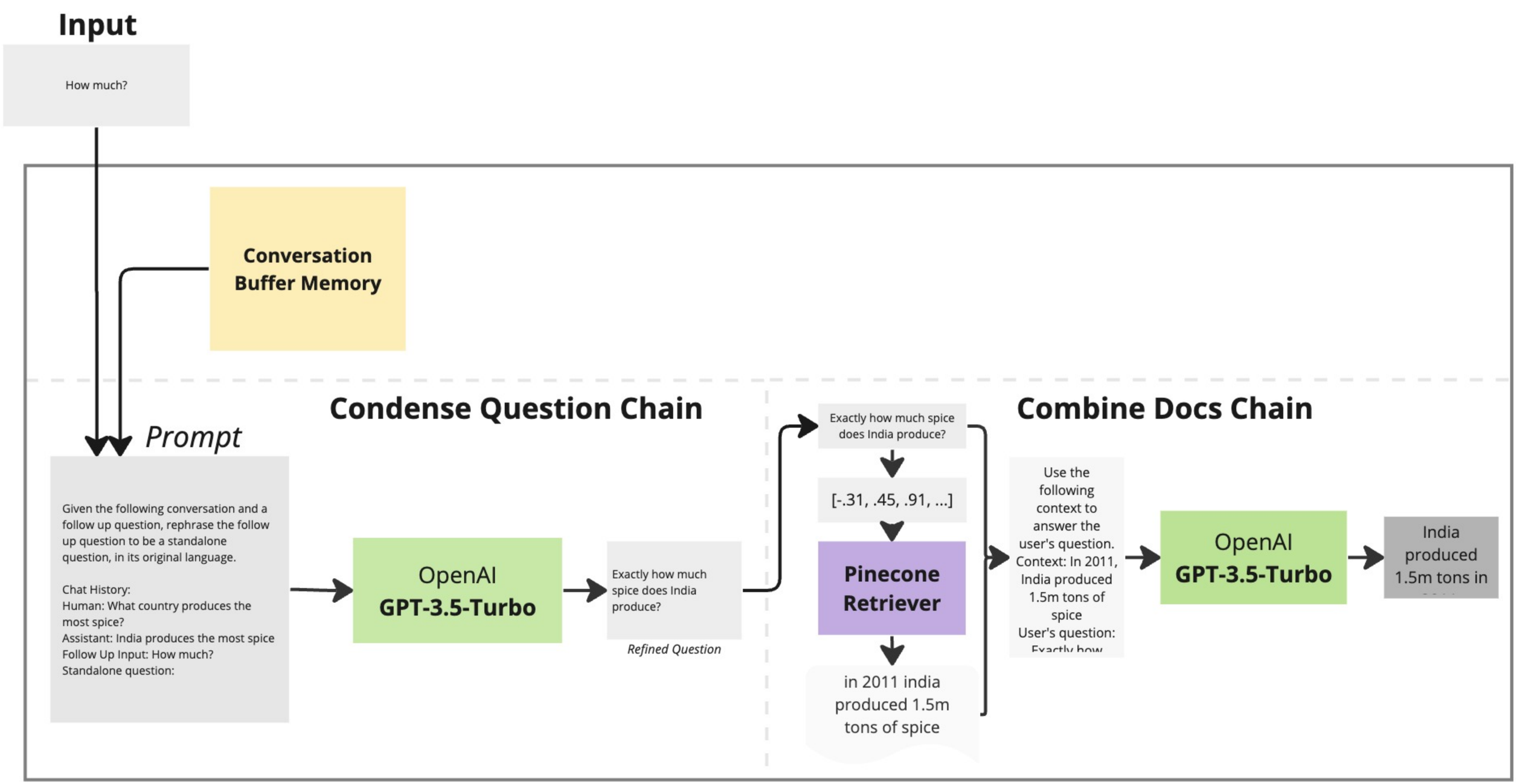
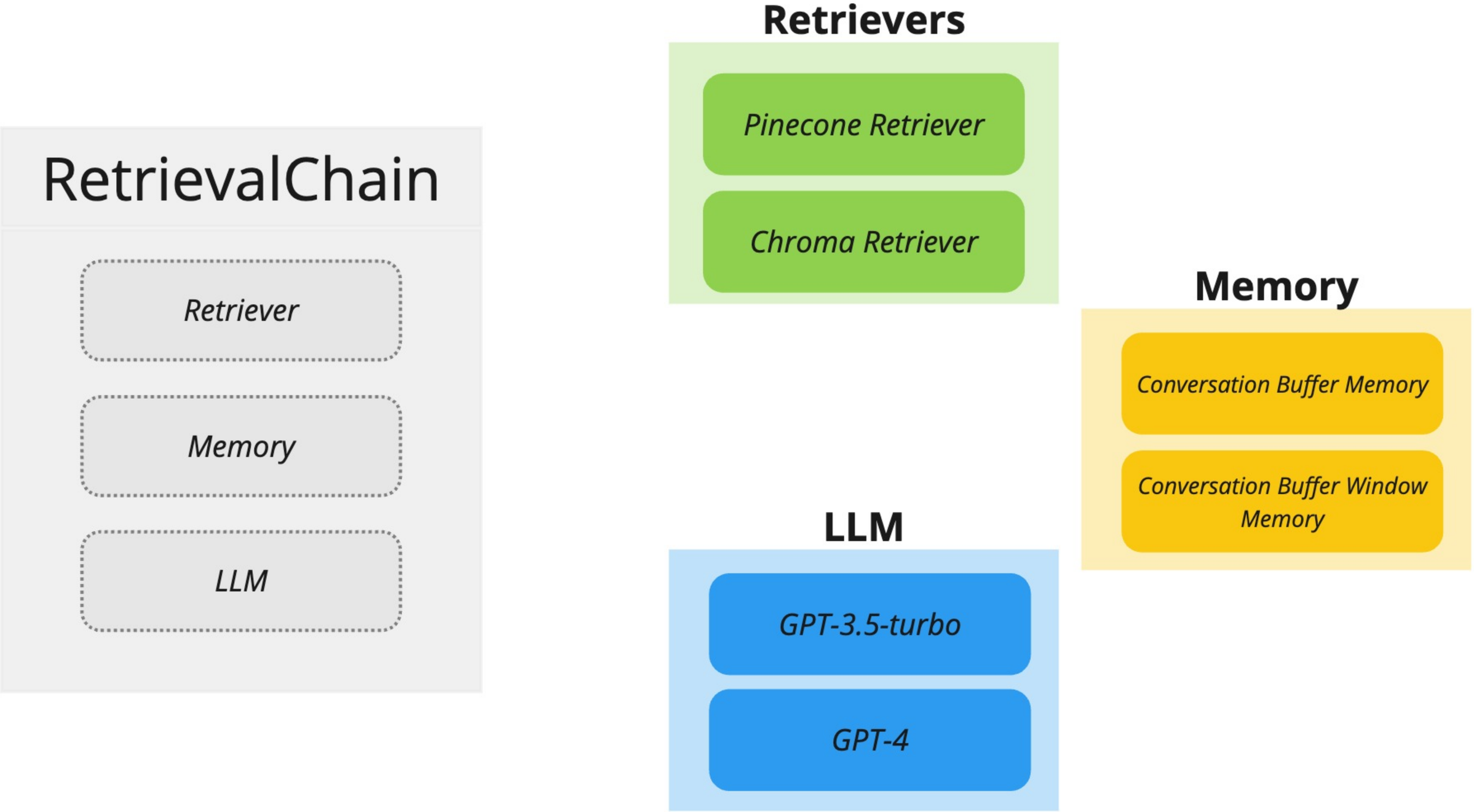
build_chat()

