Patterns for Conversational AI Applications

KYLE BROWN, IBM Corporation
STEPHEN MEIER, IBM Corporation

In this paper we will discuss a set of emerging patterns for building Conversational AI applications that interact with users through natural language. The pattern language is meant to help developers and architects in constructing these applications by providing a conceptual framework for positioning many common user interaction paradigms and technologies such as Large Language Models (LLM's).

Categories and Subject Descriptors: Computing methodologies [**Natural language generation**]: Artificial intelligence—*Natural language processing*; Software and its engineering [**Software development techniques**] Software creation and management

General Terms: Software Patterns, Artificial Intelligence

Additional Key Words and Phrases: Generative artificial intelligence, prompt engineering, conversational systems

ACM Reference Format:

Brown, K. and Meier, S. 2024. Patterns for Conversational AI Applications. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2024), 22 pages.

1. INTRODUCTION

Conversational Interfaces as a User Interface paradigm have existed nearly since the invention of interactive computer terminals and applications. A *Conversational Application* is one that uses a Conversational Interface. However, conversation has been a secondary user interface approach for the biggest part of that history. The notion of a conversational interface probably began with Alan Turing's 1950 paper Computing Machinery and Intelligence that defined the principles of the Imitation Game that became popularly known as the Turing Test. There have been some areas in which conversational user experiences have predominated – for instance, text-based computer gaming such as MUD's (Multi User Dungeons) and other descendants of the original Adventure computer game written by Will Crowther in 1975.

There were early attempts, such as Microsoft's Rover Digital Assistant (part of Microsoft Bob, released in 1995) and the Microsoft Office Assistant, Clippy, which was part of Microsoft Office from 1997 to 2003, to broadly introduce conversational interaction into business software, but these were often derided, as the technology was simply not ready to meet user expectations. As a result, conversation as an interface paradigm mostly remained the province of science fiction (such as the Enterprise's main computer in the TV Show Star Trek and all its descendants) for most of the first six decades of computing.

Conversational interfaces began a renaissance of sorts with the advent of voice-based agent technologies relying on more accurate speech-to-text and text-to-speech translation powered by deep learning neural networks.

Author's address: Kyle Brown, 4205 S Miami Blvd, Durhan, NC 27703; email: brownkyl@us.ibm.com; Stephen Meier, 3039 E Cornwallis Rd, Research Triangle Park, NC 27709; email: stephen.meier@ibm.com

Examples include Apple's digital assistant Siri, released in 2010, and IBM's Watson, which won a special tournament set up by the TV game show Jeopardy! in February 2011. These led to a small boom for personal digital assistants that was cemented by the release of Amazon Alexa in 2014 as the personal digital assistant became part of the Internet of Things (IoT) explosion at the time.

However, even these later iterations of *Conversational Applications* failed to make a wide impact on user interface design or on the way in which users interact with computers. This was because of the limitations of the Traditional Chatbot approach – that is that the set of questions that could be understood, and the set of responses that can be returned, were relatively small and fixed. Thus, for narrow purposes like asking Siri about the weather, or asking Alexa to play music from your smart speaker, they work well, but you couldn't engage with either digital assistant in general conversation. For most computer-human interaction purposes, interacting with an application-specific graphical user interface, either on the web, or in a mobile application, remained the preferred interaction paradigm.

That all changed with the introduction of Large Language Models based on the transformer architecture. The introduction of Chat-GPT in November 2022 was a user experience change on the same order of the introduction of the mobile applications for smartphones or the invention of the graphical web browser. You could interact with an LLM on almost any topic and carry on natural-seeming conversations that were not limited by a fixed set of questions and answers. This allowed new users of the technology to explode as people began using Chat-GPT for tasks that could not have been imagined only a few years ago, such as composing poetry and writing fiction in addition to more mundane tasks like summarizing articles and generating emails.

This introduction of Chat-GPT and the resulting explosion of interest in Generative AI as used in Conversational Applications has made it imperative that we begin to explore the common architectural patterns in this area. This exploration will be of interest to application developers, architects and data scientists.

We will explore the structure of Conversational Applications as a whole, discuss patterns for building them that emerged before the introduction of Generative AI, and then explore the newest emerging patterns that stem from introducing Generative AI into this space. We begin our exploration with the root pattern of our pattern language, *Conversational Application*.

2. CONVERSATIONAL APPLICATION

Any traditional graphical or web-based interaction model lends itself well to a finite set of actions arranged as a fixed, common progression. However, in many cases, the starting place for interaction may be essentially undefined and the progression unpredictable.

How do you help a user perform an action or find data if they are not sure how to perform the action or locate the data?

In many types of applications, such as customer support, it is often hard to build a web site or application that naturally enables people to go directly to the specific piece of information that they need. Users resist navigating deep hierarchies of static pages, and FAQs become unwieldly when there are more than a few dozen entries. Search tools like Google can alleviate this somewhat, but that depends on both the information developer and the user seeing eye-to-eye about the selection of keywords and other aspects of search engine optimization.

What is needed is some way to meet these users where they are - to give them an interaction that either feels like interacting with a human, or in which they may be finally interacting with a human, intermediated through a program that may first organize the category of needs and other prerequisites that a human requires to satisfy them efficiently.

Therefore,

Build a Conversational Application that interacts with users through natural language conversations by mimicking a human conversational interaction.

A *Conversational Application* is one that allows a user to interact with a computer through text input or voice input, with either text output or voice output. It parallels the idea of a conversation between two people in which information (sentiments, observations, opinions or ideas) are exchanged verbally. *Conversational Applications* give users a better, more interactive, and more customized user experience than traditional applications in that they process natural queries and provide responses in the user's natural language. These applications can be standalone or integrated into existing user interface paradigms. The structure of a generalized *Conversational Application* is shown below in Figure 1: Conversational Application.

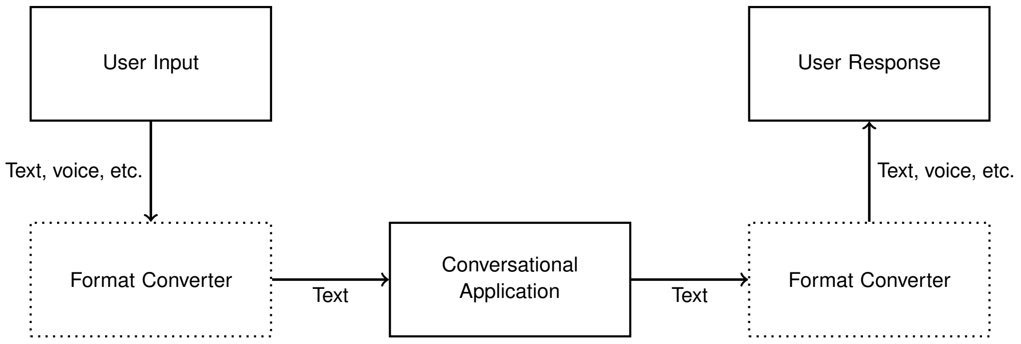

Fig. 1. Conversational Application.

In Figure 1 above we show some of the key aspects of a *Conversational Application* – the first being that it naturally communicates in text. Where there are other input forms (voice for instance) those must be converted to text for communication with the Conversational Application. Second, they are conversational – meaning a user provides an input and then waits for a response. This conversation may last over several request-response pairs, or may be a single interaction.

As described above, *Conversational Applications* are most applicable when the user requests can be ambiguous or may cover a wide variety of topics. In a traditional graphical user interface (which includes web interfaces) users must choose actions from a list of menu items, buttons or other graphical icons. That requires the user to understand at a minimum the structure of the application navigation – which means the user must share an understanding of the knowledge representation that the application designer built into the application. Imagine a vendor of spare parts for appliances; often they would build an application to allow for a search by something like part name or part number. That assumes a certain level of knowledge on the part of the person using the application. That person may not know to search for "water pump for a 2022 LG TurboWash 360 washing machine" – they instead want to know "how do I fix my washing machine when the panel reads 00 – Water error". The first instance is unambiguous, the second is more ambiguous.

The main advantage of this kind of application is that it allows for resolution of this level of ambiguity in intent on the part of the user – the user can literally ask the application to do anything or provide the answer to any question. That is also one of the biggest downsides of this kind of application – the most common answer to the universe of possible requests is "I don't know" or "I can't do that". Instead, the user interface designer of the application needs to guide the user to a specific subset of questions or a specific domain of knowledge that the Conversational Application can deal with.

