



TECHNISCHE UNIVERSITÄT MÜNCHEN

Internship Report

**Multi-agent LLM Framework for Conversational
Data Retrieval from Structured Data**

Author:	Muhammad Hamas Khan, MSCE, CIT-TUM
External supervisors:	Benjamin Pohl and Sachin Agrawal
TUM supervisor:	Prof. Dr. Florian Matthes
TUM advisor:	Phillip Schneider
Submission Date:	18 March 2024

Contents

- INTRODUCTION.....3**
- OBJECTIVES3
- METHODS AND APPROACHES.....4**
- RETRIEVAL AUGMENTED GENERATION (RAG)4
- INITIAL APPROACH - TABLE SELECTION WITH EMBEDDINGS5
- IMPROVED APPROACH - MULTI-AGENT LLM CONVERSATIONS FRAMEWORK.....6
- High-Level Overview – Our Implementation.....8*
- ADDITIONAL EXPERIMENTS.....10
- Teachable Agents.....10*
- Fine-tuning with Multi-GPU Fully Sharded Data Parallel (FSDP) and Low-Rank Adaptation (LoRA).....12*
- ANALYSIS AND DISCUSSION.....14**
- Single Agent LLM.....14*
- Multi-Agent LLM framework.....15*
- Teachable Agents.....20*
- Fine-tuning.....21*
- CONCLUSION.....24**
- REFERENCES.....24**

Introduction

The rapid development of Large Language Models (LLMs) such as OpenAI's GPT-4 has spurred significant interest among companies in diverse sectors such as automotive, software, hardware, manufacturing, and cloud computing. These organizations are eager to investigate the potential integration of these advanced language models with their enterprise data. Application areas include but are not limited to:

1. Text-to-SQL on databases – generate a SQL query to fetch results from a DB.
2. Conversational chatbots – question-answering on documents.
3. Text-to-Code – generate code to perform a task based on the user input.

This research internship aims to explore various methods for deploying LLMs on new and unseen data, specifically in the automotive industry. The investigation encompasses aspects such as Retrieval Augmented Generation (RAG) [2], Microsoft's AutoGen [4] multi-agent framework, and the fine-tuning of open-source models like Meta's Llama-2. The internship also involves the development of a web app that interacts with the GPT-4 model for data retrieval. We have data available in Databricks tables related to car trips. This data contains signal values and aggregated values for dashboards.

Objectives

The primary goal of this internship was to create a prototype framework using recent advancements in LLMs to extract conversational information from enterprise data. This prototype served as a starting point to investigate the potential scalability of the framework for enterprise-wide use beyond the internship's scope.

During the internship, our focus was on the following tasks:

- 1) We explored and developed a RAG (Embeddings with Vector Search and AutoGen multi-agents) framework for:
 - a) Performing text-to-SQL operations on tables containing sensor data.
 - b) Retrieving data from files containing documentation.
- 2) We conducted additional experiments involving teachable agents.
- 3) We also experimented with fine-tuning on a custom dataset.

Here are some example question-answer pairs:

1. **Question:** "Hey, can you show me the usage of lane assistance on the car Alpha for the past 3 months within Bavaria with engine control unit version 3?"

Answer: "Sure, there have been 5561 trips with the requested details in the past 3 months. Would you like to know which schema I used for these results?"

2. **Question:** "Hey, could you provide the sales figures of different electric models in the last quarter, broken down by countries?"

Answer: "Of course, the highest sales were for model ABC in the USA during the last quarter. You can visualize the sales of different models categorized by countries using the following Python code..."

This prototype caters to user groups comprising engineers and managers who seek detailed insights from data contained in tables and documents.

Methods and Approaches

Retrieval Augmented Generation (RAG)

Large pre-trained language models have demonstrated an ability to retain factual knowledge within their parameters and achieve top-notch results when fine-tuned for specific natural language processing tasks. However, their capacity to access and precisely manipulate this knowledge remains limited. Consequently, on tasks demanding extensive knowledge, they often fall short compared to architectures tailored specifically for those tasks [2]. While sometimes LLMs accurately answer questions, at other times, they might simply repeat random facts from their training data. This inconsistency arises because LLMs lack semantic understanding [2].

Retrieval Augmented Generation (RAG) [2] addresses these issues by enhancing LLM-generated responses by incorporating external sources of knowledge to supplement the model. Furthermore, RAG helps mitigate the risk of LLMs producing incorrect information by grounding the model on verifiable external facts [2]. This reduces the need for continuous training on new data, thereby lowering the computational and financial overheads associated with deploying LLM-powered chatbots in business settings.

Initial Approach - Table Selection with Embeddings

Initially, we adopted a straightforward approach, leveraging embeddings to conduct similarity searches on tables. Our dataset is structured within Databricks tables, encompassing information on automobile event logs across various sensors. Additionally, certain tables feature aggregated data suitable for creating dashboards, such as sales or usage metrics derived from car trip events.

To facilitate interaction with this data, we employed the LangChain framework to develop a basic chatbot. Our choice for the Large Language Model (LLM) was the OpenAI GPT-3.5 model. Within this framework, we integrated functionalities like SQL execution and metadata retrieval for schemas. Metadata proves crucial as it has descriptions of the tables and columns. Below outlines the sequence of code execution:

- 1) The user types a question.
- 2) The question is sent to the GPT-3.5 model through an API request for analysis, and the model sends back an answer.
- 3) We fetch metadata information about the schema.
- 4) The retrieved metadata contains useful details about the schema, which we process.
- 5) We exclude columns with numerical data since they don't help with similarity searches for the input query. Our focus is on the descriptive text within the columns for this purpose.
- 6) We calculate and store embeddings for both the columns and their contents using the GPT 3.5 model.
- 7) We measure the similarity between the input question and the column names and contents using cosine similarity.
- 8) To determine the final score for the table, we use a weighted sum:

$$\text{Final score} = \text{Column_Name_Score} * 0.7 + \text{Average_Column_Contents_Score} * 0.3.$$

These weights are adjusted for optimal results, with a preference given to column names.

