

# **Generative AI**

# **W3 Agenda**

- **LLM Flow orchestration**
  - **Combining multiple LLM calls**
  - **How to monitor complex LLM flow**
- **AI Agents**
  - **What is an AI Agent?**
  - **How LLMs can take actions with tools?**
- **End of Course Assignment summary**
- **Chat Bots**
  - **Biggest challenges in building chatbots**
  - **How to balance performance, costs and latency**

# **LLM Flow orchestration**

# **Combining multiple LLM calls**

# LLMs perform best when they have single, clear objective, for more complex tasks its better to split them to multiple calls

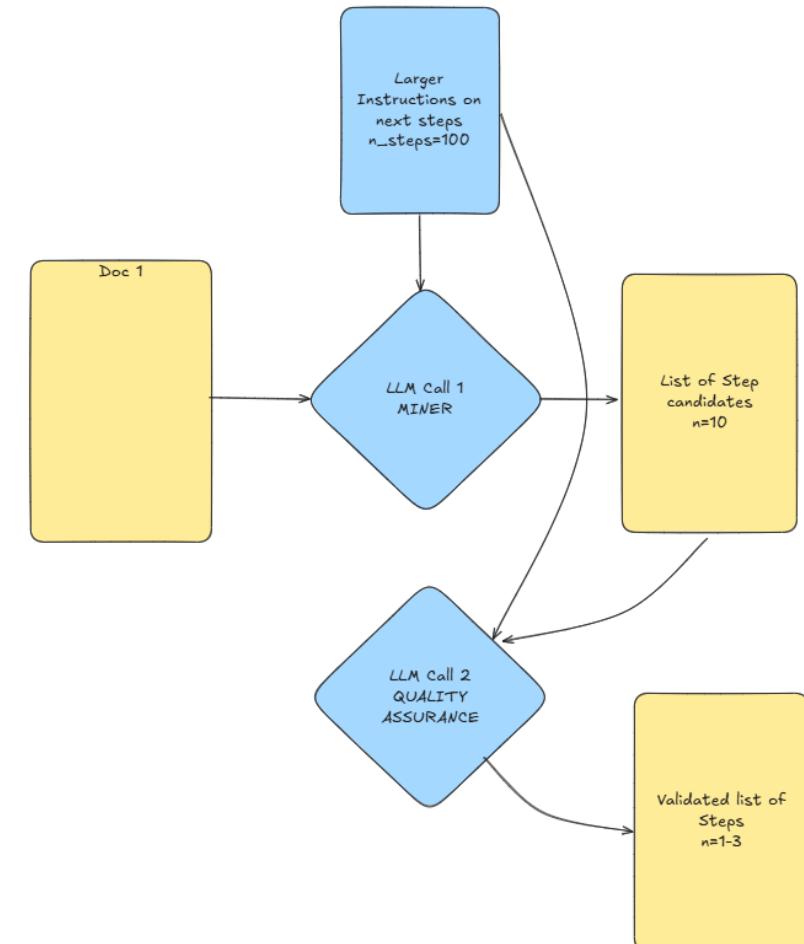


- Multiple calls can get job done with smaller models
- Performing multiple tasks e.g. mining, validation and scoring at the same time leads to worst results
- More tasks -> more prompt -> dilution of attention, which can lead to worse results
- You can use results from one stage to power following calls, or even route next steps to be taken
- Multiple calls with routing save prompt making it faster, cheaper and more attentive
- You can use programmatic validation between them

# Chaining multiple calls can give LLM different roles e.g. Miner/Extractor and Judge/Quality Assurance

Solving next steps to be taken based on the sales meeting transcript with mutliple call example

- Imagine you have a 30 min recording and need to select best 1-3 steps based on 100 possibilities
- With so much context it is hard to do in single call
- But if we select some candidate steps in first call
- And then select which of these are really best fit in 2nd call we can get a more focused results
- Output of LLM call 1 feed directly to prompt for LLM call 2 to narrow down search criteria



# Validating / Modyfing outputs programatically between calls can make systems more predictable

- If you can solve some problem in a few lines of code, DO NOT USE a multi B params model
- If you need to match some values to a finite list, you can use fuzzy match
- If you need to implement some logic like loose matching try doing it programatically instead of trying to prompt LLMs to let through anything between +/- 10% from X

Example of guardrails funciton for LLM powered real estate search

```
def search_params_guardrails(params):  
    #Remove price min threshold if it is less than 20% from max  
    if 'price' in params and all(key in params["price"] for key in ['min', 'max']):  
        if (params['price']['max'] - params['price']['min']) / params['price']['max'] <= 0.20:  
            del params['price']['min']  
  
    # Spread area thresholds by +/-10% if both 'min' and 'max' are specified and equal  
    if 'area' in params and all(key in params["area"] for key in ['min', 'max']):  
        if params['area']['min'] == params['area']['max']:  
            params['area']['min'] *= 0.9  
            params['area']['max'] *= 1.1  
  
    return params
```

