Module 5: Building Agents with LLMs, Tools & Memory

Learning Objectives

Upon completion of this module, students will be able to:

- Implement basic AI agents from scratch in Python to understand their internal workings.
- Utilize the OpenAI Agents SDK for building sophisticated AI agents, including tool integration and function calling.
- Integrate LLMs as the reasoning core of AI agents.
- Develop custom tools and functions for agents to interact with external APIs and systems.
- Understand and implement memory management strategies for agents, including short-term and long-term memory.

Key Topics and Explanations

5.1 Agent Implementation from Scratch

Understanding the fundamental components of an agent by building a simple one from the ground up provides crucial insights into how agentic systems operate.

5.1.1 Building a Simple Agent Loop

- **Concept:** The core of any agent is its continuous loop of perceiving, thinking (reasoning), and acting. This loop drives the agent's autonomous behavior.
- **Implementation:** A basic Python loop can simulate this:

5.1.2 Implementing Basic Perception and Action Mechanisms

- **Perception:** Can involve reading from files, making API calls, parsing sensor data, or interpreting user input.
- **Action:** Can involve writing to files, sending API requests, controlling external devices, or generating text responses.

5.1.3 State Management for Agents

- **Internal State:** How the agent keeps track of its own beliefs, goals, and knowledge about the environment.
- **Persistence:** Storing the agent's state so it can resume operations after being shut down or restarted.

5.2 OpenAI Agents SDK

The OpenAI Agents SDK provides a powerful and intuitive way to build agents that leverage OpenAI's models for reasoning and tool use.

5.2.1 Introduction to the SDK and its Components

- **Core Idea:** The SDK simplifies the creation of agents that can use tools, manage memory, and engage in multi-turn interactions.
- **Key Components:**
    - **Agent:** The central entity that orchestrates perception, reasoning, and action.
    - **Tools:** Functions or APIs that the agent can call to interact with the external world.
    - **Memory:** Mechanisms for the agent to retain information across turns.
    - **Messages:** The primary way agents communicate with the user and with each other (implicitly).

5.2.2 Defining Agent Capabilities and Tools

- **Tool Definition:** Tools are typically Python functions decorated or registered with the SDK, along with a description that the LLM can understand.
- **Example Tool:**

5.2.3 Function Calling and Tool Use with LLMs

- **Concept:** LLMs can be trained to detect when a user's request can be fulfilled by calling a specific tool/function and to respond with the JSON arguments needed to call that function.
- **Process:**
    1. User provides a prompt.
    2. LLM analyzes the prompt and determines if a tool is needed.
    3. If so, LLM generates a function call (tool name + arguments).
    4. The SDK executes the tool.
    5. The tool's output is fed back to the LLM for generating a natural language response.

5.2.4 Handling Agent Responses and Errors

- **Structured Responses:** Agents can be configured to provide responses in specific formats (e.g., JSON).
- **Error Handling:** Implementing mechanisms to gracefully handle errors during tool execution or LLM inference.

5.3 LLMs as Agent Brains

LLMs serve as the powerful reasoning engine for many modern AI agents, enabling them to understand complex instructions, generate plans, and make decisions.

5.3.1 Connecting LLMs for Reasoning and Decision-Making

- **Integration:** Agents typically interact with LLMs via their APIs. The agent constructs prompts that include the current observation, relevant memory, available tools, and the agent's goal.
- **Decision Process:** The LLM, based on the prompt, decides whether to:
    - Generate a direct response.

    - Call a tool.
    - Ask for clarification.
    - Update its internal state/memory.

5.3.2 Prompt Design for Agentic Behavior

- **System Prompts:** Instructions given to the LLM at the beginning of a conversation to define its role, constraints, and how it should use tools.
- **Dynamic Prompting:** Constructing prompts on the fly based on the agent's current state, observations, and memory.
- **Thought-Action-Observation (TAO) Pattern:** A common prompting pattern where the LLM is encouraged to output its internal

thought process, the action it decides to take, and then the observation from that action.

5.3.3 Managing Context Windows and Token Usage

- **Context Window:** The limited amount of text (tokens) an LLM can process at one time. This is a critical constraint for agents that need to maintain long conversations or access extensive information.
- **Token Usage Optimization:** Strategies to minimize token consumption, including:
    - **Summarization:** Summarizing past conversations or retrieved documents.
    - **Filtering:** Only including the most relevant information in the prompt.
    - **Compression:** Using techniques to compress information before feeding it to the LLM.

5.4 Tool Integration and External Systems

Tools are how agents interact with the outside world, extending their capabilities beyond just language generation.

5.4.1 Creating Custom Python Tools for Agents

- **Concept:** Any Python function can be exposed as a tool to an agent, provided it has a clear description and defined inputs/outputs.
- **Best Practices:** Tools should be atomic, well-documented, and handle errors gracefully.

5.4.2 Interacting with REST APIs, Databases, and Other Services

- **API Wrappers:** Creating Python functions that wrap external API calls (e.g., weather APIs, financial data APIs, search engines).
- **Database Connectors:** Tools that allow agents to query and update information in databases (SQL, NoSQL).
- **Web Scraping:** Tools for extracting information from websites.

