# FINAL_REPORT

June 6, 2023

## 1 Chatbot Architecture

On the backend, my chatbot uses OpenAI `GPT-3.5-turbo` as the LLM engine for responses. However, the user interaction is not straightforward. Between the user input and the raw inference against the LLM model sits several other prompt-conditioned LLMs + some Python code. The result is that there is an `Agent` who decides how to best answer the users prompt, namely by leveraging what `Tools` it has available to it. I provide a very brief overview of this concepts within the doc, followed by an evaluation of this approach.

### 1.1 Concepts

#### 1.1.1 Chains

Chains are wrappers around a sequence of components such that they are called sequentially. We can use this to feed a LLM output back into the input context for another LLM inference.

#### 1.1.2 Agents

Agents are prompt-conditioned LLMs working in conjunction with a prompt template generator and an output parser. They run in iterative loops until they determine that the task has been executed or cannot be executed. The tight coupling between each successive iteration puts a lot of emphasis on the prompt template generator and how this inital prompt is generated. It also places emphasis on the output parser–if things fall apart it can often times happen here.

The goal of Agents is to decide when to use a Tool and which Tool to use. Their initial prompt is constructed to include a list of Tool names and natural language descriptions. Different Agent types have different expectations about the stuctured format of input/out. A very popular format is as follows:

```
Plan: the plan of API calls to execute
Thought: you should always think about what to do
Action: the action to take, should be one of the tools [{tool_names}]
Action Input: the input to the action
Observation: the output of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I am finished executing the plan (or, I cannot finish executing the plan without know
Final Answer: the final output from executing the plan or missing information I'd need to re-pl
```

This format is injected into the prompt template generator, which might look something like this:

```
API_PLANNER_PROMPT = """You are a planner that plans a sequence of API calls to assist with use
```

```
You should:
1) evaluate whether the user query can be solved by the API documentated below. If no, say why
2) if yes, generate a plan of API calls and say what they are doing step by step.
3) If the plan includes a DELETE call, you should always return an ask from the User for autho

You should only use API endpoints documented below ("Endpoints you can use:").
You can only use the DELETE tool if the User has specifically asked to delete something. Otherw
Some user queries can be resolved in a single API call, but some will require several API calls
The plan will be passed to an API controller that can format it into web requests and return th

----

Here are some examples:

Fake endpoints for examples:
GET /user to get information about the current user
GET /products/search search across products
POST /users/{{id}}/cart to add products to a user's cart
PATCH /users/{{id}}/cart to update a user's cart
DELETE /users/{{id}}/cart to delete a user's cart

User query: tell me a joke
Plan: Sorry, this API's domain is shopping, not comedy.

User query: I want to buy a couch
Plan: 1. GET /products with a query param to search for couches
2. GET /user to find the user's id
3. POST /users/{{id}}/cart to add a couch to the user's cart

User query: I want to add a lamp to my cart
Plan: 1. GET /products with a query param to search for lamps
2. GET /user to find the user's id
3. PATCH /users/{{id}}/cart to add a lamp to the user's cart

User query: I want to delete my cart
Plan: 1. GET /user to find the user's id
2. DELETE required. Did user specify DELETE or previously authorize? Yes, proceed.
3. DELETE /users/{{id}}/cart to delete the user's cart

User query: I want to start a new cart
Plan: 1. GET /user to find the user's id
2. DELETE required. Did user specify DELETE or previously authorize? No, ask for authorization
3. Are you sure you want to delete your cart?
----

Here are endpoints you can use. Do not reference any of the endpoints above.
```

```
{endpoints}

----

User query: {query}
Plan:"""
API_PLANNER_TOOL_NAME = "api_planner"
API_PLANNER_TOOL_DESCRIPTION = f"Can be used to generate the right API calls to assist with a u

# Execution.
API_CONTROLLER_PROMPT = """You are an agent that gets a sequence of API calls and given their d
If you cannot complete them and run into issues, you should explain the issue. If you're able t


Here is documentation on the API:
Base url: {api_url}
Endpoints:
{api_docs}


Here are tools to execute requests against the API: {tool_descriptions}


Starting below, you should follow this format:

Plan: the plan of API calls to execute
Thought: you should always think about what to do
Action: the action to take, should be one of the tools [{tool_names}]
Action Input: the input to the action
Observation: the output of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I am finished executing the plan (or, I cannot finish executing the plan without knowi
Final Answer: the final output from executing the plan or missing information I'd need to re-pl


Begin!

Plan: {input}
Thought:
{agent_scratchpad}
"""
```

This is just one example for a particular Agent. It can also be the case that the langchain library will show an Agent pre-baked examples that follow the recommended output stucture to aid in few-shot performance.