# Validating / Modyfing outputs programmatically between calls can make systems more predictable

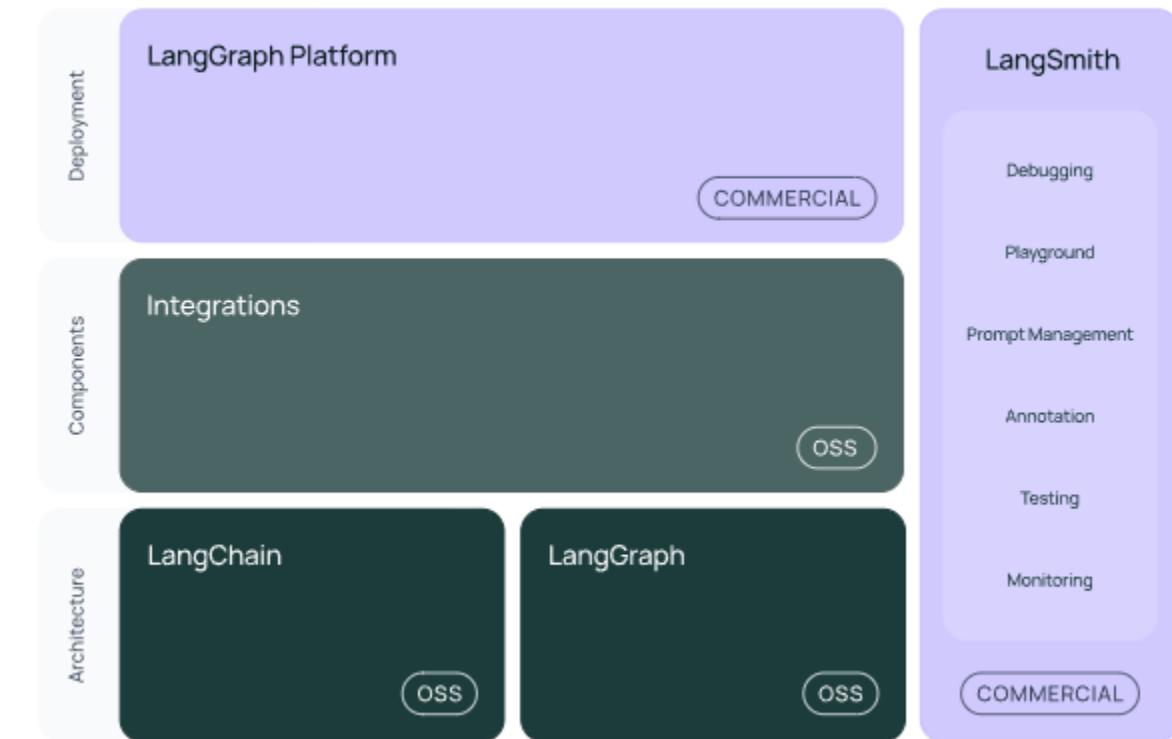
LangChain is a framework for developing applications powered by large language models (LLMs).

LangChain simplifies every stage of the LLM application lifecycle:

Development: Build your applications using LangChain's open-source components and third-party integrations. Use LangGraph to build stateful agents with first-class streaming and human-in-the-loop support.

Productionization: Use LangSmith to inspect, monitor and evaluate your applications, so that you can continuously optimize and deploy with confidence.

Deployment: Turn your LangGraph applications into production-ready APIs and Assistants with LangGraph Platform.



Source: [langchain.com](https://langchain.com)

# Most important langchain concepts [1/2]

## LLM (Large Language Model)

- Core building block that provides text generation and reasoning capabilities
- Abstracted in LangChain via classes like BaseChatOpenAI and others
- Handles prompting, token limits, and output generation
- Can be configured with parameters like temperature, max\_tokens, and model name
- Acts as the "brain" that processes textual inputs and generates responses

## Prompts

- Templates for generating instructions to LLMs
- Can incorporate variables using f-strings or LangChain template syntax
- Structured via ChatPromptTemplate, HumanMessagePromptTemplate, etc.
- Support different message types (system, human, assistant)
- Can include examples for few-shot learning

# Most important langchain concepts [2/2]

## Chain

- Connects multiple components in a processing pipeline
- Enables sequential execution of operations on data
- Can combine prompts, LLMs, and other tools in a reusable workflow
- Allows for modular, reusable components that can be composed together
- Chains can consist of other chains, creating readable levels of abstraction

## RunnableBranch:

- Provides conditional logic in LangChain workflows
- Routes execution based on predicates (conditions)
- Takes a list of (condition, runnable) pairs and a default runnable
- Enables dynamic, conditional processing paths

## RunnablePassthrough

- Passes inputs through without modification
- Often used with .assign() to add new fields to the data context
- Maintains existing data while adding or transforming specific fields
- Helps manage state throughout a complex workflow
- Enables data transformation without losing context

# Simple langchain example

```
# Define the sentiment analysis response format
sentiment_parser = JsonOutputParser()

# Sentiment analysis prompt
sentiment_prompt = PromptTemplate(
    template="""You are a sentiment analysis expert.
Review the following customer review and determine if it's positive or negative.

Review: ```{review}```

Return answer as a valid json object with the following format:
{{"positive_sentiment": boolean, "reasoning": string}}
""",
    input_variables=["review"]
)

# Create sentiment analysis chain - use proper configuration to get direct output
sentiment_chain = sentiment_prompt | llm | sentiment_parser
```

Prompt gives instructions and sets up argument

LLM manages which models is used

Parser makes sure that final output is a JSON

Runnable Passthrough.assign fetches output of one sub-chain to be fed to next steps

```
# Create the branching logic with RunnablePassthrough directly in the chain
full_chain = (
    RunnablePassthrough.assign(
        sentiment_result=sentiment_chain
    )
    | RunnableBranch(
        (
            lambda x: x["sentiment_result"]["positive_sentiment"],
            {
                "review": lambda x: x["review"],
                "reasoning": lambda x: x["sentiment_result"]["reasoning"]
            } | positive_chain
        ),
        (
            lambda x: not x["sentiment_result"]["positive_sentiment"],
            {
                "review": lambda x: x["review"],
                "reasoning": lambda x: x["sentiment_result"]["reasoning"]
            } | negative_chain
        ),
        # Default fallback
        lambda x: { "message": f"Error: Unable to determine sentiment for: {x['review']}'" }
    )
    | negative_message
)
negative_message | full_chain.invoke({"review": negative_test})
```

RunnableBranch Executes one off possible LLMs based on value of previous step

# Introduction to Langchain

## Start in W3-IIm-flows-and-monitoring

```
● from langchain.chains import LLMChain
  from langchain.output_parsers import StructuredOutputParser, ResponseSchema
  from langchain.schema import Document
  from langchain.schema.runnable import RunnablePassthrough, RunnableBranch
  from typing import Literal, List, Dict, Any
  from pydantic import BaseModel, Field

  from langchain_core.runnables import RunnableBranch
  from langchain_core.prompts import PromptTemplate

# Initialize LLM
llm = ChatOpenAI(temperature=0, model_name="gpt-4o")
```

