# DevRev's AI Agent 007
# Tooling up for Success

Mid Evaluation Report

**Team 22**


Inter IIT Tech Meet 12.0

# 1 Introduction

In today's era of conversational AI, the ability of language models to understand and respond to user queries is of the greatest importance. Very often, the user query needs an invocation of an API to complete some specific tasks. Tool selection refers to the process of choosing and determining which tools are most suitable for a particular task, based on the query, the tool name, the description of the tool, and its arguments. The problem statement deals with the tool selection given a dynamic list of tools and user query prompt. The objective is to accurately list the tool(s) with their respective argument values as the query provides. This report documents the literature review, experimentation, and benchmarking of the existing LLMs, as well as the future work.

# 2 Literature Review

## 2.1 ToolLLM

ToolLLM [1] is a general tool-use framework encompassing data construction, model training, and evaluation.

For data construction, the paper proposes, ToolBench an instruction-tuning dataset for tool use, constructed automatically using ChatGPT. The construction of ToolBench involves three key phases: API Collection: where 16,464 real-world REST APIs are gathered from RapidAPI across 49 categories; Instruction Generation, where ChatGPT creates diverse instructions for sampled APIs, covering both single and multi-tool scenarios; and then using it to search for a valid solution path (chain of API calls) for each instruction using a novel method named depth-first search-based decision tree (DFSDT).

In classical DFS algorithms, multiple child nodes are generated at each step, then those are sorted, and the highest-scoring node for expansion is selected. After greedily expanding to the terminal node, DFS backtracks to explore nearby nodes, expanding the search space. Throughout the algorithm, the most resource-intensive part is the sorting process of child nodes. If we use an LLM to evaluate two nodes at a time, it requires approximately O(n log n) complexity of OpenAI API calls, where n is the number of child nodes. We find empirically that in most cases, the highest nodes are often the nodes generated at first. Therefore, we skip the sorting process of child nodes and choose a pre-order traversal (a variant for DFS) for the tree search. This design has the following advantages:

- If the model does not retract an action (e.g., for the case of simple instructions), then DFSDT degrades to ReACT, which makes it as efficient as ReACT.
- After the algorithm finishes, the nodes explored by this method are almost the same as those found by a classical DFS search. Hence, it can also handle complex instructions that only DFS can solve.

Overall, this design achieves a similar performance as DFS while significantly reducing costs. It should also be noted that ReACT can be viewed as a degraded version of DFSDT. Therefore, although ToolLLaMA is trained on data created by DFSDT, the model can be used either through ReACT or DFSDT during inference.

For Model Training, based on ToolBench, authors fine-tune LLaMA [2] to obtain an LLM ToolLLaMA. Additionally, they train a neural API retriever, leveraging Sentence-BERT on BERTBASE to streamline API selection for ToolLLaMA. Trained on examples from ToolLLaMA's instruction set sampled a few random API examples, the retriever outperforms BM25 and OpenAI's text-embedding-ada-002 baselines in retrieval performance. This applicability of the retriever could be applied to our use case to retrieve the tools most relevant to the given user query.

For Model Evaluation, ToolEval is based on ChatGPT, which incorporates two evaluation metrics

- Pass Rate: it calculates the proportion of successfully completing an instruction within limited budgets. The metric measures the executability of instructions for an LLM and can be seen as a basic requirement for ideal tool use;
- Win Rate: authors provide instruction and two solution paths to the ChatGPT evaluator and obtain its preference (i.e., which one is better).

The authors pre-define a set of criteria for both metrics, and these criteria are organized as prompts for our ChatGPT evaluator. Multiple evaluations are conducted on ChatGPT to enhance reliability, and average results are calculated for analysis.

Besides, ToolLLaMA exhibits robust generalization to previously unseen APIs, requiring only the API documentation to adapt to new APIs effectively. Therefore, the DFSDT prompting strategy could be worthwhile, considering the dynamic nature of the toolset.

The paper then discusses the experimentation results, trying out ChatGPT, Text-Davinci-003, GPT-4, and Claude-2 as baselines and applying both DFSDT and ReACT for inference.ToolLLaMA + DFSDT demonstrates competitive generalization performance in all scenarios, achieving a pass rate second to GPT4 + DFSDT. Further, ToolLLaMA also demonstrates strong zero-shot generalization ability in an out-of-distribution tool-use dataset: APIBench. Despite not training on any of the APIs or instructions on APIBench, ToolLLaMA performs on par with Gorilla, a pipeline specifically designed for APIBench.

## 2.2 MetaTool

In scenarios where LLMs serve as intelligent agents, as seen in applications like AutoGPT and MetaGPT, LLMs are expected to engage in intricate decision-making processes that involve deciding whether to employ a tool and selecting the most suitable tool(s) from a collection of available tools to fulfill user requests. Therefore, MetaTool [3] which is a benchmark designed to evaluate whether LLMs have tool usage awareness and can correctly choose tools.

The process of using tools can be divided into four stages: Firstly, LLMs consider whether to employ a tool (1) and, if so, which tools to select (2). The tool selection process involves directly having LLMs choose from a provided tool list or selecting via a retriever. Next, LLMs configure the users' input as tool parameters (3), then handle the results from the tool (4), and finally return the outcomes to the user.

Current studies have proposed several benchmarks for tool usage for LLMs, with the main contributions being limited to stages 3 and 4. Previous research proposed datasets lacked diverse user inputs, making it hard to cover various real-world scenarios. Additionally, there is an issue of overlapping in the dataset, meaning that a user's needs can be addressed by more than one tool, which makes it challenging to conduct evaluations since user inputs can correspond to multiple tools. The second aspect is the task setting; the benchmark should include different tasks to evaluate LLMs from different perspectives, such as the reliability of the performance under different scenarios in daily life. To address these issues, paper proposes the following:

- TOOLE dataset: A comprehensive dataset that encompasses a wide range of user queries, with both single-tool and multi-tool queries. These queries are generated using various prompting methods. Moreover, to address the challenge of overlapping tool functionality, authors undertake tool merging and decomposition. They created embeddings for tool descriptions and used hierarchical clustering to find patterns, then manually merged or split tools based on functionality, guided by human expertise and practical considerations.
- Evaluation on awareness of tool usage and tool selection: The authors of the paper construct a test set to evaluate the awareness of tool usage based on TOOLE and existing instruction datasets.

In the evaluation of tool usage awareness, the metrics employed include accuracy, recall, precision, and F1 score. The findings suggest that the majority of these models exhibit suboptimal awareness of tool usage. ChatGPT has the best performance in this regard; however, the accuracy remains fairly low. In the context of tool selection evaluation, the authors introduce the Correct Selection Rate (CSR) as a metric to determine the percentage of correct selection actions. Authors formulate below four distinct tasks to evaluate the tool selection ability of LLMs :

1. High-Level Semantic Comprehension
   (a) Vicuna-7b and ChatGPT demonstrate commendable performances, each achieving around a 70% Correct Selection Rate (CSR).
   (b) Other Large Language Models (LLMs) exhibit CSR values ranging between 45% and 60%.
2. Tool Selection in Specific Scenarios
   (a) Vicuna-33b, among open-source models, distinguishes itself and even surpasses ChatGPT.
   (b) ChatGPT displays notable stability, maintaining consistent performance with only a minor CSR decline as the size of tool lists increases.
3. Exploration of Internal Hallucination and Reliability

(a) Most LLMs face challenges in accomplishing this task, struggling to recognize the absence of necessary tools in the provided list to address queries.

(b) Seven LLMs achieve a CSR of less than 10%, highlighting difficulties in internal hallucination and reliability. In contrast, ChatGPT maintains a relatively robust performance with a CSR of 50.35%, and Baichuan2 attains a 32.26% CSR.

4. Evaluation of Inference Ability

(a) ChatGPT excels in this task with a CSR exceeding 80%, showcasing strong inference ability, particularly in determining the order of using multiple tools.

(b) Baichuan2 performs less favorably, recording the lowest CSR of less than 20% in this task.

