Leveraging JSON Data for a Network Data Chatbot

A technical paper prepared for presentation at SCTE TechExpo24

David Suh
Lead Software Engineer
Cox Communications
david.suh@cox.com

Table of Contents

List of Figures

1. Introduction

Large Language Models (LLM) have become the state–of–the-art for chatbot development with unprecedented performance. With the rise of LLMs, there has been a need to develop this technology within the constraints of both practical and corporate requirements and needs. For our network data chatbot, we have the practical need to chat with our network data being collected by our backend services and accessed via REST APIs as JSON data. Other practical needs include minimizing hallucinations which are false responses, maximizing deterministic responses, and fitting prompts within the LLM's maximum context window length. In addition, corporate requirements dictate that we have sandboxed LLMs to isolate this internal data from external access. To meet these requirements, we have implemented a Retrieval Augmented Generation (RAG) framework based on the LangChain orchestrator that runs an LLM agent. The agent asks the LLM to generate formatted JSON to invoke tool functions that make up the semantic layer that connects meaning to action. It has been implemented to retrieve relevant network data in JSON snippets stored in a knowledge graph database that also holds the network relationships. Ultimately, the LLM agent will include the JSON snippets as context in further LLM prompts to get the answer that the user is asking for. This paper will discuss how we met the needs and requirements in our network data chatbot.

2. Background

2.1. Generative AI

Since the seminal 2017 Google paper "Attention Is All You Need" [1] introduced the Transformer architecture, that is the heart of the LLM, an Artificial Intelligence (AI) shift has been under way from smaller specialized neural network models to massive, generalized foundation models for generative AI [2,3].

Traditional deep learning models are designed by hand with many different choices in input, hidden, and output layers of neural networks to choose from as well as the feed forward and feed backward connections used. In addition, you usually train your whole model with your curated supervised data specific to the domain that you are interested in. With many variables, it requires specialized deep learning skills and experience to get the right model and data for your application.

In contrast, foundation models are models that have been trained on massive amounts of broad data that can be applied across a wide range of use cases and are not necessarily domain specific [4]. LLMs are one type of foundation model for text generation. You can either use the model directly or formally fine tune it by training it further with your domain specific data. It is also well known that a side effect of fine tuning is that an LLM can experience catastrophic forgetting of earlier knowledge so care must be taken [5]. Regardless of which way you go, since the model has already been established, you just need to use it as a black box for your application. This makes building LLM applications a natural fit for software developers.

Transformer-based generative AI models encode the patterns and structures of their training data as context to be able to generate new data that has similar characteristics [6]. In essence, this architecture predicts the next token based on the model, the input, and what has been generated so far by ranking the next one by probability match. This generation is called inference, and the input is called the prompt. Because the architecture is inherently continuous completion that produces the next generation from the last iteration, an output stop condition must be used to stop generation. Generation is not deterministic as different inference runs can generate different variations in the output. Because generation is also piece-by-piece, some variations can lead the generation in directions that will produce incorrect or incomplete

output. These types of generations are called hallucinations and for LLMs, it can happen convincingly in the output.

2.2. Prompt Engineering

Prompt engineering consists of techniques that optimize the text we use when interacting with an LLM to get the answer that we want for questions or any other kind of text completion. The critical factor in prompt engineering is the context window length of the LLM being used. This matters because all the techniques described below add to the input prompt that is sent to the LLM. Larger context length allows more relevant documents and data to be included so that the LLM can answer the question.

2.2.1. RAG

In 2020, Meta introduced the RAG approach to prompting LLMs in the paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" [7] that supplements the knowledge that a LLM has with new knowledge that it was not trained for. Generally, the input is used to retrieve supplemental knowledge from any number of sources like databases (DB), APIs, services, and environment. Some examples are vector DBs, graph DBs, SQL DBs, REST APIs, calculator, and the internet. The original paper used text embeddings which are vectors of numbers that represent chunks of text put into a vector DB. Relevant chunks are retrieved by similarity ranking from the vector DB like the Transformer does. However, you can use any method to isolate the key words needed to retrieve the supplemental knowledge. In essence, when using RAG, you provide the supplemental relevant documents by including them in the input prompt along with the question asked so that the LLM can generate a response with the data provided in the context [8]. It has also been shown that general foundation model LLMs with prompt techniques that use RAG can outperform domain specific, fine-tuned LLMs [9].

2.2.2. ReAct

ReAct is a prompting technique that uses both reasoning and action as steps to work out the answer to a question [10,11]. This represents the semantic layer that connects meaning to action. The action is used to allow the LLM to call out tools to use in the form of software function calls when needed. The LLM uses sequential reasoning and decision making in a thinking-out-loud manner to decide if it has enough data to answer the question or whether it needs any number of presented tools that it can call out to get more data. This encourages the LLM to follow a logical reasoning format by spelling out the ground rules about formatted reasoning and action responses that are included in the input prompt so that your application can parse them and act on them appropriately. The LLM generates task solving trajectories that direct the outputs to the next step at any one time. Consequently, this technique may make multiple sequential LLM inferences to get to the next step.

The LLM is asked to include the following responses in its outputs to drive the progression to an answer.