[62] ✓ 0.0s

## Sentiment classification

```
from langchain_core.output_parsers import JsonOutputParser

# Define the sentiment analysis response format
sentiment_parser = JsonOutputParser()

# Sentiment analysis prompt
sentiment_prompt = PromptTemplate(
    template="""You are a sentiment analysis expert.
Review the following customer review and determine if it's positive or negative.

Review: ```{review}```

Return answer as a valid json object with the following format:
{{"positive_sentiment": boolean, "reasoning": string}}
""",
    input_variables=["review"]
)

# Create sentiment analysis chain - use proper configuration to get direct output
sentiment_chain = sentiment_prompt | llm | sentiment_parser
```

[56]

# **LLM Flow Monitoring**

# Setting up Langsmith

1. Go to <https://smith.langchain.com/>
2. Make a free account
3. Go to Set Up Tracing
4. Generate API\_KEY
5. Copy first code snippet
6. Paste it to .env file
7. Remove „ ” for strings as .env file does not need them
8. Change project to something meaningful

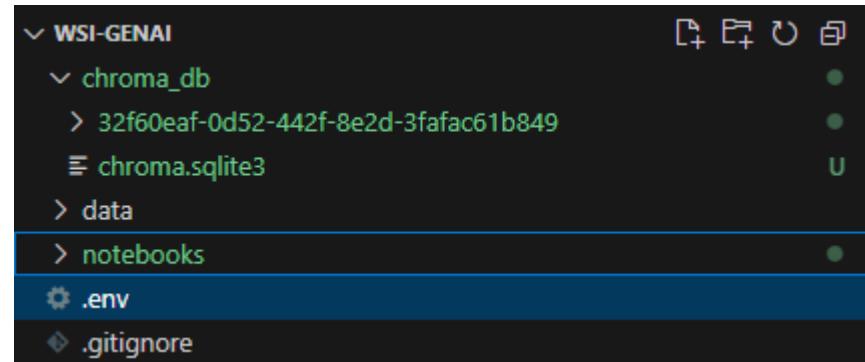
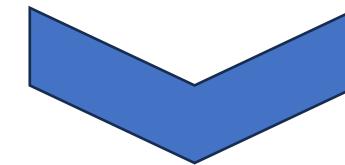
3. Configure environment to connect to LangSmith.

Project Name

pr-scholarly-length-34

```
1 LANGSMITH_TRACING=true
2 LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
3 LANGSMITH_API_KEY=<your-api-key>
4 LANGSMITH_PROJECT="pr-scholarly-length-34"
5 OPENAI_API_KEY=<your-openai-api-key>
```

Copy



# How to navigate logs for full LLM observability

The screenshot illustrates a log tracing interface designed for full LLM observability. On the left, a trace tree shows the execution flow of a process named "RunnableSequence". The tree is organized into two main sections: one for sentiment processing and one for reasoning/review generation. The sentiment section includes steps like "RunnableAssign<sentiment\_result>" and "RunnableParallel<sentiment\_result>". The reasoning/review section includes "RunnableBranch", "RunnableLambda", and "RunnableSequence" for generating reviews. Each step is accompanied by its duration (e.g., 1.00s, 0.99s) and a small icon representing the component or model used.

On the right, a detailed view of a specific "RunnableSequence" step is shown. This view is divided into three tabs: Run, Feedback, and Metadata. The Run tab displays the input message: "The service was outstanding and the food was delicious. I'll definitely come back!". The Output tab shows the generated response, which includes a message of thanks, an offer for a voucher, and a welcome message for future visits. The Feedback tab is currently inactive.

# Debugging through logs

- As LLM flows become more complex and they take multiple, non deterministic inputs (e.g. from previous steps) clear logging is crucial to understand their mistakes
- Due to non-deterministic nature it is often easier to debug logs than try to replicate all states
- Minor mistakes such as incorrect setup of some prompt inputs or missing memory are easiest to find in logs
- Small upstream changes can lead to significant shifts 2-3 calls further

# Jupyter exercise Reranking

Rerun W3-llm-flows-and-monitoring after setting up langchain

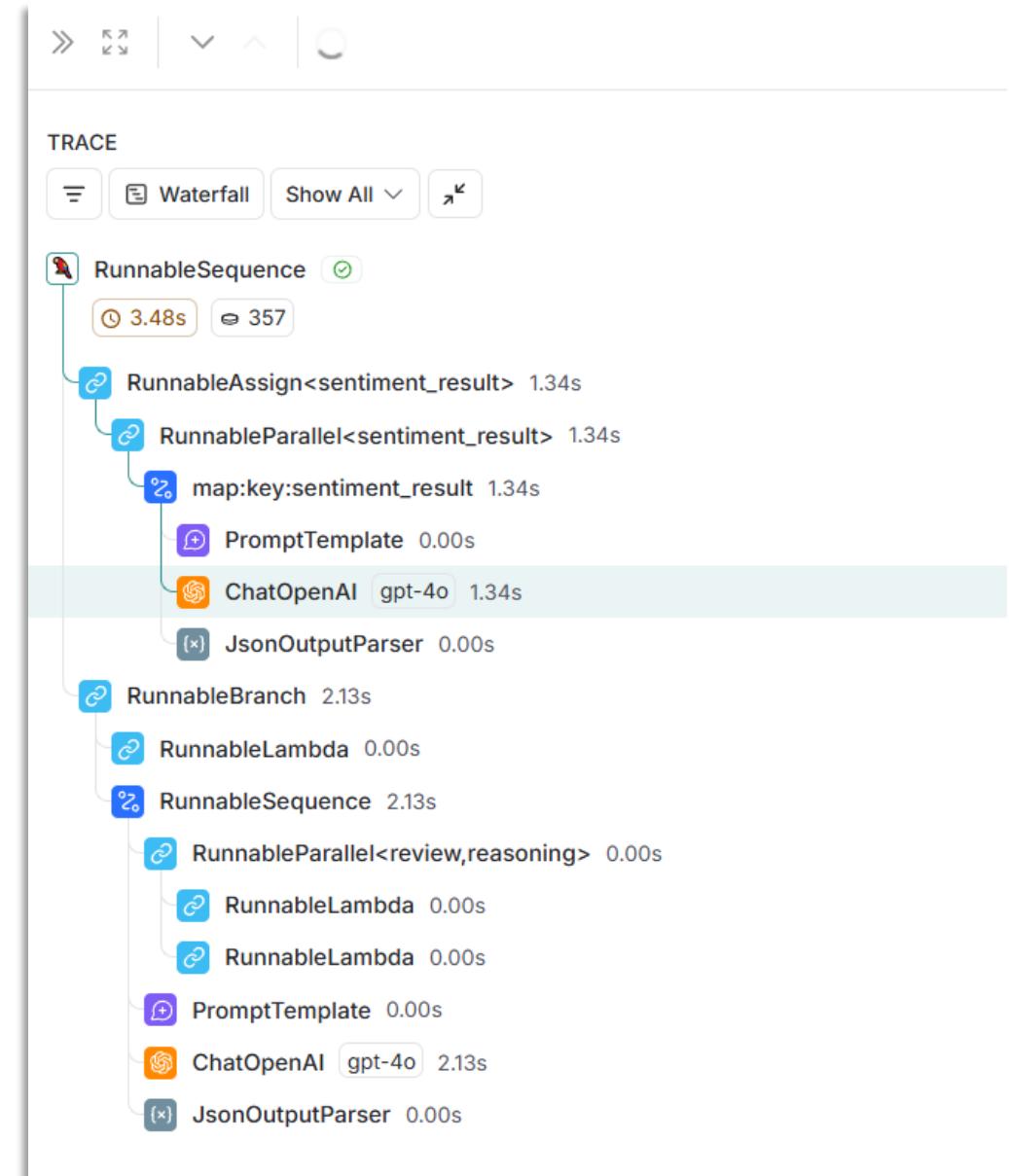
Personal > Tracing Projects > wws1-genai

