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 challengs 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 tchem to multiple calls

Split complex task to multiple flows → **More specific goal and instructions** → **Clear LLM attention focused** → **Ability to route downstream tasks**

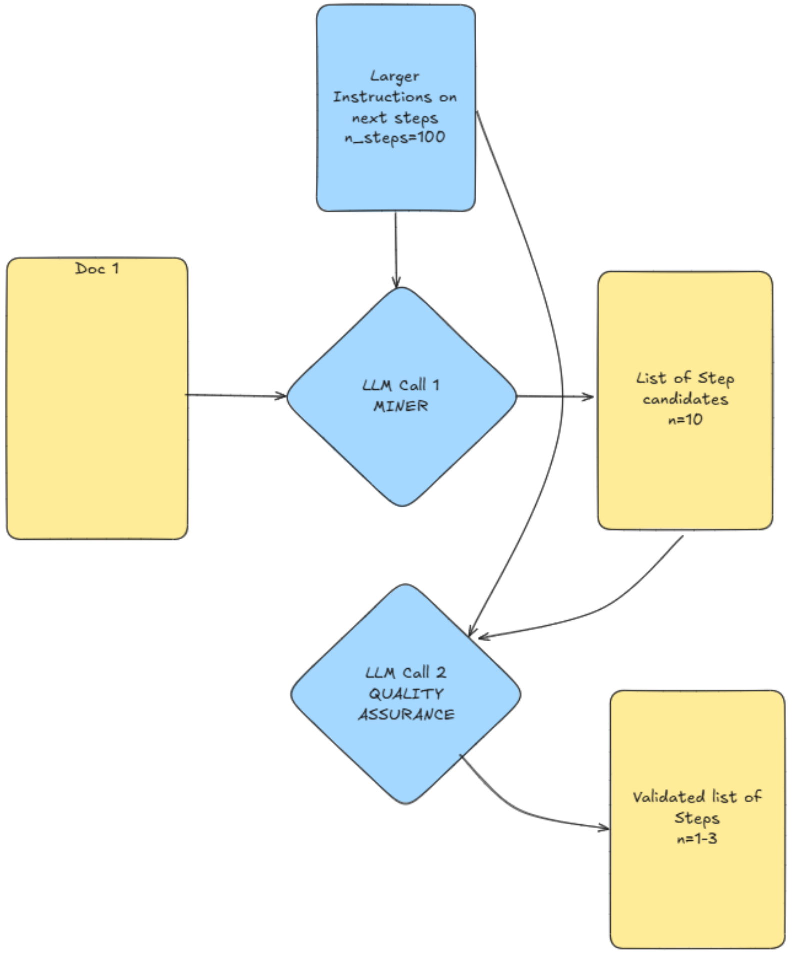
- Multiple calls can get job done with smaller models
- Performing multiple tasks e.g. mining, validation and scoring at the same time leads to worst results
- More tasks -> more prompt -> dillution of attention, which can lead to worse results

- 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 programtaic 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

```python
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
```