- 9) We calculate the table score by averaging the final scores (name + content) of all columns.
- 10) The top 3 tables with the highest scores are presented for selection.
- 11) These tables are provided to the GPT-3.5 model for SQL query generation.
- 12) The generated SQL undergoes a preliminary check with a Guardrail module to filter out any data manipulation statements (such as Delete, Merge, or Truncate) and ensure SQL query consistency using a parsing library (SQL Glot).
- 13) The validated SQL query is executed to retrieve the results.
- 14) Finally, the obtained results are fed back to the GPT model to generate a user-friendly response. We also implement additional checks to prevent API timeout exceptions, considering factors like the model's context window and rate limit.

Figure 1: Table Selection with Embeddings using LangChain. shows a high-level block diagram of this approach.

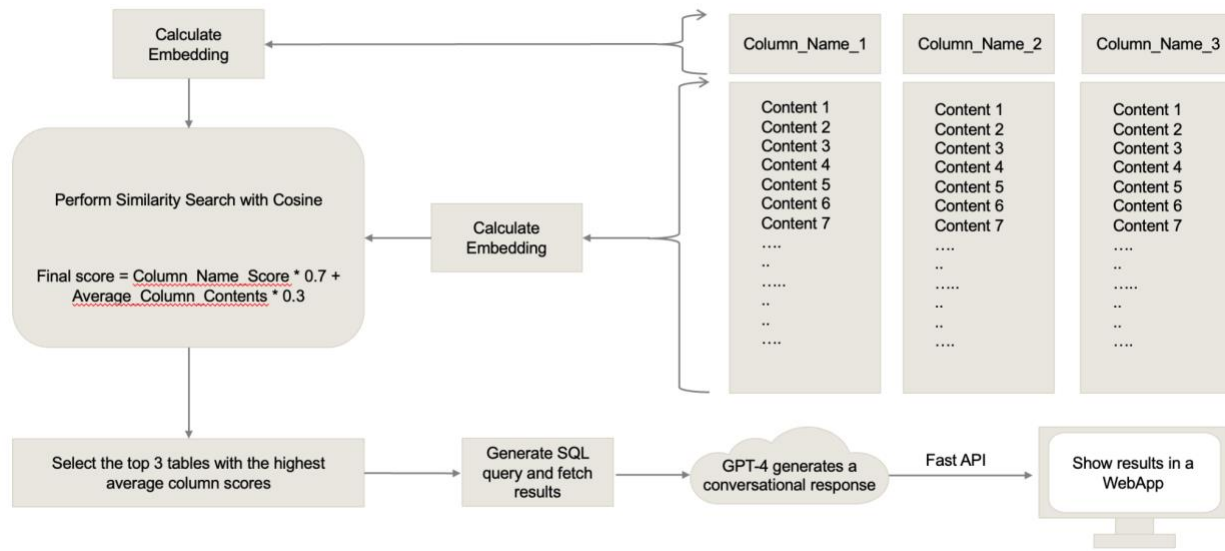


Figure 1: Table Selection with Embeddings using LangChain.

Improved Approach - Multi-agent LLM Conversations Framework

We utilized Microsoft AutoGen, an open-source framework designed to empower developers in creating applications using Large Language Models (LLMs) by enabling collaboration among multiple conversational agents [4]. These agents are flexible and adaptable, capable of operating in various modes that incorporate LLMs, human inputs, and tools as needed. With AutoGen, developers have the freedom to define interaction behaviors for these agents, leveraging both natural language and code to customize conversation patterns for different applications. By orchestrating chat interactions among multiple agents, users can easily orchestrate them to autonomously perform tasks or collaborate with human feedback [4].

Key features of AutoGen include [4]:

- 1) Multi-agent conversations: AutoGen facilitates communication among agents, allowing them to collaborate on tasks. This capability enables the development of more complex and advanced applications compared to those achievable with a single LLM.
- 2) Customization: AutoGen empowers developers to customize agents according to the specific requirements of an application. This includes the ability to choose LLMs, specify acceptable forms of human input, and integrate various tools as needed.
- 3) Human involvement: AutoGen seamlessly integrates human participation, enabling individuals to provide input and feedback to the agents as necessary.

Figure 2: AutoGen configurations [4]

(Image source: <https://arxiv.org/pdf/2308.08155.pdf>). Figure 3: AutoGen conversation example show examples of different configurations and a sample conversation.

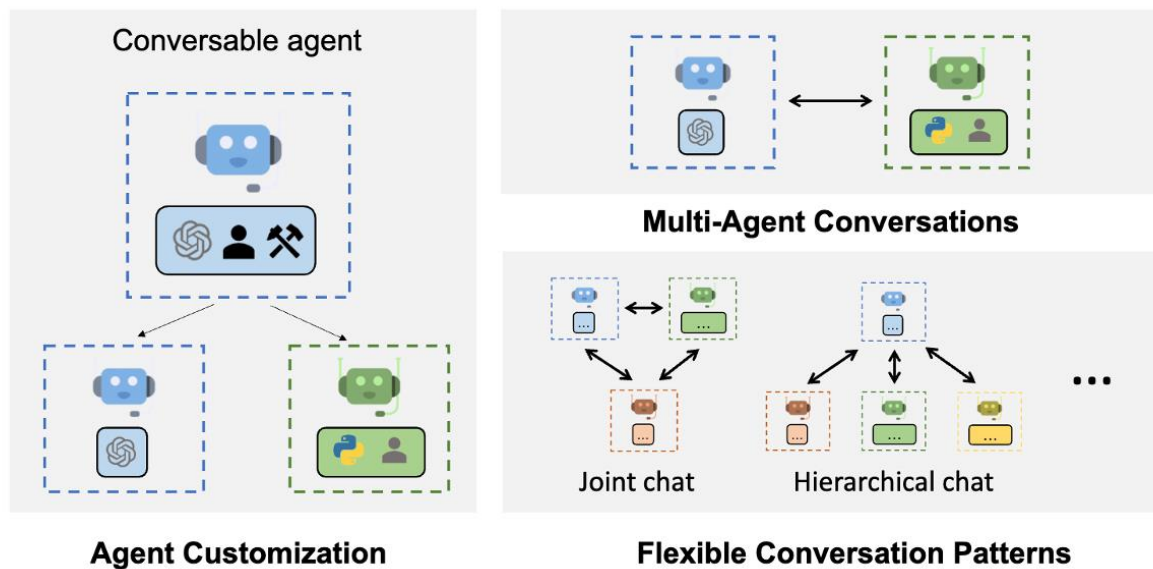


Figure 2: AutoGen configurations [4]

(Image source: <https://arxiv.org/pdf/2308.08155.pdf>).

