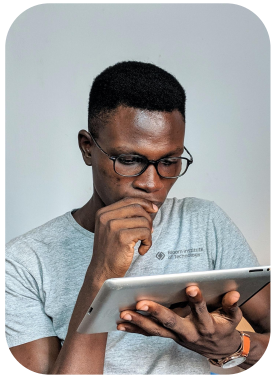
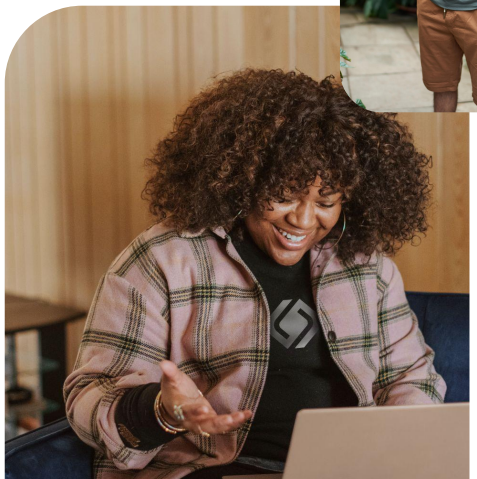
Bloom Institute of Technology

Agents

Use a language model to select a sequence of actions and act as a reasoning engine to determine which actions to take and in what order.





Core Competencies

The student must demonstrate...

1. Understanding ReAct and the basic agentic lifecycle (10 min)
2. Understanding the ReAct prompt pattern and why it works (5 min)
3. How to build an agent (20 min)
4. View agents in LangSmith (10 min)
5. Using pre-built agents (10 min)

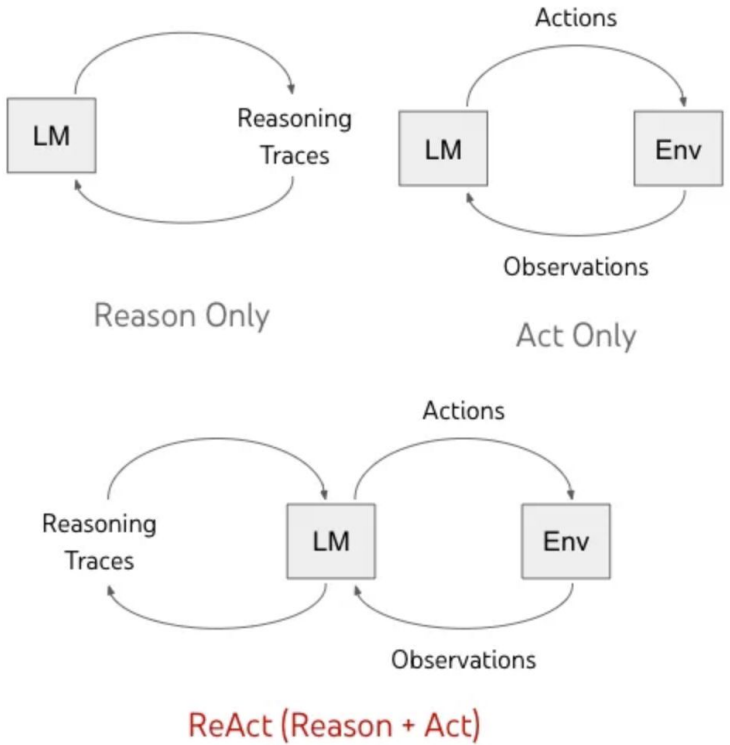
Using the *ReAct* Framework

What is ReAct?

- ReAct stands for Reasoning + Acting.
- Enables LLMs to mimic human-like operations, reasoning verbally, and taking actions to gain information.
- Allows LLMs to interact with external tools, enhancing decision-making processes.

ReAct Agents in LangChain

- The ReAct agent in LangChain utilizes the ReAct framework to select the appropriate tool based on its description.
- It is the most general-purpose action agent in LangChain, ideal for situations with multiple available tools.
- Automatically selects the right tool based on the provided descriptions.

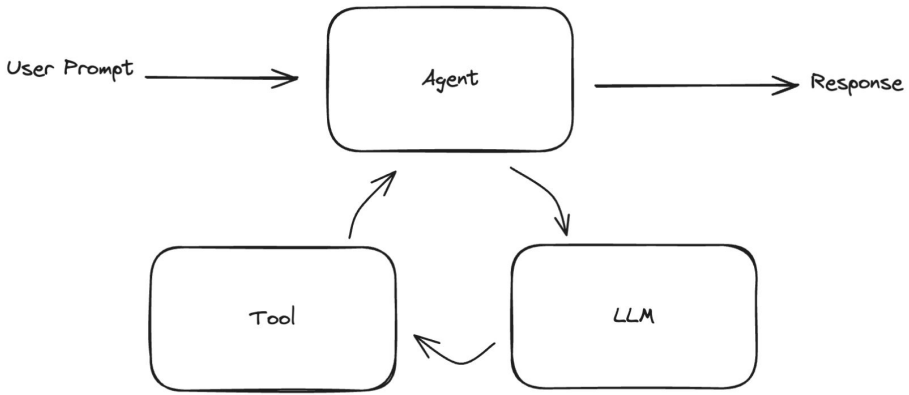


Agentic Lifecycle

Agents **use LLMs to simulate reasoning** and solve complex tasks. Developers can equip agents with tools and code, allowing them to control how the agents work.