There are many ways to build a Conversational Application, starting with the older approaches used in the historical examples we mentioned earlier, which we can call Traditional Chatbots, and then moving on to newer approaches using LLM's, as shown in Figure 2: Conversational Application Pattern Map.

While Traditional Chatbots are the simplest, most straightforward type of Conversational Application, they have been largely supplanted over the last few years by the introduction of applications based on Large Language Models. We can call that subtype of Conversational Application that uses an LLM an LLM Chatbot.

However, simply connecting a user to a Large Language Model (which is the approach taken by the simplest chatbots based on Chat-GPT) has several limitations.

An LLM may formulate a plausible sounding, yet factually incorrect answer that is commonly called a hallucination [Lewis et al. 2021]. This could include making up a biography for someone who doesn't exist or incorrectly assigning the wrong year to a historical event. Such hallucinations can be dangerous in a production environment where decision may be automated based on model output. This is something that every company should have some strategy to fight against to see the long-term benefits of generative AI. This is an important area of ongoing research [Huang et al. 2025].

When you ask a question of a large language model it formulates an answer by giving you the most statistically likely combination of words and terms that match the structure of the question you have asked. Where this statistical approach can fail is when you reach the end of the general knowledge base on which the large language model was trained. Likewise, hallucinations can occur in other situations such when the model is overfitted for a particular task and can produce text inappropriate to a request as a result [Huang et al. 2025].

The problem of hallucinations where knowledge ends led to the development of approaches to ground the Large Language Model's responses in specific examples of factual information that match the subject of a particular query. The first approach, Retrieval Augmented Generation (RAG), uses a Vector Space search to locate concepts in Vector space that are "close" to the concepts in the query, thus improving the specificity of the result returned by the Large Language Model. Long-Tail RAG combines the accuracy of pre-prepared answers of a Traditional Chatbot with the ability of the LLM and RAG to handle questions that fall outside of the narrow range covered by a Traditional Chatbot.

One can improve on the performance of RAG by re-ranking the results post the semantic search [Glass et al. 2022]. This has led to a technique known as RAG-Fusion, which employs reciprocal rank fusion (RRF) to reorder the search results prior to prompting the LLM. The RRF algorithm fuses together the results of multiple search queries based on the aggregate document score. As a result, documents ranked high amongst multiple search queries achieve a higher re-ranking. RAG-Fusion starts by prompting an LLM to generate multiple queries that are similar to the user's original query. A semantic search is performed for each of these queries and then RRF fusions together the results and improves document relevance. RAG-Fusion does incur a runtime performance cost, as more time is required for processing, however the relevance of the context provided to the LLM is greatly improved [Rackauckas 2024].

As you move to handling parts of a conversation that require more than simple factual answers, you begin to venture into the space of the Modular Reasoning, Knowledge, and Language (MRKL) pattern. This pattern introduces the idea of integrating factual information from a tool like an API or a database call into an answer to fix the answer to a specific piece of information retrieved from a factual source.

MRKL relies on the idea of a conversational router to make decisions around which subsystems should be responsible for handling parts of a conversation when the conversation may cover multiple sources of information, multiple aspects of knowledge of a domain, or multiple tools that need to be invoked.

Sometimes you also need to have an LLM explain itself and then use that explanation to set the direction of further action that it needs to take grounded in real-world information. This is the domain of the Reasoning and Action (ReAct) pattern.

Finally, Conversational Applications are often built in addition to, or within traditional graphical user interfaces. However, due to their nature as text, Conversational Applications also open new avenues of interaction – they

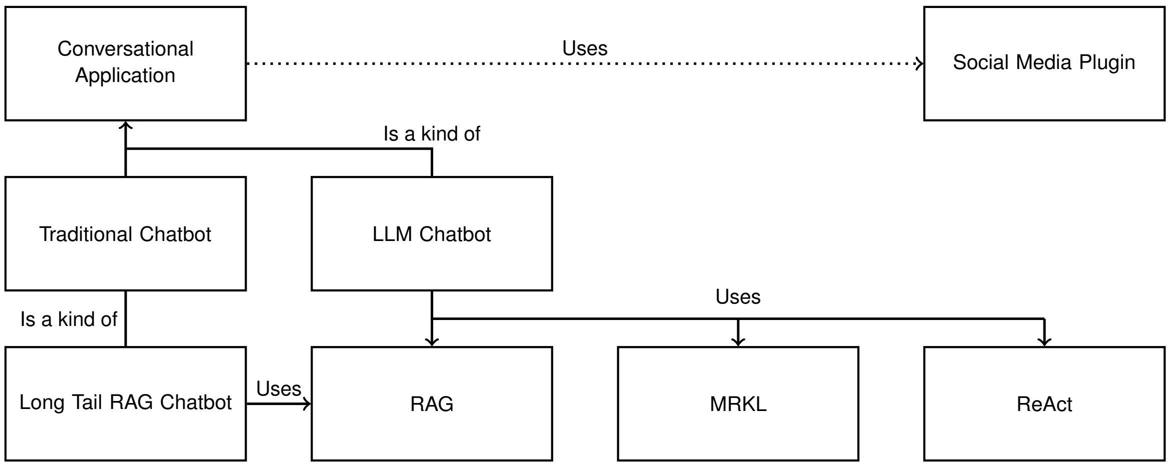

Fig. 2. Conversational Application Pattern Map.

allow you to add natural language conversation functionality to existing platforms like SMS, WhatsApp, Slack and many others through a Social Media Plugin.

3. TRADITIONAL CHATBOT

You are building a Conversational Application and are in a situation where the accuracy of information provided is important (perhaps due to life or health safety, or because of financial or reputational risk).

How do you constrain a Conversational Application to respond to specific queries where accuracy is critical with fixed, well-known answers that give guidance that has been vetted and guaranteed to be accurate by a human ahead of time?

In many situations accuracy within a Conversational Application is critical. For example, in the medical field, there are situations in which AI hallucinations or misunderstandings can be life-threatening. There have been cases (see here) where a chatbot meant to provide advice on eating disorders gave harmful advice to multiple people, causing its withdrawal from service by its owner, the National Eating Disorders Association. Likewise, there was a reported case (see here) of a man committing suicide after being encouraged to do so by a chatbot.

Even in fields where a person's life is not at stake, there have been multiple reported cases of actual or potential financial loss caused by chatbots representing businesses and providing communications that were inaccurate or that did not reflect the policies of the business. For instance, there was the humorous incident of the prankster convincing a chatbot for a GM Dealership into making an offer for a brand-new Chevy Tahoe truck for $1 (see here). In a more serious incident, Air Canada was judged liable for inaccurate information given out by its chatbot (see here).

In those cases, the possibility of AI hallucination precludes relying on an LLM, even a trained LLM grounded with RAG to provide answers. To preserve human life and health, or to eliminate the possibility of financial loss or liability, for a fixed set of questions within specific, tightly constrained domains, you must give precise answers.

Therefore,

Build a Traditional Chatbot – an intent-based system that scans for particular words and uses Natural Language Processing to identify intents and select responses.

Traditional Chatbots are not a new form of technology - in a sense they can be traced back all the way to the Eliza program written by Joseph Weizenbaum in 1966. However, recent improvements in Natural Language

Processing have made it easier and more efficient to build them than before. Most chatbots share a common set of architectural features that allow them to be built easily from existing open-source libraries or commercial web-based API's. Therefore, they can be implemented as microservices that can be called from within mobile apps, web apps, chat applications (such as Slack or Discord) and social media platforms. An example of how a Traditional Chatbot would be invoked (or embedded within) one or more other types of user interfaces, along with the relationship between the Traditional Chatbot and other parts of an enterprise application architecture, including the cloud or open-source services it relies on is shown below in Figure 3: Traditional Chatbot with other components:

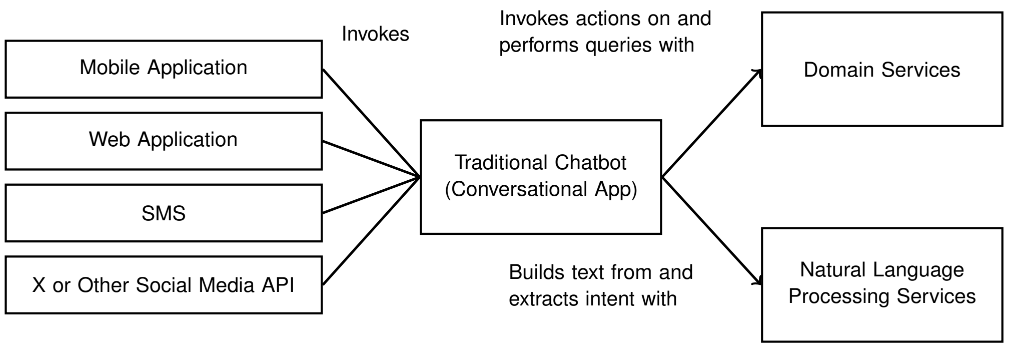
Mobile Application
Web Application
SMS
X or Other Social Media API
Invokes
Traditional Chatbot (Conversational App)
Invokes actions on and performs queries with
Domain Services
Builds text from and extracts intent with
Natural Language Processing Services

Fig. 3. Traditional Chatbot with Other Components.

