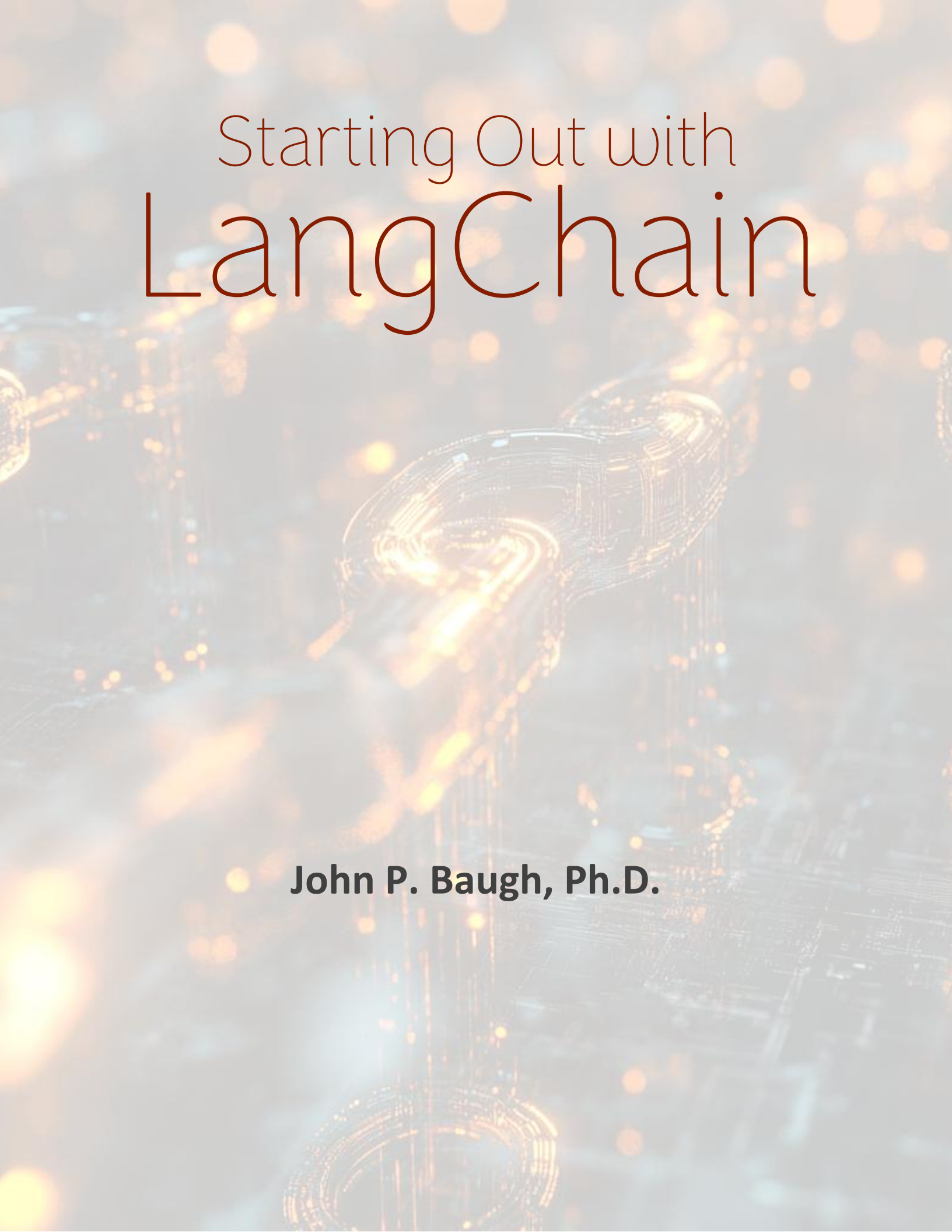


Starting Out with LangChain

John P. Baugh, Ph.D.

Starting Out with LangChain

John P. Baugh, Ph.D.

v. 2025.5.3.1

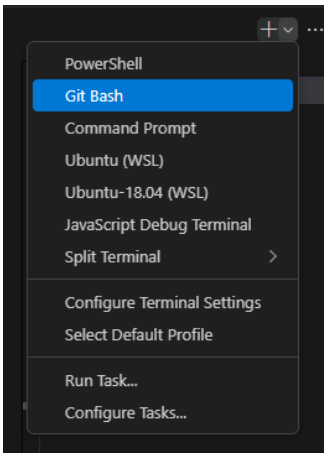
Version	Date	Notes
2025.5.3.1	5/3/2025	Initial release

Table of Contents

Setup

The examples we'll use contain instructions that assume you are using VS Code, but should be easily adaptable to other environments with which you are familiar.