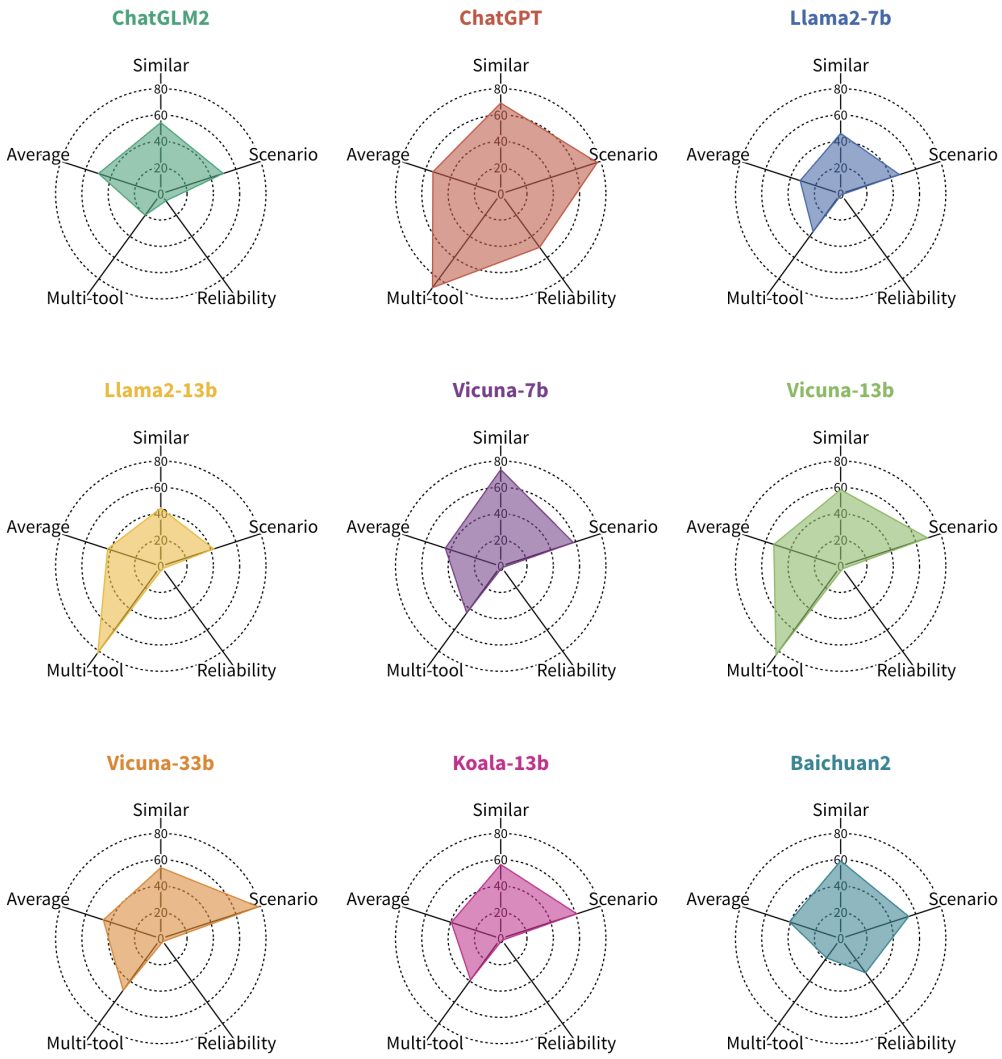


Figure 1: MetaTools evaluation results[1]

Finally, the paper suggests the following insights for tool developers:

1. The more detailed the description, the more efficient tool selection, indicating that detailed descriptions can help LLMs better understand the functionality of tools, thus improving the accuracy of tool selection.

2. Tool descriptions generated by ChatGPT are better than those provided by tool developers since descriptions

[1]Figure from the original paper

written by ChatGPT have higher quality and are more easily understood by LLMs.

In addition, the authors show that adding another dispatcher LLM enables on-the-fly tool creation and usage. The dispatcher screens each incoming task instance. If an appropriate tool is available for the task, the dispatcher directs the task to the tool user for resolution. However, if no suitable tool exists, the task is sent to the tool maker to create a new tool, which can then be used by the tool user at a later time.

Finally, the authors compare the performance of Chain-of-Thought prompting with LATM. With GPT-4 as the tool maker and GPT-3.5 as the tool user, LATM can achieve performance that is on par with using GPT-4 for both tool making and tool using, significantly outperforming CoT prompting.

## 2.3 Prompting Techniques

### 2.3.1 Chain of Thought (CoT) Prompting

LLMs struggle to perform complex reasoning tasks. Finetuning an LLM on datasets that demonstrate examples with high levels of reasoning is extremely costly to generate. Providing prompts that have a few examples (Few-Shot prompting) also produces outputs that exhibit the difficulties faced by LLMs in reasoning. Furthermore, few-shot prompting does not result in large improvements when model size increases. Chain of Thought [4] prompting provides a way to overcome these hurdles by enabling LLMs to reason for themselves.

It is typical for humans to decompose problems into intermediate steps and solve each step before giving the final answer. A chain of thought is a series of intermediate natural language reasoning steps that lead to the final output. Endowing LLMs with the ability to generate such a chain of thoughts has the following desirable attributes:

- The chain of thought allows models to decompose problems into substeps, thereby allowing the allocation of more computation to more complex reasoning-based problems.
- A chain of thought also provides a window into the reasoning of the model and thus provides greater interpretability and insight.
- Chain of thought prompting is, in principle, applicable to any task that may be solved through the usage of natural language.
- Chain of thought reasoning can be readily elicited from any LLM by including examples of chain of thought reasoning.

Thus Chain of Thought prompting demonstrates significant improvements on larger models. It is also observed that the improvement is greater on more complex tasks. A key takeaway from this paper was the fact that the order of examples can make a huge difference, to quote the paper:

> Varying the permutation of few-shot exemplars can cause the accuracy of GPT-3 on SST-2 to range from near chance (54.3%) to near state of the art (93.4%)

### 2.3.2 Chain of Thoughts with Self-Consistency

Chain of Thought prompting uses a *greedy* decoding approach, where the only the most probable reasoning path is taken. It is well known that greedy decoding is suboptimal due to the potential of errors propagating through the chain. The Self-Consistency [5] technique method hinges upon the intuition that complex reasoning problems typically admit several distinct approaches of thought to a correct, unique answer. Self-consistency works in the following manner:

1. The language model is prompted with a set of manually-written chain-of-thought exemplars.
2. A diverse set of candidate reasoning paths are generated by the language model.
3. The answers are aggregated by marginalizing out the sampled reasoning paths and choosing the answer that is most consistent amongst the generated answers.

Similar to CoT, CoT-SC prompting produces greater improvements in reasoning capabilities when applied in larger models. CoT-SC shows good improvements of 7-10% over CoT on all tested datasets, including GSM8k, AQuA, SVAMP, and ASDiv.

### 2.3.3 Contrastive Chain of Thought

The Chain of Thought (CoT) prompting method is well-known for enhancing model reasoning and improving model outputs. Logically sound reasoning greatly improves model generations, however conventional Chain of Thought prompting does not inform language models on what mistakes to avoid, and why. Thus Contrastive Chain of Thought [6] takes inspiration from learning methods employed by humans and provides both positive examples (information on what should be done and how it should be done) and negative examples (what should not be done, and how not to do the assigned task) to guide the model to reason step-by-step, while reducing mistakes.

The prompting method seeks to classify different errors by looking at the source of the errors:

- **Invalid Reasoning:** There are errors in the reasoning behind the answer.
- **Incoherent Objects:** The objects referenced in the answer are inconsistent.
- **Incoherent Language:** The language behind the reasoning is not clear or is ambiguous.
- **Irrelevant Objects:** The objects referenced in the reasoning are not relevant to the problem at hand.
- **Irrelevant Language:** The language used has no relevance to the problem at hand, or the reasoning is formulated in a different manner from the query.

Further experimentation reveals that providing negative examples of the Incoherent objects category shows higher average scores on standard datasets. The authors report that applying the technique of Self-Consistency amplifies the benefits accrued from this type of prompting significantly, by up to 10%. In comparison to standard CoT prompting, Contrastive CoT shows significantly higher scores on all tested datasets, thus proving the usefulness of in-context learning via negative examples.