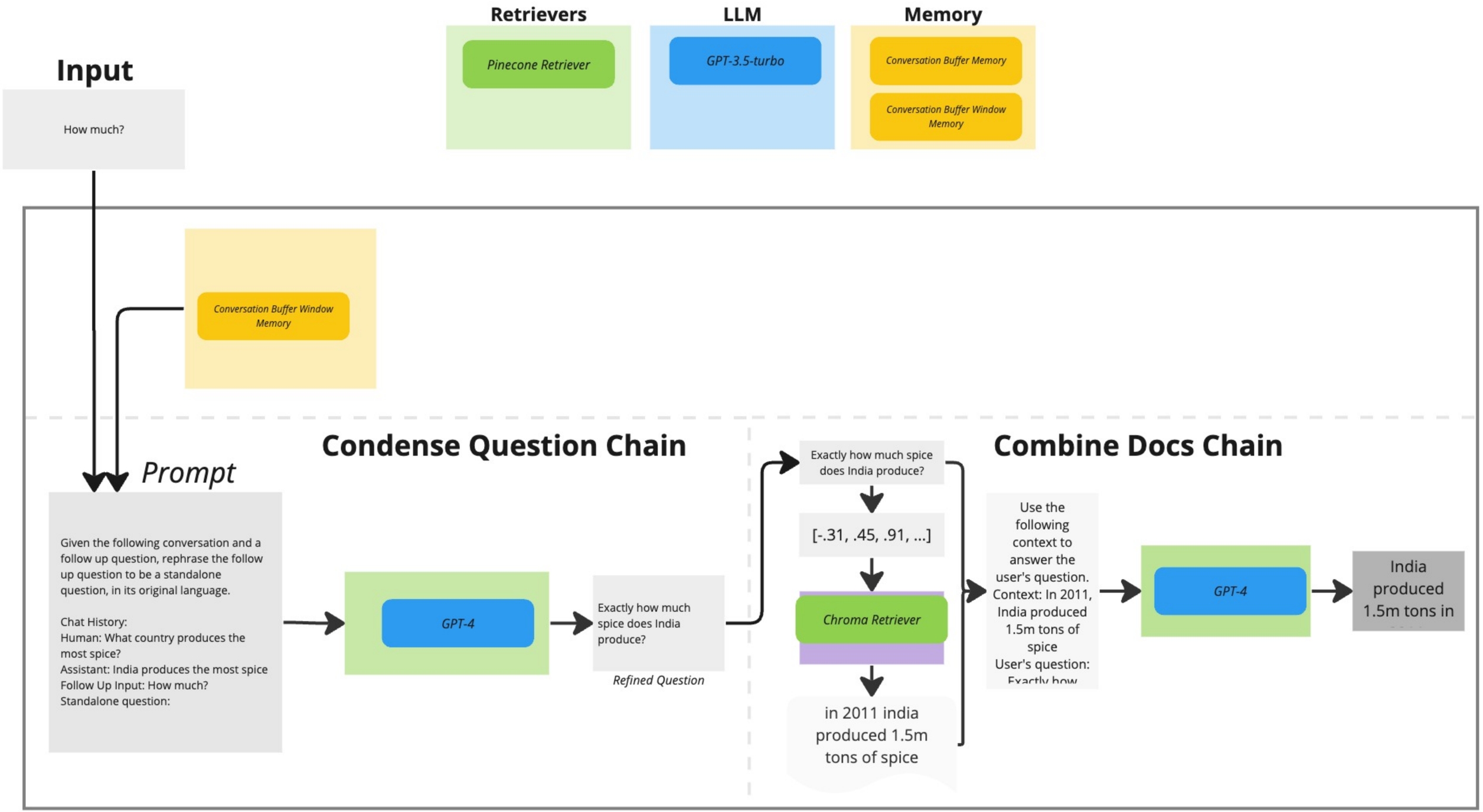
Builds and returns a Conversational Retrieval Chain to answer a user's question