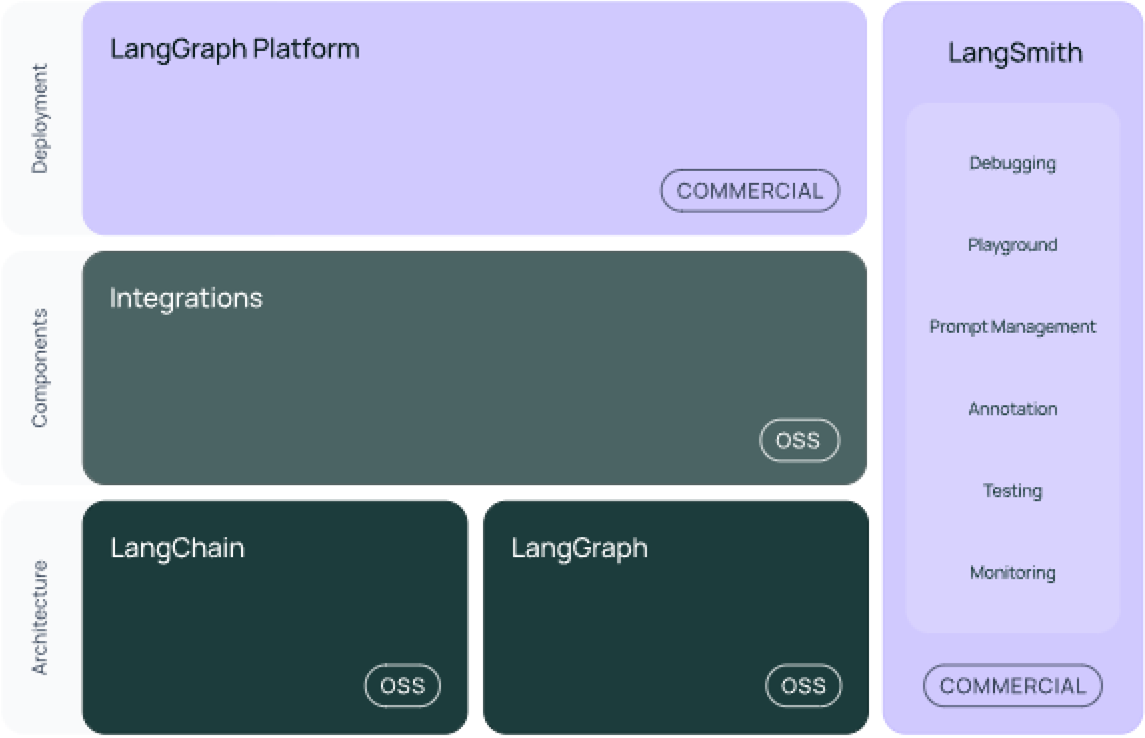
Validating / Modyfing outputs programatically between calls can make systems more predictable

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



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 consists 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

Runable Passhtorugh.asisgn fetches output of one sub-chain to be fed to next steps

```
# Create the branching  ith 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: { essage": f"Error: Unable to determine sentiment for: {x['review']}"}
    )
)

negative_messag  hain.invoke({"review": negative_test})
```

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

Introduction to Langchain

Start in W3-llm-flows-and-monitoring

```python
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

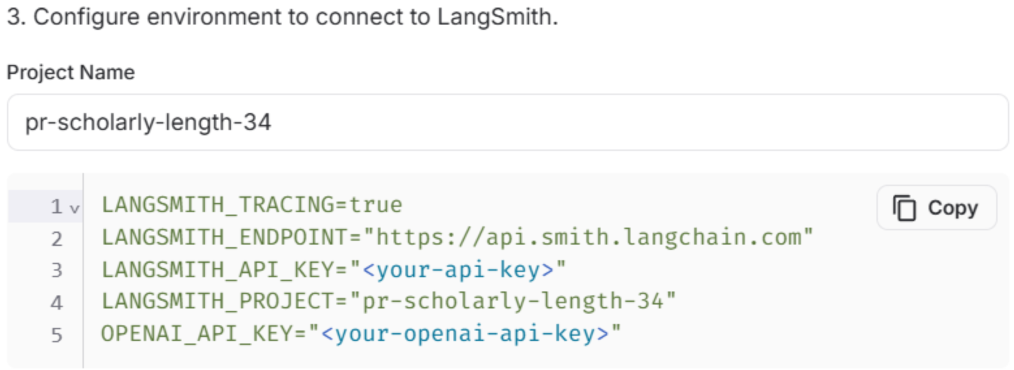
```python
from langchain_core.output_parsers import JsonOutputParser