1. Create a folder for the tutorials
    a. E.g., **langchain-tutorials**
2. Open that folder from VS Code
    a. File → Open Folder
    b. Find the **langchain-tutorials** folder
    c. Select it/Open it
3. Open a Terminal window
    a. Terminal → new Terminal (in VS Code)
    b. Change the shell type to a Unix-style shell, such as Bash



4. Create and activate a virtual environment
    a. E.g., in the Terminal, type:

```
python -m venv venv
```

5. Activate the virtual environment

Type:

```
source venv\Scripts\activate
```

Windows

Or

```
source venv/bin/activate
```

macOS or Linux

6. Install required packages

```
pip install langchain langchain-community langchain-core langchain-ollama faiss-cpu
```

7. Install Ollama if not already installed
    a. Go to https://ollama.com/download
    b. Download the installer
    c. Install it
8. Test to make sure it is running
    a. Go into the VS Code Terminal again

```
ollama list
```

That will show installed models. If you do not see llama3 installed, you should pull it down using:

```
ollama pull llama3
```

9. If you see everything working well, you may want to restart VS Code just to make sure everything is properly registered and configured before continuing

Tutorial 1: Hello LangChain

Goal

Make your first LangChain + LLaMA3 script work from scratch.

About the Technologies

LangChain is a framework that helps you build applications powered by language models (LLMs) using modular components like prompts, chains, memory, tools, and agents. This tutorial introduces the simplest LangChain pipeline, using:

- PromptTemplate to inject a topic into a prompt
- ChatOllama to run a local LLM (LLaMA 3) through Ollama
- StrOutputParser to parse the LLM's raw response as a plain string

This pipeline gives you a clear mental model:

Prompt → LLM → Parser → Output

It's a great starting point because it's fast, local, and does not require an OpenAI key or cloud access.

Instructions

1. Within VS Code, create a new file, **hello_langchain.py**
2. Write the following code:

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama import ChatOllama

# 1. Create a dynamic prompt with a variable
prompt = PromptTemplate.from_template("Tell me a fun fact about {topic}")

# 2. Use local LLaMA3 through Ollama
llm = ChatOllama(model="llama3")

# 3. Parse the output as plain text
output_parser = StrOutputParser()

# 4. Chain the prompt → model → output parser
chain = prompt | llm | output_parser

# 5. Run the chain with an input
result = chain.invoke({"topic": "octopuses"})
print(result)
```

3. Open a Terminal window
    a. Terminal → New Terminal
    b. Set the type to something like Git Bash or another Unix-based CLI
4. Run the application

```
python hello_langchain.py
```

You should get a fun fact about octopuses[1].

Additional Notes and Explanations

The `PromptTemplate.from_template("Tell me a fun fact about {topic}")` creates a reusable prompt with a `{topic}` placeholder. By invoking the chain with `{"topic": "octopuses"}`, LangChain fills in the prompt dynamically. The | symbol (which is borrowed from the Unix-style **pipe** syntax) chains the components together cleanly using LangChain's newer "runnable" syntax.

[1] The plural of **octopus** is **octopuses** in English. While some people say "octopi", this is trying to use a Latin pluralization on a Greek word, and is therefore, incorrect. Using Greek plural would be **octopodes**, but octopuses is the most common and accepted form in English.

Tutorial 2: Prompt Templates with Variables

Goal

Inject multiple variables into a prompt for custom storytelling.

About the Technologies

This tutorial builds on the previous one by showing how *multiple variables* can be inserted into a prompt. This is useful for storytelling, dialogue generation, simulations, and more.

The key technology here is again `PromptTemplate`, but now it uses two variables (`{name}` and `{hobby}`), showing LangChain's support for multi-variable templates. The pipeline is identical in structure — just with more contextual inputs.

This reflects real-world use cases like:

- Filling in user-specific information in a chatbot
- Generating product descriptions, summaries, or custom emails

Instructions

1. Within VS Code, create a new file, **prompt_template_chain.py**
2. Write the following code:

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama import ChatOllama

prompt = PromptTemplate.from_template(
    "Write a short story about a person named {name} who loves {hobby}."
)

llm = ChatOllama(model="llama3")
parser = StrOutputParser()
chain = prompt | llm | parser

result = chain.invoke({"name": "Lena", "hobby": "skydiving"})
print(result)
```

3. In the Terminal, run it

```
python prompt_template_chain.py
```

You should get a short story about Lena, who loves to skydive.

Additional Notes and Explanations

When you call `chain.invoke({"name": "Lena", "hobby": "skydiving"})`, LangChain auto-fills the prompt to:

```
Write a short story about a person named Lena who loves skydiving.
```

That prompt is passed to the LLM, and the result is parsed and printed. This tutorial reinforces how LangChain decouples prompt content from prompt formatting logic.

Tutorial 3: Structured Output with LLaMA

Goal

Ask LLaMA to return a bullet list or formatted structure.

About the Technologies

While LLMs are known for free-form responses, this tutorial shows how to guide the model into returning structured output — in this case, a bulleted list. This is essential when building apps that rely on predictable formatting, such as:

- Assistants
- Data extraction pipelines
- Multi-part reasoning chains

The structure is enforced via the prompt, not by parsing or formatting the output in code. This is still using `StrOutputParser`, which expects readable strings. Later tutorials may use `JsonOutputParser` for machine-readable output.