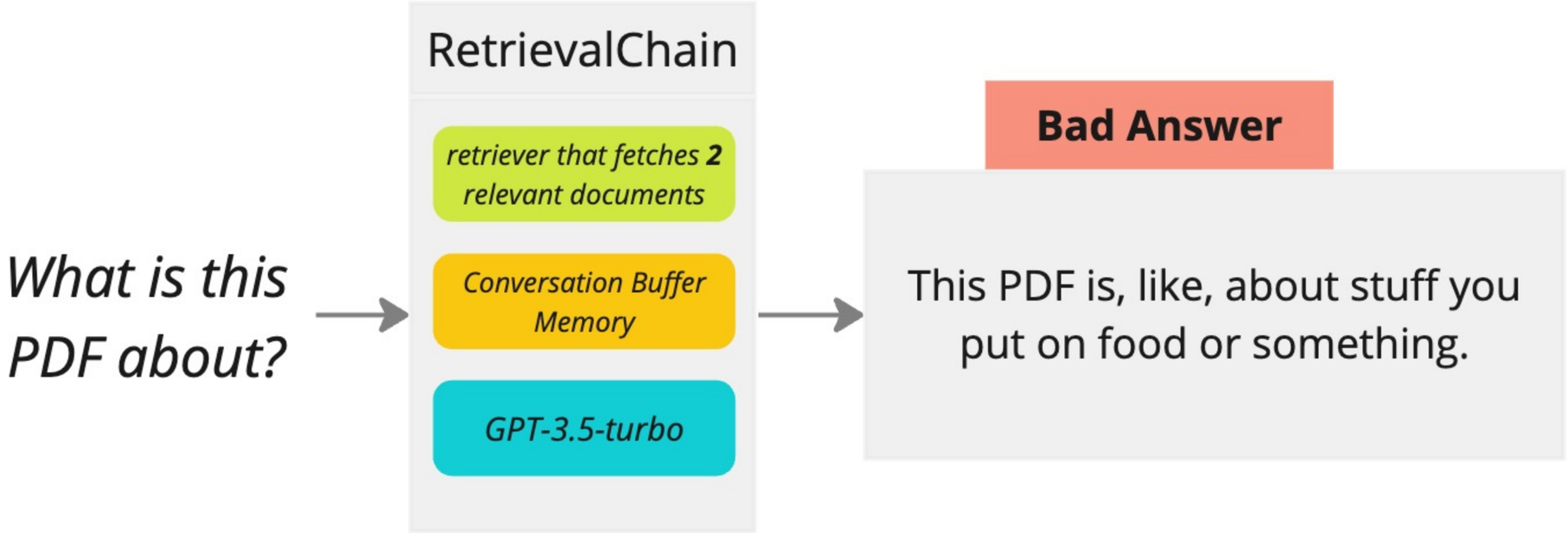
The chain should support streaming text generation