2.2.2.1. Thought

Reasoning thought at the current step that will lead to the next action. This includes deciding that it has the final answer.

2.2.2.2. Action

Action is an appropriate tool call that can be formatted in JSON for concise output parsing. Action can also be used to call out the final answer.

2.2.2.3. Observation

Response from the Action that will be used to drive the next Thought and Action.

2.2.2.4. Final Answer

Normal exit condition to return an answer to the user.

3. Needs

To supplement our family of internal web applications and backend services that provide data collection, inventory, visibility, statistics, forecast, and capacity planning, we had a need to make it easier for users to find information and data through a natural language interface.

Knowing that all LLMs hallucinate and are not deterministic, a goal was to minimize this.

There were assumptions that the response would be returned in a reasonable amount of time but that would need to be balanced by the trade-offs of speed, accuracy, and cost.

There was no decision made on how production would be deployed to our internal users so we had to be flexible with how it would be deployed. It could be deployed to internal servers or through a cloud service provider.

The following are the use cases we considered.

3.1. Use Cases

3.1.1. Questions about network devices and their connections

This paper covers this use case because it is a natural fit to what we already have. Because we already collect both router and transport device data and serve them through REST APIs as JSON data, it was feasible for a user to chat with the static and dynamic network data that we have. In addition, since we normalize device data, we have data that is consistent across different vendors for each type of hardware. Of course, if you use a single vendor, you could possibly use vendor-specific JSON. We collect summary data for every device we support so that is the only type of JSON that was considered for this use case since it covers device information, device connections, slots and ports, bundles, trails, etc.

3.1.2. Questions about general production network knowledge

This paper does NOT cover this use case because this is more of a fit for unstructured knowledge of our network that is captured in internal training materials. This use case could be covered by the original RAG framework that chunks knowledge sentences in unstructured text, creates text embeddings for each chunk which is a vector of numbers that represents the relationships and context of the chunk, and puts the vector and sentence chunks in a vector DB. On inference, the question is chunked, the text embedding vectors are calculated, a series of chunks are retrieved from the vector DB with the highest vector similarity matches. Those retrieved chunks would be put in the input prompt with the question as context for the LLM to answer from. This separate effort can be tied into the solution described later.

4. Solution Development

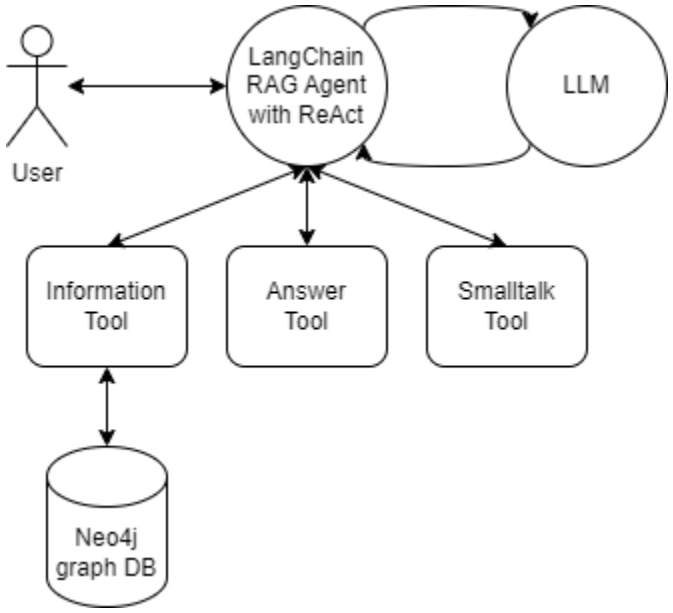

Figure 1 – RAG Agent with ReAct and Tools

4.1. Approach

Looking at our needs for the use case we planned to implement and the corporate requirements to get there, we decided on a course of action for this Proof-Of-Concept (POC) to run all services and software locally within the corporate network.

4.1.1. Local private sandboxed LLM server with open-source LLM model

Sandboxed LLMs provide privacy and security to alleviate corporate concerns about confidential and proprietary data and intellectual property. Having a local server with GPUs for the development environment allows more control of the LLM application work and allows us to keep track of the tools used to generate outputs. Production deployment can be deployed either locally or to a cloud provider.

4.1.2. RAG Agent with ReAct

An AI agent is software that executes tasks on behalf of the user asking a question [12]. Rather than the user interacting directly with the LLM, the agent acts as a liaison that gets the input question from the user and takes it from there. By removing the user from direct LLM interaction and using RAG to inject relevant data into the prompt, the agent can vet the LLM responses and minimize the hallucinations of false responses. Also, by using an agent in a loop with the LLM, we maximize deterministic behavior because an LLM can be made to stay in the loop until it responds in a correctly formatted manner through ReAct and tool specifications. Finally, by using RAG instead of fine-tuning an existing LLM, we take advantage of dynamically updated data.

4.1.3. Graph DB

A graph DB can hold network devices as nodes and JSON device summaries as node properties. You can also create relationships between device nodes. Some graph DBs can store text embedding vectors like vector DBs if needed.

4.2. Framework

4.2.1. JSON-based Agents with Ollama & LangChain

We selected the LangChain orchestrator with a ReAct agent that uses a local Ollama LLM inference server and a Neo4j graph DB to implement our RAG [13]. The agent defines the ReAct key words and the different JSON tools for the LLM to pick depending on the current state of thought.