### 2.3.4 Tree of Thoughts

The Tree of Thoughts (ToT) [7] method generalizes the Chain of Thoughts (CoT) prompting method and enables exploration over coherent and relevant units of text (referred to as thoughts) that serve as intermediate steps to answer-generation. The ”tree” in ToT comes from the studies on problem-solving that suggest humans heuristically search through a tree-like problem space, where the nodes of the tree represent partial solutions, and the branches represent operators that modify these partial solutions. While CoT prompting significantly improves problem-solving capabilities of LLMs, it still has some shortcomings:

- CoT does not explore the different continuations within a thought process - the different branches of the tree that represent the solution-space of the problem.
- Globally, it does not include any sort of heuristic-guided search like backtracking, planning or lookahead, which seems integral to human-problem solving.

ToT involves the following steps:

- **Decompose Process into Thought Steps:** ToT leverages problem structure to design and decompose the steps for the task.
- **Generate Potential Thoughts:** Generate potential thoughts for a particular tree state via CoT prompts or sequential prompting.
- **Evaluate thoughts and States:** Evaluate the current frontier of available states (via the LLM) and score the states, either via valuing each independently, or by voting.
- **Search through Tree via Algorithm:** Use a tree search algorithm such as Depth-First-Search (DFS) or Breadth-First Search (BFS) or more advanced algorithms like A*. DFS evaluates the most promising state to its conclusion, i.e until the problem is solved, or the evaluation shows it is impossible to solve the problem at the current state, while BFS maintains a set of the most promising states at each step.

ToT outshines other prompting techniques in

- **Generality:** CoT, CoT-SC etc. are all special cases of ToT,
- **Modularity:** The base LM, thought decomposition, generation, valuing and search algorithm can all be varied independently.

- **Convenience** No fine-tuning or training is needed

The authors test ToT prompting on three different problems that require problem solving skills: Game of 24 (a mathematical reasoning challenge), a creative writing task, and 5X5 mini crosswords. ToT consistently outperforms CoT and basic prompting, by upto 60% in the case of Game of 24.

### 2.3.5 Reverse Chain Prompting

Tool calling is a complex task for LLMs as they often cannot take into account the various intricacies of each tool and the complicated relationships that exist amongst the toolset. Reverse Chain [8] proposes a simple and controllable approach that enables LLMs to use external APIs via only prompting.

The reverse chain technique performs a multi-API planning task in reverse. From selecting the final API call for the given task, each preceding call is inferred backward. Rule-based constraints are employed to limit the order of planning: API selection is done first, then argument completion from query and context, and finally, backward inference to figure out which API's output can properly complete the missing arguments. These rules are not constrained to any particular task and are appropriate to various scenarios.

The rules can be summarised as below:

1. The first step is to employ LLMs to select a proper API that can directly handle a task of interest. This step is referred to as API Selection.
2. After finding the required arguments of the API, the second step is to utilize LLMs to implement argument completion. This step is referred to as Argument Completion.

The above two steps are executed iteratively till the termination condition is met, i.e., until all the arguments for the APIs are completed. Apart from this prompting technique, effects of the temperature the setting was explored. Furthermore, the effects of other popular prompting techniques like CoT (Chain of Thought), One-Shot, Few-Shot prompting, ReACT, etc. were explored. It was found that Reverse Chain outperforms the other methods by a significant margin as per the accuracy metric, on a GPT-4 constructed (and manually verified) dataset. Another takeaway from this paper is that low temperature typically leads to more reliable results.

### 2.3.6 ReAct

LLMs have demonstrated great performance in language understanding, reasoning, and acting, but there have been only a few attempts to integrate these tasks. ReAct [9] tries to incorporate actions to overcome the issue of hallucination and error propagation in CoT reasoning. The authors performed experiments to compare the performance with CoT, CoT-SC, and Act on HotpotQA and Fever, and they have comparable results and outperform the respective datasets compared to others. It also outperforms reinforcement learning methods on ALFWorld and WebShop (interactive decision-making benchmarks) by 34% and 10%, respectively. The paper tried to address why ReAct performs better than Act, CoT, and CoT-SC, and the following reasons were mentioned:

- As Act has no reasoning to guide actions, especially for composing the final answer, ReAct performs better.
- Hallucination is a major problem for CoT, resulting in high false positive rates, but ReAct uses an external knowledge base. Interleaving reasoning, action, and observation steps improve ReAct's groundedness and trustworthiness.
- Retrieving information via searching the actions is crucial for ReAct. Noninformative searches contribute to the majority of the errors in ReAct.
- ReAct + CoT-SC methods outperform CoT-SC across different numbers of samples, reaching the 21 samples performance using 3-5 samples.

Experimentation suggests that ReAct leads to superior performance on complex tasks with large action spaces, which might not fit within the context length. The key takeaway from the paper is that using the ReAct-like prompting technique to get the tool used to perform a particular action might be a better option if the tool space is too large, which might go beyond the context length.

### 2.3.7 Retrieval Augmented Generation

LLMs can be fine-tuned for common tasks such as sentiment analysis, but for complex and knowledge-intensive tasks, it's better to build LLMs that access external knowledge sources to respond to the tasks requiring additional background knowledge. This increases the reliability of the generated responses and addresses the issue of hallucination. Retrieval Augmented Generation (RAG) [10] combines external information retrieval with a text generation model.

RAG looks and acts like a standard seq2seq model, but there is an intermediary step that differentiates and elevates RAG above the plain seq2seq methods. Rather than passing the input directly to the generator, RAG instead uses the input to retrieve a set of relevant documents. The retrieved documents are then concatenated as context with the original prompt and fed to the text generation model, which produces the final result. As LLM's parametric knowledge is static, for tasks involving a dynamic knowledge base, RAG allows LLMs to skip finetuning on the latest information for reliable outputs. The paper discusses DPR, which follows bi-encoder architecture. The dense representations are created using the BERT base document encoder, the list of $k$ documents with the highest prior probability.

However, retrieving and incorporating a fixed number of retrieved passages, regardless of whether retrieval is necessary and relevant, diminishes LLM versatility or can lead to unhelpful response generation due to hallucination. Self-Reflective Retrieval Augmented Generation (Self-RAG) [11] enhances LLM's quality through retrieval and self-reflection. It first predicts whether retrieval is required or not; if it is necessary, it retrieves the document from an external source. It then predicts the document's relevancy, tries to find the supporting paragraphs, and predicts its usage. It then ranks based on these findings and provides it to the text generation model. The authors benchmarked it against Llama2, ChatGPT, Perplexity.ai, Alpaca, CoVE, Toolformer, and SAIL; it outperforms all the other models in multiple tasks.

## 2.4 Tool Retrieval Techniques

### 2.4.1 ToolDec

ToolDec [12] proposes a finite state machine (FSM) guided algorithm for decoding the tools required to answer a particular query. It states to eliminate all tool-related errors by ensuring tool names, arguments, and argument types using the FSM. The authors also claim that ToolDec achieves superior generalization on unseen tools, something we thought worth working on, given the dynamic nature of the tools.

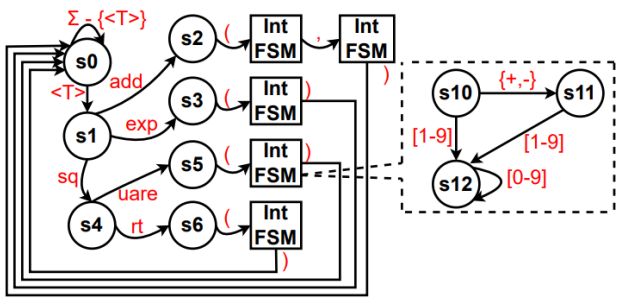


Figure 2: A Finite State Machine (FSM) constructed for math functions taking integer arguments for ToolDec[2]

The FSM is built to cater to the query end-to-end. It uses two states to determine whether the current word is a text/character unrelated to the tool or the text mode, represented by $s_0$, or the current word/character is related to a tool or the tool mode, represented by $s_1$. It uses a special token $\langle T \rangle$ to separate these two states in the FSM. It discusses the construction of the FSM's tool states by using the Trie data structure, and it inserts the names of the tools one by one into the trie. Inserting a string to the trie means going down the root till it finds a path, and the moment it does not, it creates a new node. It assumes that tool names cannot be the same, and hence, it is not usable when tool overloading (same tool name but different arguments) is used. The next task is to generate the states for argument handling in the FSM and propose an argument-type-defined FSM like "IntFSM" chained for multiple arguments. Argument-type defined FSMs are similar to the above FSM and use the transitions and states generated by converting the program rules into similar grammar rules.