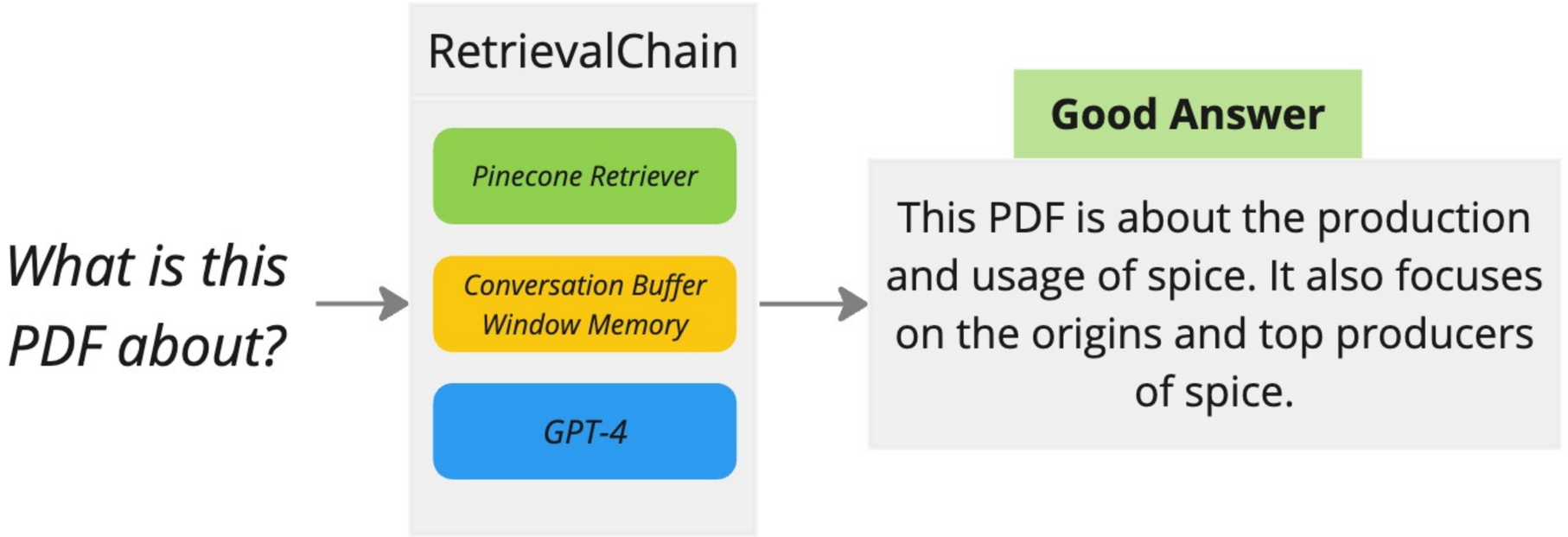
The chain should use random variations of component parts

