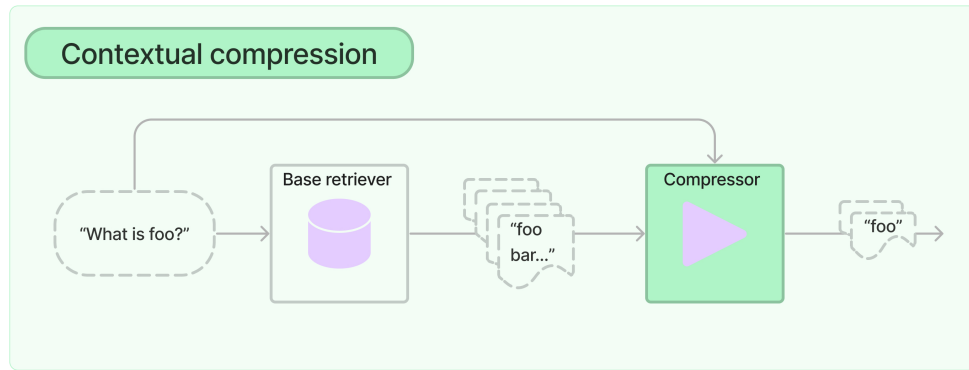


Figure 8: Contextual Copmression

To use the Contextual Compression Retriever, we need a base retriever (Here openAI or HuggingFace LLMs are used) and a document compressor.

The Contextual Compression Retriever passes queries to the base retriever, takes the initial documents and passes them through the Document Compressor. The Document Compressor takes a list of documents and shortens it by reducing the contents of documents or dropping documents altogether.

2.2 Composers

2.2.1 ChainOfThoughts Composer

The Chain of Thought (CoT) model represents a novel approach to query answering that emphasizes reasoning and transparency. It leverages the capabilities of large language models (LLMs) by providing them with a structured framework for understanding the problem space and identifying the most suitable solution.

The CoT framework relies on a knowledge base containing information about various tools and their functionalities. Each tool is described by its name, arguments, descriptions, and examples. This knowledge base empowers the LLM to understand the available resources and their capabilities.

When presented with a user query, the CoT model analyzes it to understand the task and the desired output format. Then, it engages in reasoning about the query based on its knowledge of the available tools. It considers the functionalities of each tool and its arguments to identify a chain of operations that can solve the query. This chain represents a sequence of tools, each performing a specific task with specific arguments.

Once the chain of tools is determined, the LLM generates the final output in the desired format. This output typically includes a JSON structure containing information about the tools used, their arguments, and their order in the chain.

Langchain integrates the Chain of Thought (CoT) framework to empower large language models (LLMs) with a structured approach to query answering. The LLM first analyzes the query and then

builds a chain of tools based on its knowledge base. Each tool plays a specific role in achieving the desired outcome. The LLM then executes the chain and captures the entire process, providing transparency into its reasoning. Finally, Langchain analyzes the execution and refines the knowledge base to enhance the LLM's capabilities over time.

2.3 LLM's

We use OpenAI's GPT-4 and HuggingFace embeddings for our final approach. Although we have experimented with other LLMs such as OpenAI's GPT-3.5, HuggingFace's Llama2, zephyr-7b-alpha and Open-Platypus. We also tried and tested OpenAI's embeddings.

3 Final Approach documentation

3.1 Introduction

This library is an implementation of the problem statement. Here we use gpt-4 as the primary backend and langchain as the framework to achieve our results. The task is divided into retrieval and composition. The retrieval part is done using langchain's retrievers and the composition part is done using langchain's composers. The output of the composer is parsed using the output parser.