# 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
```

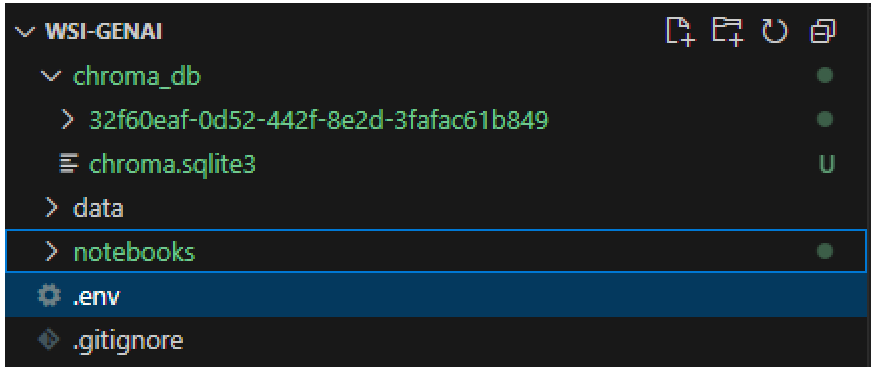
[56]

LLM Flow Monitoring

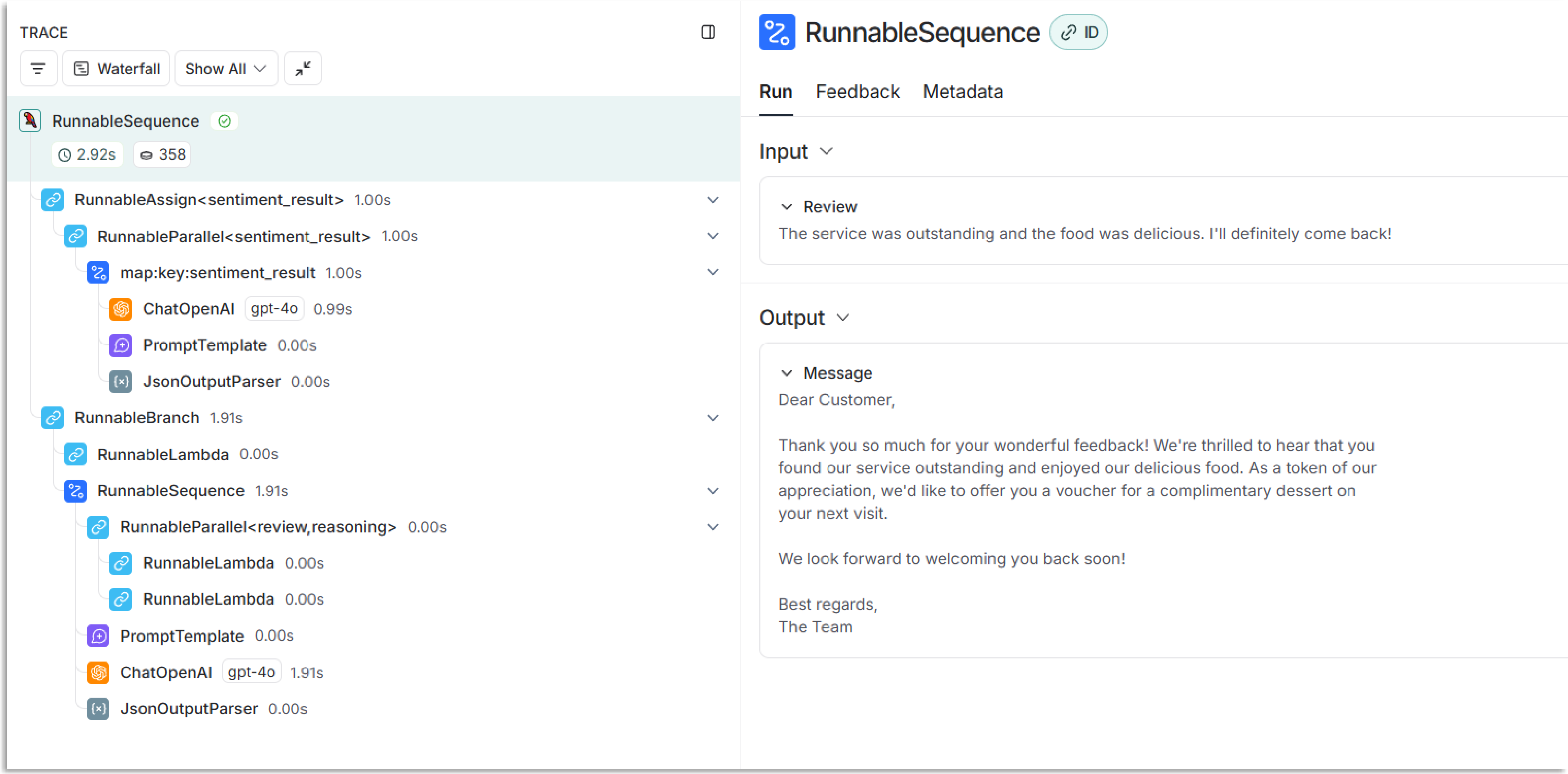
Setting up Langsmith

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 tchem
8. Change project to something meaningful





How to navigate logs for full LLM observability



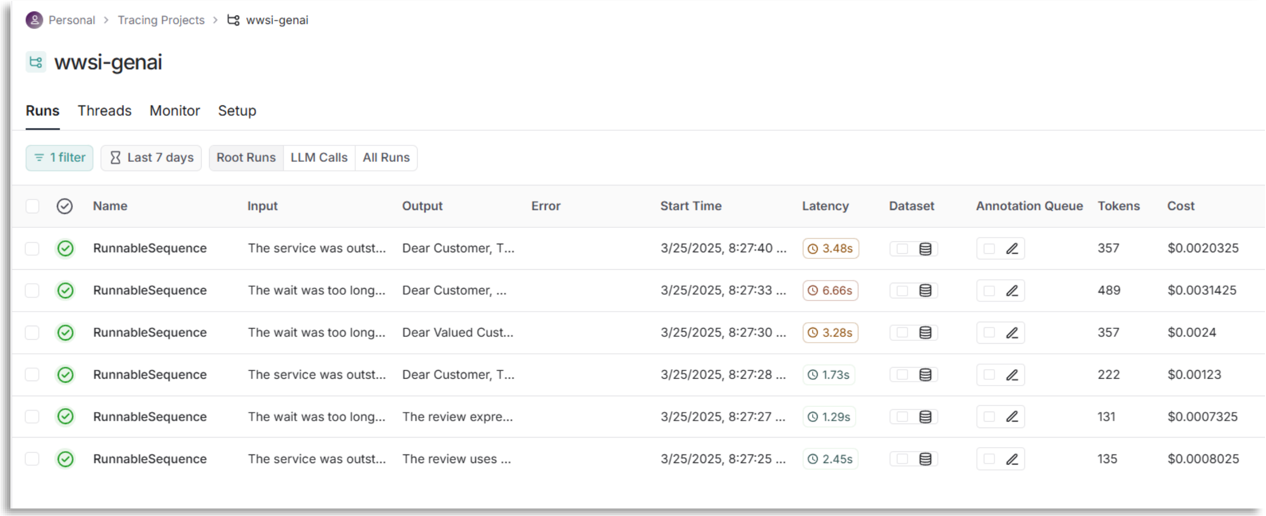
Debugging through logs

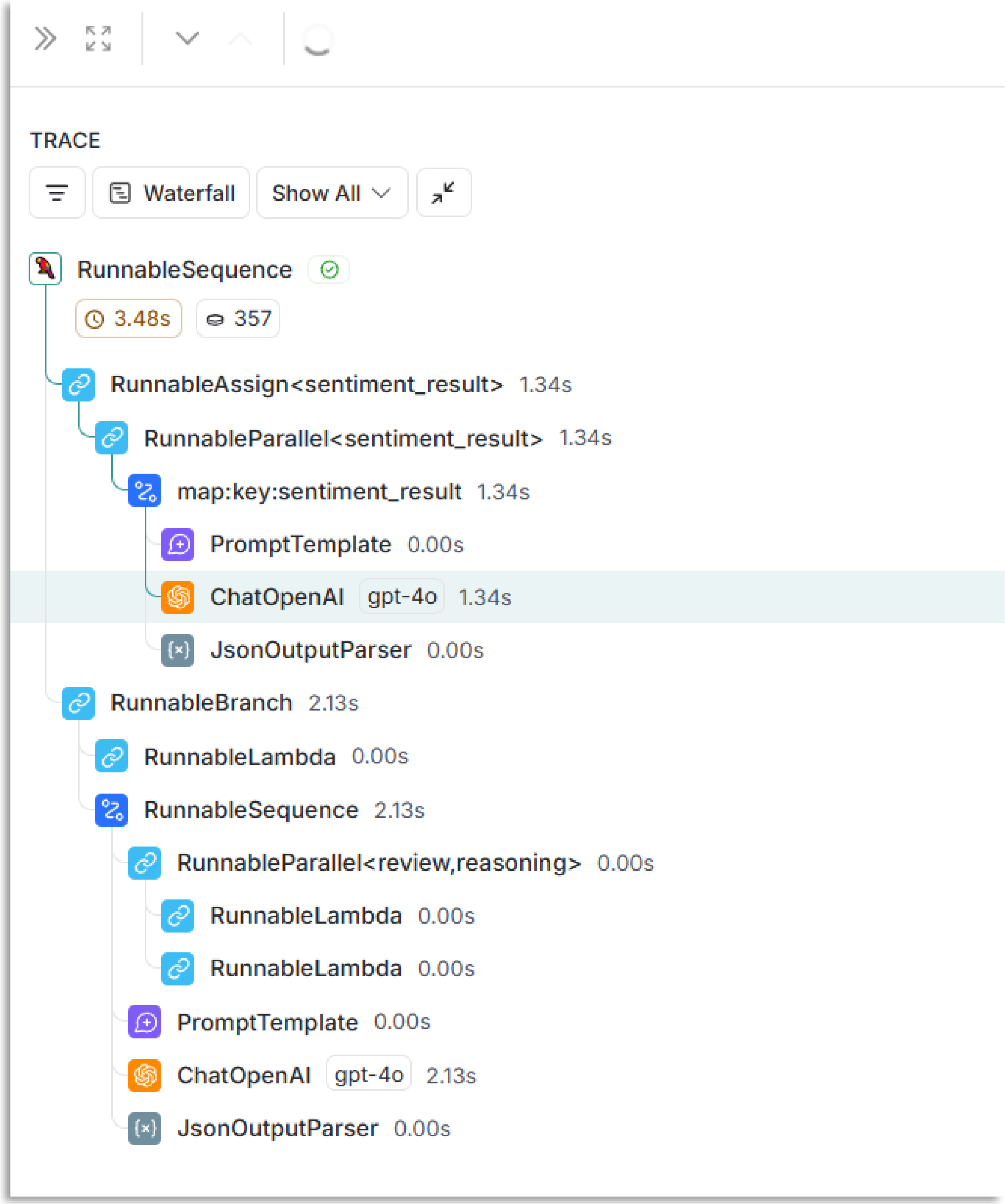
- As LLM flows become more complex and they take multiple, non deterministic inputs (e.g. from previous steps) clear logging is crucial to understand their mistakes