4.2.1.1. LangChain

Open-source LangChain is a rich framework to build applications that interact with LLMs by chaining interoperable components. It supports many LLM inference servers (e.g. Ollama), graph DBs (e.g. Neo4j), and other third-party components.

4.2.1.1.1. AgentExecutor

This is the LangChain ReAct agent that uses tools.

This agent implements a state machine that basically continuously iterates on steps that include LLM query and response pairs based on the prompt template, input, and intermediate responses. It is looking for key ReAct words and JSON tool calling in the LLM response specified in the prompt template. The "Thought:" key word is to help the LLM work through to "Action:", which is to call a JSON tool. JSON tool calling will execute the corresponding Python tool function and the return text gets put into an "Observation:" for the next step iteration. The agent needs to converge to "Final Answer:" as a normal exit condition, as detailed in the next section. Otherwise, exceeding the maximum number of steps, which we set at 10, will trigger the forced exit condition. At exit, it returns an answer from the "Final Answer:" or a statement that the "Agent stopped due to max iterations."

4.2.1.2. Tools

4.2.1.2.1. Information

Useful for when you need more information to answer questions about various device names or node names. This tool first does a Lucene search in the Neo4j DB index for the entity at 80% match to account for misspellings. If there is a device name match, the corresponding summary JSON property is retrieved and returned to the agent with text instructions to use the JSON data to answer the question.

4.2.1.2.1.1. Arguments

entity: Union[str, list[str]] = Field(description="ip device or transport node mentioned in the question")

query: Optional[str] = Field(description="user query")

4.2.1.2.2. Answer

Useful for when you have the answer. This tool assumes that because of hallucinations, the LLM has not yet decided to output "Final Answer:" for ReAct even though it does have the answer. Instead, the LLM has decided to call this tool to send the answer. This tool responds to the agent with text instructions to create a final answer with the answer sent.

It is no mistake that the argument for the answer is called query because the LLM was better at matching 2 argument names when they were common among all the tools.

4.2.1.2.2.1. Arguments

query: Optional[Any] = Field(description="answer to the query")

4.2.1.2.3. Smalltalk

Useful for when user greets you or wants to small talk. This tool passes responses from the LLM for user input that are not questions, like “Hi”.

4.2.1.2.3.1. Arguments

query: Optional[str] = Field(description="user query")

4.2.1.3. Prompt Template

For our prompt template, we modified an example ReAct agent prompt template [14] to specify a ReAct format that we are asking the LLM to adhere to. In addition, we specify our own JSON tools that would be called out by the LLM as a ReAct Action with the JSON “action” and “action_input” populated with the tool to use and the argument(s) to call it with. LLMs do not always follow instructions completely due to the non-deterministic variances in the LLM output. Therefore, it is recommended to limit the number of arguments your tool accepts to at most 3 so that you can limit the number of variations in the arguments. Our project whittled it down to at most 2, although it was still functional when we had 3, there appeared to be more instances of missing and malformed arguments. The quality of the adherence by the LLM to the JSON tool interface definition as well as to the ReAct key words is directly proportional to the quality of the LLM.

4.2.1.3.1. Complete System Message

Answer the following questions as best you can.

You can answer directly if the user is greeting you or similar.

Otherwise, you have access to the following tools:

Information - useful for when you need more information to answer questions about various device names or node names, args: {'entity': {'title': 'Entity', 'description': 'ip device or transport node mentioned in the question', 'anyOf': [{'type': 'string'}, {'type': 'array', 'items': {'type': 'string'}}]}, 'query': {'title': 'Query', 'description': 'user query', 'type': 'string'}}

Answer - useful for when you have the answer, args: {'query': {'title': 'Query', 'description': 'answer to the query'}}

Smalltalk - useful for when user greets you or wants to smalltalk, args: {'query': {'title': 'Query', 'description': 'user query', 'type': 'string'}}

The way you use the tools is by specifying a json blob.

Specifically, this json must only have a `action` key (with the name of the tool to use)

and a `action_input` key (with the input to the tool going here).

The only values that are allowed in the "action" field are: ['Information', 'Answer', 'Smalltalk']

The $JSON_BLOB must only contain a SINGLE action and action_input,

do NOT return a list of multiple actions.

There must be a valid $JSON_BLOB with all string args.

Here is an example of a valid $JSON_BLOB.

```
{{{{
   "action": $TOOL_NAME,
   "action_input": $INPUT
}}}}
```

The $JSON_BLOB must always be enclosed with triple backticks!

ALWAYS use the following format.

Question: the input question you must answer

Thought: you should always think about what to do

Action:```

$JSON_BLOB

```

*Observation: the result of the action...*

*(this Thought/Action/Observation can repeat N times)*

*Thought: I now know the final answer*

*Final Answer: the final answer to the original input question*

*Reminder to always use the exact characters `Action` when responding.*
```

Reminder to always use the exact characters `Thought` when responding.

Reminder to always use the exact characters `Final Answer` when responding.

Begin!

4.2.1.4. Ollama

Open-source Ollama is a popular LLM inference server that uses the fast open-source llama.cpp LLM inference engine. It is also supported by a wide variety of third-party software.