3.2 Prerequisites

- Python 3.10
- openai
- langchain
- sentence_transformers (optional)
- huggingface_hub (optional)

3.3 Example

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.chat_models import ChatOpenAI
```

```
from src.retrievers import VectorStoreRetriever
from src.composers import ChainOfThoughtComposer
from src.functions import devrev_functions
from src.examples import example_queries
```

```
# Initialize the chat object
```

```
chat_llm = ChatOpenAI(
```

Technical Report

```
    openai_api_key = "<YOUR_API_KEY>" ,
    temperature=0.7,
)

# Generate Examples

example_str = "".join([f"Query: {query}-Answer:\n-{functions}" for
query, functions in example_queries[:4]])

# Retrievers can be chained together to form a pipeline ,
with vs_retriever being the most basic

vs_retriever = VectorStoreRetriever(
    embeddings=OpenAIEmbeddings(
        openai_api_key = "<YOUR_API_KEY>"
    ),
    name = "vs_rtr_01",
    init_functions=devrev_functions.copy()
)

cmq_retriever = CustomMultiQueryRetriever(
    chat_llm = ChatOpenAI(
        openai_api_key = "<YOUR_API_KEY>" ,
        temperature=0.7
    ),
    vector_store=vs_retriever
)

cot_composer = ChainOfThoughtComposer(
    chat_llm=chat_llm ,
)

def agent(query):
    # Retrieve functions
    retrieved_functions = cmq_retriever.retrieve(query)

    # Compose answer
    answer = cot_composer(query, retrieved_functions, example_str)

    # Return answer
    return answer['text']
```

3.4 Documentation

3.4.1 src

It is the core library that contains the retrievers, composers, and a few functions along with some base examples to begin with

- **src.retrievers**

It contains the retrievers that are used to retrieve the functions that are similar to the query. They can cascade into each other to form a pipeline.

- **src.retrievers.VectorStoreRetriever**

It is the most basic FAISS based retriever that uses the embedding similarities to retrieve relevant functions.

init

Initializes the retriever

* **embeddings:** `langchain.embeddings`: The embeddings that are used to generate the embeddings of the functions and the query

* **name:** `str`: The name of the retriever, used for caching

* **init_functions:** `list[dict]`: The functions that are used to initialize the retriever

add_functions Adds the functions to the retriever

* **functions:** `list[dict]`: The functions that are added to the retriever

find_functions Returns the functions that are similar to the query

* **query:** `str`: The query that is used to retrieve the functions

get_retriever Returns the retriever object

- **src.retrievers.CustomMultiQueryRetriever**

It is a retriever that uses the chat model to break the query into multiple queries and then uses the vector store retriever to retrieve the functions.

init Initializes the retriever

- **chat_llm:** `langchain.chat_models`: The chat model that is used to generate the query

- **vector_store:** `src.retrievers.VectorStoreRetriever`: The vector store retriever that is used to retrieve the functions

find_functions Returns the functions that are similar to the query

- **query:** `str`: The query that is used to retrieve the functions

- **src.retrievers.CustomContextualCompressionRetriever**

It is a retriever that uses contextual compression to store documents and infer the query. It uses the vector store retriever to retrieve the functions.

`_init_` Initializes the retriever

- `chat_llm: langchain.chat_models`: The chat model that is used to generate the query
- `vector_store: src.retrievers.VectorStoreRetriever`: The vector store retriever that is used to retrieve the functions

`find_functions` Returns the functions that are similar to the query

- `query: str`: The query that is used to retrieve the functions

src.composers

It contains the composers that are used to generate the answer based on the query and the retrieved functions. These composers are built upon langchain's LLMChain classes and can inherit directly from them.

- **src.composers.ChainOfThoughtComposer**

This uses chain of thought to compose the retrieved function into a meaningful task as specified by the query.

`_init_`

- `chat_llm: langchain.chat_models`: The chat model that is used to generate the answer

`_call_`

- `query: str`: The query that is used to generate the answer
- `functions: list[dict] | list[Document()]`: The functions that are retrieved by the retriever
- `examples: str`: The examples that are used to generate the answer, from `src.examples`

src.functions

It contains the functions that are used to initialize the retrievers. The functions are taken from the dataset provided by Devrev

src.examples

It contains the examples that are passed along with the composer, it is a list of query-result pairs that are used to generate the answer. The ideal way to process it is:

```
example_str = """.join([f"Query: {query}-Answer:\n-{functions}" for query, funct
```

However, the user is free to process it however they want.

4 Sample Results

Query:

check if there are greater than 10 items in the list of work items similar to
don:core:dvr-v-us-1:devo/0:issue/1

Result:

```
[
  {
    "tool_name": "get_similar_work_items",
    "arguments": [
      {
        "argument_name": "work_id",
        "argument_value": "don:core:dvr-v-us-1:devo/0:issue/1"
      }
    ]
  },
  {
    "tool_name": "count",
    "arguments": [
      {
        "argument_name": "items",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
  {
    "tool_name": "Greater_than",
    "arguments": [
      {
        "argument_name": "left_value",
        "argument_value": "$$PREV[1]"
      },
      {
        "argument_name": "right_value",
        "argument_value": "10"
      }
    ]
  }
]
```

Stats:

Prompt Tokens: 3339

Technical Report

Completion Tokens: 250

Total Cost: \$0.0055084999999999995

Query:

Summarize high severity tickets from the customer UltimateCustomer

```
[
  {
    "tool_name": "search_object_by_name",
    "arguments": [
      {
        "argument_name": "query",
        "argument_value": "UltimateCustomer"
      }
    ]
  },
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "ticket.severity",
        "argument_value": [
          "high"
        ]
      },
      {
        "argument_name": "ticket.rev_org",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
  {
    "tool_name": "summarize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[1]"
      }
    ]
  }
]
```

Technical Report

Stats:

Prompt Tokens: 2900

Completion Tokens: 258

Total Cost: \$0.0048660000000000005

Query: List all high severity tickets coming in from slack from customer Cust123 and generate a summary of them.

Result:

```
[
  {
    "tool_name": "search_object_by_name",
    "arguments": [
      {
        "argument_name": "query",
        "argument_value": "Cust123"
      }
    ]
  },
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "ticket.severity",
        "argument_value": [
          "high"
        ]
      },
      {
        "argument_name": "ticket.source_channel",
        "argument_value": [
          "slack"
        ]
      },
      {
        "argument_name": "ticket.rev_org",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
]
```

Technical Report

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

Stats:

Prompt Tokens: 3094

Completion Tokens: 313

Total Cost: \$0.005267

5 Bonus points formulation

We used custom functions for bonus points -

Tool- _Name	Tool_Description	Argument- _Value	Argument- _Type	Argument_Description
equal_to	Checks if two values are equal.	left_value, right_value	string, string	The left-hand side value for comparison. The right-hand side value for comparison.
logical_and	Performs a logical AND operation on two conditions.	condition1, condition2	boolean, boolean	The first condition for the logical AND operation. The second condition for the logical AND operation.
logical_or	Performs a logical OR operation on two conditions.	condition1, condition2	boolean, boolean	The first condition for the logical OR operation. The second condition for the logical OR operation.
logical_not	Performs a logical NOT operation on a condition.	condition	boolean	The condition to negate using logical NOT.
Lesser_than	Checks if the left value is lesser than the right value.	left_value, right_value	int, int	The left-hand side value for comparison. The right-hand side value for comparison.
Greater- _than	Checks if the left value is greater than the right value.	left_value, right_value	int, int	The left-hand side value for comparison. The right-hand side value for comparison.
Equal_to	Checks if the left value is equal to the right value.	left_value, right_value	int, int	The left-hand side value for comparison. The right-hand side value for comparison.
resolution- _status	Gives an array of only unresolved issues.	unresolved- _issue	array	An array of only unresolved issues.
length	Returns the length of an array.	array	array	The array for which to determine the length.
if_else- _condition	Executes a conditional statement with if-else logic.	Function, If_true, If_false	string, string, string	The conditional function or expression. Action to be taken if the condition is true. Action to be taken if the condition is false.

Bonus Functions

6 Bonus points queries

e.g

Query = " If there are more than 10 high-priority tasks,
prioritize them; else, get the current user's ID"

Output