### wws1-genai

Runs Threads Monitor Setup

1 filter Last 7 days Root Runs LLM Calls All Runs

Name	Input	Output	Error	Start Time	Latency	Dataset	Annotation Queue	Tokens	Cost
RunnableSequence	The service was outst...	Dear Customer, T...		3/25/2025, 8:27:40 ...	3.48s			357	\$0.0020325
RunnableSequence	The wait was too long...	Dear Customer, ...		3/25/2025, 8:27:33 ...	6.66s			489	\$0.0031425
RunnableSequence	The wait was too long...	Dear Valued Cust...		3/25/2025, 8:27:30 ...	3.28s			357	\$0.0024
RunnableSequence	The service was outst...	Dear Customer, T...		3/25/2025, 8:27:28 ...	1.73s			222	\$0.00123
RunnableSequence	The wait was too long...	The review expre...		3/25/2025, 8:27:27 ...	1.29s			131	\$0.0007325
RunnableSequence	The service was outst...	The review uses ...		3/25/2025, 8:27:25 ...	2.45s			135	\$0.0008025



# AI Agents

# **What is an AI Agent?**

# What makes LLMs Agentic

## What Are AI Agents?

- Autonomous systems powered by AI that can perceive their environment, make decisions, and take actions
- Combine LLMs with the ability to use tools and execute tasks over multiple steps
- Designed to achieve specific goals with varying degrees of autonomy
- LLM serves as the `brain` of the system, but it also be connected to classical programs, databases, email etc

## Key Capabilities

- **Contextual Memory:** Short and long-term memory systems
- **Reasoning:** Ability to make logical inferences
- **Learning:** Improving performance through experience
- **Multimodal Understanding:** Processing different types of inputs (text, images, etc.)