It integrates the proposed solution to two baselines: ToolLLM [1] and ToolkenGPT.
As ToolkenGPT uses special tokens to call tools, ToolDec uses FSMs to enforce argument types. For ToolLLM, ToolDec uses three parts of the FSM:

[2]Figure from original paper

a. Format FSM: It enforces the "Thought, Action, Action Input" syntax of ReAct [9] prompting technique.

b. Function name FSM: It guarantees that a decoded function name is always valid.

c. JSON-based function argument FSM: It enforces the JSON-formatted arguments for the function name decoded in the function name FSM.

The paper then discusses the experimentation results on ToolkenGPT + ToolDec and compares them with 0-shot ChatGPT w/o tools, LLaMA w/ tools + CoT, LLaMA w/ tools + ReAct, ToolkenGPT, and ToolkenGPT + Backtrace. The results show a significant reduction in time while maintaining accuracy similar to the ToolkenGPT + Backtrace.

### 2.4.2 ControlLLM

The ControlLLM [13] framework is designed to enable Large Language Models (LLMs) to leverage diverse tools across various modalities for the resolution of intricate real-world tasks. It comprises three specialized components:

1. **Task decomposition:** It breaks down the user prompt into subtasks with well-defined inputs and outputs. It is different from task planning as it only breaks down the user's request into several parallel subtasks and summarizes the input resources for each subtask from the user request. It does not need to know what tools to use or how to use them. The objective of this stage is to achieve three goals: Decomposing input query into manageable subtasks, determining relevant search domain, and inferring the input and output resource types from the instruction context.
2. **Thoughts on Graph Paradigm for Task Planning:** Employing a heuristic approach, this component navigates an optimal solution path on a graph that illustrates tool dependencies. The initial step involves graph construction, comprising Resource Nodes identified by type and Tool Nodes characterized by description, arguments, and return type. Two types of edges exist: Tool-Resource Edges, revealing tools capable of generating specific resources, and Resource-Tool Edges, indicating resources required to operate a tool. Subsequently, a Depth First Search-based algorithm traverses the graph, utilizing a tool selection function, F, to explore all potential paths from input resource nodes to the output resource node. The algorithm terminates upon reaching the anticipated output node or exceeding a predefined maximum length, yielding a list of tool sequences as solutions.
3. **Execution Engine:** This component incorporates a toolbox for the efficient scheduling and execution of the solution path. The execution engine parses solutions into a sequence of Actions during this stage. Each action is linked to particular tool services, implementable through manually crafted mapping tables or an automatic scheduler guided by strategic methodologies.

To evaluate ControlLLM's effectiveness across tasks of varying complexity, a benchmark is established. Experimental results showcase substantial enhancements in tool utilization. The Authors claim that ControlLLM attains a 98% success rate in overall solution evaluation metrics for challenging tasks, surpassing the best baseline, which achieves only a 59% success rate.

### 2.4.3 Craft

Craft [14] is a general tool creation and retrieval framework for LLMs. For tool creation, it creates a toolset for a given specific task. It is achieved through an automated process involving prompting LLM to generate code to solve the training problem. For inference, it identifies and retrieves relevant tools for the given query. It proposes a retrieval component that takes the query, names of the tools, and their documentation through a multi-view function. Authors prompt the LLM to generate the function names and the documentation based on the query. Then, it adopts a similarity measure, considering the following: a) The original query used to create the tool, b) The tool's function name, and c) The documentation of the function. It gets top k tools from the similarity of the given query and three considerations separately and ranks them based on frequency; it also filters out the tools that only appear once. If the resulting toolset is empty, it asks the model to generate the code to perform the query directly without invoking any tool.

The authors compare CRAFT with baseline methods in four categories:

- **Basic Reasoning without Tools:** This technique is based purely on the reasoning ability of the model. The authors used CoT prompts to generate the rationales before answers without using tools. Craft outperforms basic reasoning on Tabular data and Math data.

- **Tool Learning:** LLMs learn to use the provided tools without creating and retrieving tools. CRAFT outperforms by a margin on the VQA task and comparable results on the rest.
- **Different LLM-Created Tools:** To create a tool for the task, LATM samples 3 examples from the training set, and CREATOR creates one specific tool for each test case in the inference time. CRAFT outperforms LATM by a margin and gets comparable results with CREATOR but is better.
- **Alternative Retrieval Methods:** The Authors compare it with tool retrieval approaches, focusing on the similarity measure between the query and the API names. It uses SimCSE and BM25 similarity. CRAFT outperforms by a margin on all tasks other than tabular data, where the results are comparable with BM25 being slightly better.

The key takeaway from the paper is to use the similarity search technique to find the tools and examples given a query as an experiment.

### 2.4.4 Toolchain*

The Toolchain* [15] algorithm serves as a planning algorithm designed for Large Language Models (LLMs), employing a tree search-based approach that exhibits superior efficiency relative to algorithms within its class. It incorporates the A* search algorithm through the integration of a task-specific cost function. The authors assert its outperformance of state-of-the-art baselines in planning and reasoning tasks by an average of 3.1% and 3.5%, respectively, while necessitating significantly less time—7.35x less for planning tasks and 2.31x less for reasoning tasks.

Categorically, planning algorithms can be classified into four overarching categories: Open Loop Systems (such as CoT[4]), Greedy Closed Loop Systems (like ReAct[9]), Closed-Loop Systems (such as AdaPlanner), and Tree Search-Based Systems (such as Tree of Thoughts[7]).

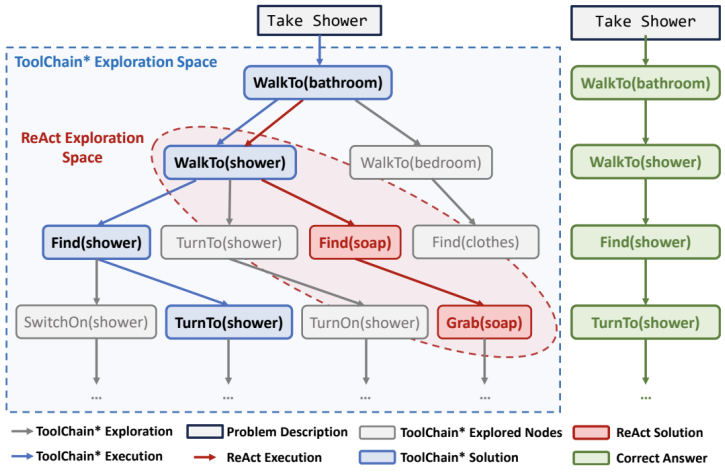


Figure 3: Case study of ToolChain* and ReAct on Virtual Home dataset. Compared to ReAct with a unidirectional search (red), ToolChain* effectively enlarges search space (blue) with tree structures[3]

This algorithm falls within the fourth category—a best-first search algorithm, divisible into three distinct stages: Selection, Expansion, and Update. During the selection phase, the algorithm identifies the most optimal node from the tree's frontier, representing the yet-to-be-explored leaf nodes. Subsequently, in the expansion stage, the selected node undergoes expansion by furnishing the LLM with API definitions and demonstration examples, generating potential actions in a singular step, thereby enhancing the algorithm's efficiency. The final Update step involves the modification of the frontier with new nodes, and costs are computed for subsequent iterations. The algorithm's cost function draws inspiration from the A* algorithm [16], comprising two components: g(n), representing the distance from the root to node n, and h(n), denoting the heuristic approximation of the distance from node n to the terminal node.

While the algorithm demonstrates a commendable success rate, it is imperative to acknowledge its associated drawback in terms of latency. Each query demands a duration of hundreds of seconds, rendering it impractical for our specific use case.