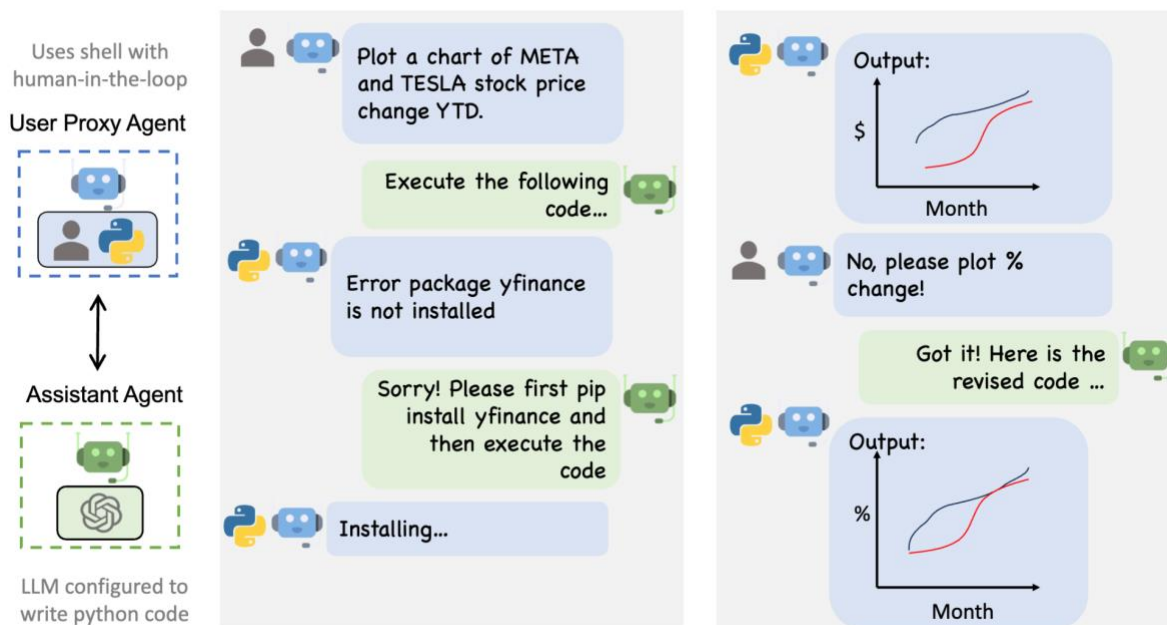


Figure 3: AutoGen conversation example [4]

(Image source: <https://arxiv.org/pdf/2308.08155.pdf>).

High-Level Overview – Our Implementation

We created a multi-agent LLM program using AutoGen. In our setup, each agent is assigned specific functions or tools based on its role. Here are the key components of our system:

- 1) User Proxy Agent: This agent serves as the administrator, responsible for communicating with other assistant agents, finalizing responses, and soliciting user feedback.
- 2) Group Chat Manager: This agent is responsible for initializing all the agents (including the user proxy, RAG agent, teachable agent, and assistant agents). It also keeps track of all previous chat messages.
- 3) Assistant Agents: These agents have different roles:
 - Planner: Responsible for outlining a plan for the next steps of action.
 - SQL Executor: Generates SQL queries and has access to the SQL execution function.
 - Database Manager: Has access to functions for retrieving schema metadata.

Here's how the execution typically flows:

1. When a user asks a question, the chat manager agent kicks off the conversation by setting up the admin and assistant agents.
2. The chat manager then directs the question to the admin agent, which interacts with other assistant agents based on the conversation's context and user feedback.
3. Initially, a RAG agent checks if the metadata holds the answer to the user's query.
4. If it does, the RAG agent replies with relevant information. If not, the admin requests assistant agents to retrieve the required data from the database.
5. A database manager grabs the schema metadata and processes it.
6. The SQL executor agent crafts and executes the SQL query to fetch data from the pertinent table.
7. An assistant agent summarizes the findings and sends the response back to the admin for wrapping up the chat.
8. If the user has more questions, the conversation can continue with further inquiries from the admin.

The agents respond based on their prompts and the tools at their disposal, like fetching schema metadata, planning actions, or presenting results. Additionally, system prompts can be used to refine the messages further.

We structured the schema metadata to include details such as table names, columns, data types, and additional information like the size of column contents. This schema metadata is organized as a nested dictionary and can be utilized by other agents during conversations.

Figure 4: Overview of the program flow of the multi-agents. offers a broad overview of the program flow. It's crucial to understand that the sequence of execution depicted for the admin and assistant agents is a typical pattern but can vary based on the specific question in different conversations. Since this is a conversation and not a rigid sequence, the flow can adapt depending on the context. For instance, although control typically passes from the admin to the RAG agent, if a user requests to amend a SQL query due to an error, the admin can directly involve the engineer agent to rectify and re-execute the SQL query for accurate results.

Furthermore, we implemented guardrails to parse the syntax of generated SQL queries using the SQL Glot library. These guardrails include checks to prevent data manipulation queries such as Delete, Truncate, and Merge.