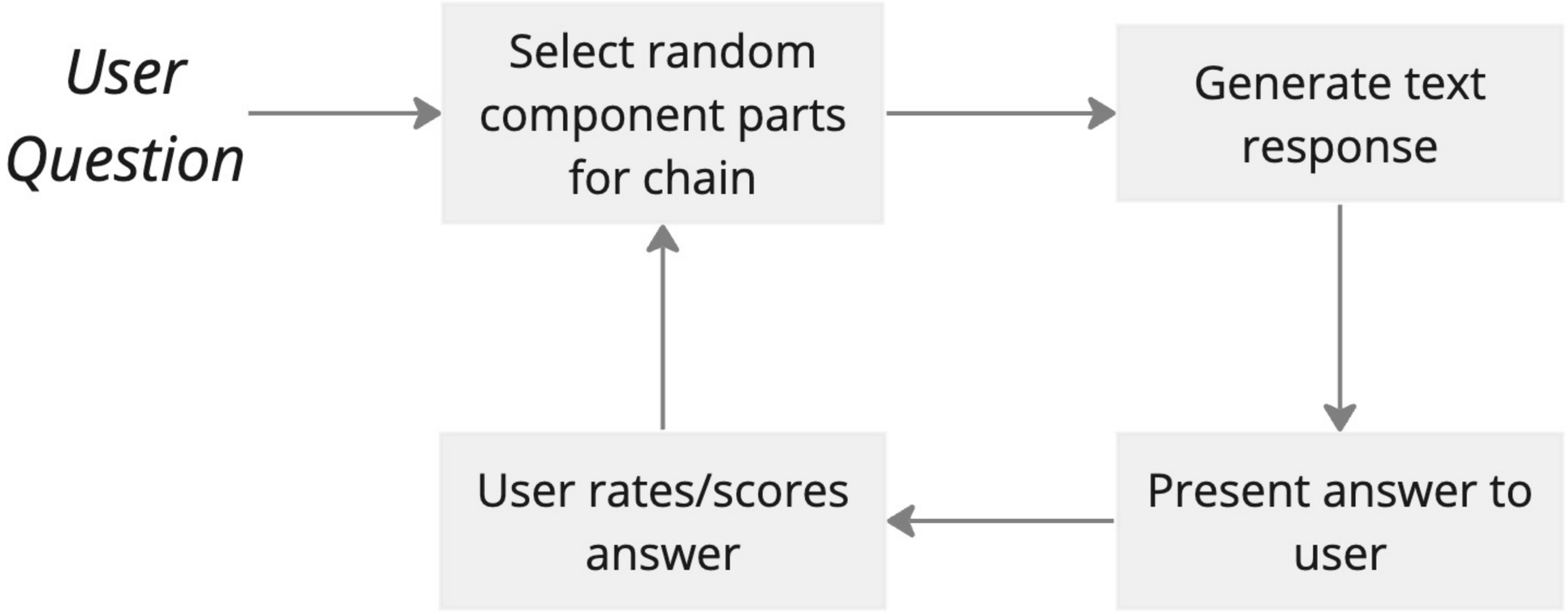

Input
How much?
Conversation Buffer Memory
Condense Question Chain
Prompt
Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question, in its original language.
Chat History:
Human: What country produces the most spice?
Assistant: India produces the most spice
Follow Up Input: How much?
Standalone question:
OpenAI GPT-3.5-Turbo
Exactly how much spice does India produce?
Refined Question
Exactly how much spice does India produce?
[-.31, .45, .91, ...]
Pinecone Retriever
in 2011 india produced 1.5m tons of spice
Combine Docs Chain
Use the following context to answer the user's question.
Context: In 2011, India produced 1.5m tons of spice
User's question:
OpenAI GPT-3.5-Turbo
India produced 1.5m tons in


RetrievalChain
Retriever
Memory
LLM
Retrievers
Pinecone Retriever
Chroma Retriever
Memory
Conversation Buffer Memory
Conversation Buffer Window Memory
LLM
GPT-3.5-turbo
GPT-4


Retrievers
Pinecone Retriever
LLM
GPT-3.5-turbo
Memory
Conversation Buffer Memory
Conversation Buffer Window Memory
Input
How much?
Conversation Buffer Window Memory
Condense Question Chain
Prompt
Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question, in its original language.
Chat History:
Human: What country produces the most spice?
Assistant: India produces the most spice
Follow Up Input: How much?
Standalone question:
GPT-4
Exactly how much spice does India produce?
Refined Question
Exactly how much spice does India produce?
[-.31, .45, .91, ...]
Chroma Retriever
in 2011 india produced 1.5m tons of spice
Combine Docs Chain
Use the following context to answer the user's question.
Context: In 2011, India produced 1.5m tons of spice
User's question:
GPT-4
India produced 1.5m tons in


RetrievalChain
retriever that fetches 2 relevant documents
Conversation Buffer Memory
GPT-3.5-turbo
What is this PDF about?
Bad Answer
This PDF is, like, about stuff you put on food or something.


RetrievalChain
Pinecone Retriever
Conversation Buffer Window Memory
GPT-4
What is this PDF about?
Good Answer
This PDF is about the production and usage of spice. It also focuses on the origins and top producers of spice.


User Question
Select random component parts for chain
Generate text response
Present answer to user
User rates/scores answer

Component Maps

Retrievers

name	component
pinecone_2	*Pinecone Retriever that fetches top **2** docs*
pinecone_3	*Pinecone Retriever that fetches top **3** docs*

Memory

name	component
buffer	*Conversation Buffer Memory*
window	*Conversation Buffer Window*