- Due to non-deterministic nature is is often easier to debug logs than try to replicate all states
- Minor mistakes such as incorrect setup 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





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

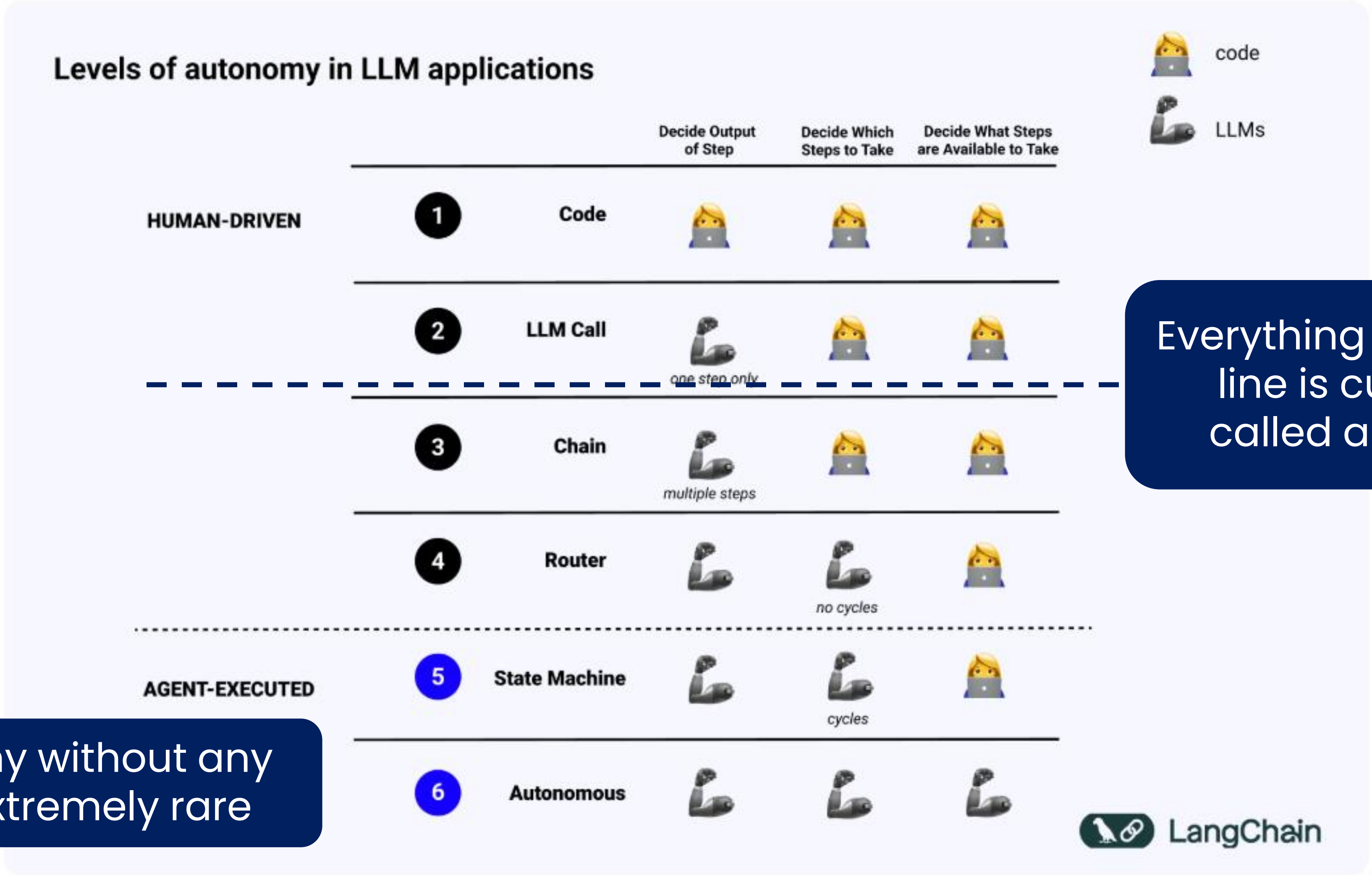
•**Tool Integration**: Access to APIs, databases, code execution, web browsing

•**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



Everything below this line is currently called an agent