# Interacting with online world through tools and iterative improvements is what makes Agentic LLMs so powerful

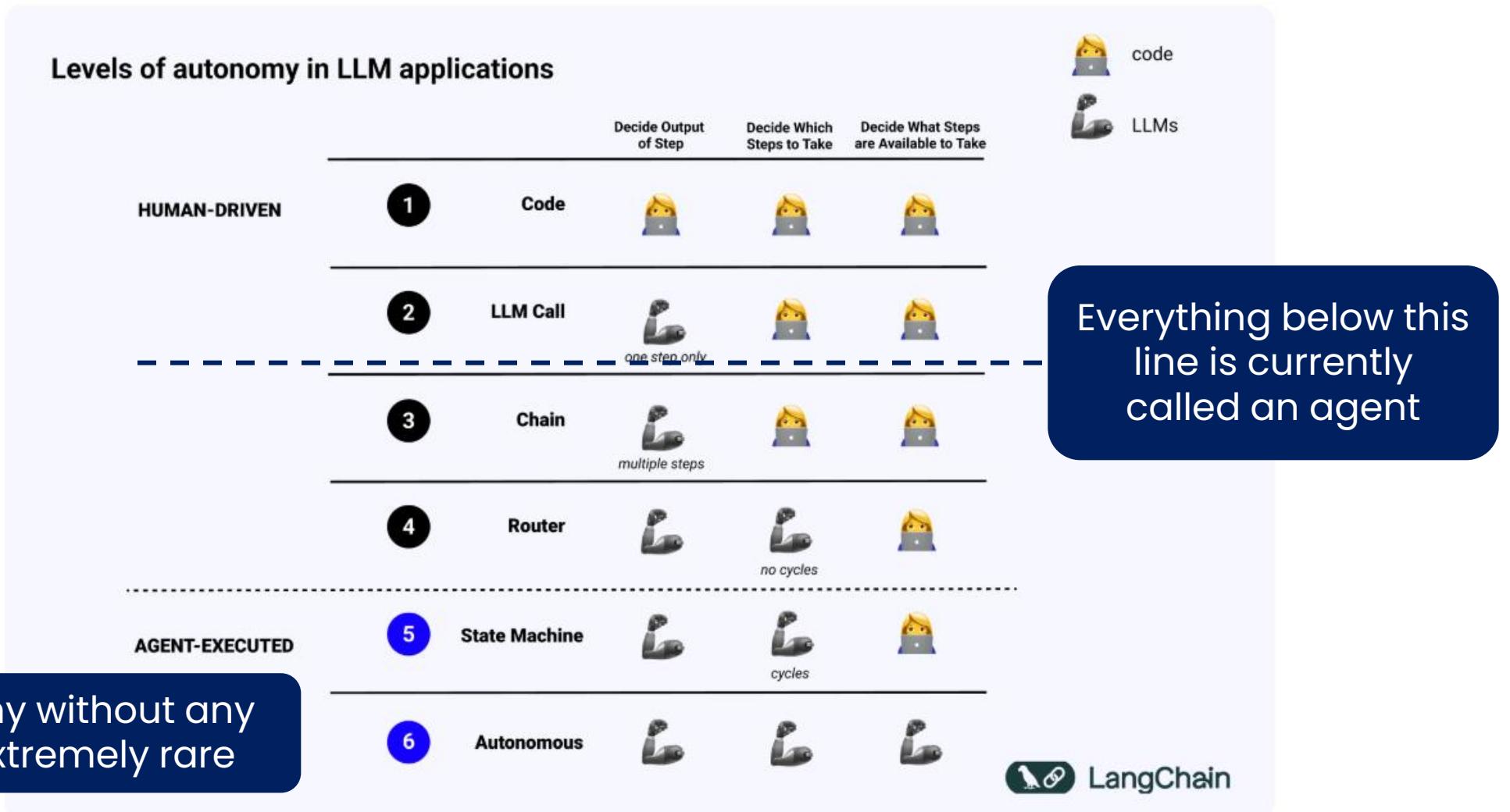
## Iterative Decision Making

- **Plan-Execute-Reflect Loop:** Form a plan, take actions, evaluate results
- **Self-Correction:** Adjust strategy based on feedback and outcomes
- **Decomposition:** Break complex tasks into manageable subtasks
- **Memory Management:** Maintain context across multiple decision points
- **Re-planning:** Update approach when facing unexpected obstacles

## How AI Agents Use Tools

- **Tool Integration:** Access to APIs, databases, code execution, web browsing
- **Tool Selection:** Choose appropriate tools based on the current task requirements
- **Tool Chaining:** Coordinate multiple tools to solve complex problems
- **Tool Augmentation:** Overcome LLM limitations (computation, up-to-date info, specific actions)