How It Works

1. The system receives input from the user.
2. Based on the input, the agent decides whether to use a tool and determines the appropriate input for that tool.
3. The tool is called and the output is recorded as an observation.
4. The history of the tool, tool input, and observation is fed back into the agent.
5. The agent repeats the process until no further tools are needed.
6. The agent responds directly to the user once all necessary tools have been utilized.



How to Build a Research Agent

Objective: Build an agent using LangChain that can research, review, and write reports on specific topics. This agent should be capable of gathering relevant information, analyzing it, and producing well-structured, comprehensive reports.

Overview of Tasks

1. **Initialize a LLM**: We will use GPT-3.5-turbo for simplicity.
2. **Define Tools**: Set up the tools that the agent will use for research, review, and report writing.
3. **Define a Prompt**: Create a prompt that outlines the goal for the agent.
4. **Call Agent Executor**: Utilize the `AgentExecutor` inside LangChain, a runtime for the agent.
5. **Stream Outputs**: Stream the agent's outputs and content to the terminal for real-time monitoring.

Define Tools

Before we start coding, make sure to: (1) Obtain an OpenAI API key and a free Tavily search key. (2) Add these keys to your coding environment.

Ask yourself: ***What tools does our research agent need***, or in other words, what actions must it be able to perform?

1. **Tool:** Web Search
   How: Tavily Search API
   Action: Gather Information

2. **Tool:** Report Analysis and Writing
   How: Call OpenAI with context
   Action: Analyze and write report

3. **Tool:** Report Saving
   How: Interact with file system
   Action: Save the report

```python
@tool
def web_research(query: str) -> str:
    """
    Performs a web search.
    """
    tool = TavilySearchResults()
    docs = tool.invoke({"query": query})
    web_results = "\n".join([d["content"] for d in docs])

    return web_results

@tool
def write_report(research_data: str) -> str:
    """
    Writes a report based on topic and research data.
    """
    writer = ChatOpenAI()
    prompt = " " # Will fill this in later
    chain = prompt | writer | StrOutputParser()
    return chain.invoke({"research_data": research_data})

@tool
def save_to_file(report: str, filename: str) -> str:
    """
    Saves the report to a file.
    """
    with open(filename, "w", encoding="utf-8") as file:
        file.write(report)

    return f"Content saved successfully to {filename}"
```

Initialize LLM and Build Prompts

You must write two prompts for OpenAI. The first prompt, used inside the `write_report` tool, is for analyzing data and creating the structured report, while the second prompt is for your agent to help it choose tools, make observations, and reason its way to an output.

Inside write_report tool:

```
system_prompt = "You are a writer that writes
clear and concise reports on the context you are
given. Your audience is business executives."

prompt = ChatPromptTemplate.from_messages(
      [
          (
              "system", system_prompt
          ),
          (
              "user",
              "context: {research_data}"
          ),
      ]
    )
```

For the agent:

```
# Initialize the base LLM (using OpenAI's GPT model
here for simplicity)
llm = ChatOpenAI()

agent_prompt = "You are research assistant. When
the report is written, save it to a file"

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", agent_prompt),
        ("user", "{input}"),
        MessagesPlaceholder(
            variable_name = "agent_scratchpad"
        ),
    ]
  )
```

Create and Run Agent

- The code initializes a list of tools (web_research, write_report, save_to_file) and binds them to a LLM, then creates an agent using these tools and a specified prompt.
- The AgentExecutor is set up to manage the agent's runtime, executing actions, passing outputs back to the agent, and streaming the agent's content to the terminal.

```
tools = [web_research, write_report, save_to_file]
llm_with_tools = llm.bind_tools(tools)

agent = create_tool_calling_agent(llm, tools, prompt)

# The agent executor is the runtime for an agent.
# This is what actually calls the agent, executes the actions it chooses, passes the
# action outputs back to the agent, and repeats.
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# `list` allows the agent's content to stream to the terminal
query = "Write a report about the mysterious gpt2-chatbot that's on lmsys.org."
list(agent_executor.stream({"input": query, "agent_scratchpad": ""}))
```

Pre-built Agents in LangChain

LangChain provides pre-built agents that are **ready to use and serve as examples for building your own**. LangChain Toolkits are collections of tools for specific tasks, easily extendable as needed.

Access the pre-built toolkits list here.

```python
# Pre-built example: sql_agent

from langchain_community.utilities import SQLDatabase
from langchain_community.agent_toolkits import create_sql_agent
from langchain_openai import ChatOpenAI
from constants import DB_URI
from langchain_community.agent_toolkits import SQLDatabaseToolkit

db = SQLDatabase.from_uri(DB_URI)
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)

toolkit = SQLDatabaseToolkit(db=db, llm=llm)
agent_executor = create_sql_agent(toolkit=toolkit,
agent_type="openai-tools", verbose=True)

agent_executor.invoke(
"List all the employees with salaries greater than 1000."
)
```

Pre-built agents can be:

- Tailored to Your Needs: Pre-built implementations can be customized to suit your needs.
- Leveraged for Learning: Review the source code to understand how to implement tools for complex scenarios.

Hands-on Homework.

Create a chatbot using an agent. The agent must be able to take a user prompt and decide whether to do a web search, database lookup, RAG, or a combination to resolve the query.

Note: The tools can be mocked by having them return static strings.