4.2.1.4.1. Open-source LLM

Originally, the medium mixtral:8x7b-instruct-v0.1-q8_0 model from Mistral was used at 8-bit quantization (to fit better in the GPUs we were using) because other people noted that it performed well as an agent [15]. However, during testing, it was inconsistent and slow.

After much testing of different LLMs, we chose the small mistral:7b-instruct-v0.3-fp16 unquantized model from Mistral. This newer, smaller LLM proved to be more consistent, faster, and more accurate than the others we tested that were of reasonable size. This was important because agents are in an iterative loop with the LLM, so the LLM needs to be fast to be used repeatedly in one run. On the other hand, accuracy in retrieving data from JSON was very good but not perfect.

Quantization is the process of creating smaller models from original models by replacing the model weights with quantized versions of the weights which lowers the amount of storage needed to store the weights and also accelerates the evaluation at the cost of reduced response quality. The difference is like having a lossy model compared to the original lossless model. This impacts the output quality as it depends on how closely the models track each other.

We decided to stay with the unquantized model as the chosen LLM was still small enough to fit in our GPUs. Otherwise, we periodically had some incorrect or irrelevant text included in the output when we tested with even an 8-bit quantized version of the model.

Ultimately, the best LLMs are ones that can follow directions well and output correctly formatted responses.

4.2.1.5. Neo4j

Open-source Neo4j is one of the leading graph databases in use today. It is also supported by a wide variety of third-party software.

4.2.1.5.1. Network Data Ingest

Network data ingest is done by scripts that get JSON device summaries for every device in the network from our backend REST APIs. This data is parsed, and device nodes are added to the graph DB for each device. The JSON device summary is also added to a device node as a property. Furthermore, connection relationships are added for device nodes that are connected to each other through interfaces. To search for a device node by its name, indexes are created. For advanced queries in the future, the LLM could be prompted to generate query expressions in the Neo4J query language which would allow for an extension of this framework beyond searches by name.

It is intended that the scripts be run daily to update the graph DB data so that it is in sync with the current backend REST API data.

4.3. Integration

4.3.1. LangServe

The open-source LangServe framework helps developers deploy LangChain runnables and chains as a REST API. It offers a built-in playground user interface to send questions to the LangChain application through a browser. The same LangServe application can be deployed to production through its REST APIs.

5. Testing

Testing was done through the LangServe built-in playground user interface.

5.1. Sample Question Steps

Where is <x> located?