An important aspect of a Traditional Chatbot that is different than many of the other Graphical Interaction Models like Web Pages or GUI's is that a chatbot can help the user navigate amongst through a decision tree of choices. Rather than a window with fields for name, address, and phone, the chatbot can ask questions of the user interactively: What is your name? What is your Address? What is your phone number? If the information has already been provided, it can simply skip the question or ask the questions in different orders.

The chatbot itself must take advantage of a set of Natural Language Processing Services that typically parse incoming text and then extract the "intent" (e.g., the relevant domain concept in terms of predetermined entities and actions) from the incoming text. Likewise, the chatbot will then either directly or through the intermediation of the Natural Language Processing Services call one or more Domain Services to perform queries whose results are then formatted by the Natural Language Processing Services as output text, or to perform some set of actions based on the user's intent. This sort of typical Chatbot architecture is shown below in Figure 4: Traditional Chatbot Components:

For instance, for simple cases, the open-source Chatterbot library is a machine-learning based conversational dialog engine for Python. It allows you to build or customize logic adaptors that take in preprocessed statements from the user and either provide you the best match to a response or invoke other adaptors for more specific domain types (like date/time or math). The system learns from a set of provided example statements in order to determine the best response.

One of the most sophisticated, systems for developing Traditional Chatbots was the IBM Watson natural language engine, derived from the research system that won the game show Jeopardy! against top human competitors in 2011. This has now evolved into the Watson Assistant suite of services, that work from a set of provided Intents (which are goals learned from multiple examples) to understand how to respond to a particular statement. Likewise in Watson, you can define specific Entities that are found in the conversation and even specify a conversational flow for a complex interaction that includes slots to hold the different entities you identify (such as the time for a dinner reservation). Actions are the tasks that the system undertakes on a user's behalf (such as searching for information as in at what times tables are open at the restaurant).

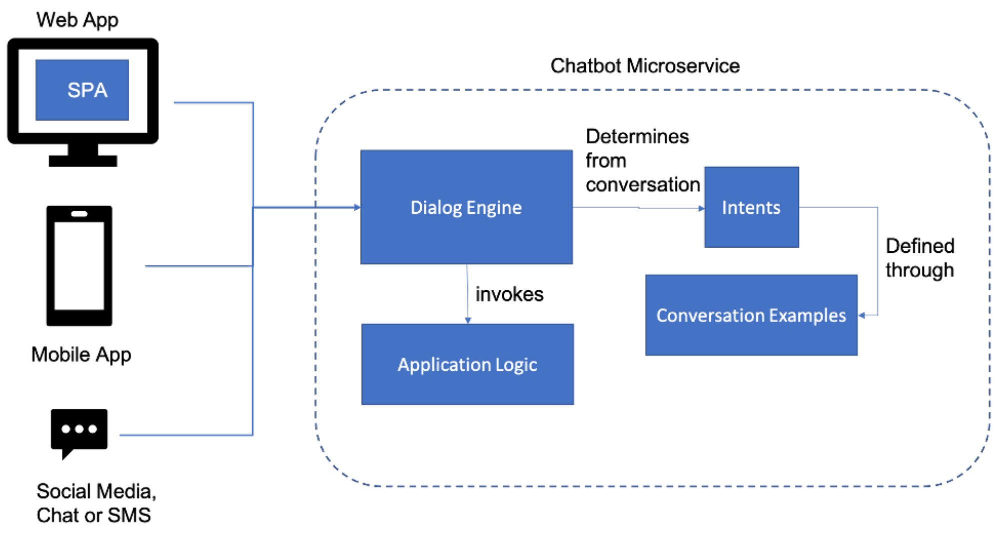

Fig. 4. Traditional Chatbot Components.

Somewhere in between the two ends lie the Amazon Lex service, which is connected to the suite of services that includes the Alexa voice service. It also allows you to define Bots built up from Intents derived from samples but fulfills each matched intent with a custom function built on AWS Lambda.

The biggest problems with Traditional Chatbots is that they are both enormously expensive to implement because each intent must be determined by a human, which requires not only deep domain expertise, but data science expertise to describe the variations of each particular intent and conversation example, but also limited in how many questions that they can answer for the very same reasons.

Another problem with Traditional Chatbot is that it is representing threads of conversation that continue across multiple question/answer pairs is challenging. A Traditional Chatbot will answer each question by parsing the natural language and comparing it against its set of intents – it may not realize that a question asked earlier may define a specific intent out of a set of possible intents. If you want a more natural conversational flow, you want to allow for conversational context that spans multiple question/answer pairs.

Any software development project is ultimately constrained on some resource, but the most important tradeoff is usually time vs. money. If you need a solution quickly but can afford to spend ongoing money on a large AI model that is pre-trained to handle conversation, hosted on the cloud or running on an in-house GPU inference cluster (which are expensive, but are a one-time charge) that is often an acceptable tradeoff.

An important consideration in building a Traditional Chatbot is that you must consider that it is a user interface channel into an Enterprise application much like any other user interface choice. That means you must plan for connecting requests from your chatbot into the back-end API's that your system presents, although due to the conversational nature of the interaction, these API's often end up being heavily mediated to strip down large amounts of data into smaller answers more uniquely suited to the nature of chat.

4. LLM CHATBOT

You are building a Conversational Application, and you want to provide answers that are well-structured natural language. However, you are limited by available time for development and data science.

How do you develop a Conversational Application quickly and cheaply without having to predefine all the possible combinations of intents and fixed answers required by a Traditional Chatbot?

One of the most constrained resources in most development shops is developer time. When you need a Conversational Application, you want to spend your limited development time on those tasks that are unique to your business – not on giving basic conversational abilities to a chatbot. However, that is exactly what happens when you build a Traditional Chatbot – you often end up spending lots of development time on defining things like synonyms and alternatives for intents that trigger a prepared response. What's more, data science time is also limited. Researching potential answers, defining test cases, etc., all take time.

As discussed in Traditional Chatbot, representing conversational threads (e.g. maintaining conversational context) is challenging in the simple question-answer pair format of a Traditional Chatbot. This has led developers to look for opportunities to address this issue with approaches that offer better "memory" of previous conversations.

This is where LLM's have come to the forefront of the range of possibilities for addressing these issues. Large Language Models exist with relatively large "context windows" allowing not only the most recent question, but potentially a number of previous Q&A interactions to be presented to the LLM as part of a prompt.

Therefore,

Develop an LLM Chatbot that uses the generational capabilities of Large Language Models to generate plausible answers to natural language queries.

Since the introduction of ChatGPT in November 2022, the advantages of using large language models in chatbot implementations have been abundantly clear. The main advantage of developing a chatbot with a large language model is that it is easy to get the chatbot to handle standard queries requiring general knowledge. Instead of laboriously defining a large set of potential questions and answers and constructing a chatbot from them, development instead mostly consists of prompt engineering – or constructing specific prompts to the LLM that will give the best output for a particular set of desired questions and perhaps fine tuning – or updating the weights of a large language model by training on a domain-specific dataset.

As noted above, this makes chatbot development relatively easy. However, prompt engineering is an emerging skill set that is often difficult for companies and development teams to find, requiring existing engineers to have to retrain to gain this skill set.

The good news is that many frameworks like LangChain for development of Conversational Applications that use LLM's are both widely available, and well-documented and understood. That means that in most cases, only a relatively limited time is usually needed for prompt engineering and/or fine tuning to achieve good results.