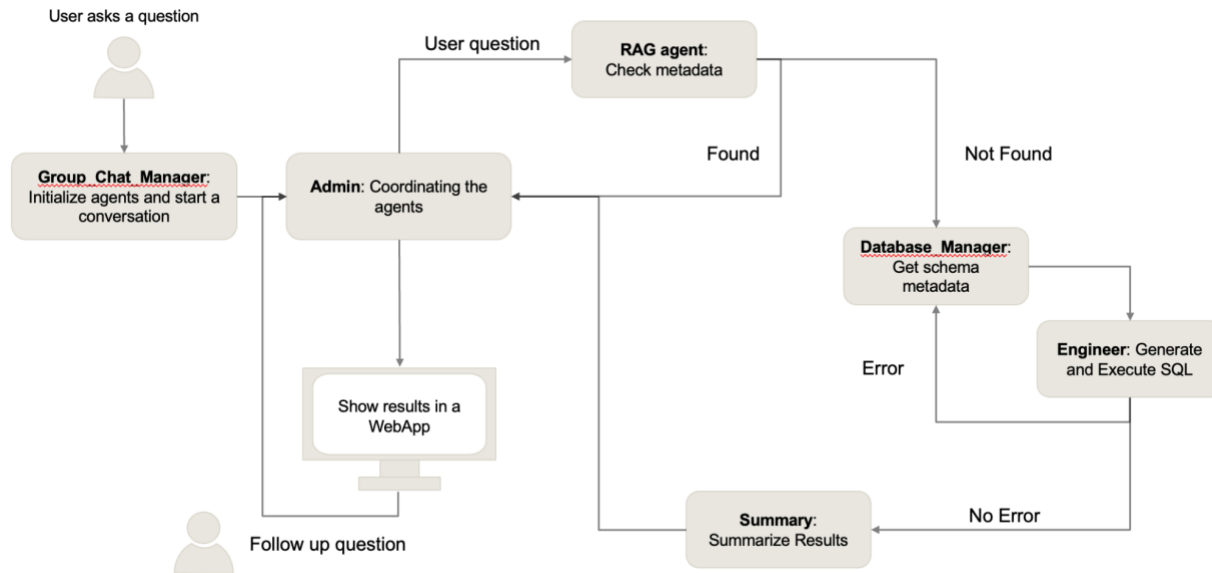


Figure 4: Overview of the program flow of the multi-agents.

Additional Experiments

Teachable Agents

Conversational assistants based on Large Language Models (LLMs) can remember ongoing chats with users and learn from user input during conversations. However, once a chat ends or becomes too lengthy for the LLM to manage effectively, the assistant's memories and learned information are typically lost. This means that in subsequent chats, users may need to repeat instructions or information multiple times [3].

Teachability addresses these challenges by storing user teachings across chat sessions in a long-term memory system implemented as a vector database. Instead of storing all memories in the chat context, which would consume a lot of space, individual memories (referred to as "memos") are retrieved into the conversation context only when needed [3]. This enables users to teach the assistant important facts and skills just once, with the assistant recalling them in future conversations.

To make informed decisions about memo storage and retrieval, the teachable agent uses a `text_analyzer` agent to identify and adjust the text as necessary for remembering facts, preferences, and skills [3]. Figure 5: Teachable agent overview

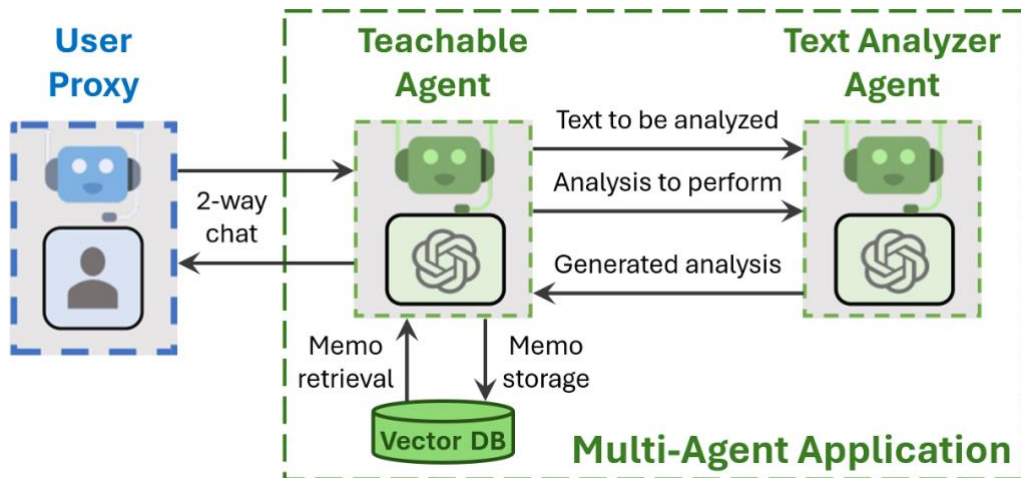


Figure 5: Teachable agent overview [3]

(Image source: <https://microsoft.github.io/autogen/blog/2023/10/26/TeachableAgent/>)

Below is a response from the text_analyzer agent:

1. teachable_agent (to text_analyzer):

*“Is there any part of the text that requests the agent to perform a task or solve a problem?
Answer with just one word, Yes or No”*

2. HTTP Request:

“https://abcxyz.openai.azure.com/openai/gpt-4-32k/chat/completion”

3. text_analyzer (to teachable_agent):

“Yes”

Fine-tuning with Multi-GPU Fully Sharded Data Parallel (FSDP) and Low-Rank Adaptation (LoRA)

Dataset Generation

We have documentation detailing the code of various internal components of cars. This documentation outlines parent and inherited child classes, along with their respective functions. Additionally, it includes hierarchy descriptions such as `parent_class`, `child_class`, and `child_class_functions`. We extracted and processed this documentation from HTML files into a JSON format. Due to restrictions outlined in OpenAI's [Terms of Use](#), we could not utilize OpenAI models to create a dataset for commercial purposes. Instead, we employed the open-source Meta Llama-2 model to generate an instruction dataset comprising question-answer pairs.