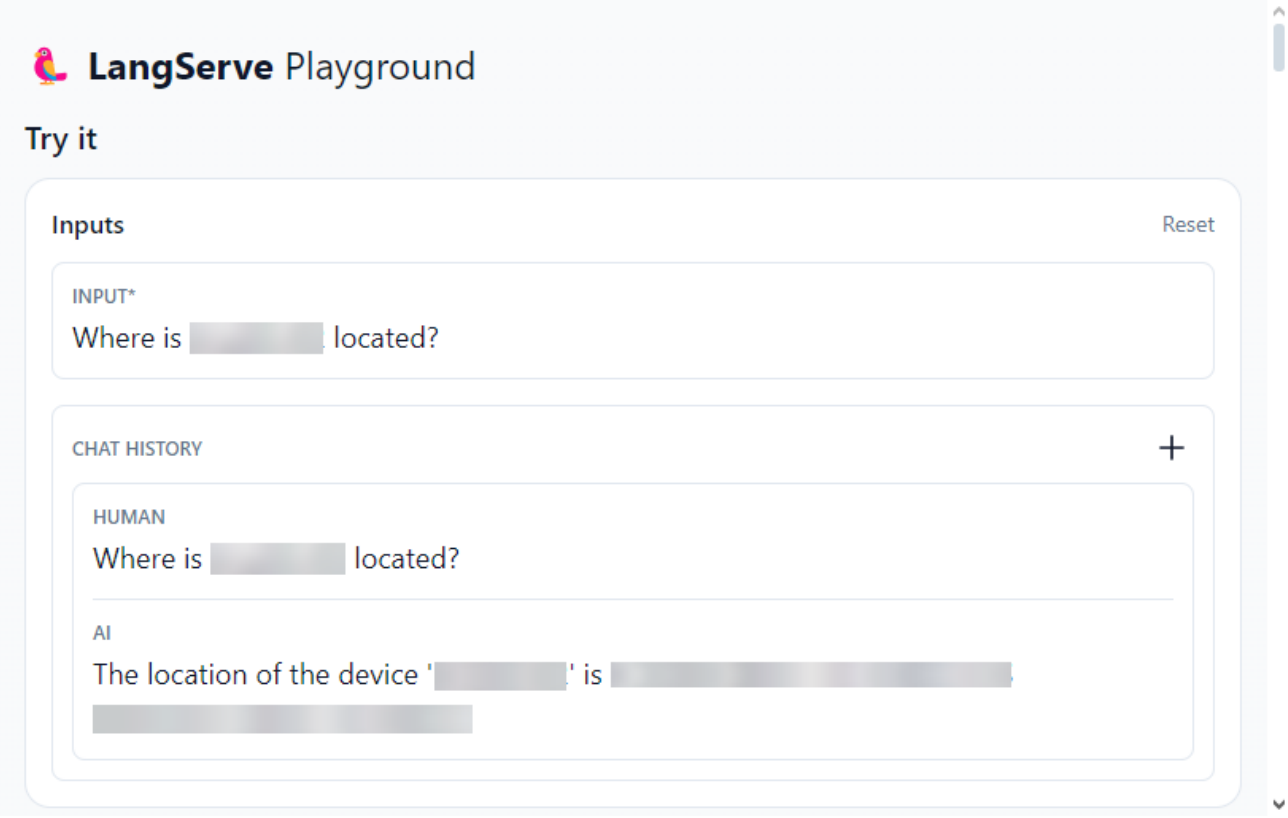

Figure 2 – User Question and Answer

ChatPromptTemplate 3 hours ago


```
{
  "messages": [
    {
      "content": "Answer the following questions as best you can.\nYou can answer directly
if the user is greeting you or similar.\nOtherwise, you have access to the following
tools:\n\nInformation - useful for when you need more information to answer questions about
various device names or node names, args: {'entity': {'title': 'Entity', 'description': 'ip
device or transport node mentioned in the question', 'anyOf': [{'type': 'string'}, {'type':
'array', 'items': {'type': 'string'}}]}, 'query': {'title': 'Query', 'description': 'user
query', 'type': 'string'}}\nAnswer - useful for when you have the answer, args: {'query':
{'title': 'Query', 'description': 'answer to the query'}}\nSmalltalk - useful for when user
greets you or wants to smalltalk, args: {'query': {'title': 'Query', 'description': 'user
query', 'type': 'string'}}\n\nThe way you use the tools is by specifying a json
blob.\nSpecifically, this json must only have a `action` key (with the name of the tool to
use)\nand a `action_input` key (with the input to the tool going here).\nThe only values
that are allowed in the \"action\" field are: ['Information', 'Answer', 'Smalltalk']\nThe
$JSON_BLOB must only contain a SINGLE action and action_input,\ndo NOT return a list of
multiple actions.\nThere must be a valid $JSON_BLOB with all string args.\nHere is an
example of a valid $JSON_BLOB.\n```\n{\n    \"action\": $TOOL_NAME,\n    \"action_input\":
$INPUT\n}\n```\nThe $JSON_BLOB must always be enclosed with triple backticks!\n\nALWAYS use
the following format.\n\nQuestion: the input question you must answer\nThought: you should
always think about what to do\nAction:```\n$JSON_BLOB\n```\nObservation: the result of the
action... \n(this Thought/Action/Observation can repeat N times)\nThought: I now know the
final answer\nFinal Answer: the final answer to the original input question\n\nReminder to
always use the exact characters `Action` when responding.\nReminder to always use the exact
characters `Thought` when responding.\nReminder to always use the exact characters `Final
Answer` when responding.\nBegin!\n",
      "additional_kwargs": {},
      "response_metadata": {},
      "type": "human",
      "name": null,
      "id": null,
      "example": false
    },
    {
      "content": "Where is [redacted] located?",
```