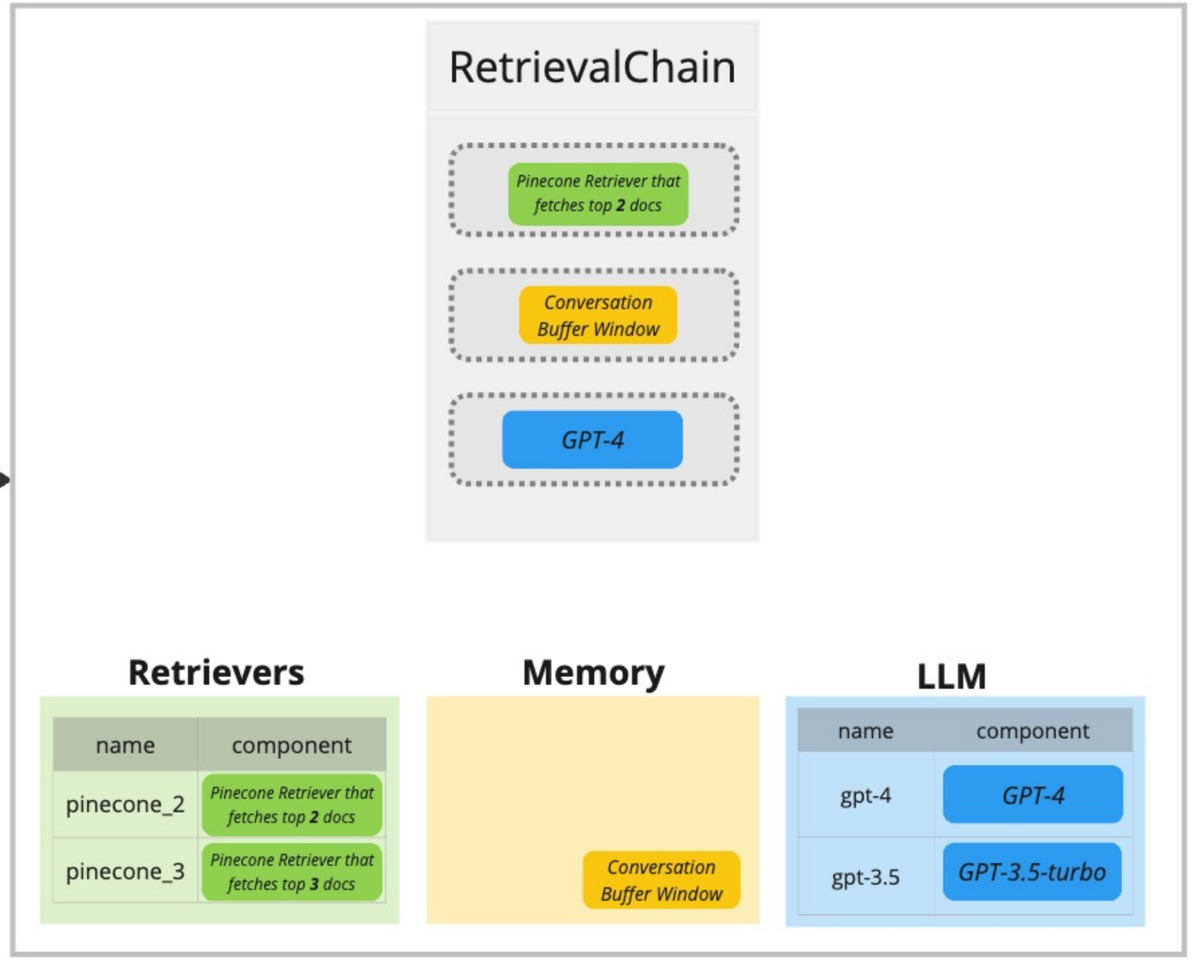
LLM

name	component
gpt-4	*GPT-4*
gpt-3.5	*GPT-3.5-turbo*

Dictionaries where the key is an identifier for a component configured in a particular way

First message of conversation

User Message tied to Conversation '123'