```
[
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "issue.priority",
        "argument_value": "high"
      },
      {
        "argument_name": "limit",
        "argument_value": 10
      }
    ]
  },
  {
    "tool_name": "length",
    "arguments": [
      {
        "argument_name": "array",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
  {
    "tool_name": "Greater_than",
    "arguments": [
      {
        "argument_name": "left_value",
        "argument_value": "$$PREV[1]"
      },
      {
        "argument_name": "right_value",
        "argument_value": 10
      }
    ]
  },
]
```

```
{
  "tool_name": "if_else_condition",
  "arguments": [
    {
      "argument_name": "Function",
      "argument_value": "$$PREV[2]"
    },
    {
      "argument_name": "If_true",
      "argument_value": "$$NEXT[0]"
    },
    {
      "argument_name": "If_false",
      "argument_value": "$$NEXT[1]"
    }
  ]
},
{
  "tool_name": "prioritize_objects",
  "arguments": [
    {
      "argument_name": "objects",
      "argument_value": "$$PREV[3]"
    }
  ]
},
{
  "tool_name": "who_am_i",
  "arguments": []
}
]
```

7 Applet

AI Agent 007 is a Streamlit application designed to leverage OpenAI's GPT-3 model for answering queries based on a set of predefined functions. The application utilizes the Langchain library for embeddings and chat models, creating a seamless pipeline for intelligent query responses.

8 Applet documentation - AI Agent 007

8.1 Overview

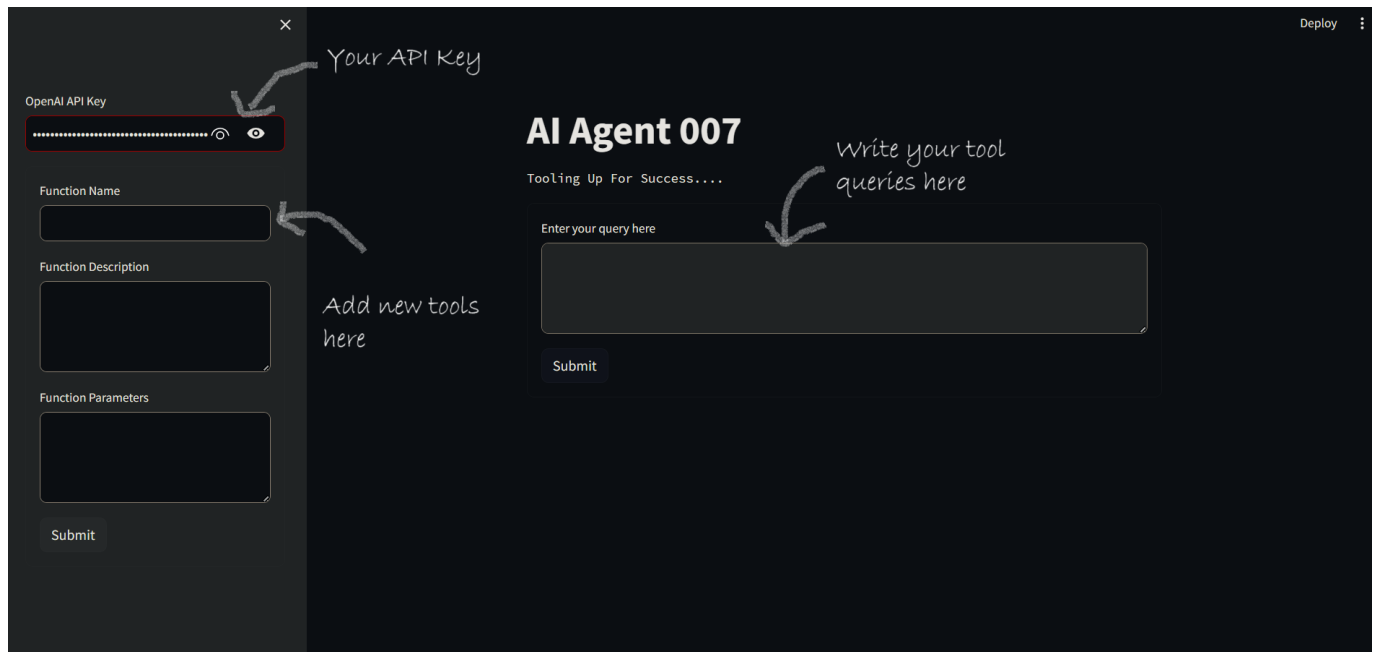


Figure 9: Query Retrieval

AI Agent 007 is a powerful Streamlit application designed to leverage OpenAI's GPT-3 model for answering queries based on a predefined set of functions. The application utilizes the Langchain library, incorporating embeddings and chat models to enhance its capabilities.

8.2 Installation

1. Clone the repository.
2. Install the required dependencies using pip:

```
pip install -r requirements.txt
```

8.3 Application Composition

The application is structured with a user-friendly interface and robust backend components:

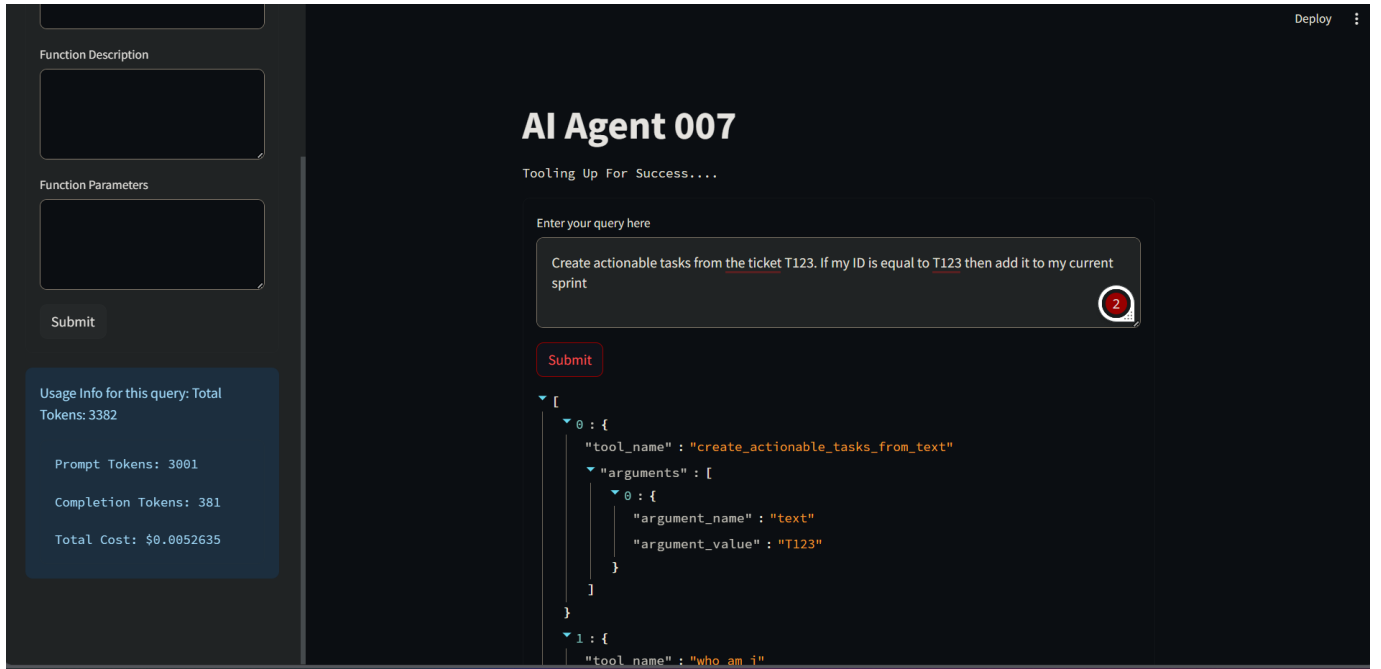


Figure 10: Query Retrieval Example 1

8.3.1 Framework Requirements

- **Streamlit:** Provides the interactive web interface.
- **json:** Used for handling JSON data.
- **langchain:** Incorporates Langchain library for embeddings and chat models.
- **os:** Handles operating system functionalities.
- **logging:** Implements logging for tracking and debugging.
- **datetime:** Manages date and time-related operations.

8.3.2 Library Requirements

- **VectorStoreRetriever:** Retrieves functions based on vector embeddings.
- **CustomMultiQueryRetriever:** Utilizes the chat model to break queries into multiple parts.
- **ChainOfThoughtComposer:** Composes answers using a chain of thought approach.
- **devrev_functions:** Predefined functions for initialization.
- **example_queries:** Example queries for testing.



Figure 11: Query Retrieval Example 2

8.4 Usage

1. **OpenAI API Key:** Enter your OpenAI API Key in the sidebar.
2. **Function Addition:** Use the sidebar form to add custom functions.
 - Enter the function name, description, and parameters.
 - Click 'Submit' to add the function.
3. **Chat Interface:** Input queries in the designated text area and click 'Submit'.
4. **Response Display:** View the JSON response, usage information, token count, and cost in the main window.

8.5 Application Workflow

1. **Initialization:**
 - The application initializes the Langchain chat model and retriever components.
 - Functions are loaded from the `devrev_functions` dataset.
2. **User Interaction:**
 - Users can add custom functions through the sidebar form.