# Levels of Autonomy in LLM applications

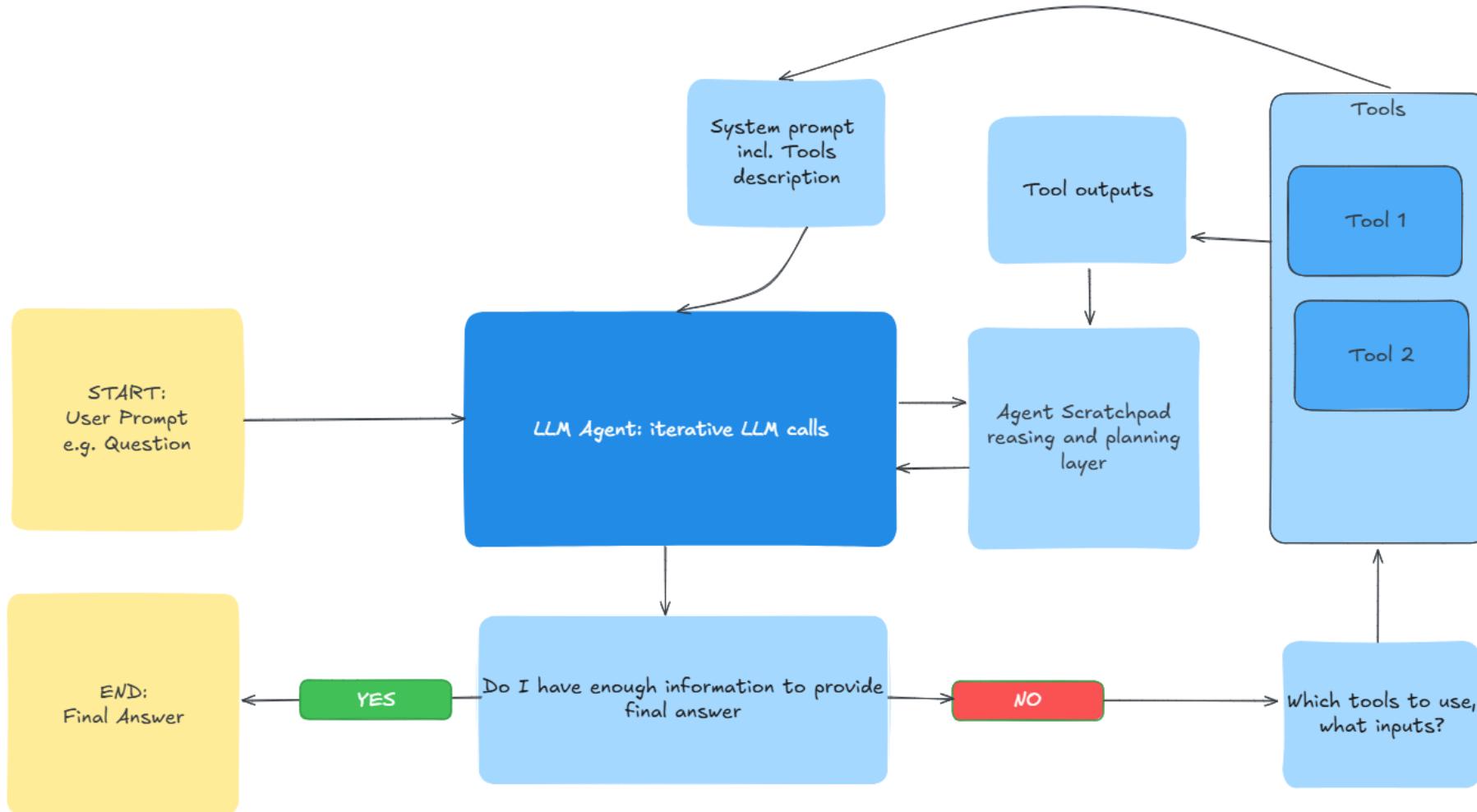


**Empowering LLMs to take  
actions with tools**

# **Why do agents need tools?**

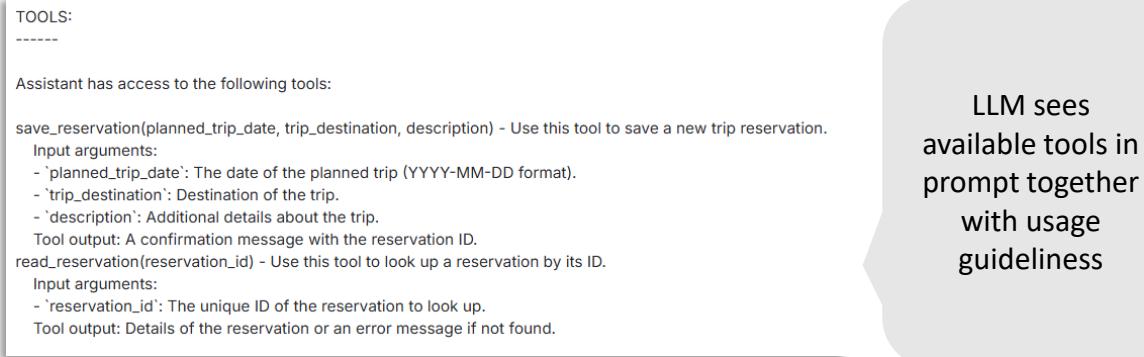
- LLMs are more powerful than just chatbots -> tools allow them to interact with online world through API, Databases, Email or anything that you can trigger with a script
- Tools also enable them to leverage more data for answers, by query API, SQL Databases etc
- Tools can improve reliability where LLMs typically fail e.g. Math, Physics, Word Counting, Dates processing -> if LLMs are smart enough to use a calculator why should they try to replicate it??

# How to convert LLM calls into iterative LLM Agent?



# How tools work from LLM calling perspective

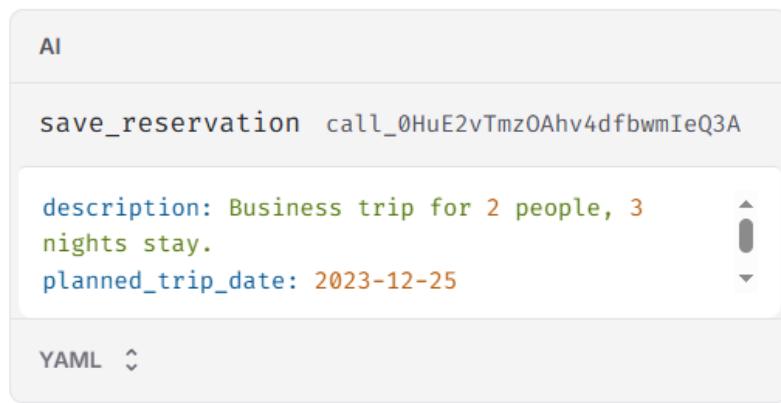
Call #1



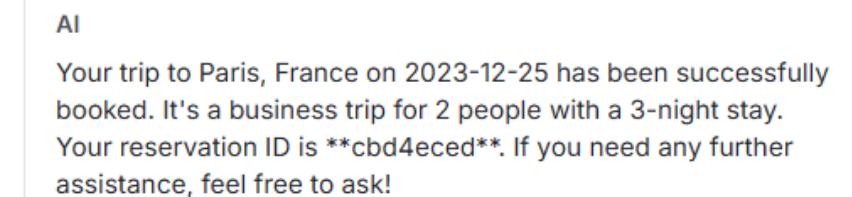
Call #2



Output ▾



Output ▾



# Jupyter exercise Reranking

Start with W3-  
agent\_with\_tools\_BLANK.py

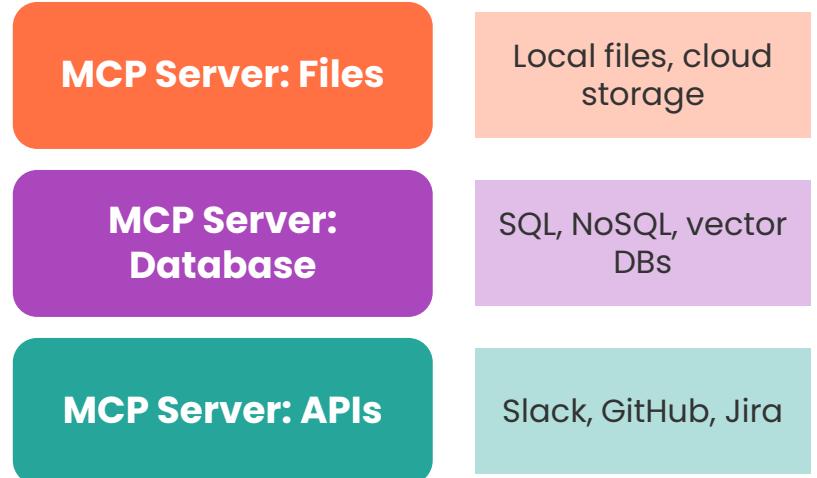
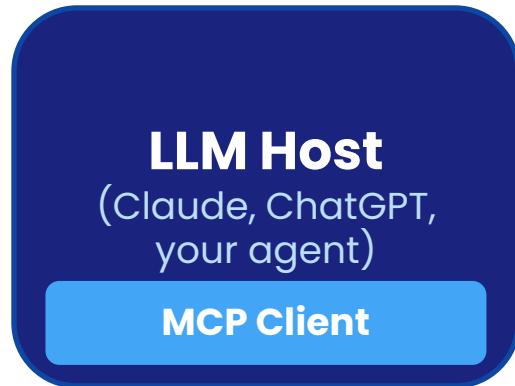
```
135 When you are using a tool, remember to provide all relevant context for the tool to execute the task, especially  
134 Pay attention if user is asking about sale or rent offers.  
135  
136 If you gave the user some recommendations in previous messages and he agrees with them use those recommendations  
137 When analyzing tool output, compare it with Human question, if it only partially answered it explain it to the user  
138  
139 To use a tool, please use the following format:  
140 Thought: Do I need to use a tool? Yes  
141 Action: the action to take, should be one of [{tool_names}]  
142 Action Input: the input to the action  
143 Observation: the result of the action  
144  
145 ... (repeat Thought/Action/Observation as needed)  
146 Final Answer: The response to the user including relevant information from the tools  
147  
148 Begin!  
149  
150 New input:""  
151 | ),  
152 | ("human", "{input}"),  
153 | MessagesPlaceholder(variable_name="agent_scratchpad"),  
154 | ]  
155 )  
156  
157  
158  
159  
160 prompt = prompt.partial(tools=render_text_description(tools), tool_names="", ".join([t.name for t in tools]))  
161 llm_with_tools = llm.bind(tools=[convert_to_openai_tool(tool) for tool in tools])  
162  
163  
164 agent = (  
165 | RunnablePassthrough.assign(  
166 | | agent_scratchpad=lambda x: format_to_openai_tool_messages(x["intermediate_steps"]),  
167 | | )  
168 | | prompt  
169 | | llm_with_tools  
170 | | OpenAIToolsAgentOutputParser()  
171  
172  
173 agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=False, handle_parsing_errors=False)  
174  
175  
176  
177 def run_agent_with_query(query):  
178 | return agent_executor.invoke({"input": query})  
179  
180 if __name__=="__main__":  
181 | query = "I want to book a trip on 2023-12-25 to Paris, France. 2 people for 3 nights. Its a business trip"  
182 | output = run_agent_with_query(query)  
183 | breakpoint()  
184  
185 | query_2 = "What is the status of reservation 9c89a904?"  
186 | output_2 = run_agent_with_query(query_2)
```