[3]Figure from original paper

## 2.5 Miscellaneous

### 2.5.1 Tool Manipulation

Large Language Models have shown great results in many cases, but when they are augmented with tools there performance decreases, the paper [17] discussed about the challenges faced by LLMs in Tool manipulation capability and ways to resolve this and enhance them. There is a big performance gap between open and closed LLMs and they face 3 key challenges. Firstly, open-source models often struggle to identify API names accurately. Secondly, without demonstration examples, open-source LLMs often fail to populate the appropriate values for API arguments. Thirdly, open-source LLMs tend to produce non-executable generation, such as natural language beyond the desired code; they provide the order and usage of APIs which are not in the correct order and also not executable.

The techniques used to solve the above-mentioned errors are:-

- **Model alignment:** Providing API knowledge to LLMs by providing their documentation and performing instruction tuning. This reduces hallucination by the model and uses correct APIs instead of non-existent ones. Closed LLMs already internalized knowledge about APIs during training, which is not true with open-sourced LLMs. To provide API usage examples, it uses templates and random values to generate more data by adding those random values to the template.
- **In-context demonstration retriever:** Similar to retrieval-augmented generation, it retrieves most semantically similar examples from the group of examples and give it to the LLM to provide contextual understanding. Due to the sequence length of the LLMs, retrievers help in reducing the prompt size given to the LLMs. LLMs also make mistakes in choosing argument values for the APIs, which we can reduce by giving perfect in-context demonstration of examples to it.
- **System Prompt:** They are provided with pre-defined system prompts to provide guidelines for the api calls. This reduces the human effort for the task and adds a natural language style to the generated outputs.

### 2.5.2 ToolAlpaca

ToolAlpaca [18] automatically generates a tool-use corpus and aims to add tool-use abilities even to small language models. It generates a tool-use corpus using a multi-agent simulation environment, it performs well for unseen examples, and has generalized tool-use capabilities. Fine-tuning smaller language models to acquire the capacity for tool usage on a limited range of tools, which lacks the ability to generalize to unseen tools. so we should fine-tune them on a corpus containing highly diversified tool-use instances. Unfortunately, such a diversified corpus is currently unavailable.

ToolAlpaca consists of Three components:

- **Toolset construction :** If we provide the list of APIs with just a brief introduction, it prompts LLMs to produce detailed, structured documentation for each tool. A standardized format is name, introduction, description, function documentation, OpenAPI specification. Different LLMs are used for the generation of different components of the above format. This constructs a diverse and structured toolset that closely resembles real-world scenarios. This comprehensive dataset assists language models in understanding the functionality and usage of each tool.
- **Tool-use instance generation via multi-agent simulation:** They are generated by a simulation environment aimed at emulating the multi-step interactions among language models, users, and tools by making LLMs serve as different kind of agents.It uses multi-turn interplay of three vitrual LLM agents: the user, the tool executor, and the assitant. The format of the instance is Instruction, Actions, Response.
- **Model Training:** By using the instances generated above, different compact language models like Vicuna are finetuned. The evaluation is done on the different dataset generated by this framework and it is evaluated on the basis of procidure it followed, the final response it gave and the overall process.

The finetuned model performs well better than the actual model, and the results were comparable to ChatGPT in the tool-use scenarios, this can be further improved by improving the quality of the data it is trained on.

### 2.5.3 LATM

LLMs As Tool Makers (LATM) [19], where LLMs create their own reusable tools for problem-solving. LATM pipeline can be divided into two stages: 1) Tool Making: A powerful yet more expensive model serves as the tool maker to generate generic and reusable tools from a few demonstrations; 2) Tool Using: A lightweight and cheaper model serves as the tool user to use the tool to solve various instances of the task. The tool-making stage can be further divided into three sub-stages:

1. **Tool Proposing:** The tool maker makes an attempt to generate the tool (Python function) from a few training demonstrations, if the tool is not executable, report the error and generate a new one (fix the issues in the function);
2. **Tool Verification:** The tool maker runs unit tests on validation samples, if the tool does not pass the tests, report the error and generate new tests (fix the issues in function calls in unit tests)
3. **Tool Wrapping:** Wrapping up the function code and the demonstrations of converting a question into a function call from unit tests, preparing usable tools for tool users.

## 2.6 Autonomous Agents

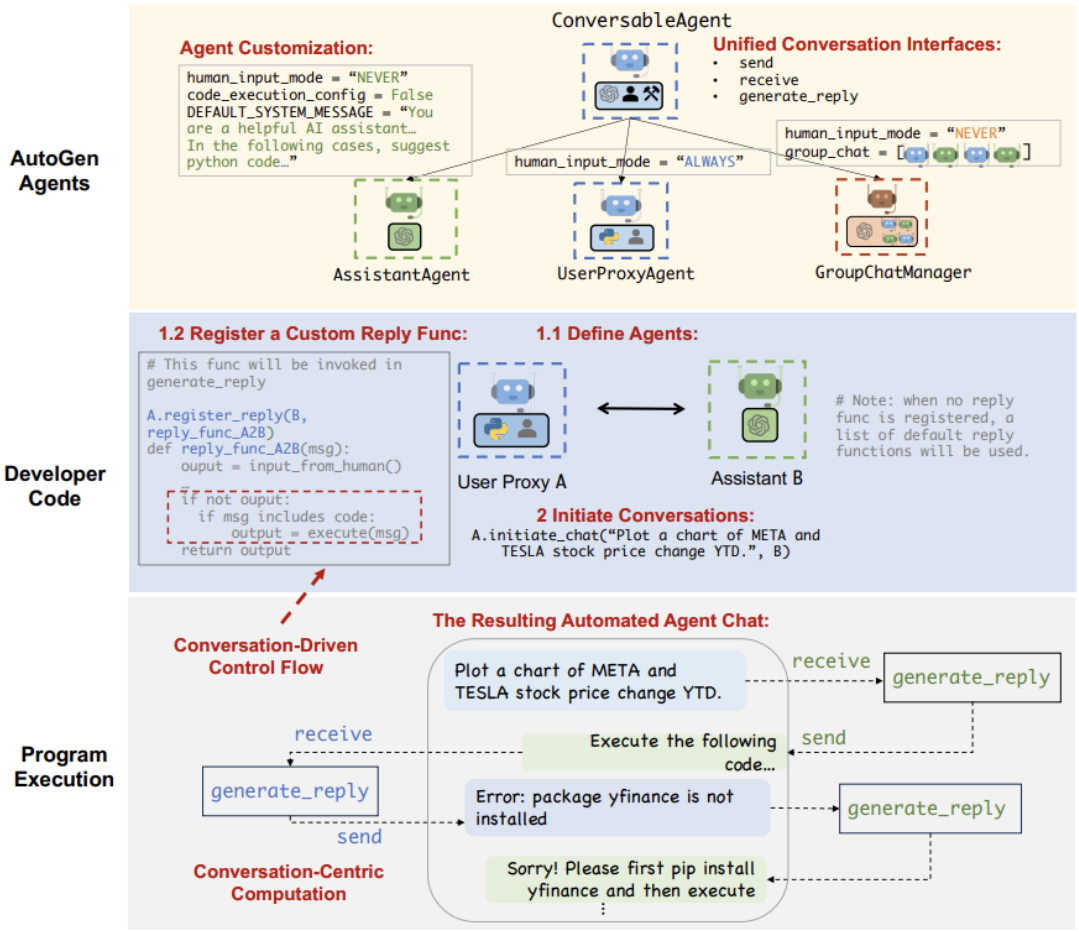


Figure 4: Illustration of how to use AutoGen to program a multi-agent conversation. The top subfigure illustrates the built-in agents provided by AutoGen, which have unified conversation interfaces and can be customized. The middle sub-figure shows an example of using AutoGen to develop a two-agent system with a custom reply function. The bottom sub-figure illustrates the resulting automated agent chat from the two-agent system during program execution.[4]

### 2.6.1 AutoGen

AutoGen [20] is an open-source Multi-Agent Framework by Microsoft Research that allows the user to build LLM Applications. It serves as a generic framework for building diverse applications of various complexities and LLM capacities. It has two components:

1. **Customizable and Conversable Agents:** AutoGen uses a generic design of agents that can leverage LLMs, human inputs, tools, or a combination of them. When configured properly, an agent can hold multiple turns of conversations with other agents autonomously or solicit human inputs at certain rounds, enabling human agency and automation. The conversable agent design leverages the strong capability of the most advanced LLMs in taking feedback and making progress via chat and also allows combining capabilities of LLMs in a modular fashion.
2. **Conversation Programming:** AutoGen adopts a programming paradigm centered around inter-agent conversations. It streamlines the development of intricate applications via two primary steps: defining a set of conversable agents with specific capabilities and roles and programming the interaction behavior between agents via conversation-centric computation and control. Both steps can be achieved via a fusion of natural and programming languages to build applications with a wide range of conversation patterns and agent behaviors.