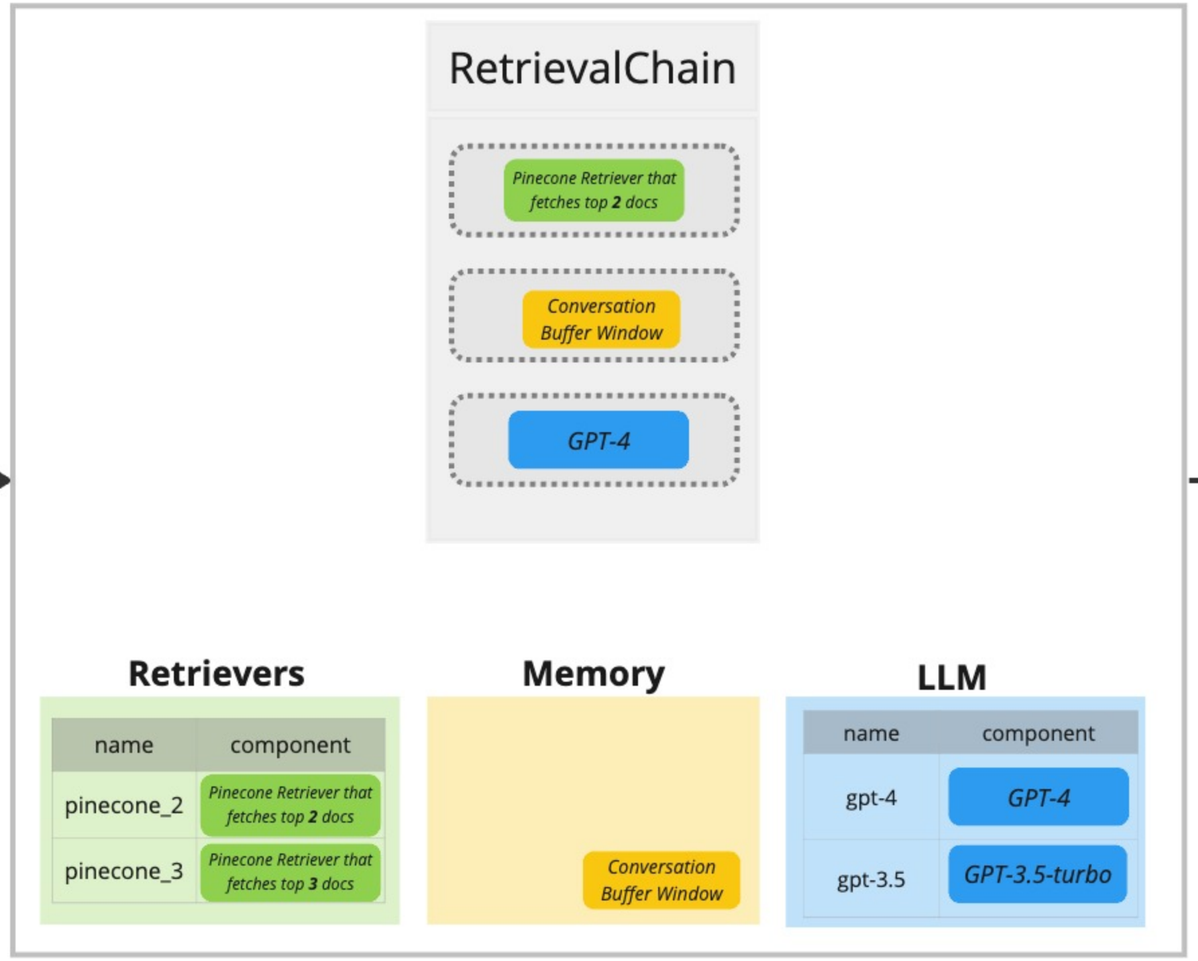
SQLite Database List of Conversations

id	llm	memory	retriever
123	gpt-4	window	pinecone_2

This table already exists in the database

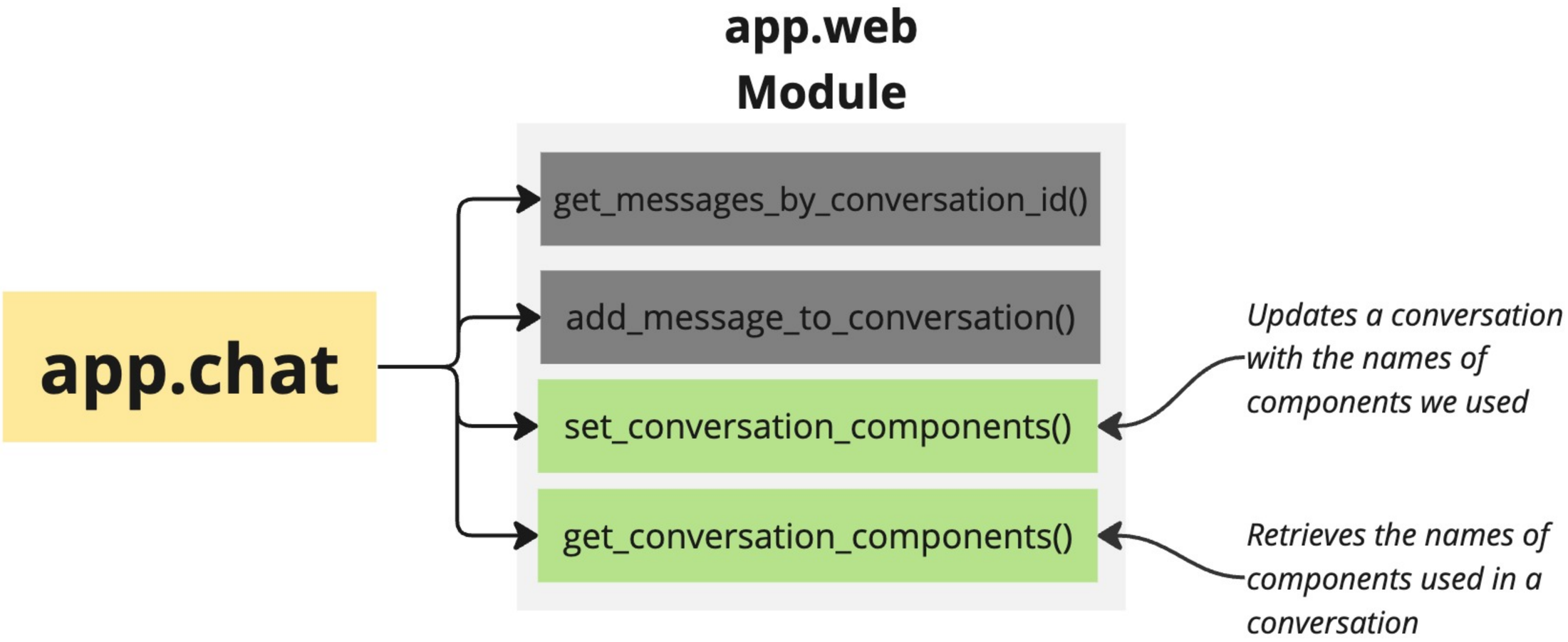
Second, third, fourth, etc message of conversation