# **Model Context Protocol**

## **(MCP)**

# What is MCP (Model Context Protocol)?

A standardized protocol that lets AI models discover and use tools from any MCP-compliant server



## Key Concepts:

- **MCP Client:** Built into the host app, discovers and calls MCP servers
- **MCP Server:** Exposes tools, resources, and prompts via standardized interface
- **Protocol:** JSON-RPC based, supports tool discovery, execution, and resource access
- **Plug-and-Play:** Connect any MCP server to any MCP-compatible host without custom code

# MCP vs Traditional Tools: When to Use Each

## Traditional Tools (Function Calling)

- Tools defined per-application
- Custom integration for each LLM
- Tight coupling to your code
- Full control over tool behavior
- Best for: single-purpose agents, prototypes, custom pipelines

## MCP (Model Context Protocol)

- Tools are reusable across apps
- Standardized interface for all LLMs
- Loose coupling via protocol
- Growing ecosystem of servers
- Best for: multi-tool agents, enterprise apps, tool sharing

### When to choose MCP:

Use MCP when you need to **share tools** across multiple agents, want to use **community-built servers**, or building **production systems** that may switch LLM providers. Traditional tools work best for quick prototypes and tightly-integrated single-agent apps.

# What MCP Really Is (More Than Shareable Tools)

## MCP Provides

- **Tools** – Shareable, reusable actions
- **Resources** – Data/context (files, DB records)
- **Prompts** – Reusable prompt templates
- **Dynamic Discovery** – LLM asks “what can you do?” at runtime

## Can Tools Do Everything MCP Does?

**Technically yes** – but you'd be rebuilding:

- Your own communication protocol
- Your own discovery mechanism
- Custom integration per LLM provider
- What community has already built

**Key Insight:** MCP solves the  $N \times M$  problem – without it, N agents each need custom code for M tools

## Do Traditional Tools Still Make Sense?

**Absolutely!** They're **complementary**, not competing. Use tools for simple, tightly-integrated cases. Use MCP when you need interoperability, reusability, or want to leverage community servers.

# Decision Guide: When to Use Tools vs MCP

## ✓ Use Traditional Tools When...

- Building quick prototypes or experiments
- Single-agent apps with 1-5 simple tools
- Tools tightly coupled to your business logic
- Internal tools you'll never share
- Committed to one LLM provider
- Learning/educational projects

## ✓ Use MCP When...

- Sharing tools across multiple agents
- Using community-built servers (Slack, GitHub, etc.)
- Building production systems
- May switch LLM providers in future
- Need many integrations (5+ external services)
- Want to contribute tools to the ecosystem

## Practical Examples

"I need my agent to call my own Python function"

→ **Tools**

"I want to connect to Slack, GitHub, and a database"

→ **MCP (use existing servers)**

"Building a quick POC this afternoon"

→ **Tools**

"Production system, might switch from Claude to GPT"

→ **MCP**

"I want full control over execution logic"

→ **Tools**

# Chat Bots

# What are chatbots

## Key Concepts

- **Definition:** AI systems designed for human-like conversation
- **Evolution:** Rule-based → ML-powered → LLM-powered
- **Capabilities:** Text completion, knowledge access, tool use
- Most popular topic of 2024 -> real-life showed that Chat bots are harder than they seem
- Chatbots combine LLMs, Tools and managing short and long conversation memory

## Real-World Applications

- Customer service automation
- Virtual assistants (Siri, Alexa)
- Technical support
- Educational tutoring
- Healthcare triage

## Limitations and Risks

Chats are pretty much as open as its possible -> this creates a lot of risks including:

- Toxic behaviour
- Prompt Injection
- Customers trying to get benefits from ChatBot errors -> they have unlimited tries

# 5 steps to building your first bot with memory

## Step 1: Choose a Foundation Model

- Select an appropriate Large Language Model (like GPT-4o in this example)
- Consider capabilities, cost, and performance requirements for your use case

## Step 2: Design the Agent's Personality

- Create a system prompt that defines how the agent should behave
- Set boundaries and establish the agent's role and tone
- Enable the agent to maintain consistent personality across interactions

## Step 3: Implement Memory Management

- Store conversation history to provide context continuity
- Allow the agent to recall previous interactions and user details
- Memory can be as simple as saving messages to a file or database

## Step 4: Build the Communication Loop

- Create a function to handle user input and generate responses
- Process the conversation context before sending to the LLM
- Return responses in a structured format that maintains the conversation flow

## Step 5: Enhance with Additional Capabilities

- Add tools for specific tasks (like web search, calculations, etc.)
- Implement error handling and fallback responses
- Consider privacy features and user identification mechanisms