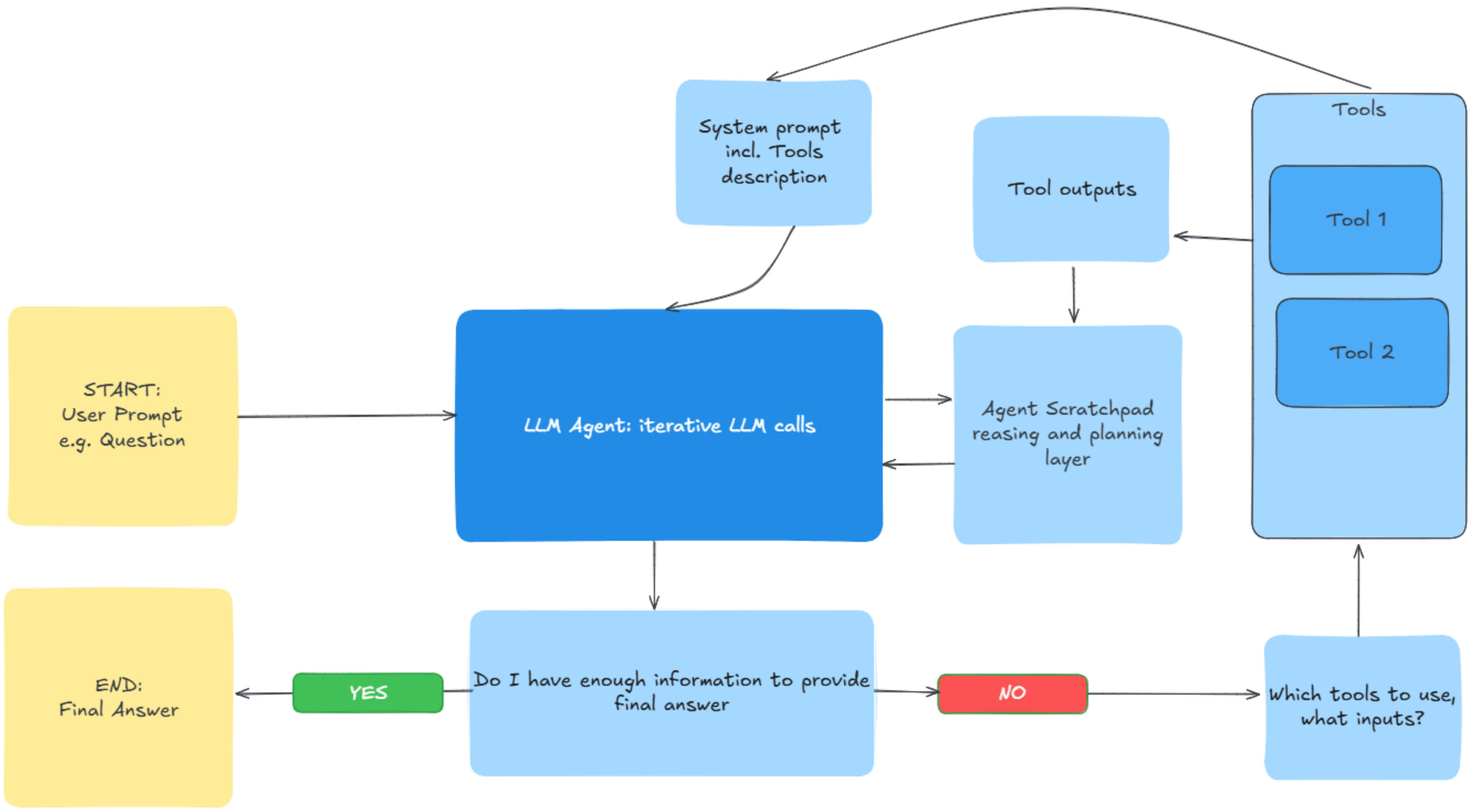
Level 6 Autonomy without any control is still extremely rare

Empowering LLMs to take actions with tools

Why do agents need tools?

- LLMs are more powerfull 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

TOOLS:

Assistant has access to the following tools:

save_reservation(planned_trip_date, trip_destination, description) - Use this tool to save a new trip reservation.
  Input arguments:
  - `planned_trip_date`: The date of the planned trip (YYYY-MM-DD format).
  - `trip_destination`: Destination of the trip.
  - `description`: Additional details about the trip.
  Tool output: A confirmation message with the reservation ID.
read_reservation(reservation_id) - Use this tool to look up a reservation by its ID.
  Input arguments:
  - `reservation_id`: The unique ID of the reservation to look up.
  Tool output: Details of the reservation or an error message if not found.

LLM sees available tools in prompt together with usage guideliness

Output

AI

save_reservation call_0HuE2vTmzOAhv4dfbwmIeQ3A