Figure 3 – Agent Telling LLM the System Message and Asking the Question

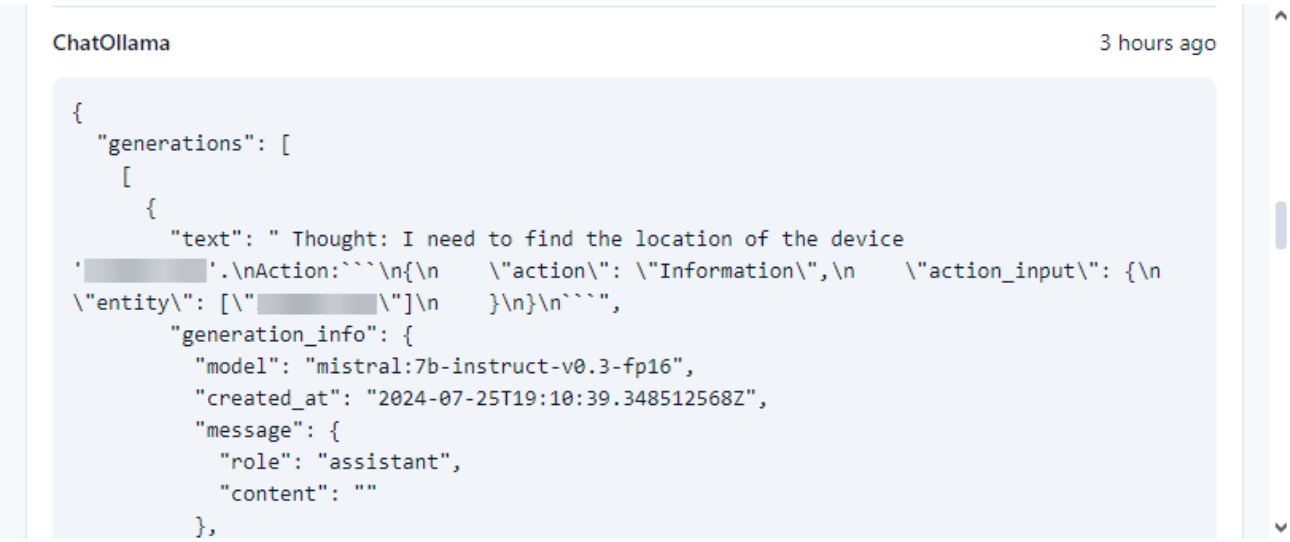

Figure 4 – LLM Telling Agent To Use Information Tool

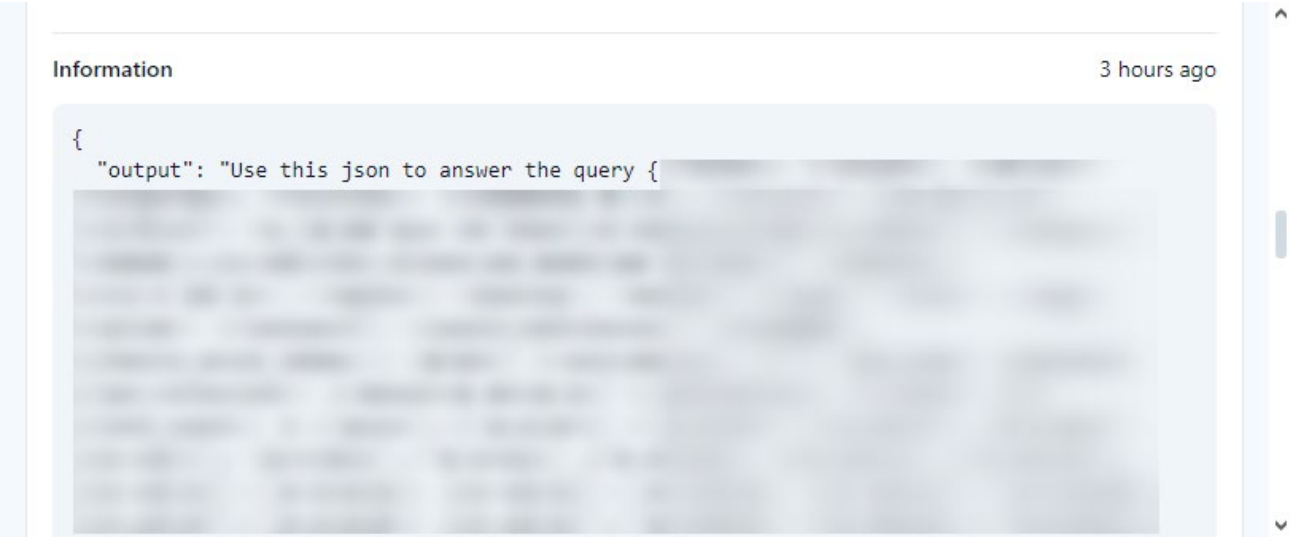

Figure 5 – Agent Telling LLM To Answer Question Using JSON Returned From Tool

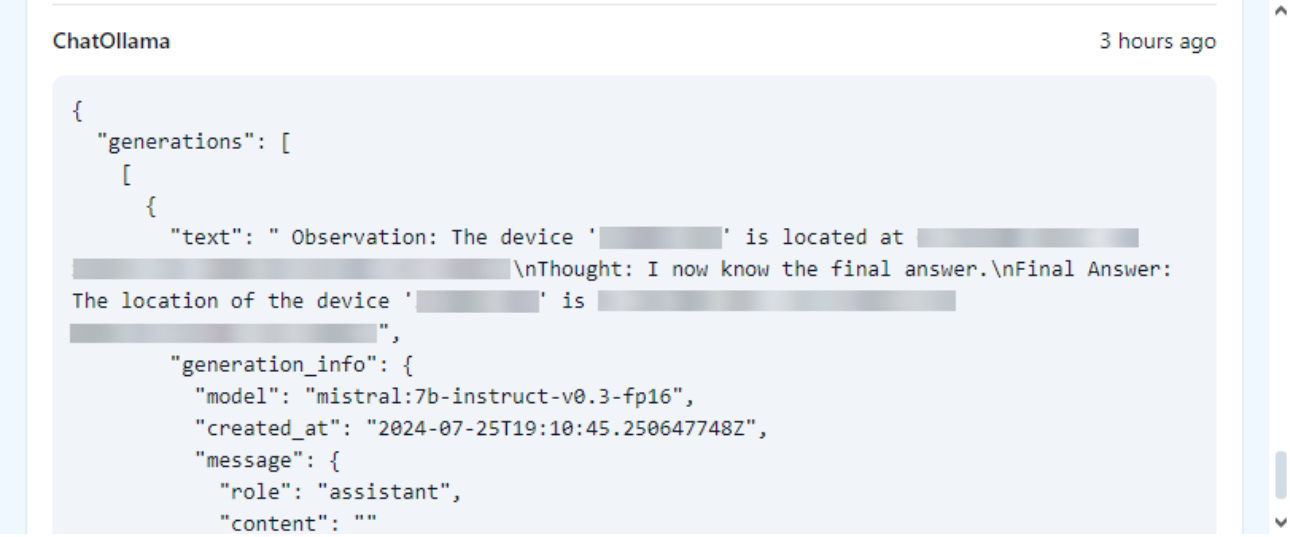

Figure 6 – LLM Telling Agent To Output The Final Answer

5.2. Other Sample Questions

What are the <y> for <x>?