 - Queries are entered into the chat interface.



Figure 12: Query Retrieval Example 3

3. Processing:

- The application employs the `CustomMultiQueryRetriever` to find functions similar to the query.
- The `ChainOfThoughtComposer` is then used to generate an answer based on the query and retrieved functions.

4. Results Display:

- The JSON response is displayed in the main window.
- Usage information, including token counts, cost, and additional metrics, is shown in the sidebar.

8.6 Note

Ensure that you have a valid OpenAI API Key to use this application. The key should start with "sk-".

9 References

1. <https://arxiv.org/pdf/2304.11477.pdf>
2. <https://github.com/Cranial-XIX/llm-pddl>
3. <https://arxiv.org/abs/2210.03629>
4. https://python.langchain.com/docs/modules/agents/agent_types/react
5. <https://arxiv.org/abs/2308.07317>
6. <https://github.com/arielnlee/Platypus>
7. <https://arxiv.org/pdf/2211.10435.pdf>
8. <https://github.com/reasoning-machines/pal>
9. <https://arxiv.org/abs/2306.06891>
10. <https://github.com/soochan-lee/RoT>
11. <https://arxiv.org/abs/2306.06891>
12. <https://github.com/kyegomez/tree-of-thoughts>
13. <https://huggingface.co/HuggingFaceH4/zephyr-7b-alpha>
14. <https://platform.openai.com/docs/introduction>
15. <https://huggingface.co/meta-llama/Llama-2-13b>
16. <https://pypi.org/project/sentence-transformers/>
17. <https://huggingface.co/docs/hub/index>
18. https://python.langchain.com/docs/modules/data_connection/retrievers/contextual_compression/
19. https://python.langchain.com/docs/modules/data_connection/retrievers/contextual_compression/
20. https://python.langchain.com/docs/modules/data_connection/retrievers/MultiQueryRetriever
21. <https://streamlit.io/>