### 1.1.3 Tools

Tools are wrappers around APIs or other programming interfaces. The Agent selects which Tool to use based on the Tool's natural language description. Under the hood, the LLM then predicts how to actually use the tool by having its implementation passed into context. The Langchain library provides the logic to orchestrate this generation and passing of appropriate context.

### 1.1.4 Other Concepts

I found all of these concepts to be highly relevant. This project is also built around the idea of retrieval augmentation. The ReAct and Self-ask papers are also very pertinent.

## 1.2 Pros of this approach

- Everything is in natural language! It makes it very easy to understand what a given tool does, what the Agent output is, etc. This is one of the obvious perks of LLMs.
- Retrieval augmentation can generalize and makes LLMs work for data outside of the training set. Being able to build vector databases of recent information that a LLM has never seen before is fantastic.
- Agents decide which tools to use and when. This makes extending the framework and adding additional Tools a breeze.
- Tools can help to avoid hallucination. Rather than asking a LLM to do math, we can ask it to pass the right input to an actual calculator. This can increase the quality of results and the capacity for more complex tasking.
- Chat history or rolling context provides a very natural way to interact with a chatbot, where you can ask follow-up questions and the chatbot does not respond from a blank slate.
- LLMs give us the ability to pass (relatively) huge amounts of tokens into them! This is great for summarization or extraction over a cohesive corpus.

## 1.3 Cons of this approach

- LLMS and especially Agents can be extremely inconsistent. Sometimes it's from hallucinations, sometimes from outputparsing failure; this means the Agent either hallucinated extra or incorrect Tool parameters / API paremters, or failed to structure its output in a structured form that the next iteration of the Chain could consume.
- Not only are Agents inconsistent, but they're very difficult to trace and debug–there are so many layers and abstractions. Things look very simple and natural to the end-user, but if you have to dig into the Langchain library itself it becomes opaque very quickly.
- There is a certain finesse to everything using natural language descriptions. Sometimes the addition, removal, or location of a single word can yield completely different results.
- LLMs are slow in general, but retireval augmentation from a vector database can be excruciatingly slow depending on how that database was built, which LLM you're working with, and how the database retriever is passing chunks to the Agent.
- How your data is parsed, ordered, embedded, and chunked all matters. For smaller context LLMs, they may not receive all the chunks necessary to correlate data at the beginning of a document with data at the end of a document.
- There is not always enough context for things to remain as simple as we hope. There are tactics for dealing with large context but all of these have tradeoffs.

## 2 Next Steps

The following are all ideas for potential add-ons to this project and things I'd like to explore or have already built and need to integrate.

- Add large context models MPT-7B and run them locally. This requires a wild amount of RAM and GPU RAM, but could dramatically simplify the data flow.
- Create a CustomTool for YouTube search. This would find an appropriate video(s) and return the URL.
- Create a CustomTool for YouTube transcription. This tool would be used to run `youtube-dl -f 'bestaudio[ext=m4a]'` on a given YouTube URL and pass the resulting audio to a model like OpenAI Whisper to produce a text transcript.
- Create a CustomTool for transcript summarization. This would take a text transcript and pass it to a large context summarization model, like BERT-led-large-book-summary, responding with a concise summary of the transcript.
- Create a CustomTool for writing UW colloquia reviews based on transcript summaries. I may or may not have already built all the above.
- Create a CustomTool for YouTube video recommendations. In particular, it could use context about which UW CSE courses you are interested in a recommend a relevant colloquia archive. This would require storing the transcripts and/or summaries in a vector database.
- Create a CustomTool for vectorizing an aribitrary URL. This would give the chatbot the ability to learn from any arbitrary document and then use the rest of its toolkit to complete the task.
- Create a CustomTool for building VectorQuery{X} Tools. This tool would contain logic to provide some best practices when creating a vector embedding (i.e. which search algo, k number of docs, which chain type to use, etc).
- Create a CustomTool for crawling a model database (e.g, Huggingface and building Tools that use their Model Card for the natural language description to decide when and how to callout to specialized models. It could either use the HuggingFace inference API or download the models directly. – This could actually build (execution) Agents by extracting what structure the input/output for a given model is. The execution Agent could be added to the overall task workflow by running under a planning Agent.
- Turn the agent on it's own code base and ask it to write a summary report. This is essentially retrieval augmentation again, but perhaps directly without a vector database involved.