User Message tied to Conversation '123'



SQLite Database List of Conversations

id	llm	memory	retriever
123	gpt-4	window	pinecone_2


app.web
Module
app.chat
get_messages_by_conversation_id()
add_message_to_conversation()
set_conversation_components()
get_conversation_components()
Updates a conversation with the names of components we used
Retrieves the names of components used in a conversation

```python
def build_chat(chat_args):
    # Assume it is a follow-up message
    components = get_conversation_components(chat_args.conversation_id)

    print(components) # -> { "llm": "gpt-3.5-turbo" }

    if components["llm"] == "gpt-3.5-turbo":
        retriever = build_retriever(
            chat_args,
            k=2
        )
    elif components["llm"] == "gpt-4":
        retriever = build_retriever(
            chat_args,
            k=3
        )
```

↓

```python
def build_retriever(chat_args, k):
    search_kwargs = {
        "filter": { "pdf_id": chat_args.pdf_id },
        "k": k
    }
    return vector_store.as_retriever(
        search_kwargs=search_kwargs
    )
```

Different Retrievers might require different config

name	component
pinecone_2	*Pinecone Retriever that fetches top **2** docs*
pinecone_3	*Pinecone Retriever that fetches top **3** docs*
chroma_spice_2	*Chroma Retriever that finds first 2 docs that contain the word "spice"*
chroma_3	*Chroma Retriever that finds first 3 docs*

```
{
    "filter": {"pdf_id": chat_args.pdf_id },
    "k": 2
}
```

```
{
    "filter": {"pdf_id": chat_args.pdf_id },
    "k": 3
}
```

```
{
    "where": {"pdf_id": chat_args.pdf_id}
    "where_doc": {"$contains": {"text": "spice"} }
}
```

```
{
    "where": {"pdf_id": chat_args.pdf_id}
    "where_doc": {"$contains": {"text": "technical"}}
}
```

We aren't going to add in Chroma, this is just an example

```python
def build_chat(chat_args):
    conversation_id = chat_args.conversation_id
    # Assume it is a follow-up message
    components = get_conversation_components(conversation_id
        )

    print(components) # -> { "retriever": "pinecone_2" }

    if components["retriever"] == "pinecone_2":
        retriever = build_pinecone_retriever(
            chat_args,
            k=2
        )
    elif components["retriever"] == "pinecone_3":
        retriever = build_pinecone_retriever(
            chat_args,
            k=3
        )
    elif components["retriever"] == "chroma_spice_2"
        retriever = build_chroma_retriever(
            chat_args,
            n_results=2,
            where_doc={"$contains": {"text": "spice" }}
        )
    elif components["retriever"] == "chroma_3"
        retriever = build_chroma_retriever(
            chat_args,
            n_results=3
        )
```

```python
def build_pinecone_retriever(chat_args, k):
    search_kwargs = {
        "filter": { "pdf_id": chat_args.pdf_id },
        "k": k
    }
    return vector_store.as_retriever(
        search_kwargs=search_kwargs
    )
```

```python
def build_chroma_retriever(chat_args, n_results, where_doc):
    search_kwargs = {
        "where": { "pdf_id": chat_args.pdf_id },
        "n_results": n_results,
        "where_document": where_doc
    }
    return vector_store.as_retriever(
        search_kwargs=search_kwargs
    )
```

```python
def build_chat(chat_args):
    conversation_id = chat_args.conversation_id
    # Assume it is a follow-up message
    components = get_conversation_components(conversation_id
        )

    print(components) # -> { "retriever": "pinecone_2" }

    if components["retriever"] == "pinecone_2":
        retriever = build_pinecone_retriever(
            chat_args,
            k=2
        )
    elif components["retriever"] == "pinecone_3":
        retriever = build_pinecone_retriever(
            chat_args,
            k=3
        )
    elif components["retriever"] == "chroma_spice_2"
        retriever = build_chroma_retriever(
            chat_args,
            n_results=2,
            where_doc={"$contains": {"text": "spice" }}
        )
    elif components["retriever"] == "chroma_3"
        retriever = build_chroma_retriever(
            chat_args,
            n_results=3
        )
```

Many arguments are hard coded and tied to the name of the component