The JSON file contains our processed classes alongside their descriptions and functions. We formulate questions by incorporating additional context based on these descriptions. These question-context pairs are then passed to Llama-2 for inference using custom and system-level prompts. The responses obtained are used to populate the answer fields of the dataset. Some of these responses include hierarchical information about the data, such as the relationship between different classes. We structure our dataset like the [Stanford Alpaca](#) dataset, as shown in Figure 6: Alpaca Dataset sample.

```
{
  "instruction": "Identify the odd one out.",
  "input": "Twitter, Instagram, Telegram",
  "output": "Telegram"
},
{
  "instruction": "Explain why the following fraction is equivalent to 1/4",
  "input": "4/16",
  "output": "The fraction 4/16 is equivalent to 1/4 because both numerators and denominators are divisible by 4. Dividing both the top and bottom numbers by 4 yields the fraction 1/4."
},
{
  "instruction": "Write a short story in third person narration about a protagonist who has to make an important career decision.",
  "input": "",
  "output": "John was at a crossroads in his life. He had just graduated college and was now facing the big decision of what career to pursue. After much deliberation, he decided that he wanted to be an accountant and help the financially disadvantaged. He had always been good with numbers and enjoyed seeing the tangible results of his work. \n\nJohn enrolled in accounting courses and initially found it quite challenging. He had to learn multiple systems and regulations quickly, but he worked hard and eventually excelled in his studies. After a few years, John started working at an accounting firm in his city. He was eager to put his knowledge of taxes and accounting to use in a real-world setting.\n\nJohn loved his job, as it let him express his creativity in finding strategies to save his clients money. After a few years at the firm, he became a senior accountant and was asked to manage bigger and more challenging cases. He was now a respected figure in the financial industry, but he still remembers when he was just a recent college graduate, unsure of the direction in which his life would take him."
},
{
  "instruction": "Render a 3D model of a house",
  "input": "",
  "output": "<nooutput> This type of instruction cannot be fulfilled by a GPT model."
},
{
  "instruction": "Evaluate this sentence for spelling and grammar mistakes",
  "input": "He finnished his meal and left the resturant",
  "output": "He finished his meal and left the restaurant."
},
}
```

Figure 6: Alpaca Dataset sample. Our generated dataset has a similar structure.

The generated dataset comprises approximately 2000 pairs of instructions and corresponding outputs, with some pairs containing additional input fields. This dataset includes information about classes, such as descriptions, functions, and hierarchy. It was utilized for training-test splits during the fine-tuning process.

Fine-tuning & Training

The model we used to fine-tune is Meta's Llama-2 7 billion parameters because of its open-source license. We accessed the model using Hugging Face in the Azure ML Studio platform. The compute resources we used in Azure ML Studio are as:

4 x NVIDIA Tesla V100 16GB vRAM
32 \$ per hour for 4 GPUs

To improve efficiency in computational cost and training time without sacrificing performance, we employed Parameter Efficient Fine Tuning (PEFT) [1]. PEFT allows for the adaptation of large pre-trained models to various downstream tasks without the need to fine-tune every single parameter. This approach significantly cuts down on computational and storage costs while still producing results comparable to fully fine-tuned models.

One specific technique within PEFT is known as Low-Rank Adaptation (LoRA) [1]. In natural language processing, a common practice involves pretraining large models on general domain data and then adapting them to specific tasks or domains. However, as we scale up to larger models, fully fine-tuning all parameters becomes impractical.

LoRA addresses this challenge by freezing the pre-trained model weights and incorporating trainable rank decomposition matrices into each layer of the Transformer architecture. This approach significantly reduces the number of parameters that need to be fine-tuned for downstream tasks. For example, compared to fine-tuning GPT-3 175B with the Adam optimizer, LoRA can reduce the number of trainable parameters by a factor of 10,000 and decrease GPU memory requirements by threefold [1].

We encountered an issue with the Llama-2 7 billion model not fitting into GPU memory, requiring approximately 24 Gigabytes of memory. While we had access to four GPUs, each with 16 Gigabytes of memory, fitting the entire model into a single GPU was not feasible. Consequently, we needed to devise a method to distribute the model across multiple GPUs so that it could fit into the available memory.

The solution came in the form of PyTorch's Fully Sharded Data Parallel (FSDP) API [5]. Unlike traditional data-parallel training, where each GPU maintains a copy of the model's parameters, gradients, and optimizer states, FSDP shards all these components across data-parallel workers [5]. Additionally, it offers the option to offload the sharded model parameters to CPUs. This approach allowed us to effectively distribute the model across the available GPUs, overcoming the memory limitation and enabling efficient training [5]. Consequently, we successfully distributed the 24 Gigabytes model across four GPUs, with each GPU accommodating approximately 6 Gigabytes of memory allocated for the model.

Analysis and Discussion

Single Agent LLM

The typical and widely used approach for tasks such as information retrieval in a Large Language Model (LLM) framework is the single-agent method. In LangChain, we developed a single-agent program that employs table selection through embeddings, which proves effective for certain scenarios, such as straightforward data retrieval from databases. However, when we attempt to expand its capabilities to include fetching data from documents alongside database queries, its performance in generating accurate responses diminishes. This decline occurs because a single agent, which operates with a single prompt, tends to produce generic responses rather than breaking down the task into distinct steps. For instance, consider a single-agent prompt that requires conversational abilities, the ability to fetch schema metadata, generate and execute SQL queries, and interpret and summarize the results, all simultaneously. Typically, in single-agent prompts, these instructions are presented sequentially, such as "first perform task 1, then task 2," and so forth. While the agent can accurately respond to simple questions that align with this sequential structure, it struggles when faced with more complex inquiries that require simultaneous handling of multiple tasks.

When users engage with the agent in conversational exchanges and ask follow-up questions, the quality of responses tends to decrease. This decline occurs because the agent is directed by the initial prompt to adhere to a specific set of instructions. Additional follow-up questions overwhelm the agent, as it is not programmed to deviate from its prescribed instructions and instead break down the problem into smaller, more manageable steps.

Due to these factors, interacting with a single-agent system often feels unnatural and non-conversational since it rigidly adheres to its predefined instructions. For instance, imagine a scenario where the agent retrieves information from a database, and the user then asks a follow-up question based on that information. In such cases, the single agent will attempt to fetch the data from the database again, even though the answer might be found in accompanying documentation.

Another significant issue arises in scenarios where a single agent is equipped with multiple functions or tools, such as those for accessing schema metadata, generating and validating SQL queries with guard rails, and visualizing results. In such instances, the single agent may occasionally choose the wrong function or tool during execution, leading to inaccurate results.

