- What we will Learn Today?
  - Local LLM:
  - Connecting to Local LLMs:
  - AI Agents:
  - AI Agent Frameworks:
  - LangGraph:
  - AI Agent Components:

What we will Learn Today?

☑ Setup Local LLM.
☑ Connect to Local LLM via API.
○ AI Agents.
○ AI Agents Frameworks.
○ Dive Deeper into LangGraph
○ Learn about AI Agent Components

Local LLM:

- There are various tools which allow us to setup LLMs on our local machine.
- The most used tools are:
  1. LM-Studio:
     - Runs llama.cpp backend.
     - Supports lot of hardware (CPU, Intel/NVIDIA/AMD GPU).
     - We will be using this in our course.
  2. Ollama:
     - Simple Setup

        - Well supports CPU, NVIDIA GPU.
        - Partial support for AMD GPU.
    3. vLLM:
        - Handles heavy server workload well.
        - Used in production environment.
        - Works best on NVIDIA GPU.
- Various Open Source LLMs are:
    - Qwen3
    - Gemma
    - GPT-OSS
    - Llama
    - Phi

Connecting to Local LLMs:

- The local LLMs are exposed by the tools via the `/v1/chat/completions` API.
- These models can be accessed via the OpenAI Client.

```
from openai import OpenAI

client = OpenAI(
        base_url="<base url of tool>",
        api_key="sk-1234...anything"
    )

responses = client.chat.completions(
        model="<name of the model>,
        messages=[{"role": "user", "content: "Hi"}]
    )
```

Base URL	Tool
LM Studio	http://localhost:1234/v1
Ollama	http://localhost:11343/v1

Base URL	Tool
vLLm	http://localhost:6379/v1

AI Agents:

> **Agents ≠ simple workflows.**
> Workflows are *static*; agents are *dynamic*.

- **AI Agent** = **LLM + Tools + Memory + Context**
- They don't just follow predefined steps—they make decisions about *what to do next*.

AI Agent Frameworks:

- There are various frameworks for building Agents available.
- These agents abstract the low level tasks and provides a simplified API.
- Frameworks:
  1. LangGraph
  2. CrewAI
  3. Agent SDK
- Advantages:
  - Simple to use.
  - Has some nice optimizations inbuilt.
  - Has prebuilt templates, tools, etc
- Disadvantages:
  - Hides the underlying prompts and LLM calls.
  - Harder to Debug.
  - Can make simple tasks complex.

LangGraph:

- LangGraph is an AI Agent SDK that uses Graph Flow Execution.
- Defined with nodes and edges.
- Data Processing is done by *Nodes* and flow is decided using *Edges*.
- Provides **Functional** and **Graph** API.
- Define *State* using a *TypedDict*.
- **Node**:
    - Takes *State* as input and returns `json` output with keys of the *State*.
- **Edge:**
    - Use `add_edge` method.
- **Conditional Edge**:
    - Takes *State* as input and returns the name of the *Node* to go to.

```
from langgraph.graph import StateGraph, START, END
from typing import TypedDict, Literal
from pydantic import BaseModel
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="<model-name>")

class AgentState(TypedDict):
    sentence: str
    sentiment: Literal["positive", "negative", "neutral"]
    score: float

class SentimentResponse(BaseModel):
    sentiment: Literal["positive", "negative", "neutral"]
    score: float




def get_sentiment(state: AgentState):
    response =
llm.with_structured_output(SentimentResponse).invoke(f"""
Calculate the sentiment of the given sentence.
## Sentence:
```

```
{state.get("sentence")}
""")
    response = SentimentResponse.model_validate(response)
    return {
        "sentiment": response.sentiment,
        "score": response.score
    }

def positive_sentiment(state: AgentState):
    print(f"The sentence {state.get("sentence")} is Positive with
score = {state.get("score")}")
    return {}

def negative_sentiment(state: AgentState):
    print(f"The sentence {state.get("sentence")} is Negative with
score = {state.get("score")}")
    return {}

def neutal_sentiment(state: AgentState):
    print(f"The sentence {state.get("sentence")} is Neutral with score
= {state.get("score")}")
    return {}

def _route(state: AgentState):
    sentiment = state.get("sentiment")
    if sentiment == "positive":
        return "positive_sentiment"
    if sentiment == "negative":
        return "negative_sentiment"
    if sentiment == "neutral":
        return "neutral_sentiment"


graph = StateGraph(AgentState)

graph.add_node("get_sentiment", get_sentiment)
graph.add_node("positive_sentiment", positive_sentiment)
graph.add_node("negative_sentiment", negative_sentiment)
graph.add_node("neutral_sentiment", neutral_sentiment)

graph.add_edge(START, "get_sentiment")
graph.add_conditional_edge("get_sentiment", _route)

graph.add_edge("positive_sentiment", END)
graph.add_edge("negative_sentiment", END)
graph.add_edge("neutral_sentiment",  END)

agent = graph.compile()

result = agent.invoke({
    "sentence": "I had a very boring day at college today!"
})
```

- Production Agent Structure:

```
my-app/
├── my_agent # all project code lies within here
│   ├── utils # utilities for your graph
│   │   ├── __init__.py
│   │   ├── tools.py # tools for your graph
│   │   ├── nodes.py # node functions for your graph
│   │   └── state.py # state definition of your graph
│   ├── __init__.py
│   └── agent.py # code for constructing your graph
├── .env # environment variables
├── pyproject.toml # package dependencies
└── langgraph.json # configuration file for LangGraph
```

AI Agent Components:

- **LLM**
  - The *reasoning engine* of the agent
  - Handles text understanding, generation, and decision-making
- **Memory**
  - Keeps track of previous interactions
  - Enables personalization, recall, and continuity

- **Tools**
  - Extend the LLM’s raw capabilities
  - Examples: web search, database queries, API calls, code execution
- **Context**
  - The immediate “working set” of information
  - Includes system instructions, user queries, and retrieved documents