Instructions

1. Create a new file, **structured_output_llama.py**
2. Write the following code:

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama import ChatOllama

prompt = PromptTemplate.from_template(
    """
    Generate a structured list of three interesting facts about {topic}.

    Format the response like this:
    - Fact 1: ...
    - Fact 2: ...
    - Fact 3: ...
    """
)

llm = ChatOllama(model="llama3")
parser = StrOutputParser()
chain = prompt | llm | parser

result = chain.invoke({"topic": "space travel"})
print(result)
```

3. Run it

```
python structured_output_llama.py
```

Additional Notes and Explanations

The triple-quoted string allows you to write a *well-formatted instruction prompt*, telling the model exactly how to reply (with three facts, in bullet format). This is a soft form of structure — you're instructing the model with good prompt engineering rather than enforcing it programmatically.

Tutorial 4: Multi-Turn Chat with a Loop

Goal

Introduce a basic loop that lets users chat with a local LLM repeatedly, simulating multi-turn conversation (but without memory yet).

About the Technologies

This tutorial introduces how to build a simple, text-based chat interface with LangChain and a local LLaMA model via Ollama.

You'll build a while-loop that accepts user input continuously, sends it to the model, and prints out the result — like a chatbot, This version does not use memory, so it behaves statelessly: each user message is processed without regard to prior messages.

Instructions

1. Create a new file, **multi_turn_chat_loop.py**
2. Write the following code:

```
from langchain_ollama import ChatOllama

llm = ChatOllama(model="llama3")

print("Chat with LLaMA (type 'exit' to quit):")
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    response = llm.invoke(user_input)
    print("LLaMA:", response.content)
```

3. Run the program

```
python multi_turn_chat_loop.py
```

Test it with questions of your choosing, and then type **exit** to exit the program.

- How many cats are there worldwide?
- What's the most popular food in the United States?
- Tell me a Chuck Norris joke.

Additional Notes and Explanations

This is the most basic multi-turn loop.

There's no memory or prompt formatting — just a model call on each user input.

You can extend this by saving previous inputs and re-feeding them to the model, or adding LangChain memory (next tutorial).

Tutorial 5: Memory with LangChain

Goal

To build a local conversational assistant that remembers previous messages across turns using `RunnableWithMessageHistory` and `InMemoryChatMessageHistory`. This allows for realistic, multi-turn dialog with context retention.

About the Technologies

LangChain now uses a more modular and declarative memory system:

- **`ChatPromptTemplate`**: Builds a conversation prompt using message roles (system, human, etc.).
- **`InMemoryChatMessageHistory`**: Stores messages from the conversation as chat messages (not raw strings), allowing re-insertion into the next prompt.
- **`RunnableWithMessageHistory`**: Wraps any LangChain chain or pipeline and injects message history automatically based on session ID.
- **`ChatOllama`**: LangChain wrapper around the Ollama LLaMA3 model for local inference.

This architecture is scalable, reusable, and avoids deprecated APIs (e.g., `ConversationBufferMemory`).

Instructions

1. Create a new file, **memory_with_langchain.py**
2. Write the following code:

```
from langchain_ollama import ChatOllama
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

# Step 1: Load local LLaMA model
llm = ChatOllama(model="llama3", temperature=0)

# Step 2: Prompt with placeholder for chat history
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    MessagesPlaceholder(variable_name="chat_history"),
    ("human", "{input}")
])

# Step 3: Chain the prompt and model
base_chain = prompt | llm

# Step 4: Session-based message history manager
session_store = {}
```

```
def get_session_history(session_id: str):
    if session_id not in session_store:
        session_store[session_id] = InMemoryChatMessageHistory()
    return session_store[session_id]

# Step 5: Wrap the base chain with memory logic
chat_chain = RunnableWithMessageHistory(
    base_chain,
    get_session_history,
    input_messages_key="input",
    history_messages_key="chat_history"
)

# Step 6: Simple terminal loop
print("Chat with LLaMA (with memory). Type 'exit' to quit.")

while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break

    result = chat_chain.invoke(
        {"input": user_input},
        config={"configurable": {"session_id": "local_user"}}
    )

    print("LLaMA:", result.content)
```

3. Run it

```
python memory_with_langchain.py
```

Try it with something like the following (You type the stuff next to “You:” and wait on a response):

```
You:  My name is John
LLaMA:  ...
You:   Tell me a funny joke about dogs
LLaMA:  ...
You:   What’s my name again?
LLaMA:  ...
```

You should verify that it remembers your name, demonstrating that the memory is working.

Additional Notes and Explanations

Component	Explanation
`InMemoryChatMessageHistory()`	Stores messages in memory for a specific session

`RunnableWithMessageHistory`	Handles injecting message history into `{chat_history}` placeholder
`session_id="local_user"`	Controls which session's memory is used
`MessagesPlaceholder`	Ensures memory is inserted into the right spot in the prompt

Tutorial 6: Manual Tool Execution (ReAct style)

Goal

Use the **ReAct** (**Reasoning + Acting**) framework, where the model selects tools based on the prompt and you execute them via code.

About the Technologies

This tutorial introduces manual tool usage where the model is guided to think step-by-step and call a tool like calculator[...] based on its reasoning.
You'll parse the model's response using regex and invoke matching Python functions yourself. This shows how to integrate LLM reasoning with real-world actions.

Instructions

1. Create a new file, **manual_tool_execution.py**
2. Write the following code:

```
import re
from langchain_ollama import ChatOllama

def calculator(expression: str) -> str:
    try:
        return str(eval(expression))
    except Exception as e:
        return f"Error: {e}"

llm = ChatOllama(model="llama3")

# Prompt style for ReAct
prompt = """You are a helpful assistant who uses tools.

Available tool: calculator[expression]

Respond using this format:
Thought: ...
Action: calculator[2 + 2]
Observation: ...
Final Answer: ...
"""

user_input = input("Ask a math question: ")
response = llm.invoke(prompt + f"Question: {user_input}")
response_text = response.content

print("--- Raw Model Response ---")
print(response_text)
```

```
# Extract tool usage from response
match = re.search(r"Action: calculator\[(.*?)\]", response_text)
if match:
    expression = match.group(1)
    result = calculator(expression)
    print("--- Tool Execution ---")
    print(f"calculator({expression}) = {result}")
else:
    print("No valid tool call found.")
```

3. Run it

```
python manual_tool_execution.py
```

Additional Notes and Explanations

This shows early-stage tool use where the LLM suggests what tool to use but doesn't call it directly.
You extract the tool + input manually and run the function in Python.
This prepares you for full agent-style chaining and agent frameworks later.

Recap: Tutorials 1-6

Summary of Tutorials

Tutorial #	Title	Key Concepts Covered	Skills Gained
1	Hello LangChain	PromptTemplate, ChatOllama, StrOutputParser	Create your first LLM pipeline with a dynamic prompt
2	Prompt Templates with Variables	Multiple input variables in a prompt	Build and invoke reusable prompts for storytelling or templated outputs
3	Structured Output with LLaMA	Prompt formatting, bullet lists	Control the structure of model outputs through instruction
4	Multi-Turn Chat Loop	Basic loop, stateless LLM interaction	Use input() to drive repeated LLM calls and build a chatbot-like experience
5	Memory with LangChain	RunnableWithMessageHistory, InMemoryChatMessageHistory	Maintain conversation history across turns using LangChain memory
6	Manual Tool Execution (ReAct)	ReAct pattern (Thought → Action → Observation → Final Answer), re.search, tool functions	Parse and run tool calls based on model reasoning; simulate agent-like behavior

Summary of LangChain Classes and Components

Class / Component	Source	Purpose
`PromptTemplate`	`langchain.prompts`	Creates a single-string prompt with variables (e.g., `{topic}`) for use with text or chat models. Used in basic prompting.
`ChatPromptTemplate`	`langchain_core.prompts`	Builds prompts using structured chat messages (system, human, ai, `MessagesPlaceholder`).

		Used with chat models like LLaMA 3.
`MessagesPlaceholder`	`langchain_core.prompts`	Placeholder in a `ChatPromptTemplate` where prior messages (memory) are inserted automatically.
`StrOutputParser`	`langchain_core.output_parsers`	Parses a language model's raw output and returns it as a simple string. Useful for basic pipelines.
`ChatOllama`	`langchain_ollama`	LangChain wrapper for using Ollama-hosted local chat models (like llama3). Acts like a Runnable.
`ChatOpenAI (mentioned in notes)`	`langchain_openai`	Chat model wrapper for OpenAI (used when comparing cloud-based models).
`Runnable`	`langchain_core.runnables`	A base class for anything you can run or chain together. Prompts, models, and output parsers are all Runnable.
`RunnableWithMessageHistory`	`langchain_core.runnables.history`	Wraps any `Runnable` (like a chain) and injects memory into the prompt using a message history.
`InMemoryChatMessageHistory`	`langchain_core.chat_history`	Stores a list of messages (human/AI) in memory. Used for short-term conversational memory in session-based apps.
`invoke()`	`All Runnable subclasses`	Executes the prompt/model chain with a given dictionary of variables and returns the result.
`from_template(...)`	`PromptTemplate, ChatPromptTemplate`	Factory method to create a prompt from a string or list of message roles.

`input_messages_key / history_messages_key`	`Used in RunnableWithMessageHistory`	Specify which inputs in your data are the current message and which hold the historical chat.
`config={"configurable": {"session_id": "xyz"}}`	`In invoke()`	Allows you to pass a session ID for tracking and persisting memory per user/session.
`re.search(...)`	`Standard Python re module`	Used to extract tool name and arguments from LLM output when manually parsing ReAct-style outputs. Not from LangChain, but crucial in Tutorial 6.

Tutorial 7: Tool Use with a Custom AgentCore Class

Goal

Refactor your tool-driven LLM logic into a clean, reusable AgentCore class stored in its own file. Then use a separate script to define tools and run the agent against user input.

This sets the foundation for scalable, multi-agent systems.

About the Technologies

We're applying **modular design principles** to the agent system. This is a software engineering step — not a new LangChain feature — but it's essential for scaling your framework.

You'll:

- Define a standalone AgentCore class in AgentCore.py
- Define tool logic and execute AgentCore from another script
- Improve maintainability and testability by separating concerns

Instructions

1. Create a new file, **AgentCore.py**
2. Write the following code:

```
import re
from langchain_ollama import ChatOllama

class AgentCore:
    def __init__(self, model, tools: dict[str, callable], description: str):
        self.llm = model
        self.tools = tools
        self.description = description

    def _extract_action(self, text: str):
        match = re.search(r"Action:\s*([\w_]+)\[(.*?)\]", text, re.DOTALL)
        if match:
            return match.group(1), match.group(2)
        return None, None

    def _execute_tool(self, tool_name: str, param: str):
        if tool_name not in self.tools:
            return (
                f"⚠️ Tool '{tool_name}' is not available.\n"
                f"Available tools: {', '.join(self.tools.keys())}"
            )
        try:
            return self.tools[tool_name](param)
```

```
        except Exception as e:
            return f"⚠️ Tool execution failed: {e}"

    def run(self, user_input: str):
        prompt = f"{self.description}\n\nQuestion: {user_input}"
        response = self.llm.invoke(prompt).content

        print("--- Raw Model Response ---")
        print(response)

        tool_name, param = self._extract_action(response)
        if tool_name:
            result = self._execute_tool(tool_name, param)
            print(f"\n--- Tool Used ---\n{tool_name}({param}) = {result}")
        else:
            print("No valid tool action detected.")
```

3. Create a new file, **agent_test_runner.py**
4. Write the following code:

```
from langchain_ollama import ChatOllama
from AgentCore import AgentCore

# Define tools
def calculator(expr: str) -> str:
    try:
        return str(eval(expr))
    except Exception as e:
        return f"Invalid expression: {e}"

def facts(topic: str) -> str:
    return f"Fun fact about {topic}: It is surprisingly interesting!"

tools = {
    "calculator": calculator,
    "facts": facts
}

# Agent behavior description (ReAct-style)
description = """
You are a helpful assistant who uses tools.

Available tools:
- calculator[expression]
- facts[topic]

Use the following format:
Thought: ...
Action: tool_name[parameters]
Observation: ...
Final Answer: ...
```

```
"""

# Load local LLaMA model
llm = ChatOllama(model="llama3", temperature=0)

# Create the agent
agent = AgentCore(model=llm, tools=tools, description=description)

# User interaction loop
print("Ask me something (type 'exit' to quit):")
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break
    agent.run(user_input)
```

5. Run it

```
python agent_test_runner.py
```

Try it with some of the following (wait for response after hitting Enter/Return after each one):

```
You: What is 25 * (4 + 1)?
You: Tell me a fact about octopuses.
You: Do you have a weather tool?
```

Additional Notes and Explanations

- **Tool functions** are plain Python functions mapped by name in the tools dictionary
- **Regex pattern** extracts the tool name and parameter safely from the model's response
- **run()** handles reasoning, execution, and printing results
- This structure makes it easy to:
    - Add new tools
    - Use the agent in different environments (CLI, web app, etc.)
    - Extend to multiple agents (next tutorial)

Tutorial 8: Multi-Agent Routing with Orchestrator

Goal

Route user questions to the correct specialized agent (`math` or `info`) using a simple keyword-based orchestrator. This allows agents to have different toolsets and behaviors.

About the Technologies

You now have multiple `AgentCore` instances — one for math, one for facts. The `Orchestrator` decides which one to use based on keywords.

While this routing is simple, it mirrors the architecture of larger agentic systems where routing is later powered by classifiers, embeddings, or flow engines like LangGraph.

Instructions

1. Create a new file, **Orchestrator.py**
2. Write the following code:

```
class Orchestrator:
    def __init__(self, agents: dict[str, object]):
        self.agents = agents

    def route(self, user_input: str) -> str:
        lowered = user_input.lower()
        if any(word in lowered for word in ["math", "calculate", "+", "-", "*", "/"]):
            return "math"
        else:
            return "info"

    def run(self, user_input: str):
        agent_name = self.route(user_input)
        print(f"[Orchestrator] Routed to agent: {agent_name}")
        agent = self.agents.get(agent_name)
        if agent:
            agent.run(user_input)
        else:
            print("No agent found for the request.")
```

3. Create a new file, **multi_agent_runner.py**
4. Write the following code:

```
from langchain_ollama import ChatOllama
from AgentCore import AgentCore
from Orchestrator import Orchestrator

# Define tools
def calculator(expr: str) -> str:
    try:
```

```
        return str(eval(expr))
    except Exception as e:
        return f"Error: {e}"

def facts(topic: str) -> str:
    return f"Here's a fun fact about {topic}!"