What are the members of <z> in <x>?

What is the <y> of <x>?

What is the <y> for <x>?

Get the neighbors of <x>?

Get <y> for <x>?

Get <z> state for <x>?

How many <y> are there for <x>?

How many <y> are vacant for <x>?

What <y> are vacant for <x>?

6. Results

This POC worked well as a QA chatbot for JSON network data. Preliminary results show that for direct explicit questions about key names in the JSON summary data, it consistently answered correctly roughly 80% percent of the time. However, it was dependent on the questions being clear on what was asked and whether the JSON key names were related to that. For indirect or vague questions, answer consistency and correctness dropped. Overly verbose questions could confuse the LLM in pinpointing the device name in question.

Examples of where it did well are questions about location and other directly named JSON keys.

It gave incomplete answers when there were too many JSON elements that related to the question and only a subset was retrieved. This may have been due to those elements being a couple of JSON levels down or not being in a JSON array, but we have not investigated further yet. An example is when there were many neighbors connected to a network device.

One common incorrect case was when the JSON key name being asked for was close to the meaning of other names. The chatbot would get confused and sometimes return answers related to the other meaning. One example is asking for the site of a network device, which is an explicit key name. This was too close to location in terms of meaning and sometimes the chatbot would return the location. Another example is when it was asked about slots, but it returned information about ports.

There was a case where it would mistake retrieved names that included three periods as an IP address.

There was also a case where it incorrectly answered a question with unrelated JSON data it ended up focusing on. With hallucinations, LLMs can follow a narrow myopic reasoning path or blow up the scope to all the data in the JSON.

7. Lessons Learned

The lesson learned from this POC is that you need a multitude of mitigation strategies to balance the shortcomings of the LLM.

7.1. Mitigation Strategies

7.1.1. Trust but Verify

We look at the possible state of answers that the user is trying to find to be on a ranked truth scale going from lie, incomplete truth, ignorance, truth.

The user starts at ignorance because they have a question they need an answer to.

If the response from the LLM is incorrect, that is a lie which is worse than ignorance because we are being misled by the LLM.

If the response from the LLM is incomplete, that is incomplete truth which is still worse than ignorance because we are still being misled by the LLM.

Truth in the answer is the goal, but it is better to be ignorant than to be misled. Ideally, the chatbot should say that it does not know instead of incorrect or incomplete responses. That is why our strategy going forward should center around first trusting the LLM response if it is not from known problem questions but also give a supplemental link to the specific network device page in our existing network data web application to verify.

7.1.2. Blacklist Known Problem Questions

We encourage testing as much as possible to find questions that do not return full truth from the JSON network data. Testing can work to find problematic questions because JSON data is structured data that LLMs recognize and process so there is more consistency in handling than unstructured data. For example, if a certain type of question regularly causes the LLM to respond incorrectly or incompletely, the chatbot should respond that it does not know so that the user is not misled. The user can follow up with the supplemental link provided to find the truth. For example, when asking for the neighbors of a network device, the LLM returns a subset of the neighbors.

We are currently looking for the best way to do this.

7.1.3. Loose Tool Arguments

There were many times during the chatbot development that tool calls failed because the LLM did not format the tool arguments correctly as specified. Because of this, we made our tool arguments more accepting of variations, such as a string argument that was given to us as a list or an argument that was not even a string. Tool arguments are made loose with the liberal use of *Optional*, *Union*, and *Any* Python type hints to account for LLM hallucinations that do not quite get the arguments right. This worked out well as we could convert any JSON element into a string and strip unnecessary characters.

Also, limiting the tool arguments to at most two and using the same set of argument names across all tools limited the number of bad variations that the LLM could make.

7.1.4. Prompt Template Optimization

Throughout the development of the chatbot application, it was clear that determining what we say in the prompt template is more of an art than science. Even when we explicitly state the format of acceptable ReAct statements and JSON tool calling, the LLM does not always follow instructions. We resorted to subtle nudges to push it in that direction.

More occurrences of words like “must” and “always” seemed to emphasize what we wanted. Also, reminder statements to use ReAct key words like “Action”, “Thought”, and “Final Answer” were added to try to keep these critical key words front and center.

In addition, the Answer tool was added to give the LLM another way to get to “Final Answer” if it did not explicitly say it that way. Convergence to “Final Answer” is a normal exit condition to respond to the user.

7.1.5. Agent Loop Hooks

Since the ReAct agent loop tries to parse either an “Action” or “Final Answer” at each loop step, if they were either missing from the LLM output or were malformed, the default error response to the LLM was “Invalid or incomplete response”. However, this error response was too vague for the LLM to correct itself on the next LLM output.

We used a LangChain AgentExecutor hook to respond with explicit instructions to correct itself. We responded with “Invalid or incomplete response. Please provide either a valid Action with all string args or a Final Answer.” After this, the LLM consistently corrected itself and converged on the final answer. We found that always giving explicit instructions matters to LLMs.

8. Next Steps

We are on the road to production. Because our chatbot is an internal application, we are better able to monitor and work to adjust the performance. Also, since we are close to the users, we can get valuable feedback.