5.4.3 Handling Structured and Unstructured Data for Tool Inputs/Outputs

- **Structured Data:** Using Pydantic or dataclasses to define clear schemas for tool inputs and outputs, ensuring data consistency.
- **Unstructured Data:** Processing natural language inputs and outputs, often requiring additional LLM calls for parsing or generation.

5.5 Memory Management for Agents

Memory is crucial for agents to maintain context, learn from past experiences, and access external knowledge.

5.5.1 Short-term Memory (Context Window, Scratchpad)

- **Concept:** Information that the agent needs to access immediately and frequently, typically within the current conversation or task.
- **Implementation:** Often managed by the LLM's context window, or by maintaining a simple list/buffer of recent interactions (a

scratchpad).

- **Challenges:** Limited by the LLM's context window size, requiring strategies for summarization or truncation.

5.5.2 Long-term Memory (Vector Databases, Knowledge Graphs)

- **Concept:** Persistent storage for vast amounts of information that the agent might need to recall over extended periods or across different tasks. This includes factual knowledge, past experiences, and learned skills.
- **Vector Databases:** Specialized databases designed to store and query high-dimensional vector embeddings. Useful for semantic search and finding semantically similar information (e.g., ChromaDB, Qdrant, FAISS).
- **Knowledge Graphs:** Structured representations of knowledge that store entities and their relationships. Enable complex querying and reasoning (e.g., Neo4j, RDF stores).

5.5.3 Retrieval Augmented Generation (RAG) for Knowledge Retrieval

- **Concept:** A technique that combines the generative capabilities of LLMs with the ability to retrieve relevant information from external knowledge bases. Instead of relying solely on the LLM's pre-trained knowledge, RAG allows the LLM to

access up-to-date and specific information, reducing hallucinations and improving factual accuracy.

- **Process:**
  1. **Retrieval:** Given a query, retrieve relevant documents or chunks of text from a knowledge base (often stored in a vector database).
  2. **Augmentation:** The retrieved information is then provided to the LLM as part of the prompt.
  3. **Generation:** The LLM generates a response based on its internal knowledge and the augmented context.

Study Guide for Module 5

Self-Assessment Questions

1. Describe the core components of a simple agent loop that you would implement from scratch. What is the purpose of each component?

2. How does the OpenAI Agents SDK simplify the process of building AI agents compared to implementing everything from scratch?
3. Explain the concept of "function calling" in the context of LLMs and agent development. Provide a conceptual example of how an LLM might use a tool to answer a user query.
4. Why are LLMs considered the "brains" of many modern AI agents? How do agents typically interact with LLM APIs?
5. What is the significance of the LLM's context window in agent design? What strategies can be employed to manage token usage effectively?
6. When designing a custom tool for an agent, what are some best practices to ensure it is effective and reliable?
7. Differentiate between short-term and long-term memory for an AI agent. Provide examples of technologies or approaches used for each.
8. Explain the concept of Retrieval Augmented Generation (RAG). How does RAG improve the performance and factual accuracy of LLM-powered agents?
9. How can structured data (e.g., using Pydantic) be beneficial when defining tool inputs and outputs for agents?
10. In what scenarios would you choose to use a vector database for an agent's memory? What kind of information would you store there?

Practical Exercises

1. **Implement a Simple Agent:** Write a Python script to create a basic agent that simulates the Perceive-Think-Act loop. The agent should:
    - Perceive a simple input (e.g., a string representing an event).
    - "Think" by printing a decision based on the input and a simple internal rule.
    - "Act" by printing an action based on its decision.
    - Run the loop for a few iterations.

2. **OpenAI Agents SDK Tool Integration (Conceptual/Setup):** Set up a basic environment for the OpenAI Agents SDK (or a similar framework like LangChain). Define a simple tool (e.g., a function to look up a predefined fact). Write a basic agent that can use this tool when prompted.
3. **LLM as a Reasoning Engine:** Using an LLM API (e.g., OpenAI), craft a system prompt that instructs the LLM to act as a decision-making agent. Provide it with a scenario and a set of available actions, and ask it to choose the best action and explain its reasoning.
4. **RAG Simulation (Conceptual):** Imagine you have a small collection of text documents. Describe how you would use a vector database (conceptually) and an LLM to answer a question that requires retrieving information from these documents. Outline the steps involved in the RAG process.
5. **Custom Tool Development:** Write a Python function that simulates interacting with an external API (e.g., a mock weather API that returns a fixed temperature for a given city). Define its inputs and outputs clearly, ready to be integrated as an agent tool.

Further Reading and Resources

- **Libraries/Platforms:**
  - OpenAI Agents SDK Documentation (Note: The OpenAI Assistants API is the successor to the original Agents SDK, offering similar functionalities. Focus on the Assistants API for current best practices).
  - LangChain Documentation (especially sections on Agents and Memory).
  - LlamaIndex Documentation (especially sections on RAG and Vector Stores).
  - ChromaDB Documentation
  - Qdrant Documentation
- **Articles/Blogs:**
  - "Building AI Agents with LLMs" (various articles from OpenAI, Google AI, and independent researchers).
  - "Retrieval Augmented Generation (RAG)" explained.

    - Tutorials on integrating LLMs with external tools.
- **GitHub Repositories:**
    - Explore the `panaversity/learn-agentic-ai` repository for practical examples of agent implementation and Dapr integration.
    - Look for examples of agents built with LangChain or LlamaIndex.