As anyone with experience with ChatGPT can attest, the answers provided by an LLM are usually very convincing. The issue is that while the answers always sound very confident, they may not be factually correct. When this happens, which means a statistically likely, but factually incorrect answer is given, this is called a hallucination and is one of the main drawbacks of using LLM's.

There are other drawbacks as well. One possibility is the generation of text containing Hate, Abuse and Profanity. You may want to filter both input and output against the possibility of both prompts asking for the generation of inappropriate content, and for the generation of inappropriate content. Often that is done by combining encoder-only models with classifiers to judge the resulting text as falling into an inappropriate category, but that can also be done with traditional NLP techniques.

Another potential drawback is reputational risk stemming from an exploit against the LLM itself. Many social media users will be familiar with the words "ignore all previous instructions and…" which has been used for detecting the presence of an LLM in social media. This is known as "Jailbreaking" and is the process of manipulating prompts to generate content in an unexpected or harmful way. Like in HAP filtering, Jailbreaking is something developers of LLM chatbots must protect against.

Another potential drawback is cost, especially when using the largest available models. Larger models have higher computational cost due to the higher amount of GPU computation that is required to generate text from those models. That cost is passed along to the user of the model either through having to purchase or rent additional Graphics Processing Units (GPU)'s on the cloud, or by the vendors of AI model API's such as OpenAI. Thus, you want to use the smallest, most cost-efficient model you can while still getting the most effective responses.

Finally, there is a set of potential security issues that need to be addressed as well. If you train a model on confidential or private data, there is no way to prevent it leaking out to anyone who uses that model. Once information is encoded into the weights of a neural network there are no good ways to keep the information private – you can only filter responses after they are generated.

That last concept is one of the reasons why Retrieval-Augmented Generation (RAG) is a popular and important pattern. In RAG, you can keep confidential or private data outside of the LLM itself in a separate vector database, which you can then impose security restrictions on. Likewise, with the Reasoning and Action (ReAct) Pattern, you can also keep confidential or private data outside of the LLM by connecting to enterprise systems with an API. In both cases, the data is passed into the LLM through the prompt and thus does not become encoded in the LLM weights.

5. RETRIEVAL-AUGMENTED GENERATION (RAG)

You are building a Conversational Application, and you need to make sure that the answers provided by the LLM are grounded in a set of known facts, and that the possibility of hallucination is reduced as much as possible.

How do you improve the accuracy of an LLM when you have a large, but constantly changing corpus of information that the LLM can draw from?

Training and fine-tuning a large language model (LLM) is a difficult and an expensive task. It is unreasonable to expect that development teams, especially those without deep expertise in training and fine-tuning models will have the expertise, time, or ability (in terms of GPU compute power) to perform fine-tuning or especially retraining of an LLM.

What's more, there are situations in which it's not feasible to train or fine-tune a model for a particular circumstance, purely based upon the type of questions that you are building a system to answer. If you have a set of information that changes frequently, then a retrained LLM will give answers based on the last version of the data that it was trained on. This can be an issue if you are training it to answer questions about data such as contracts, policies, or other legal documents where giving an answer based on older data could create legal liability or cause financial harm. What you need is a mechanism to allow the LLM to generate answers but based on a combination of general knowledge of the world plus knowledge specific to the particular situation at hand.

Therefore,

Use Retrieval Augmented Generation (RAG) to combine the generative power of an LLM with the accuracy of a text search for specific, accurate answers drawn (as much as possible) from a known corpus.

Retrieval-augmented generation (RAG) provides a way for application developers to pass documents to the model that are relevant to a particular user query. The components of the RAG pattern in its most common form are shown in Figure 5: RAG Pattern Components.

For example, if an employee in your company was to ask about details related to your human resource (HR) policies the LLM will not be able to provide an accurate answer because the model was not trained on data related to you HR policies. However, with RAG you can add the relevant snippets of HR documentation which the model can synthesize and summarize to provide the user with an answer that is better tailored to your company.

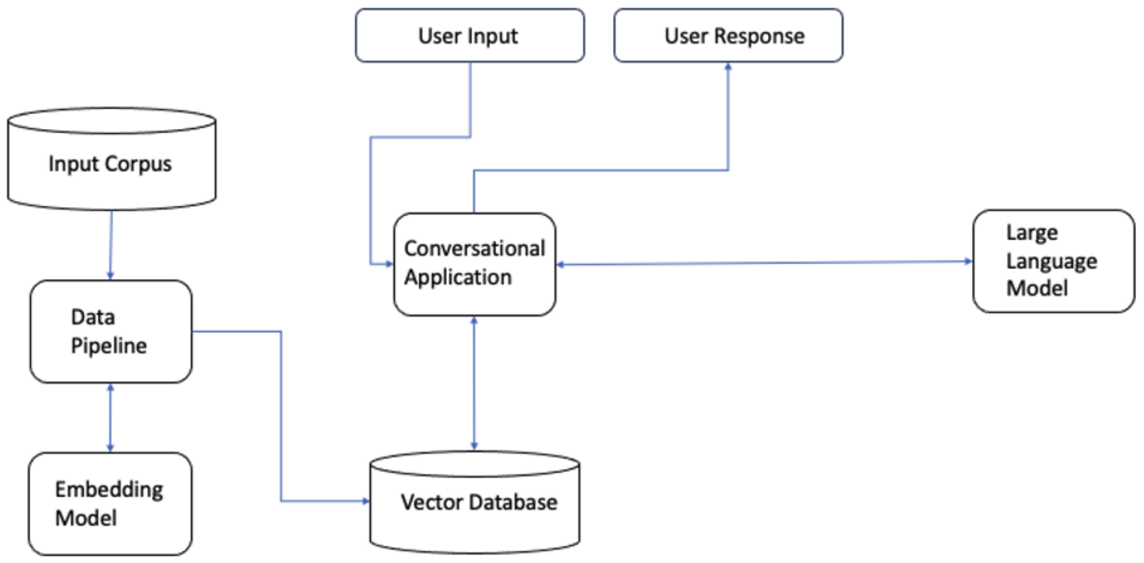

Fig. 5. RAG Pattern Components.

There are many possible architectural designs one can leverage to implement RAG. Generally speaking, the approach is broken down into the following steps:

(1) The user asks a question to the model.
(2) The user's query is converted to an embedding vector that describes the location of the concepts in the query in vector space.
(3) The embedding vector is used to retrieve a list of semantically similar documents that are "nearby" in vector space.
(4) One or more of these semantically similar documents are added to the in-context memory of the LLM via prompt engineering.
(5) The prompt, containing both the user's query and relevant documentation, is submitted to the model.
(6) The model sends back a response that is more contextual given the included documentation.

To retrieve the relevant documents via semantic search the documents must first be converted to embedding vectors and stored in a vector database. A vector database allows for efficient indexing, such as Hierarchical Navigable Small Worlds (HNSW) [Briggs 2023], and a variety of similarity metrics one can use to calculate the distances between vectors (for examples see [Mil 2023]).

The major benefits of RAG include that the accuracy of the resulting text produced by the chatbot is often significantly higher than when an LLM is simply asked the question without the additional grounding provided by the documentation retrieved by the vector search. Metrics such as the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) score are often used to measure the accuracy of the response generated by the LLM. This is done by measuring the number of overlapping n-grams between the LLM's response and the ground truth, human annotated, response. A high degree of overlap indicates that the LLM has managed to generate a response similar to how a human expert would response to that particular question [Lin 2004]. ROUGE gives us an understanding of how well the initial documentation was incorporated into the final answer provided by the LLM. What's more, since the Vector database can be updated at any time, this approach makes it possible to provide answers that reflect updates to knowledge nearly immediately.

The largest downside of RAG is that the data science effort required to identify, clean, embed and maintain the Vector database can be substantial. There is also effort required to determine the appropriate way to split up or “chunk” the data (each vector can only represent a relatively small piece of information) for storage in the database, and sometimes contextual information present in the structure of the original data source (such as document headers, footnotes, etc.) can be lost.

5.1 Example Prompt

Below is an example prompt plus model response that is leveraging the RAG pattern. It is worth noting that in this example the model is also providing a reference to the source it used for generating the response to the user. This is an important addition that adds an increased level of explainability. If the user decides to review the reference material at a later point, they have the reference there available to them.