```yaml
description: Business trip for 2 people, 3 nights stay.
planned_trip_date: 2023-12-25
```


YAML

LLM decides to use tool -> 1st output is just a tool trigger

Call #2

TOOL

save_reservation call_0HuE2vTmzOAhv4dfbwmIeQ3A

"Reservation created successfully! Your

YAML RAW

1st call output is fed to tool, triggering an execution and a tool output

Output

AI

Your trip to Paris, France on 2023-12-25 has been successfully booked. It's a business trip for 2 people with a 3-night stay. Your reservation ID is **cbd4eced**. If you need any further assistance, feel free to ask!

Tool output is fed to 2nd LLM call to power final response, or trigerring another iteration

Jupyter exercise Reranking

Start with W3-agent_with_tools_BLANK.py

```
Pay attention if user is asking about sale or rent offers.

If you gave the user some recommendations in previous messages and he agrees with them use those recommendatio
When analyzing tool output, compare it with Human question, if it only partially answered it explain it to the

To use a tool, please use the following format:\n\n```\n
Thought: Do I need to use a tool? Yes\n
Action: the action to take, should be one of [{tool_names}]\n
Action Input: the input to the action\n
Observation: the result of the action\n

... (repeat Thought/Action/Observation as needed)
Final Answer: The response to the user including relevant information from the tools

Begin!

New input:""",
        ),
        ("human", "{input}"),
        MessagesPlaceholder(variable_name="agent_scratchpad"),
    ]
)




prompt = prompt.partial(tools=render_text_description(tools), tool_names=", ".join([t.name for t in tools]))
llm_with_tools = llm.bind(tools=[convert_to_openai_tool(tool) for tool in tools])



agent = (
        RunnablePassthrough.assign(
            agent_scratchpad=lambda x: format_to_openai_tool_messages(x["intermediate_steps"]),

            )
        | prompt
        | llm_with_tools
        | OpenAIToolsAgentOutputParser())

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=False, handle_parsing_errors=False)


def run_agent_with_query(query):
    return agent_executor.invoke({"input": query})

if __name__=="__main__":
    query = "I want to book a trip on 2023-12-25 to Paris, France. 2 people for 3 nights. Its a business trip"
    output = run_agent_with_query(query)
    breakpoint()

    query_2 = "What is the status of reservation 9c89a904?"
    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×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...	✓ Use MCP 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	• 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