# Descriptions
math_description = """
You are a specialized math agent.

You MUST use the following tool:
- calculator[expression]

Use this EXACT format and syntax:

Thought: <your reasoning>
Action: calculator[expression]
Observation: <result of tool>
Final Answer: <conclusion>

DO NOT write "calculator expression".
DO NOT skip the brackets [].
DO NOT use quotes.
DO NOT omit the Action step.

If you fail to follow this format, your answer will not be accepted.
"""


info_description = """
You are an info agent.

You MUST use one of these tools to answer:
- facts[topic]

Respond ONLY using this exact ReAct format:

Thought: <your internal reasoning>
Action: facts[topic]
Observation: <result of running the tool>
Final Answer: <your complete answer>

Do NOT make up new tool names.
Do NOT add quotes around the parameter.
Do NOT skip the Action step.
Do NOT say anything outside of the ReAct format.
"""

# Create models and agents
llm = ChatOllama(model="llama3", temperature=0)
math_agent = AgentCore(llm, {"calculator": calculator}, math_description)
```

```
info_agent = AgentCore(llm, {"facts": facts}, info_description)

# Orchestrator
router = Orchestrator({
    "math": math_agent,
    "info": info_agent
})

# Interactive loop
print("Ask something (type 'exit' to quit):")
while True:
    user_input = input("You: ")
    if user_input.lower() == "exit":
        break
    router.run(user_input)
```

5. Modify the **_extract_action** method within the **AgentCore** class:

```
  def _extract_action(self, text: str):
      match = re.search(r"Action:\s*[`\"]?([\w_]+)\[(.*?)\][`\"]?", text,
  re.DOTALL)
      if match:
          tool = match.group(1)
          raw_param = match.group(2)
          param = raw_param.strip().strip('"').strip("'")
          return tool, param
      return None, None
```

This cleans it up a bit to:

- Match both `tool[param]` and quoted inputs like `tool["param"]`
- Strip unnecessary quotes and whitespace from parameters

6. Run it

```
python multi_agent_runner.py
```

Additional Notes and Explanations

You've now created a **multi-agent environment**. This design makes it easy to add more agents later (e.g., RAG agents, API agents, planners). Routing logic can evolve as needed.

Tutorial 9: Add doc_search Tool (Retrieval via Embeddings + FAISS)

In this tutorial, we build directly on Tutorial 8. Make sure you've completed the multi-agent setup, as we'll be adding a new tool (`doc_search`) to the existing `info_agent`.

Goal

Create a `doc_search` tool that lets your agent search a real document (e.g. an employee handbook) using vector embeddings + similarity search with FAISS. You'll make this searchable using the nomic-embed-text model via Ollama.

About the Technologies

Component	Role
`TextLoader`	Reads and loads .txt documents
`RecursiveCharacterTextSplitter`	Breaks the doc into overlapping chunks
`OllamaEmbeddings`	Embeds text chunks using a local embedding model
`FAISS`	Fast, in-memory vector store for similarity search
`doc_search[query]`	A tool function added to the info agent

This is a complete **RAG (Retrieval-Augmented Generation)** loop — except you're doing it **locally** without any OpenAI API keys.

Something else we're going to continue doing is **prompt engineering** in order to ensure our model uses the correct agents, current tools, and to reduce the probability of hallucination[2].

What is RAG?

RAG stands for **Retrieval-Augmented Generation**. It's a technique that combines two major steps:

1. **Retrieve** relevant information from a knowledge source (like a document or database)
2. **Generate** a response using a language model that incorporates that retrieved context

This solves a common problem: *LLMs don't know your private data.*

Instead of training a custom model, RAG lets you *plug your own content* (e.g., employee handbooks, technical docs) into an AI assistant that uses that data *at inference time* — meaning it remains up-to-date and efficient.

What is an Embedding?

An **embedding** is a numerical representation of text — a vector — that captures its semantic meaning.

For example:

[2] A **hallucination** occurs when a language model generates a response that sounds plausible, but is factually incorrect, made up, or not grounded in the provided data or tools.

- "vacation policy" and "paid time off" may have very similar embeddings because they mean similar things
- Embeddings allow you to perform similarity searches: find chunks of text that “mean the same” as a user’s query, even if the words don’t exactly match

In LangChain, we use models like `nomic-embed-text` (via Ollama) to convert text chunks and queries into embeddings.

What is FAISS?

FAISS stands for **Facebook AI Similarity Search**. It’s a fast, in-memory search engine for *embedding vectors.* It is a *specific implementation* of something called a **vector store.**

A **vector store** stores the output of an **embedding model** — numeric vectors representing the meaning of each text chunk. When a user asks a question, it is embedded and compared to these stored vectors to find the most relevant information. This is how the RAG process “retrieves” meaningfully similar text, even if the words don’t match exactly.

FAISS:

- Stores many embeddings efficiently
- Quickly finds the top N vectors (text chunks) that are **most similar** to your query vector
- Is used under the hood in LangChain when you call vectorstore.similarity_search(...)

So in RAG:

- Embeddings represent the meaning
- FAISS retrieves similar meanings quickly
- LangChain uses both together to plug in external knowledge to the model

What is ReAct?

ReAct is short for **Reasoning + Acting**, a prompting strategy developed to help LLMs:

1. Think step-by-step (like CoT — Chain of Thought)
2. Use tools or take external actions
3. Reflect on results and build better responses

Standard format for ReAct:

```
Thought: <Why the agent is doing something>
Action: tool_name[parameter]
Observation: <What the tool returned>
Final Answer: <The conclusion to the original user input>
```

This structure is now considered best practice when designing tool-using agents — and it’s why we use it so explicitly in `info_description` and `math_description`.

So, is ReAct standard in LangChain?

Yes. In fact:

- LangChain's built-in agents use a variant of ReAct
- When you create agents with create_tool_calling_agent() or AgentExecutor, they often use prompts inspired by ReAct
- By spelling it out clearly in your agent description, you're **guiding the LLM to think and act step by step** — crucial for tool-based reasoning

For our use cases in the tutorial, using the explicit ReAct format helps:

- Structure tool reasoning predictably
- Parse the tool usage reliably (Action: tool[param])
- Separate internal thought from external action
- Train your users and LLM to behave consistently

It's also **much easier to debug**, as you can see each stage: Thought → Action → Observation → Final Answer.

In short:

- **ReAct** is the thinking framework
- **Embeddings** are the meaning representations
- **FAISS** (the **vector** store) is how we search them.

Together, they power **RAG** systems — allowing your agents to reason, fetch knowledge, and answer accurately.

Instructions

1. Install dependencies that will be needed
    a. Some of these may already be installed, but some may be needed still

```
pip install langchain langchain-community langchain-ollama langchain-text-
splitters faiss-cpu
```

OPTIONAL:

If you're on a GPU, you could also try:

pip install faiss-gpu

2. Pull the embedding model:

```
ollama pull nomic-embed-text
```

This is a lightweight embedding model optimized for fast local use.

3. Create a ***data file*** named **company_guide.txt**
    a. Place it in the same folder with the Python files
4. Copy and paste the contents into it:

```
Section 1:  Employee Benefits
Our company offers comprehensive health insurance including medical, dental,
and vision coverage.
```

```
Employees are eligible after 30 days of employment.
We also provide a 401(k) plan with a 4% employer match after 6 months.

Section 2: Vacation and PTO
Full-time employees accrue 15 days of paid vacation per year.
Unused PTO may roll over up to 5 days into the next calendar year.
Sick leave is tracked separately and accrues at 1 day per month.

Section 3: Remote Work Policy
Employees are allowed to work remotely up to 3 days per week with manager
approval.
   Fully remote positions are available in engineering, design, and support
   teams.
   Remote workers must attend mandatory quarterly on-site meetings.
```

5. Create a new file, **tools.py**
6. Write the following code:

```
from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_ollama import OllamaEmbeddings
from langchain_community.vectorstores import FAISS

# Load and split the guide
loader = TextLoader("company_guide.txt")
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50)
chunks = splitter.split_documents(docs)

# Embed using local Ollama embedding model
embeddings = OllamaEmbeddings(model="nomic-embed-text")
vectorstore = FAISS.from_documents(chunks, embeddings)

def doc_search(query: str) -> str:
    results = vectorstore.similarity_search(query, k=1)
    return results[0].page_content if results else "No relevant information
found."
```

7. Open **multi_agent_runner.py**
8. Import the new tool at the top

```
from tools import doc_search
```

9. Update the **info_agent** tools and description:

```
info_tools = {
    "facts": facts,
    "doc_search": doc_search
}
```

```
info_description = """
You are an information agent.

You have access to these tools:
- facts[topic] → Use this for general or public knowledge.
- doc_search[query] → Use this to look up employee and company-specific
policies, documents, and internal guides.

Only use one tool per question.

Follow this exact ReAct format:
Thought: ...
Action: tool_name[parameter]
Observation: ...
Final Answer: ...

NEVER use the facts tool for company-specific policies or internal documents.
NEVER invent tools.
Do NOT use quotes.
Do NOT skip the Action step.
Do NOT use parentheses for tools — use square brackets like tool_name[param].
Do NOT put brackets around the entire action.
The format is: Action: tool[param] — NOT Action: [tool param]

FORMAT EXAMPLES (copy this pattern exactly):

Action: doc_search[remote work policy]
Action: facts[octopuses]


If you fail to follow this format, your answer will not be accepted.
"""
```

10. Now, pass the updated tools into the agent:

```
info_agent = AgentCore(llm, info_tools, info_description)
```

11. Run it

```
python multi_agent_runner.py
```

Some sample input:

```
You: What's the remote work policy?
You: What benefits do new employees get?
You: Tell me about vacation policies
```

Expected behavior:

- Agent gets routed to `info`
- LLM calls `doc_search[remote work policy]`

- tools.py vectorstore returns the relevant paragraph from *company_guide.txt*
- Final Answer is based on the Observation

Additional Notes and Explanations

- You've added true knowledge grounding to your agent — this mimics how internal knowledge bases work.
- You're using embeddings + retrieval without any external services — perfect for private, local-first use cases.
- This pairs beautifully with the structured `AgentCore` system.
- When we modified the prompts (e.g., info_description, math_description), this is an example of **prompt engineering**.

Tutorial 10: Enhanced Tool Handling and Error Feedback

Goal

Improve the reliability and clarity of your agent's behavior by:

- Handling cases where tools are missing or misused
- Catching and displaying runtime errors from tool execution
- Giving actionable, developer-friendly error feedback
- Helping debug malformed or missing Action steps from the LLM

This makes your agent more robust, easier to troubleshoot, and safer in real-world usage.

About the Technologies

`AgentCore.run()`

This method manages the full cycle: it prompts the LLM, extracts the tool name and parameter, executes the tool, and handles the output. Enhancing this method gives us full control over the agent's reliability.

`_extract_action(text: str)`

This function uses a regular expression to extract tool usage like:

```
Action: tool_name[parameter]
```

Improving this function helps detect malformed Action: lines and gives users guidance on how to fix them.

`_execute_tool(tool_name: str, param: str)`

This method is where tool functions are called dynamically. Adding validation here allows us to:

- Detect missing tools
- Catch exceptions like `ZeroDivisionError`, eval misuse, etc.
- Print clear messages instead of crashing

Even if the LLM generates correct thoughts and tools *most of the time*, it's still an open-ended model and might:

- Misspell tool names
- Skip Action: altogether
- Format the action incorrectly
- Pass invalid parameters

Catching these issues in code improves trust and maintainability.

Instructions

The following will be inside **AgentCore.py**:

1. Improve the `_extract_action()` making it more robust

```
def _extract_action(self, text: str):
    import re
    # Match Action: tool[param] or fallback Action: [tool param]
    match = re.search(r"Action:\s*(?:([\w_]+)\[(.*?)\]|\[([\w_]+)\s+(.*?)\])",
                      text,
                      re.DOTALL)
    if match:
        tool = match.group(1) or match.group(3)
        raw_param = match.group(2) or match.group(4)
        param = raw_param.strip().strip('"').strip("'")
        return tool, param
    return None, None
```

2. Update the `_execute_tool()` **to add error handling and messaging**

```
def _execute_tool(self, tool_name: str, param: str):
    if tool_name not in self.tools:
        return (
            f"The tool '{tool_name}' is not available.\n"
            f"Available tools: {', '.join(self.tools.keys())}"
        )
    try:
        return self.tools[tool_name](param)
    except Exception as e:
        return f"Tool '{tool_name}' failed to execute. Error: {e}"
```

3. Improve the `run()` with better output and fallback

```
def run(self, user_input: str):
    prompt = f"{self.description}\n\nQuestion: {user_input}"
    response = self.llm.invoke(prompt).content

    print("\n--- Raw Model Response ---")
    print(response)

    tool_name, param = self._extract_action(response)

    if tool_name:
        result = self._execute_tool(tool_name, param)
        print(f"\n--- Tool Used ---\n{tool_name}({param}) = {result}")
    else:
        print("\nNo valid Action step found.")
        print("Hint: Expected format is: Action: tool[param]")
        print("     Example: Action: doc_search[vacation policy]")
```

4. Run the program
    a. You don't really need to *change* the multi_agent_runner.py

```
python multi_agent_runner.py
```

Some test cases you might consider:

```
You: What is 5 * (6 + 1)?
You: What's the company policy on remote work?
You: Tell me about unknown_tool[stuff]
You: Just answer directly without using a tool
```

Expected results if you use the test cases above:

- First two trigger proper tool use
- Third gives "tool not available" warning
- Fourth triggers "No valid Action step found"

Additional Notes and Explanations

Why do we need both `_extract_action` and `_execute_tool` to handle errors?

- `_extract_action()` validates **formatting** (was the LLM structured properly?)
- `_execute_tool()` validates **content** (was the tool real? did it run successfully?)

Both are needed for a robust agent architecture.

Could this be done in LangChain's default agents?

Sort of — LangChain's `AgentExecutor` handles some errors internally. But by implementing your own `AgentCore`, you've created *full visibility and control*, which is essential for education and production-level systems.

To summarize, you now have:

- Full feedback on invalid/missing tool calls
- Helpful error messages for malformed Action lines
- Cleaner, more stable agent execution loop

This prepares your agent system for even more complexity — including retry logic, user clarification prompts, or LangGraph flow branching.

Note on LangChain's `AgentExecutor`:
In production-grade LangChain applications, a built-in class called `AgentExecutor` is commonly used to wrap agents and tools into a structured execution loop that handles tool parsing, error fallback, and memory integration. However, in this tutorial, we've built our own lightweight orchestrator logic manually to better understand how tool parsing, routing, and error handling work behind the scenes. This hands-on approach makes it easier to later appreciate the abstraction and power of `AgentExecutor`, while giving us full control over how we direct tool execution and process model outputs.