6. LONG-TAIL RAG

You are building a Conversational Application with a Q&A interface. You have a few hundred very well-constrained "Frequently Asked Questions" but there are many thousands of other questions and subjects your users may ask about that are outside of the FAQ list.

How do you handle the "long tail" of questions in a Q&A experience?

The basic problem with a Traditional Chatbot is that the set of questions that it can handle are limited. It essentially only knows how to answer a fixed set of questions that map to a particular set of terms or Intents that can be extracted using NLP capabilities from the questions asked in the chat. The problem is that the set of questions that a human can ask is much larger (practically infinite) than the set of specific intents that can be identified and for which answers can be pre-planned for a Traditional Chatbot.

The reason that Traditional Chatbot technologies have been successful is that in many domains the most asked questions represent a large percentage of the total set of questions that are practically asked. If, for instance, you are conversing with a Chatbot that is part of an airline booking website, most interactions will be about a small set of questions - what are the flights between these cities and when do they leave and arrive? How much is a ticket on this flight? What is the cheapest ticket between these destinations if I’m flexible in my time? What seats are available on this flight? It would be a rare interaction that is not about some combination of origin and destination, time, price and seats. However, you can easily imagine those rare interactions - that might include questions like "Will my particular child’s car seat fit on this particular plane seat?" Until recently, the only option for answering questions like this would involve bringing a human into the conversation loop to answer the question. With the advent of Large Language Models, this has changed.

However, Large Language Models are not a panacea for all conversational use cases. The problem is that LLM’s have the potential to hallucinate or provide factually incorrect, yet statistically likely answers, all depending upon how they have been trained. The autoregressive nature of LLMs means that the next likely token in the sequence is conditional upon all previous tokens. The model learns which of these previous tokens to pay the most attention to during training, but at inference time these weights remain constant and the model can pick up on spurious correlations between tokens. Even in patterns like Retrieval Augmented Generation there is still the potential for hallucination, even when the answers are mostly drawn from a well-defined corpus of information expressed as embeddings. So, the set of questions that can be answered is much larger (again, potentially infinite) but the accuracy of the answers may be suspect, especially as the questions range farther afield from the original training material.

```
Given the following extracted parts of a long document and a
question, create a final answer with references ("SOURCES").
If you don't know the answer, just say that you don't know. Don't try to make up an answer.
ALWAYS return a "SOURCES" part in your answer.

QUESTION: Which state/country's law governs the interpretation of the contract?
=========
Content: This Agreement is governed by English law and the parties submit to the
exclusive jurisdiction of the English courts in relation to any dispute (contractual
or non-contractual) concerning this Agreement save that either party may apply to any
court for an injunction or other relief to protect its Intellectual Property Rights.
Source: 28-pl

Content: No Waiver. Failure or delay in exercising any right or remedy under this Agreement shall
not constitute a waiver of such (or any other) right or remedy. 11.7 Severability. The invalidity,
illegality or unenforceability of any term (or part of a term) of this Agreement shall not affect
the continuation in force of the remainder of the term (if any) and this Agreement. 11.8 No
Agency. Except as expressly stated otherwise, nothing in this Agreement shall create an agency,
partnership or joint venture of any kind between the parties. 11.9 No Third-Party Beneficiaries.
Source: 30-pl

Content: (b) if Google believes, in good faith, that the Distributor has
violated or caused Google to violate any Anti-Bribery Laws (as defined
in Clause 8.5) or that such a violation is reasonably likely to occur,
Source: 4-pl
=========
FINAL ANSWER: This Agreement is governed by English law.
SOURCES: 28-pl
```

Fig. 6. The example prompt above is from the open-source project hwchase17/langchain [Chase 2022].

Therefore,

Combine the accuracy of a Traditional Chatbot with an LLM, particularly using Retrieval Augmented Generation to only rely on the LLM for the "long tail" of questions that aren't covered by the FAQ approach of a Traditional Chatbot

As you might expect, a Long Tail RAG implementation is a combination of the components of two previous patterns, as shown in the diagram below Figure 7: Long Tail RAG components:

In this approach, the Conversational Application begins by first analyzing the user input with a Dialog Engine as in the Traditional Chatbot approach. The Dialog Engine breaks down the intents of the request, and if a match is found, that match is returned from the list of Defined Responses previously constructed. If no match is found, then the Conversational Application instead performs a search on the Vector Database and constructs an appropriate response by sending the results of that search in a prompt to the Large Language Model before then sending the response to the user.

The advantage of this approach is that you can be extremely accurate in providing specific responses that exactly match the known intents identified by the developers. The disadvantage of this is that identifying appropriate intents and formulating responses becomes yet one more aspect of analyzing the problem that the data science team responsible for also developing the data pipeline and maintaining the vector database used for RAG.

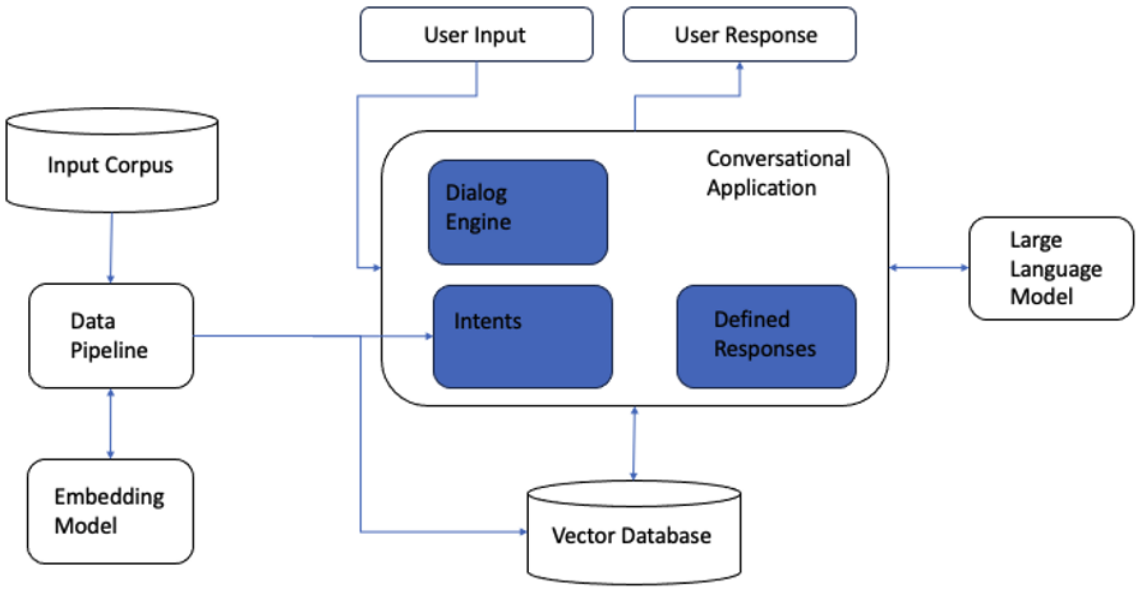

Fig. 7. Long Tail RAG Components.

Where this approach has limitations is in being able to make decisions about the direction of the conversation – this approach is good for limited Q&A systems but cannot handle more general conversations outside of the capability of the LLM. That is the domain of the Modular Reasoning Knowledge and Language (MRKL) pattern.

7. MODULAR REASONING, KNOWLEDGE, AND LANGUAGE (MRKL)

You are building a Conversational Application that needs to carry on conversations that go beyond the simple retrieval from a single known corpus of information and formulating an answer.

How do you handle situations that require retrieval of information from different knowledge bases or different types of reasoning, such as both arithmetical and language-based knowledge?

Often a question requires multiple types of reasoning to formulate a response. As we have stated previously, large-language models are wonderful at statistical analysis in determining the most likely response to a question based on the structure of language and the corpus of information on which it was trained. Where they tend to fall down is when being asked to perform tasks that require the combination of two or more types of reasoning, such as the following question:

Take the ordinal position in the alphabet of the letter "F", multiply that number by 4, and return the letter in the alphabet having that position.

A human would know that this task should be broken down into two types of reasoning, arithmetic and language. The first and last tasks are language-related tasks – for instance, knowing what an ordinal position in the alphabet means, and knowing that the answer to the first part of the question is 6. The middle part of the question asks for arithmetical reasoning – for instance, knowing that 6X4=24. In general, large language models are not terrific at arithmetical reasoning – while they may be able to work out a simple multiplication problem, a more complex problem may be beyond their capabilities. That means that they must be extended with those capabilities, but in a way that integrates into their base language capabilities.