AutoGen is a very generic framework that can be repurposed for our task by creating multiple agents, such as a decomposer, a planner, and a critic, which can all work together to provide the solution to our problem statement.

### 2.6.2 MetaGPT

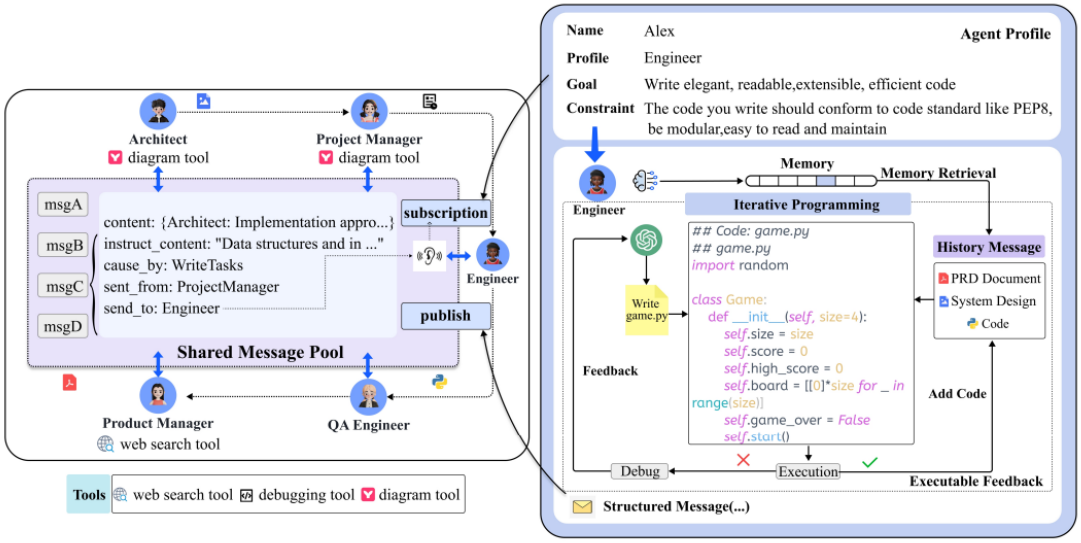


Figure 5: An example of the communication protocol (left) and iterative programming with executable feedback (right). Left: Agents use a shared message pool to publish structured messages. They can also subscribe to relevant messages based on their profiles. Right: After generating the initial code, the Engineer agent runs and checks for errors. If errors occur, the agent checks past messages stored in memory and compares them with the PRD, system design, and code files.[5]

MetaGPT [3] is a meta-programming framework for LLM-based multi-agent systems. MetaGPT encodes Standardized Operating Procedures (SOPs) into prompt sequences for more streamlined workflows, thus allowing agents with human-like domain expertise to verify intermediate results and reduce errors.Furthermore, authors introduce a novel

[5]Figure from original paper

[5]Figure from original paper

executive feedback mechanism that debugs and executes code during runtime, significantly elevating code generation quality.

1. **Agents in standard operating procedure**
    - Specialization of Role: In MetaGPT, the agent's profile, encompassing details such as their name, profile, goal, and role-specific constraints, is explicitly specified. Alongside, the initialization of role-specific context and skills is performed.
    - Workflow across Agents: By defining the roles and operational skills of the agents, the establishment of basic workflows becomes possible. The workflow in this work adheres to Standard Operating Procedures (SOP) in software development, ensuring that all agents operate in a sequential manner.
2. **Communication Protocol**
    - Structured Communication Interfaces: The authors establish a schema and format for each role and request that individuals provide the necessary outputs based on their specific role and context.
    - Publish-Subscribe Mechanism: A shared message pool facilitates direct communication among agents. Agents contribute structured messages to the pool and have transparent access to messages from other entities. This eliminates the need for agents to inquire and wait for responses, thereby improving communication efficiency. To prevent information overload, a subscription mechanism is implemented. Agents, during task execution, utilize role-specific interests to extract relevant information from the pool. This mechanism allows agents to focus on task-related information while avoiding distractions from irrelevant details.
3. **Iterative Programming with executable feedback** An executable feedback mechanism is introduced after the initial code generation. In this process, the Engineer is tasked with writing code based on the original product requirements and design.The Engineer iteratively improves the code using its historical execution and debugging memory.

In extensive experiments, MetaGPT achieves state-of-the-art performance on multiple benchmarks.We could use it to create an SOP for our tool set assigning roles like tool-designer,QA engineer etc creating multiple agents that could work all together.

# 3 Experimentation

## 3.1 Simple Prompting

To begin with, we constructed a simple prompt for ChatGPT (GPT-3.5-turbo) involving a basic initialization for the LLM as an intelligent and helpful agent, to which we added the list of tools in a JSON format. To further improve upon the performance, we added a few examples from the PS description, along with the reasoning behind the answers [Fig: 8]. The query was concatenated with this simple prompt and then used for generation. We made use of CoT prompting to guide the construction of the JSON. Examples were provided such that the entire reasoning and answer were generated within a single JSON, under different keys, for cheap and easy retrieval of answers.

| Model Name | Tool Correctness | Argument Correctness | Exact Match |
|---|---|---|---|
| GPT-3.5-turbo | 68.51% | 62.03% | 55.55% |
| zephyr-7b-beta | 83.33% | 81.66% | 66.66% |
| Mistral-instruct-0.1 | 59.25% | 60.18% | 44.44% |

Table 1: Simple prompting results on GPT-3.5, zephyr-7b, Mistral-7B-Instruct

GPT-3.5-turbo performs well with the prompt, providing fairly accurate results in a conveniently usable JSON. To further supplement our experimentation, we tested the efficacy of open-source models like zephyr-7b and Mistral-7B-Instruct. Results on zephyr-7b were very favorable, especially when considering the small size and low operational costs of the model.

## 3.2 RAG for few-shot examples

During experimentation with the above closed and open-source LLMs, we observed a tendency for models to perform much better on questions that either utilized similar tools as the example queries provided or were variations or extensions of the exemplar queries. To exploit this, we created a RAG-augmented pipeline [Fig: 6] that fetches the $k$ most relevant examples from some source of examples (say a JSON), and utilizes these examples in the prompt detailed above. This further improved generation. In fact, we find that the LLM strongly references any examples provided to it to generate future answers, as mistakes introduced into these examples will propagate into the generation. For example, an exemplar referencing a tool by an improper name will skew the output towards using this improper name.

| Model Name | Tool Correctness | Argument Correctness | Exact Match |
|---|---|---|---|
| GPT-3.5-turbo | 94.44% | 88.88% | 88.88% |
| zephyr-7b-beta | 64.81% | 56.48% | 44.44% |

Table 2: RAG for few-shot examples results on GPT-3.5, zephyr-7b, Mistral-7B-Instruct

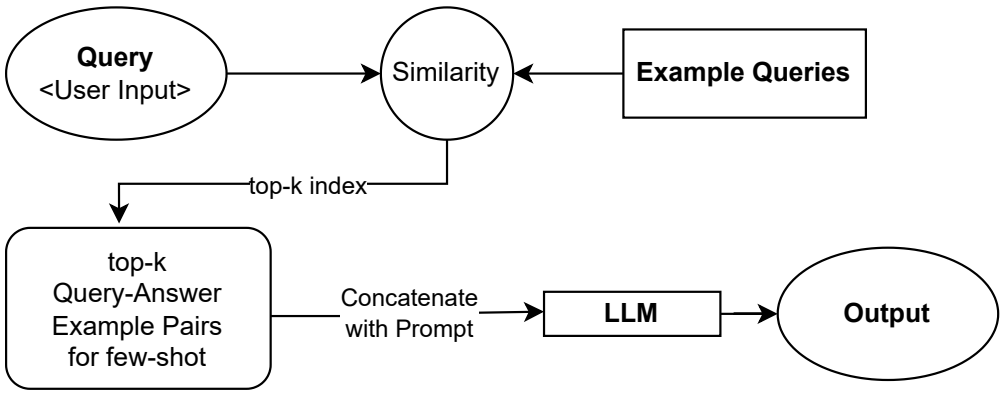


Figure 6: RAG Pipeline

## 3.3 ChatGPT Tools + Task Decomposition

ChatGPT provides inbuilt functionality that seeks a list of tools with their respective names, descriptions, and arguments. Given a query Q, and a set of tools $T$ in the tools argument, it identifies a tool that could be called by mapping the arguments. The limitation of this functionality is that it will produce only one tool. This can be easily overcome by decomposing the query into multiple intermediate subtasks. Task decomposition is a prompting technique that is inspired by ControlLLM [13]. These multiple subtasks can now be treated as a set of subqueries, for solving the task. The prompt [Fig: 9] for tool selection will take place for each subquery in order to provide a multi-tool response. This method is zero-shot learning and can be used to call the listed functions too. The response has been shown below.