Additionally, implementing new features in a single-agent system requires additional programming focused on enhancing its output. Conversely, in a multi-agent LLM system, the LLM itself handles more tasks without explicit programming. However, the multi-agent approach introduces uncertainty in the output since it is more heavily reliant on the LLM's control. To clarify,

in situations where the LLM writes SQL queries or code, multi-agent LLM conversations can autonomously correct errors in the SQL queries, generated code, or GPT-4 context window-related errors. Conversely, in a single-agent system, additional programming and checks are necessary to verify the generated output. This is because the multi-agent system operates with greater autonomy and treats the problem as a conversation rather than strictly adhering to specific instructions.

In summary, single agents excel at handling one-dimensional tasks aligned with their instructed prompts. However, for many real-life business scenarios where problems require decomposition into smaller tasks for generating accurate answers in a stepwise fashion, single agents often fall short. This underscores the value of multi-agent conversations as an enhanced framework for tackling such challenges more effectively.

Multi-Agent LLM framework

We aim to enhance the LLM retrieval framework with the following improvements:

1. Incorporate document retrieval alongside database queries for generating conversational responses.
2. Expand the framework's capabilities to address new challenges through the integration of additional features.
3. Enhance conversational abilities and enable the framework to respond effectively to follow-up questions.
4. Implement a self-correcting mechanism to address any issues that arise during task completion, such as generating code or executing SQL queries.

With these requirements in mind, we opted to utilize the Microsoft AutoGen [4] framework. Here's an analysis of its performance across various aspects.

Addition of new functionalities with specialized agents

When comparing AutoGen to a single-agent system, it's notably simpler to incorporate new functionalities, especially if these functionalities can be implemented with new instances of the LLM model, essentially creating new agents. For instance, when introducing document retrieval alongside text-to-SQL query results, AutoGen simply required the addition of a RAG agent. Conversely, achieving the same outcome in a single-agent system demands more effort. This isn't to suggest that the multi-agent approach effortlessly handles everything with minimal effort. Additional programming is still necessary to extend functionalities to agents and ensure bug-free operation. However, the process is more straightforward and efficient for agents to tackle problems.

For instance, to integrate table selection with embeddings, as described in the Methods and Approaches section with multi-agents, we can create a new assistant agent named "table_selector" with access to a function containing the logic for table selection with embeddings. We then define the role of this agent in its prompt and integrate it into conversations with administrators and other assistant agents by initializing it with the group chat manager.

Conversational response

The multi-agent system produces more conversational responses. This is because each agent is designated a specific role, enabling them to collaborate with the admin and other agents to accomplish tasks. Consequently, this structure results in more natural and conversational responses.

Ability to reason and plan with LLMs

One of the key advantages of employing a multi-agent LLM framework is the transparency it offers regarding the agents' action plan to tackle a task. This transparency stems from the inclusion of a planner agent, which devises a step-by-step course of action. An example of the planner agent's response in the initial iteration, concerning a query related to retrieving sound settings information of a car, is outlined below:

1. Retrieve the schema metadata to gain an understanding of the database structure.
2. Identify the table containing relevant information about sound settings.
3. Execute a SQL query on the identified table to obtain the necessary data. Aggregate results to ensure they remain within the context window.
4. Interpret the results of the SQL query to extract the required information.

In the subsequent iteration, once the table has been identified as "sound_table_23," the planner's response is as follows:

1. Formulate a SQL query derived from the user input.
2. Execute the query on the "sound_table_23" table.
3. Summarize and interpret the outcome of the SQL query to obtain the necessary information.

This process illustrates how the planner agent generates a course of action, and the admin agent selects the next relevant agent accordingly to achieve the desired results.

Self-correction on error messages

The multi-agent LLM framework enables self-correction of error messages within code during conversations. Let's illustrate this with an example: When a user poses a query requiring database results, the agent responsible for generating and executing SQL constructs a query accordingly. However, if the query yields excessively long results or contains syntax errors, any resulting error messages become part of the conversation. Similarly, if a follow-up question pertains to writing code for visualizing or analyzing the data further, errors such as missing variables, runtime issues, various exceptions, or GPT-4 context window errors might occur. These errors are integrated into the conversation loop, allowing the LLM model to attempt error correction by regenerating the SQL or Python code. However, the agents aren't always successful at rectifying errors, particularly if they are logical in nature. Hence, the error correction capabilities perform similarly to what one might expect from a typical user attempting to debug code using ChatGPT [4].

Problems with the AutoGen multi-agent LLM framework

Previous sections predominantly highlighted the benefits of multi-agent LLMs. Now, let's delve into some challenges we encountered.

Controlling Conversation Flow

A significant issue with these multi-agent LLM conversations is the challenge of controlling conversation flow, occasionally resulting in output uncertainty. This uncertainty implies that outputs may vary when the user poses the same question multiple times. The problem often stems from incorrect selection of the next speaker, or agent. Below are scenarios where this issue surfaced:

1. The RAG agent occasionally struggled to retrieve results from the information stored in the metadata. Ideally, we would prefer to grant control to the RAG agent to verify if the answer to a question already exists in our preloaded metadata. However, there were instances where the admin would directly opt for the SQL_executor or planner agent to retrieve results, even though the answer was available in the metadata.
2. In scenarios involving teachable agents, the optimal approach would involve the admin initially consulting with the teachable agent to check if the answer exists in the saved memos from previous conversations. However, the admin would sometimes incorrectly select the database agent to retrieve results instead.

Agents Looping and Termination

Another issue arises when agents unnecessarily loop without terminating the conversation. This typically happens when complex questions are posed, especially when lengthy results or SQL joins are involved. In scenarios where the response requires extensive processing time by the GPT-4 model, or when SQL joins result in excessively long outputs, the agents may continually loop as they struggle to find an answer to conclude the chat.

Human Feedback

Human feedback plays a crucial role in guiding conversations and providing input in case of errors or undesirable results within multi-agent systems. Nevertheless, there are instances where agents fail to act upon this feedback.