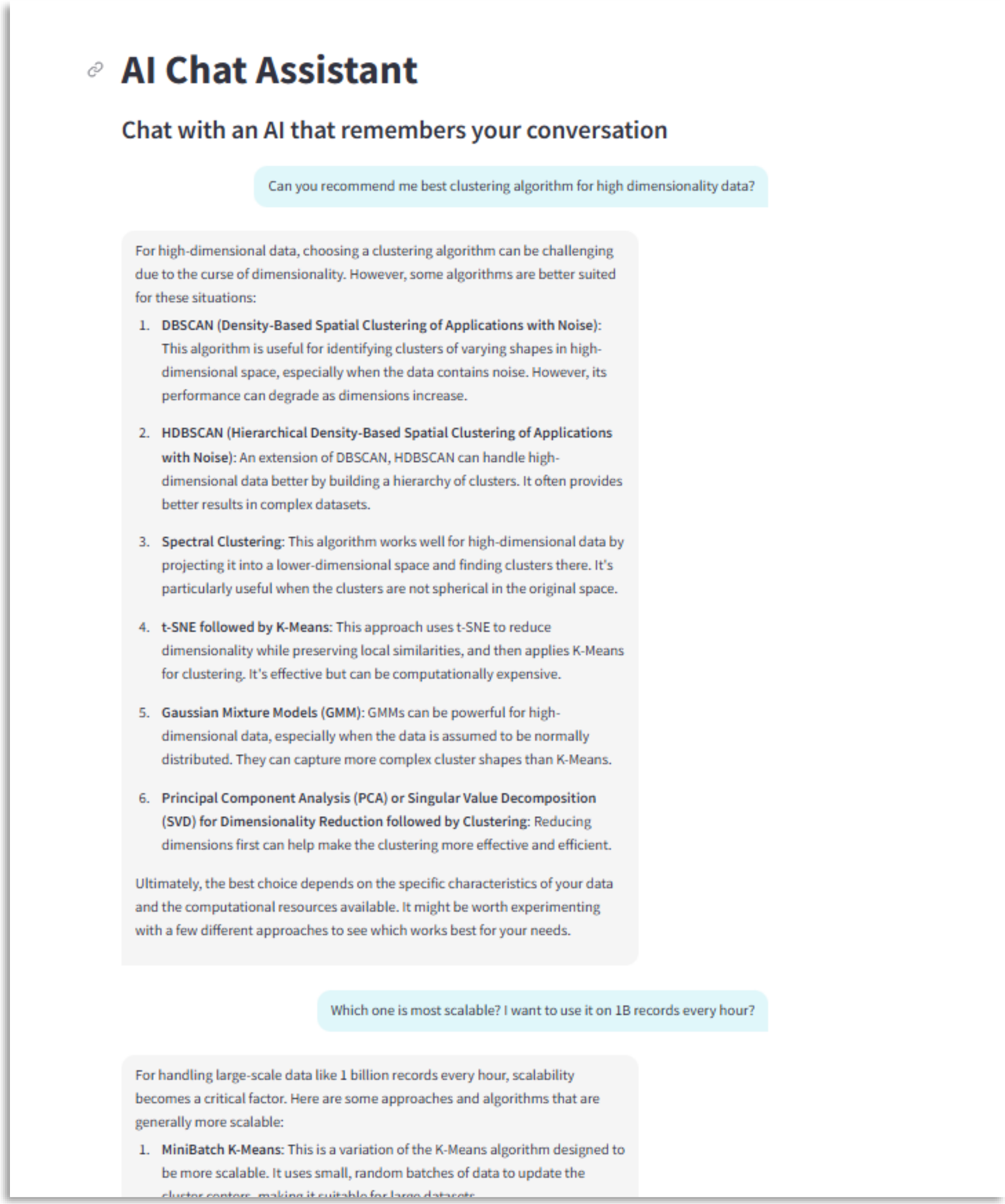
Jupyter exercise Reranking

Start with W3_chat_with_memory_BLANK.py

```python
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}")
```

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
- Fortunatelly 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



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
- 500 customer reviews with true sentiment labels
- Available trips csv

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 behavious 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