Therefore,

Develop a multi-module system of agents that allows for different types of reasoning, knowledge retrieval or action in a Conversational Application.

The MRKL (pronounced "miracle") design pattern was developed to provide large language models (LLMs) with access to current information and task-specific reasoning without having to retrain or fine-tune the model [Karpas et al. 2022].

The design consists of a Conversational Router, and task-specific modules or agents. The router is responsible for routing a request to the correct module in order to execute on a specific task. An agent might perform an arithmetic operation, make a database API call, or trigger an automation pipeline that is responsible for performing any number of subsequent tasks. Through the interface between router and agent one can provide novel information to the model as an additional context for which it can leverage to make subsequent decisions or to make a more informed response. We show the components of this design in Figure 8: MRKL Components.

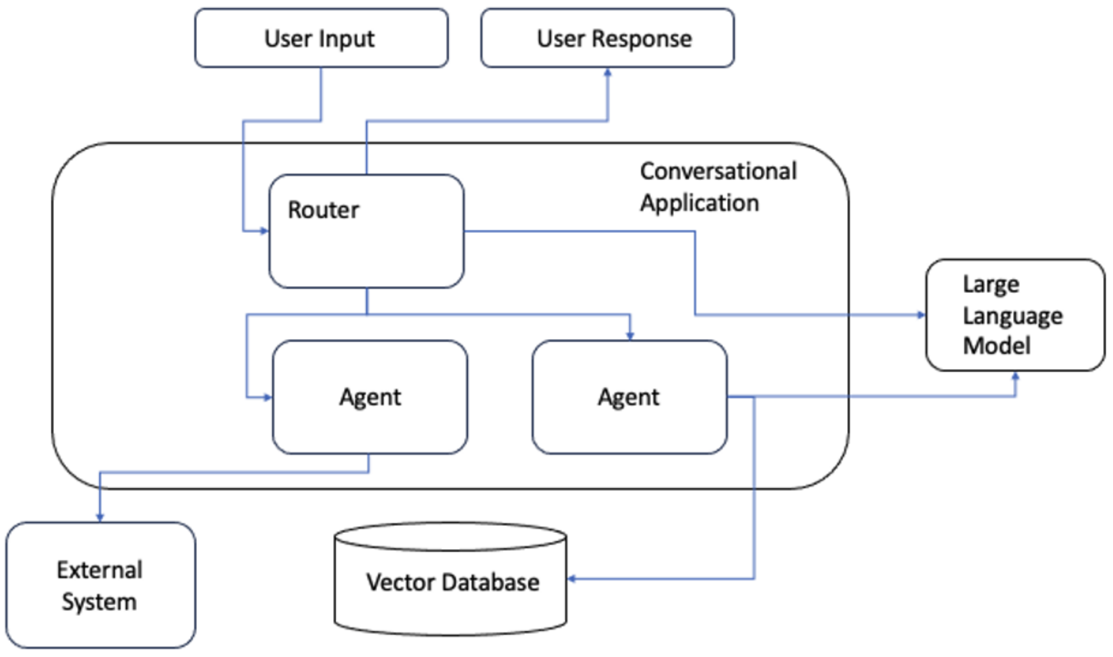

Fig. 8. MRKL Components.

7.1 Prompting the Router

The process starts with a user query and instructions detailing how we would like the LLM to respond. It is important that the response be formatted correctly so that we can pass the output to the task-specific module without any issues. Below is an example of implementing MRKL via prompt engineering. In this example the results of each task-specific request is provided back to the model via in-context memory.

The example above is made to be somewhat general to facilitate usage among a broad spectrum of use cases. However, there are several important aspects of the prompt above that are worth mentioning:

```
Answer the following question as best you can. You have access to the following tools:

read_file: Read file from disk
list_directory: List files and directories in a specified folder

Use the following format:

Action: the action to take, should be one of [read_file, list_directory]
Action Input: the input to the action
Observation: the result of the action
... (this Action/Action Input/Observation can repeat N times)
Final Answer: the final answer to the original input question

Begin!

Question: {user_query}
```

Fig. 9. The example prompt above is from the open-source project hwchase17/langchain [Chase 2022].

(1) The ‘read_file‘ and ‘list_directory‘ tools are mentioned along with a description of what the tool does.

(2) Instructions are provided for how the model should respond and how the model should expect to get back the resulting output from performing a task.

(3) Instructions for what should be said when a final answer to the user’s query is ready.

The flow above also allows for multiple tools to be called, one after another. This allows for an internal monologue to take place where the model performs an action with a particular input, gets back the result of that action, and then decides whether to call another action or respond back with the final answer. This entire script for this internal monologue is provided back to the model as part of the context to each text generation request. This script is often referred to as the agent’s scratchpad.

7.2 Argument Extraction

When a request is made by the LLM, or router, to perform a particular task it is important that the request payload is correctly formatted and aligns with the tool’s specification. This is especially important if one is to consider implementing such a design in a production system. However, this can be mitigated against by post-process polishing or making subsequent calls to the LLM and request for the error to be corrected.

7.2.1 *Error Correction Instructions.* Below is an example for how errors in the LLM’s request payload can be mitigated against by requesting for the model to self-correct.

In this example a user first asks a question related to data passed in via a Pandas dataframe. For example, the user might be interested in summary statistics for their dataset. Once the user has articulated their question, the MRKL pattern is used and the LLM is asked to generate the Python code required to answer the user’s question. In this case the tool being used is the Pandas Python library and in order to get back a meaningful answer the LLM is constrained to the Python code syntax as well as the particular interface design for the Pandas library.

Allowing for the model to correct its mistakes can be more efficient than trying to programmatically infer what the model intended. For very simple mistakes however, such as a missing semicolon or poor indentation, it is more efficient to simply polish the response by correcting the mistake on the fly.

```
For the task defined below:
{orig_task}

you generated this python code:
{code}

and this fails with the following error:
{error_returned}

Correct the python code and return a new python code (do not import anything) that fixes the
above-mentioned error. Make sure to prefix the python code with <startCode> exactly and suffix the
code with <endCode> exactly.
```

Fig. 10. The example prompt above is from the open-source project gventuri/pandas-ai [Venturi 2023].

7.3 Explainability

Knowing which tool to route to can provide an extra layer of explainability that you would not be able to achieve otherwise. This becomes even more apparent when coupled with other patterns, such as ReAct. With billions of parameters, LLMs are notorious for being difficult to interpret and this leads to difficulty in explaining how, or why, the output is what it is. With the MRKL design is it possible to keep track of which tools were leveraged for a particular user query and the output of these actions can help explain why the model can to the conclusion that it did.

8. REASON AND ACTION (REACT)

You are building a Conversational Application, and you need to be able to understand and document the process by which a Large Language Model arrives at its answers. What's more, you want the application to ground its results in verified information that is obtained from trusted sources.

How do you get an LLM to ground itself in objective reality?

There are many situations in which you need an LLM to explain its reasoning process. However, when you use chain-of-thought prompting alone, a model can only use its own internal representations to generate reasoning traces. This can result in hallucination as the reasoning becomes removed from objective reality. What is needed is a method to both reason and ground itself in observations from the outside world through interaction with the external world.

Therefore,

Capture the internal monologue of the Large Language Model by breaking the REeasoning process down into specific steps that are logged as part of an ongoing Chain of Thought. Use the output of that chain of thought to take ACTions on the real world and record the responses as part of the ongoing reasoning chain.

The ReAct pattern is based on the intuition that internal monologues are useful when performing complex tasks. As we reason through a task, we often reflect on the information being processed. This reflection can lead to subsequent actions, such as a Google search, or creating a task list. This stream of thought helps guide our decision-making process and helps us break down complex tasks into a series of smaller tasks. What is remarkable is that Large Language Models (LLMs) can generate a similar train of thought and that this approach also helps improve the language model's ability to perform some complex tasks [Lewis et al. 2021].

We show the structure of this pattern in Figure 11: ReAct Components below. The Generative AI model (shown twice) can interact with different tools to make observations of the outside world, such as executing a program, or taking a measurement. It can also interact with different databases of knowledge to find specific information that ground its results, much like as in the Retrieval Augmented Generation (RAG) pattern.