LLM Context Window and Rate Limit Errors

Handling large outputs that surpass the context window of the GPT model is a significant consideration. We initially employed GPT-3.5 and later transitioned to utilizing the GPT-4 and GPT-4-32k models as our use cases expanded. In many instances, resolving this issue involved rewriting the SQL query to summarize results through aggregation. Alternatively, we utilized a tokenizer module to divide tokens into smaller batches that fit within the context window. Rate limit timeout exceptions were addressed by retrying with exponential backoff strategies. Moreover, both the context window and rate limit settings are adjustable within the Azure model settings.

Potential Solutions

Addressing such issues typically involves an empirical approach, which includes testing different configurations of the LLM model (such as GPT-4 versus GPT-4 32k), adjusting agent settings like `human_input_mode` and `code_execution_config`, enhancing prompts with additional instructions, evaluating the number of agents, and ensuring agents are equipped with the correct functions. These functions are vital as they encompass the logic necessary to obtain desired results. For instance, a function may involve fetching schema metadata containing valuable information such as table names, columns, and their content types. Alternatively, human feedback can also be utilized to address runtime issues during ongoing chats.

It's crucial to emphasize the difficulty in resolving such issues and consistently providing perfect answers due to challenges in replicating the problem for debugging purposes. Replicating problematic responses proves uncertain because the sequence of agent execution and responses isn't consistent each time, even for the same question. For instance, when a user poses a question and receives an incorrect answer, pinpointing the issue becomes challenging because the next time the same question is asked, the agent may execute in a different sequence, due to the conversational nature.

In summary, enhancing the accuracy or quality of responses from these agents entails striking the right balance among agent parameters, as discussed, and equipping them with the necessary tools to arrive at correct answers. Furthermore, predicting the conversation flow for debugging incorrect answers requires further investigation.

Processing Time

In this section, we'll examine the processing time of a multi-agent LLM. To do this, we prepared a set of 100 question-SQL pairs. These pairs resemble the example below, although some are longer and involve multiple SQL joins:

Question:

“What kind of sound settings with preset 12, bass settings 2 and ECU version 5 are present in car Beta for the past 3 months in North America?”

Answer:

```
SELECT * FROM sound_settings_3 WHERE car_type = 'Beta' AND preset = 12 AND bass_settings = 2 AND ecu_version = 5 AND region = 'North America' AND DATE_SUB(CURDATE(), INTERVAL 3 MONTH) <= date_column;
```

On average, the time it takes from a user asking a question to the admin agent providing the final answer is approximately 35 seconds. We evaluated the final framework using various questions from the set of 100 question-SQL pairs. For simpler questions, the response time can be as short as 15 seconds. However, for more complex questions that necessitate multiple interactions between agents, the response time can exceed a minute. The factors contributing to this slower response time can include:

1. Agents looping and not terminating.
2. Agents talking about redundant information. Usually, this happens when the incorrect next agent is selected as discussed previously.
3. GPT-4 related delays including but not limited to exceeding context window related errors.
4. Agents taking time to self-correct buggy code or incorrect SQL syntax errors.
5. Typical programming delays like timeout exceptions to reconnect to the database or delays with API requests for the WebApp backend deployed in the cloud.

Reducing processing time, particularly when scaling the framework to accommodate more users, requires additional effort and investigation.

Teachable Agents

The exploration of teachable agents or learnable agents presents a promising avenue of research for various business applications. Within the automotive industry, there's an active investigation into their potential utility, such as voice assistants adjusting to user preferences or context-aware LLMs generating tailored responses from past interactions. Initially, we conducted tests on teachable agents using our database, but they were not integrated with the multi-chat system. To clarify, the teachable agent was not incorporated into the group chat manager and was tested independently.

During the stand-alone tests, the teachable agent demonstrated its ability to save crucial information as memos to the Chroma DB even after the program was closed. When conversations resumed, it recalled these saved memos, loading them for reference. If a question was repeated that had already been answered in the past, the teachable agent adeptly retrieved the answer from its memory instead of re-requesting GPT-4 to generate a response.

We proceeded to test the integration of the teachable agent into our multi-agent framework. We included the teachable agent in the group chat manager along with its necessary functions to ensure its participation in conversations. However, two issues arose:

1. Firstly, there were instances where past memos weren't retrieved as expected. This occurred because the admin agent failed to correctly designate the teachable agent at the appropriate times, instead mistakenly selecting another assistant agent like the engineer to speak next.
2. Secondly, the text analyzer component of the teachable agent sometimes saved redundant information from previous conversations, particularly in lengthy exchanges. As all conversation data was analyzed and saved into memos upon program termination, unnecessary and repetitive content was stored. Examples of such redundant information include incorrect SQL queries, metadata related to the agents, and superfluous dialogue. Additionally, it saved lengthy results of queries when saving only the SQL query would have sufficed.

Below are examples of saving redundant agent metadata and obvious talk:

Saving redundant agent metadata:

Input: What is the name of the agent and its role? What function was called and what were the arguments?

Output: {'Agent_Name': 'teachable_agent', 'Role': 'assistant', 'Function_Call': {'Function_Name': 'fetch_database_schema', 'Arguments': '{}'}}}

Saving redundant talk:

Input: "What was the name of the agent I was interacting with and what role did they play?"

Output: {'Agent_Name': 'teachable_agent', 'Role': 'user', 'Message': "Of course! I'm here to help. What would you like to talk about?"}

The primary reason behind the issues with the teachable agent stems from the ongoing nature of its integration with the group chat manager agent, which has not undergone comprehensive testing yet. Additionally, there is room for improvement in the text analyzer logic, particularly in how it sends blocks of memos to the GPT-4 model. Currently, the model is prompted to determine the importance of the requested block, responding with a simple Yes or No. However, this approach is rudimentary and warrants further exploration [3].

Regarding the business applications we discussed, it's clear that teachable agents would require customization to suit specific use cases. Currently, its integration with other agents of AutoGen is not flawless. However, when operating independently, the teachable agent shows promise, functioning as expected.