# Jupyter exercise Reranking

Start with  
w3\_chat\_with\_memory\_BLANK.py

```
def chatbot_response(user_input: str, conversation_id: Optional[str] = None) -> Dict[str, Any]:
    """
    Generate a response from the chatbot with memory

    Args:
        user_input: The user's query
        conversation_id: Optional ID to maintain conversation context
            If None, a new conversation will be started

    Returns:
        Dictionary with response and conversation_id
    """
    # Generate a new conversation ID if not provided
    if not conversation_id:
        conversation_id = str(uuid.uuid4())[:8]

    # Get conversation history
    messages = get_conversation_history(conversation_id)

    # Add user message to history
    messages.append({"role": "human", "content": user_input, "timestamp": datetime.now().isoformat()})

    # Format messages for the prompt
    formatted_history = format_messages_for_prompt(messages)

    # Generate response
    chain = prompt | llm
    response = chain.invoke({"chat_history": formatted_history, "input": user_input})

    # Add AI response to history
    messages.append({"role": "ai", "content": response.content, "timestamp": datetime.now().isoformat()})

    # Save updated conversation
    save_conversation(conversation_id, messages)

    return {
        "response": response.content,
        "conversation_id": conversation_id
    }

if __name__ == "__main__":
    # Example of a new conversation
    message1 = "Hi, my name is Alice. How are you today?"
    result = chatbot_response(message1)
    print(f"Conversation ID: {result['conversation_id']}")
    print(f"Human: {message1}")
    print(f"AI: {result['response']}\n")

    # Continue the same conversation
    conv_id = result['conversation_id']
    message2 = "I'm planning a trip to Spain next month. Have you been there?"
    result2 = chatbot_response(message2, conv_id)
    print(f"Human: {message2}")
    print(f"AI: {result2['response']}
```

# How to build UI for Demos in minutes?

- One of the key reasons why LLMs accelerated AI adoption is by reducing barriers of entry to get smart predictions
- Once we start interacting with LLMs through API, we go back to code, which is harder to read by non-tech people
- Fortunately Streamlit allows us to quickly create mockup Chat/App interface for easier testing and demo
- SoTa LLMs are amazing at building these aps, often in single prompt

## AI Chat Assistant

Chat with an AI that remembers your conversation

Can you recommend me best clustering algorithm for high dimensionality data?

For high-dimensional data, choosing a clustering algorithm can be challenging due to the curse of dimensionality. However, some algorithms are better suited for these situations:

1. **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):** This algorithm is useful for identifying clusters of varying shapes in high-dimensional space, especially when the data contains noise. However, its performance can degrade as dimensions increase.
2. **HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise):** An extension of DBSCAN, HDBSCAN can handle high-dimensional data better by building a hierarchy of clusters. It often provides better results in complex datasets.
3. **Spectral Clustering:** This algorithm works well for high-dimensional data by projecting it into a lower-dimensional space and finding clusters there. It's particularly useful when the clusters are not spherical in the original space.
4. **t-SNE followed by K-Means:** This approach uses t-SNE to reduce dimensionality while preserving local similarities, and then applies K-Means for clustering. It's effective but can be computationally expensive.
5. **Gaussian Mixture Models (GMM):** GMMs can be powerful for high-dimensional data, especially when the data is assumed to be normally distributed. They can capture more complex cluster shapes than K-Means.
6. **Principal Component Analysis (PCA) or Singular Value Decomposition (SVD) for Dimensionality Reduction followed by Clustering:** Reducing dimensions first can help make the clustering more effective and efficient.

Ultimately, the best choice depends on the specific characteristics of your data and the computational resources available. It might be worth experimenting with a few different approaches to see which works best for your needs.

Which one is most scalable? I want to use it on 1B records every hour?

For handling large-scale data like 1 billion records every hour, scalability becomes a critical factor. Here are some approaches and algorithms that are generally more scalable:

1. **MiniBatch K-Means:** This is a variation of the K-Means algorithm designed to be more scalable. It uses small, random batches of data to update the cluster centers, making it suitable for large datasets.

**End of course  
assignment**

# Tasks definition

## Travel agency chatbot

- RAG with FAQs
- Check trips availability from csv
- Basic memory
- Guardrails against toxic behaviour

### DATA

- 500 FAQs
- Available trips csv

## Customer Satisfaction processing bot

- Extract all mentioned locations with NER
- LLM classification and summary based on customizable prompt
- Decide whether to assign discount or recommend next trip

### DATA

- 1000 customer reviews for hotels with true sentiment labels (data\final\_assignment\customer\_surveys\_hotels\_1k.json)\*
- Available trips csv

\* Subsampled and balanced data from <https://huggingface.co/datasets/argilla/tripadvisor-hotel-reviews>

# Travel agency chat bot hints

## Expected features

### Base

- Answer questions about company travel policies passed on the FAQ files
- Working memory -> being able to use contents from previous messages in answers

### Extra

- Streamlit app for easier interaction
- Ability to fetch trips details from a file based on its id
- Guardrails against toxic behaviors and jailbreaking

## Most important steps

### Base

- Vectorize FAQs based on `W2-llm-calling-and-vector-search`
- Prepare RAG tool based on previous notebook and `W3-agent\_with\_tools`
- Implement memory stored in json file based on `W3\_chat\_with\_memory`

### Extra

- Retool streamlit app from `W3-chat\_app` to show this chat
- Connect tool from and `W3-agent\_with\_tools` for interactions with csv
- Read about jailbreaking and toxicity filters, try preventing them with system prompt instructions

# Customer Satisfaction processing bot hints

## Expected features

### Base

- Fetch Customer Satisfaction records from folder and classify sentiment
- Send a customized discount for negative reviews
- Recommend 3 best trips for positive reviews
- Save output to a csv file

### Extra

- Use NER to extract all mentioned locations and save them to csv
- Compare classification metrics vs ground truth data

## Most important steps

### Base

- Retool chain from `W3-llm-flows-and-monitoring` to work with new data and instructions
- Find a solution to find best trips related to NER -> can be purely RAG, possibly expanded by filtering by metadata e.g. location

### Extra

- Leverage model from `W2-NER-Intro` to implement location processing into pipeline
- Calculate accuracy, recall and precision of sentiment analysis and visualize results

# **Customer Satisfaction processing Hints**

- Extract all mentioned locations with NER
- LLM classification and summary based on customizable prompt
- Decide whether to assign discount or recommend next trip

## DATA

- 500 customer reviews with golden labels
- Available trips csv