Before releasing the tool more widely, we will need to get a better rate of good responses, which will be challenging but not impossible. We plan to test more to find parts of the summary JSON data that are problematic. We will see if we have better results when those JSON elements are refactored to be lists rather than a sequence of object elements. In addition, we plan to add the corresponding JSON schema

with the JSON data. The JSON schema describes what all the JSON key names mean, and this may help the LLM find the JSON data that is relevant to the question asked.

As a fallback, we are looking into the ability to blacklist problem questions that are related to problematic parts of the JSON. By responding that the chatbot cannot answer those questions, we direct them to the supplemental link in the network data web application that will have the answer.

Other improvements may be obtained by improving the exact step-by-step dialog that the ReAct agent goes through with the LLM. This requires access to the LangChain component chain with our own callback functions for better control.

We will also investigate using multiple agents which would be a way to detect incorrect or incomplete answers. A simple setup could have 2-3 agents answer the same question at the same time and a similarity function could be run on the 2-3 answers to see if they generally match. That would be a basic validation of the answers. With 3 agents you have either a 2 out of 3 consensus or a deadlock of 3 different answers. The answer would be handled appropriately with the consensus answer being the answer returned to the user or the deadlock having the user told that the chatbot cannot answer that question. Speed of response and cost would be factors as we would need to run multiple agents in a timely manner with large enough GPU resources.

Finally, we need to make our information tool tie into the other use case of retrieving general production network knowledge. This could be done by expanding the information tool to add retrieval of relevant unstructured information from a vector DB using a vector similarity function.

8.1. LangGraph

The biggest shortcoming of LangChain is that you have less control of the agent processing than you want because it is built on abstractions upon abstractions [16]. The chain of coupled components you set up to get executed are like plugin configurations that are designed to work together. Having the agent automatically run like a black box is great when all you need is the default operation, but it gets in the way when you want to do more with it. They provide limited control through specific response overrides and callback functions, but you need to understand the internals to use those callback functions.

LangGraph is an extension of LangChain for building agent and multi-agent systems that is made to be controllable [17]. You can use it to create custom agents with custom flow control. It also has a more accessible persistence layer that lets you inspect and edit the agent thought process. These are the reasons why we are looking into migrating our chatbot to LangGraph.

9. Conclusion

This POC was successful in meeting both the need for a network data chatbot and the corporate requirements we faced. At its best, for clear questions where there were corresponding key names in the JSON summary, preliminary results were roughly 80% consistently correct. However, more LLM and agent mitigating strategies need to be put into place for the chatbot to handle more types of questions for production. As a backstop, we should add supplemental links in the answers for the user to be able to verify answers. User verification and being able to see when the AI system does not know is the best way for the user to build trust. Of course, all these results are dependent on the quality of the LLM, and future results are expected to be better as LLMs get better. We will continue to test with new open-source LLMs as they come out.

Abbreviations

LLM	large language model
AI	artificial intelligence
RAG	retrieval augmented generation
JSON	JavaScript object notation
REST	representational state transfer
API	application programming interface
DB	database
POC	proof-of-concept
QA	question-answer
GPU	graphics processing unit
IP	internet protocol

Bibliography & References

[1] “Attention Is All You Need”, Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, https://arxiv.org/abs/1706.03762

[2] The History of AI, https://www.nocode.ai/the-history-of-ai/

[3] Traditional AI vs Foundation Models, https://www.nocode.ai/traditional-ai-vs/

[4] Foundation model, https://en.wikipedia.org/wiki/Foundation_model

[5] “An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning”, Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, Yue Zhang, https://arxiv.org/abs/2308.08747

[6] Generative artificial intelligence, https://en.wikipedia.org/wiki/Generative_artificial_intelligence

[7] Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, https://research.facebook.com/publications/retrieval-augmented-generation-for-knowledge-intensive-nlp-tasks/

[8] RAG is Just Fancier Prompt Engineering, https://analyticsindiamag.com/rag-is-just-fancier-prompt-engineering/

[9] Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine, Can Generalist Foundation Models Outcompete Special-Purpose Tuning? Case Study in Medicine - Microsoft Research

[10] ReAct Prompting, https://www.promptingguide.ai/techniques/react

[11] “ReAct: Synergizing Reasoning and Acting in Language Models”, Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, Yuan Cao, https://arxiv.org/abs/2210.03629

[12] What is an AI Agent?, https://botpress.com/blog/what-is-an-ai-agent

[13] JSON-based Agents With Ollama & LangChain, https://medium.com/neo4j/json-based-agents-with-ollama-langchain-9cf9ab3c84ef

[14] neo4j-semantic-ollama, https://github.com/langchain-ai/langchain/blob/master/templates/neo4j-semantic-ollama/neo4j_semantic_ollama/agent.py

[15] Open-source LLMs as LangChain Agents, https://huggingface.co/blog/open-source-llms-as-agents

[16] why we no longer use LangChain for building our AI agents, https://www.octomind.dev/blog/why-we-no-longer-use-langchain-for-building-our-ai-agents

[17] AI agents in LangGraph, https://x.com/hwchase17/status/1798386148982878477?t=Nj2MFJwAgMivqNhDVetVLA&s=03