**Example 1**

```
# Query
'Summarize work items similar to don:core:dvrv-us-1:devo/0:issue/1'

# Decomposed subqueries
['Use the "get_similar_work_items" tool to get a list of work items similar to
    "don:core:dvrv-us-1:devo/0:issue/1".', 'Use the "summarize_objects" tool to
    summarize the list of work items obtained from the previous step.']

# Tool calls
```

```
[[ChatCompletionMessageToolCall(id='call_WUMlOlXBvpTYbNjJzVf18KBk',
    function=Function(arguments='{\n "work_id":
    "don:core:dvrv-us-1:devo/0:issue/1"\n}', name='get_similar_work_items'),
    type='function')]

[[ChatCompletionMessageToolCall(id='call_F7OoffVmRyKNRHtMpDlZit8X',
    function=Function(arguments='{\n "objects": ["work1", "work2", "work3"]\n}',
    name='summarize_objects'), type='function')],
    [ChatCompletionMessageToolCall(id='call_gzgNaVxXiS9lHbvPZqaFzL5r']]]
```

**Example 2**

```
# Query
'Prioritize my P0 issues and add them to the current sprint'

# Decomposed subqueries
['Use the "works_list" tool to filter for P0 issues.', 'Use the "prioritize_objects"
    tool to prioritize the list of P0 issues.', 'Use the "get_sprint_id" tool to get
    the ID of the current sprint.', 'Use the "add_work_items_to_sprint" tool to add
    the prioritized P0 issues to the current sprint.']


# Tool calls
[[ChatCompletionMessageToolCall(id='call_W3xKwRxncPG7Lsa0YQE9CWHJ',
    function=Function(arguments='{\n "issue.priority": "[\'p0\']"\n}',
    name='works_list'), type='function')],
    [ChatCompletionMessageToolCall(id='call_FOkxVkqeMyFHRdIyvEb2ifk8',
    function=Function(arguments='{\n "objects": "[\'issue1\', \'issue2\',
    \'issue3\']"\n}', name='prioritize_objects'), type='function')],
    [ChatCompletionMessageToolCall(id='call_R6MAcyyOzGD0RoMYTZjmagbp',
    function=Function(arguments='{}', name='get_sprint_id'), type='function')],
    [ChatCompletionMessageToolCall(id='call_hXghZIzau2VtErZR2KHHNwGW',
    function=Function(arguments='{\n "work_ids": ["issue1", "issue2", "issue3"],\n
    "sprint_id": "current_sprint"\n}', name='add_work_items_to_sprint'),
    type='function')]]
```

In case of no tool calls, it gives the following output:

**Example 3**

```
# Query
'What is the meaning of life?'

# Decomposed subqueries
["I'm sorry, but I don't have the answer to that question."]

# Tool calls
[None]
```

### 3.4 ToolBench

We attempted to use ToolBench with ToolLLaMA-v2 as a base model to evaluate the model on questions from a generated dataset. ToolBench utilises uses an LLM to choose potential tools to call at each step, and calls the tool in actuality, and performs more work on the outputs of the tool call. We tried to provide dummy functions for ToolBench to call, however this resulted in various errors and inconsistent outputs. Also, utilising models other than ToolLLaMA-v2 resulted in nonsensical outputs. As such, we were not able to run the latest models with high-context length on ToolBench.

# 4 Future Work

## 4.1 Tree-based Search Graph

In addressing the challenge of selecting the most suitable tool, a tree graph-based search algorithm emerges as a promising solution. The foundation step involves constructing a graph structure that represents relationships between various tools, where nodes denote specific tools, and edges encapsulate criteria such as supported data formats and authentication methods. The given query is then translated into a subgraph, aligning nodes with query criteria. Leveraging a tree graph-based search algorithm, traversal from the root to the leaf nodes evaluates the tool compatibility based on the edge and node attributes. A scoring mechanism, incorporating weighted criteria, qualifies the tool's fitness for the query. This algorithm produces a ranked list of tools, facilitating a nuanced selection process.

To incorporate a tree-based search algorithm into the model for selecting the optimal tool, we will start by constructing a graph structure where nodes represent distinct tools, and edges encapsulate relevant criteria such as data format compatibility and authentication methods. Each node and edge will be attributed with relevant metadata. We will populate this tool graph with accurate information from the tool documentation. The tree-based search algorithm will traverse the graph, evaluating tools based on criteria represented by nodes and edges. A scoring mechanism, with weighted criteria, will quantify tool compatibility. The algorithm will output a ranked list of tools.

## 4.2 MetaGPT-Like AI Agent

In refining our model, we aim to offer a streamlined experience by exclusively providing the essential tools for API calls. Inspired by MetaGPT's adept natural language understanding, our model will prioritize interpreting complex user queries. Similar to MetaGPT's coherent response generation, our model will furnish clear and comprehensive explanations regarding the recommended tools and the rationale behind their suitability for a given query. The model will be able to accept new tools on-the-fly, aligning with the dynamic nature of the field. These enhancements aim to create an intuitive, context-aware, and user-centric tool-using system, providing users with a smooth response to their queries.

## 4.3 Finite state machine (FSM) guided algorithm

The approach outlined in ToolDec, utilizing a finite state machine (FSM) for decoding tools, offers a systematic method to enhance tool retrieval. To incorporate this methodology into our model, we propose integrating a similar FSM-guided algorithm for identifying and decoding tools essential for API calls. Firstly, our model will incorporate a tool-related FSM, inspired by the ToolDec approach, to eliminate errors by validating tool names, arguments, and argument types. The FSM will be designed with two states, differentiating between text unrelated to tools and tool-related text. The construction of tool states within the FSM will leverage the Trie data structure, ensuring efficient handling of tool names. Additionally, our model will extend this FSM approach to encompass argument handling. States for argument handling will be generated, potentially adopting an argument-type-defined FSM similar to "IntFSM" proposed by ToolDec, especially beneficial for scenarios involving multiple arguments.

# 5 Conclusion

Thus, we present the details of our in-depth study of the following for our Mid-Eval Report:

- The tool-use framework ToolLLM, and tool benchmarks MetaTool and ToolBench,
- Various prompting techniques including Chain of Thought (CoT), Contrastive Chain of Thought, Reverse Chain prompting, CoT with Self-Consistency, ReAct, and Tree of Thoughts,
- Retrieval Augmented Generation techniques such as Self-RAG,
- Tool Retrieval Techniques such as ToolDec, ControlLLM, Craft, and ToolChain* which aim to retrieve the proper set of tools for a task

Guided by our study of these cutting-edge methods, we performed experiments on the best instruct-tuned open-source and closed-source models. We enlisted several of the prompt methods [Appendix: 5] studied to generate high-quality

prompts to elicit the best outputs from these models. Insights gleaned from these experiments drove us to improve our few-shot prompt examples via RAG and task decomposition.

For our future work, we propose to improve our pipeline either via tree based search algorithms for tool selection, or via FSM-based retrieval like ToolDec, due to its low error rates. Single-prompt ToT or Reverse Chain methods could be devised via careful study, providing much better, consistent and coherent outputs.