8.1 Hallucinations

One of the benefits of the ReAct pattern is that it can help fight against hallucinations by keeping the model's stream on thought in-context for further text generation calls. Keeping the stream of thought in the LLM's memory helps remind the LLM of the information it has synthesized up to this point and helps keep the series of actions on track and aligned with the ultimate goal by revisiting overarching themes.

8.2 Chain of Thought

Below is one example of the kind of internal monologue one could expect from a LLM if prompted correctly. The chain of thought is broken down into a few different sections:

(1) Question: This is the question being asked of the LLM by the user.
(2) Thought: This articulates the LLM's reasoning behind performing a particular action while often also summarizing the observations from the previous action.
(3) Action: This is the action being proposed by the LLM (see here for additional details). This section is often formatted to make the interface between LLM and API seamless (i.e. JSON or the like).
(4) Observation: The output, or feedback, from performing a particular action.

This internal monologue, or chain of thought, is what makes up the context provided to the LLM at each step in the generation process. The example below is made up of five model generation calls and each input prompt passed to the model includes the entire context to help facilitate the reasoning and acting execution loop.

The internal monologue is prompted by prepending the model's generated text with "Thought:". This nudges the model towards sampling tokens which explain the observations need to draw the conclusion. Without prepending this tag, the model will simply continue generating additional questions. Another important aspect to the example below is the additional examples that are provided to the models. These examples help further nudge the model towards generating the type of response we desire.

8.3 Exception Handling

Similar error correction techniques can be applied to ReAct as we discuss in MRKL. In addition, during the chain of thought the LLM may also enter periodic loops where the same thought and action gets applied multiple times. We refer to these events as a momentary stint of insanity as the behavior is similar to the definition of insanity often attributed to Albert Einstein [Wilczek and Magazine 2015].

One way to handle these types of situations is to leverage meta-prompts as a form of exception handling. Meta-prompts take in as input the entirety of the conversation and ask the LLM how the agent (in this case another LLM) could have improved upon the actions performed [Goodman 2023]. This is another form of reflective prompting where pointing out the issue often leads to an improved answer the second time around.

9. SOCIAL MEDIA PLUGIN

Users may have interactions with an organization outside of the typical interactions through approaches like Native Mobile Applications or Web Applications. Social media outlets such as Mastodon, Threads, X and Facebook,

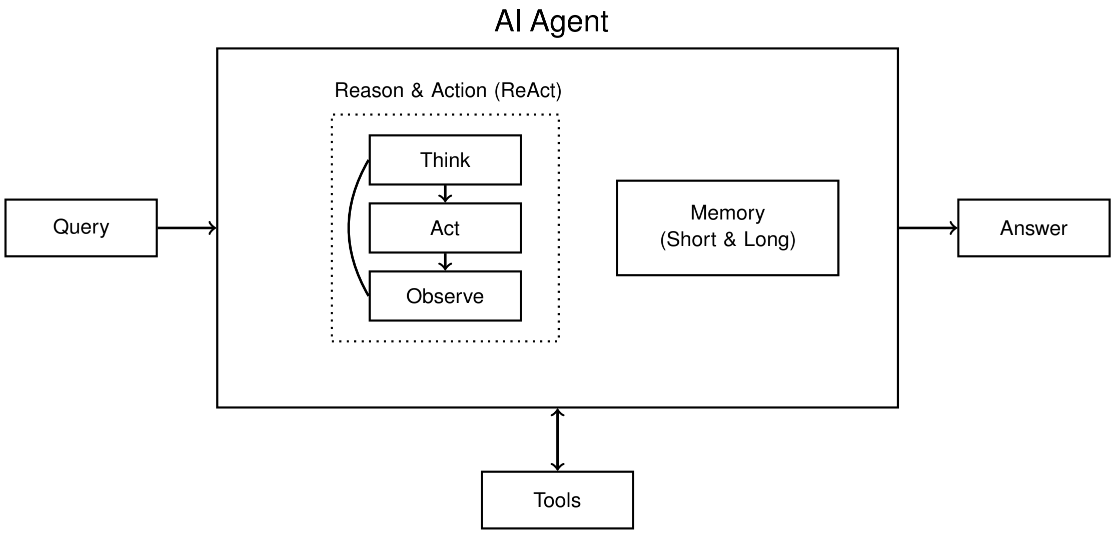

Fig. 11. ReAct Components.

```
Question: What is the elevation range for the area that the eastern sector of the Colorado orogeny
extends into?
Thought: I need to search Colorado orogeny, find the area that the eastern sector
of the Colorado orogeny extends into, then find the elevation range of the area.
Action: Search[Colorado orogeny]
Observation: The Colorado orogeny was an episode of mountain building (an orogeny) in Colorado and
surrounding areas.
Thought: It does not mention the eastern sector. So I need to look up eastern sector.
Action: Lookup[eastern sector]
Observation: (Result 1 / 1) The eastern sector extends into
the High Plains and is called the Central Plains orogeny.
Thought: The eastern sector of Colorado orogeny extends into the High
Plains. So I need to search High Plains and find its elevation range.
Action: Search[High Plains]
Observation: High Plains refers to one of two distinct land regions
Thought: I need to instead search High Plains (United States).
Action: Search[High Plains (United States)]
Observation: The High Plains are a subregion of the Great Plains. From east to west,
the High Plains rise in elevation from around 1,800 to 7,000 ft (550 to 2,130 m).
Thought: High Plains rise in elevation from around 1,800 to 7,000 ft, so the answer is 1,800 to
7,000 ft.
Action: Finish[1,800 to 7,000 ft]
```

Fig. 12. The example prompt above is from the open-source project hwchase17/langchain [Chase 2022].

or internal social platforms such as Slack are important forms of interaction that should be part of an overall communication strategy with internal and external users.

How do you interact with users of a Conversational Application outside of custom applications?

Users aren't always where you want them to be. They won't always go to your website or download your application. However, social media is pervasive. Your users are already there. What's more, social media tends to be where users ask questions that are perhaps relevant to your company.

The explosion of social media is related to the fact that most people now use a mobile device as their primary means of accessing application functionality. This encourages responding quickly to notifications, of which social media apps generate a plethora. What is more, chat platforms of various types tend to be the predominant application type most in use on these devices. Unfortunately, which platform predominates depends on location, culture, and a variety of other reasons that lead to a proliferation of platforms if you want to have reach across geographies, ages and other demographic factors. The platform that could help you reach your preferred target market could range from WeChat in the Chinese market to Facebook in the U.S. market (at least for the over-40 age range – for younger users it could be Discord). For the enterprise market, you may be most concerned with reaching your users on platforms like Microsoft Teams or Slack.

What often happens is that people look for help when they are stuck with application issues on social media sites; this could be something like Slack, Facebook or X, or perhaps a more specialized site like LinkedIn. The key here is that humans trust other humans to give them advice - often more so than they would trust the very same information if it were found on a website.

Therefore,

Build common Social Media Plugins that use the API of the social media application to parse and create posts, DM's and other artifacts of social media from within Conversational Applications.

A Social Media Plugin paired with a Conversational Application that can parse natural language text and formulate responses in natural language. This is especially true of situations where we want to automate or semi-automate the process of responding to messages (such as on Threads) raising complaints or questions about a product or service. This architecture is shown below in Figure 13: Social Media Plugin.

In Figure 13 we show how Social Media Plugin works with a Chatbot (either Traditional Chatbot, LLM Chatbot or Long Tail RAG) to parse queries and formulate responses, but it requires the use of a social media API to obtain those queries and to post the responses. Another common pattern used by a Social Media Plugin is the BFF pattern from [Newman] which act as an adapter to any additional domain services needed to obtain information or take actions based on the user's requests. For instance, in IBM we have built a system for customer support following this architecture with Slack as the social media API that can tie into our back-end Support Ticket Management service to both retrieve existing ticket details for the user and the LLM and to open new tickets when necessary. The BFF allows the Chatbot and Social Media Plugin to be isolated from the details of this back-end service.

One of the key aspects of a successful Social Media Plugin is that the representation of the User should be carried over from the Social Media Plugin into other types of user interfaces that the user interacts with. Thus, if a user asks for help on social media with a task, the results of actions taken through social media should be available on the platform on which they may have originally tried the task, be that a native mobile application, web application or other platform.