Fine-tuning

We conducted fine-tuning on a custom dataset comprising approximately 2000 instruction and output pairs. This dataset was created using inference from the Llama-2 7 billion model, as discussed earlier. Initially, we began the training process using the original Alpaca dataset, which consisted of around 60000 question-answer pairs. Due to the lengthy training time of approximately 11 hours per epoch, we only ran one epoch initially to verify that everything was functioning correctly and that the model was being saved properly.

Subsequently, we loaded the custom dataset and trained it for 15 epochs, employing the following LoRA configurations:

```
alpha_pattern: {},
base_model_name_or_path: meta-llama/Llama-2-7b-hf,
bias:none,
fan_in_fan_out: false,
inference_mode: true,
init_lora_weights: true,
lora_alpha: 32,
lora_dropout: 0.05,
peft_type: "LORA",
r: 8
```

We set up four nodes to distribute the model weights across four GPUs, with each GPU consuming approximately 6GB of memory. Here are the training configurations:

```
enable_fsdp: bool = True
batch_size_training: int = 2
batching_strategy: str= packing
context_length: int=4096
gradient_clipping_threshold: float = 1.0
num_epochs: int = 15
lr: float = 1e-5
weight_decay: float = 0.0
gamma: float = 0.85
use_fp16: bool= True
peft_method: str = lora
```

In our initial attempts, we encountered an issue where the distributed model could not be saved because the evaluation loss was undefined, represented as NaN (Not a Number). To address this, we adjusted the batch size from 4 to 2 and changed the learning rate from 1e-4 to 1e-5. These changes allowed the model to be successfully saved after training completion. Here are some notable results from the training process:

```
Avg_train_prep = 6.940
Avg_train_loss =1.937
Avg_eval_prep = 7.941
Avg_eval_loss = 3.718
Avg_epoch_time = 14760.374
```

When we tested the saved model using sample questions from our multi-agent LLM work, we found that the fine-tuned model did not yield comparable results to the multi-agent LLM. Instead, it mostly produced hallucinated responses. Here's an example of a question-response pair from the fine-tuned model:

Answer this question:

What kind of data is present in the file driver_assistance_saved_command.html?

Model output:

There are 2 types of struct in this file driver_assistance_saved_command.

This is my understanding as the file has a struct that contains 2 fields.

The id is a long type.

Description <pre> ... </pre>

Discussion on Fine-tuning

Upon testing the fine-tuned model, we observed that it predominantly produced inaccurate responses, often hallucinating information. In contrast, the multi-agent responses were significantly more refined. This was because the multi-agent approach either derived its responses from the RAG metadata we saved or retrieved results from the database using its agents and functions. The only discernible advantage we identified for fine-tuning at this stage was the inference time. On average, the fine-tuned model provided results within 5 seconds, whereas the multi-agent approach, as previously discussed, could take anywhere from 15 seconds to over a minute depending on the question.

Fine-tuning with multi-GPU training presents challenges in optimization. We invested considerable time testing various configurations of training parameters before achieving convergence and successfully saving the trained model. Additionally, training is also expensive as the cost for just the final run of training on a small dataset can be estimated for 15 epochs (average epoch time ~ 4 hours) with 32 \$ per hour for 4 GPUs. It's important to note that there are additional costs incurred in setting up the appropriate training environment for multiple GPUs.

Fine-tuning is advantageous for guiding responses in a specific tone and enhancing inference time. However, multi-agents with RAG offer a superior approach for several reasons:

1. **Scalability:** Multi-agents with RAG can effortlessly scale to accommodate new and larger datasets as they require less effort to adapt to new data. In contrast, fine-tuning would necessitate retraining or employing advanced techniques like transfer learning.
2. **Predictability and Error Correction:** Multi-agents with RAG generate responses in a stepwise manner, making them more predictable. This stepwise approach enables easy identification and correction of errors, thus improving accuracy over time. Fine tuning on the other hand is more of a black box, especially for one who is not an expert in LLM training and optimization.

Fine-tuning was conducted towards the end of my internship as an additional experiment. Our primary time spent during fine-tuning was on tasks such as dataset generation using Llama-2, setting up a multi-GPU environment with FSDP in Azure ML Studio, and conducting basic training on our test data. However, fine-tuning demands thorough investigation to optimize results, necessitating a significant time investment beyond the scope of this internship.

Conclusion

Throughout the internship, exploration and experimentation were conducted in the domain of Large Language Models (LLMs), focusing primarily on the development of a multi-agent LLM framework within the AutoGen environment. The objective was to enhance information retrieval capabilities along with conversational responsiveness.

The multi-agent framework is a step closer to creating Artificial General Intelligence (AGI). It works by having different agents work together like a team to handle many tasks at once, just like how AGI would handle various challenges. This approach helps break down big problems into smaller, easier parts for better problem-solving.

The analysis and discussion sections highlighted both the strengths and limitations of single-agent and multi-agent approaches in LLM frameworks. Challenges encountered during the internship, such as controlling conversation flow and integration issues with teachable agents, underscored the importance of systematic testing in multi-agent LLM development. Additionally, the comparison with fine-tuning revealed the trade-offs between faster inference times and the superior scalability of RAG approaches.

The internship has been a very valuable learning experience, providing hands-on experience in recent advancements in LLMs. The insights gained will serve as a foundation for future endeavors in scaling conversational LLM systems within enterprises.

References

- [1] Hu, Edward J., et al. "LoRA: Low-rank adaptation of Large Language Models." *arXiv preprint arXiv:2106.09685* (2021).
- [2] Lewis, Patrick, et al. "Retrieval-Augmented Generation for knowledge-intensive NLP tasks." *Advances in Neural Information Processing Systems* 33 (2020): 9459-9474.
- [3] Loynd, Ricky. "AutoGen's Teachable Agents." 26 October 2023.
<https://microsoft.github.io/autogen/blog/2023/10/26/TeachableAgent/>.
- [4] Wu, Qingyun, et al. "AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework." *arXiv preprint arXiv:2308.08155* (2023).
- [5] Zhao, Yanli, et al. "PyTorch FSDP: experiences on scaling Fully Sharded Data Parallel." *arXiv preprint arXiv:2304.11277* (2023).