# References


[1] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

[2] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[3] Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. Metatool benchmark for large language models: Deciding whether to use tools and which to use, 2023.

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

[5] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[6] Yew Ken Chia, Guizhen Chen, Luu Anh Tuan, Soujanya Poria, and Lidong Bing. Contrastive chain-of-thought prompting, 2023.

[7] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

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

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

[10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[11] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Learning to retrieve, generate, and critique through self-reflection, 2023.

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

[13] Zhaoyang Liu, Zeqiang Lai, Zhangwei Gao, Erfei Cui, Zhiheng Li, Xizhou Zhu, Lewei Lu, Qifeng Chen, Yu Qiao, Jifeng Dai, and Wenhai Wang. Controlllm: Augment language models with tools by searching on graphs, 2023.

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

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

[16] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

[17] Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models, 2023.

[18] Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases, 2023.

[19] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2023.

[20] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.

# Appendix

## Evaluation Metrics used in the Experimentation

```python
def evaluate(ans, gt):
  scoring = True        # Scoring determines whether to continue scoring or not
  m1 = 0                # m1 is the number of correct tool names till scoring
  t1 = 0                # t1 is the total number of tool names
  m2 = 0                # m2 is the number of correct arguments till scoring
  t2 = 0                # t2 is the total number of arguments for all tools
  if len(ans) != len(gt):       # if the number of tools is not the same, return 0
    return 0, 0, 0
  for a, t in zip(ans,gt):      # for each tool in the answer and ground truth
    t1 += 1                     # increment the total number of tools
    if a["tool_name"] == t["tool_name"]:        # if the tool names are the same
      m1 += scoring                             # increment the number of correct tool names
    else:
      scoring = False                           # else, stop scoring
    if len(a["arguments"]) != len(t["arguments"]):      # if the number of arguments is not the same
      scoring = False                                   # stop scoring
    aargs = {}
    targs = {}
    sc = True                                           # A flag to stop scoring in the next iteration as soon as an argument is wrong
    for aarg, targ in zip(a["arguments"],t["arguments"]):
      aargs[aarg["argument_name"]] = aarg["argument_value"]
      targs[targ["argument_name"]] = targ["argument_value"]
    for targ,tv in targs.items():                       # for each argument in the ground truth
      if aargs.get(targ, "") == tv:                     # if the argument name and value are the same
        m2+=scoring                                     # increment the number of correct arguments
      else:
        sc = False                                      # else, stop scoring
      t2+=1                                             # increment the total number of arguments
    if not sc:
      scoring = False                                   # if an argument is wrong, stop scoring
  return m1/t1, m2/t2, m2 == t2            # return the tool_correctness, argument_correctness, and whether all arguments are correct (binary)
```

Figure 7: Evaluation Metrics Code Snippets

We propose three metrics to evaluate the experiments we performed:

- **Tool Correctness:** Tool Correctness gives the ratio of correct predicted tool names before the first tool name or argument is wrong to the total number of tools.
- **Argument Correctness:** Argument Correctness gives the ratio of correct predicted argument names and values for all the tools before the first tool name or argument is wrong to the total number of arguments for all the tools.
- **Exact Match:** Exact Match is a binary metric giving 1 if all the tool names and argument names and values are exactly matched to the ground truth.

We only marked the correctness to the first tool name or wrong argument name/value because once the model gets a tool or argument wrong, the subsequent calls will be practically wrong because it won't produce the desired result.

## Synthetic Generated Queries

Since the provided dataset was small, we generated new queries and their corresponding JSON outputs using simple prompting [Fig: 10] on ChatGPT. The queries generated were complex and ensured the use of multiple tools and their arguments for better quality of generated dataset.

Examples of a few generated queries and the JSON output:

**Generated Query 1: Find all the tasks owned by the current user that need a response and are associated with Rev organizations REV-123 and REV-456. Sort them by priority.**

---


```
[
    {
        "tool_name": "who_am_i",
        "arguments": []
```

```
    },
    {
      "tool_name": "works_list",
      "arguments": [
        {
          "argument_name": "owned_by",
          "argument_value": "$$PREV[0]"
        },
        {
          "argument_name": "ticket.needs_response",
          "argument_value": true
        },
        {
          "argument_name": "ticket.rev_org",
          "argument_value": [
            "REV-123",
            "REV-456"
          ]
        },
        {
          "argument_name": "type",
          "argument_value": [
            "task"
          ]
        }
      ]
    },
    {
      "tool_name": "prioritize_objects",
      "arguments": [
        {
          "argument_name": "objects",
          "argument_value": "$$PREV[1]"
        }
      ]
    }
  ]
```

---

**Generated Query 2: Find work items in the ẗriageẗstage with priority p0 or p1 created by user DEVU-123 and owned by CAPL-264. Summarize them.**

---

```
[
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "stage.name",
        "argument_value": [
          "triage"
        ]
      },
      {
        "argument_name": "issue.priority",
        "argument_value": [
          "p0",
          "p1"
        ]
      },
```

```
        {
          "argument_name": "created_by",
          "argument_value": [
            "DEVU-123"
          ]
        },
        {
          "argument_name": "owned_by",
          "argument_value": [
            "CAPL-264"
          ]
        }
      ]
    },
    {
      "tool_name": "summarize_objects",
      "arguments": [
        {
          "argument_name": "objects",
          "argument_value": "$$PREV[0]"
        }
      ]
    }
  ]
```

---

**Generated Query 3: Find all tasks owned by the current user that are associated with the Rev organization REV-789 and have a source channel of slack. Sort them by priority and summarize them.**

---

```
[
  {
    "tool_name": "who_am_i",
    "arguments": []
  },
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "owned_by",
        "argument_value": "$$PREV[0]"
      },
      {
        "argument_name": "ticket.rev_org",
        "argument_value": [
          "REV-789"
        ]
      },
      {
        "argument_name": "ticket.source_channel",
        "argument_value": [
          "slack"
        ]
      },
      {
        "argument_name": "type",
        "argument_value": [
          "task"
        ]
      }
```

```
    ]
  },
  {
    "tool_name": "prioritize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[1]"
      }
    ]
  },
  {
    "tool_name": "summarize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[2]"
      }
    ]
  }
]
```

## Prompts

**System:** You are a highly intelligent AI assistant that can help users with questions about a certain wide range of topics involving customer relationship management (CRM). You specialize in writing API calls that are correct, simple, and concise.

{
"tools": <Toolset in JSON format>
}

EXAMPLES:

**User:** <QUERY EXAMPLE 1>

**Assistant:** <ANSWER EXAMPLE 1 in JSON FORMAT with REASONING>

for k examples

**User:** <QUERY EXAMPLE k>

**Assistant:** <ANSWER EXAMPLE k in JSON FORMAT with REASONING>

**User:** <USER QUERY>

Figure 8: Prompt for Simple Prompting and RAG

## Task decomposition

**System:**

The following is a friendly conversation between a human and an AI. The AI is professional and parses user input to several tasks focusing more on keywords without changing context of the query. If the AI does not know the answer to a question, it truthfully says it does not know. The AI will be provided with a set of tools, their descriptions, and the argument in them. The AI then must breakdown the task into fewer intermediate smaller subtasks, every subtask should use one tool and must be unique to minimize the cost of using the tool. Focus more on the description of tools while creating the subtasks from a query. Here is the list of tools:

```
{
   "tools": <Toolset in JSON format>
}
```

**User:**

Query: <USER QUERY>

## ChatGPT Tools

**System:**

You are an intelligent Assistant who is expert at using respective necessary tool or tools from a list when given a query. There can be multiple tools.

**User:**

<QUERY EXAMPLE 1>

Figure 9: Prompt for ChatGPT Tools + Task Decomposition

You are a highly intelligent AI assistant that can help users with questions about a certain wide range of topics involving customer relationship management (CRM). You specialize in writing API calls that are correct, simple, and concise.

{
  "tools": <Toolset in JSON format>
}

QUERY EXAMPLES:

EXPLANATION:

JSON OUTPUT: <Output in JSON format>

Generate a new queries using the toolset and the JSON output for it.

Generated queries with the corresponding JSON output

Figure 10: Prompt for Synthetic Dataset Generation