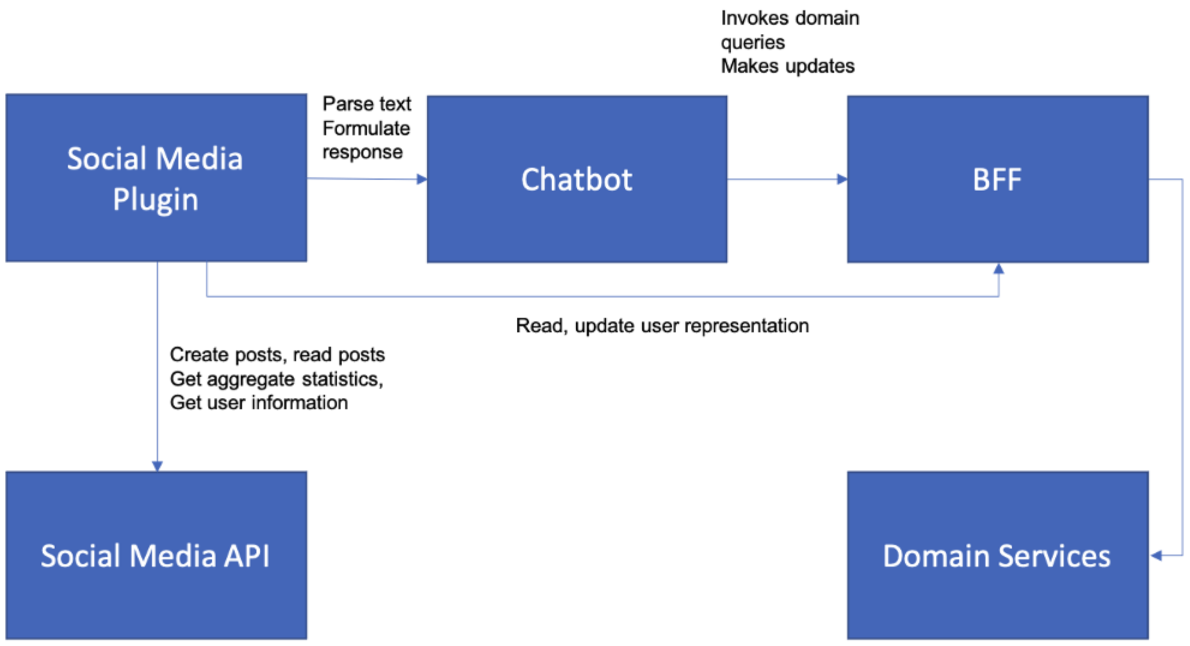

Fig. 13. Social Media Plugin.

10. CONCLUSION: PUTTING IT ALL TOGETHER

In our review of several standard patterns for Conversational Applications, we have shown the required parts that are necessary for large-scale implementation of Generative AI. When you combine the patterns together, you can end up with an architecture that resembles the following (see Figure 14: Combined Patterns).

This combined architecture begins with common UX code built upon a foundation of Social Media Plugins that allow us to centralize the interaction with the various social media and other platforms. All conversations flow to a centralized router (part of the MRKL Pattern) that then determines the direction in which to take the conversation. The conversation can be routed to one or more specialized agents that can handle unique areas of knowledge or information domains.

These agents can be implemented as Traditional Chatbots (to cover those areas where accuracy of specific responses is absolutely necessary) or as LLM Chatbots as part of a Long Tail RAG, as RAG agents (where accuracy is important, but up-to-date information is required), or as agents that take advantage of the ReACT pattern, where a chain of thought (captured as a Reasoning Trace) is needed by the LLM to determine when and how to interact with a system in the outside world.

At IBM we have employed this architecture in multiple implementations of our Conversational Applications. The AskHR architecture for managing HR issues uses an architecture that embodies almost all of these patterns, as does the Unified Chat architecture that we use for our external customer support. Likewise, the internal AskIBM architecture for internal Q&A is evolving quickly in this direction.

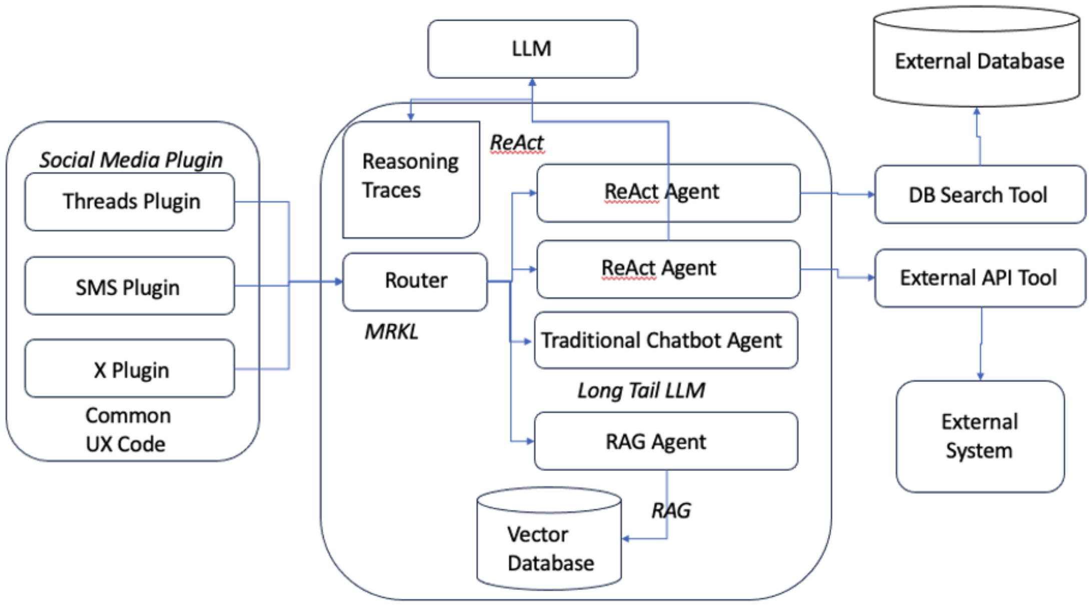

Fig. 14. Combined Patterns.

11. ACKNOWLEDGMENTS

The authors would like to thank their Shepherd, Ademar Aguilar, as well as the members of the PLoP 2024 Reading Group, including Erica Inoue, William Schrieber, Takumi Ayukawa, Max Moundas, and Hind Milhelm for their help and thoughtful comments that have helped to improve this paper.

REFERENCES

2023. Metric Types. `https://milvus.io/docs/metric.md`. (2023). Accessed: 2025-02-20.

James Briggs. 2023. Hierarchical Navigable Small Worlds (HNSW). `https://www.pinecone.io/learn/series/faiss/hnsw/`. (2023). Accessed: 2025-02-20.

Harrison Chase. 2022. LangChain. (Oct. 2022). `https://github.com/langchain-ai/langchain`

Michael Glass, Gaetano Rossiello, Md Faisal Mahbub Chowdhury, Ankita Rajaram Naik, Pengshan Cai, and Alfio Gliozzo. 2022. Re2G: Retrieve, Rerank, Generate. (2022). `https://arxiv.org/abs/2207.06300`

Noah Goodman. 2023. Meta-Prompt: A Simple Self-Improving Language Agent. `https://noahgoodman.substack.com/p/meta-prompt-a-simple-self-improving`. (April 2023). Accessed: 2025-02-20.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems* 43, 2 (Jan. 2025), 1–55. `DOI:http://dx.doi.org/10.1145/3703155`

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. 2022. MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. (2022). `https://arxiv.org/abs/2205.00445`

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. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. (2021). `https://arxiv.org/abs/2005.11401`

Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. `https://aclanthology.org/W04-1013/`

Zackary Rackauckas. 2024. Rag-Fusion: A New Take on Retrieval Augmented Generation. *International Journal on Natural Language Computing* 13, 1 (Feb. 2024), 37–47. `DOI:http://dx.doi.org/10.5121/ijnlc.2024.13103`

Gabriele Venturi. 2023. PandaAI: the conversational data analysis framework. (April 2023). `https://github.com/sinaptik-ai/pandas-ai`

Frank Wilczek and Quanta Magazine. 2015. Einstein's Parable of Quantum Insanity. `https://www.scientificamerican.com/article/einstein-s-parable-of-quantum-insanity/`. (Sept. 2015). Accessed: 2025-02-20.

Received June 2024; revised September 2024; accepted February 2025