**🤖 1. What is an AI Agent?**

**📌 Simple Definition:**

An **AI Agent** is like a **single robot** (software program) that is **trained to do just one job** automatically, without needing help from humans.

**✅ Examples:**

* A **chatbot** on a shopping website that answers **only order-related questions**.
* A **banking bot** that can **check your account balance**.

🎯 Each of these is **designed for one task only**.

**🧠 How it works:**

* Follows **predefined rules** (like a fixed script).
* Cannot change its task or make its own decisions beyond the script.

**🧠 2. What is Agentic AI?**

**📌 Simple Definition:**

**Agentic AI** is like a **team of AI agents** working together to **solve a bigger and more complex problem** — and they can **learn, adapt, and make smart decisions** on their own.

**✅ Example:**

A **smart home system** with:

* One AI for **lights**
* One AI for **thermostat**
* One AI for **appliances**

Together, they:

* Check if someone is in the room.
* Turn off lights and AC to **save electricity**.
* Talk to each other, learn from surroundings, and adjust actions automatically.

**🧠 How it works:**

* Uses **real-time data** (like sensors or health monitors).
* Makes **independent decisions**.
* Learns over time — like a human becoming smarter with experience.

**🆚 Summary of Differences**

| **Feature** | **AI Agent** | **Agentic AI** |
| --- | --- | --- |
| 🔨 Task Type | One simple task | Many tasks working together |
| 🔁 Autonomy | Limited | High – can think and adapt |
| 🧠 Learning | No learning, follows rules | Learns from experience |
| 🤝 Collaboration | Works alone | Works in a team of agents |
| 🧭 Decision Making | Based on predefined rules | Based on real-time situations |

**🧪 More Examples:**

**AI Agent:**

* A **customer service chatbot** that checks order status.

**Agentic AI:**

* A **Personalized Health Assistant** that:
  + Checks your medical history
  + Monitors your fitness data
  + Gives health advice
  + Adapts if your health changes

**📚 Real-Life Analogy:**

**Think of it like:**

🧍 **AI Agent** = A single employee doing a **repetitive job** (like a cashier who scans products).

👥 **Agentic AI** = A **smart team of employees** who talk, think, adapt, and manage a **whole store** — lights, air conditioning, customer queries, stock management — **all by themselves**.

**🧠 Imagine a Real Project in a Company**

Let’s say a client from company **XYZ** gives a software project.

**The usual steps:**

1. ✅ **Requirement Gathering**
   * Done by Business Analysts or Product Managers.
2. 🗓️ **Sprint Planning**
   * The Project Manager or Scrum Master plans the work into small chunks (sprints).
3. 💻 **Development**
   * Developers work on assigned tasks.
4. 🧪 **Testing**
   * QA team tests the application and reports bugs.
5. 🔍 **Code Review**
   * Senior developers or peers review the written code.

**👇 Now, imagine doing all this with Agentic AI!**

This means: instead of humans doing these tasks, we **create AI agents** for each of them.

**🧩 Agentic AI System Setup:**

| **Task** | **AI Agent** |
| --- | --- |
| Drafting requirements | ✍️ Agent that uses an LLM (like ChatGPT) to convert rough ideas into proper tasks |
| Developer work | 👨‍💻 Agents that write code for each task (Agent 1, Agent 2...) |
| Testing | 🧪 Testing Agent that runs test cases and finds bugs |
| Code review | 🔍 Code Review Agent that reviews and gives feedback |
| Workflow control | 🧭 A main agent that manages all others to keep the system running |

These agents **talk to each other** and **work together** to finish the project.

**🤖 Real-World Example: AI Doing Developer's Job**

Let’s say a requirement says:

"Build a login system with email and password."

**How Agentic AI would work:**

1. ✍️ **Requirement Agent**: Converts this into a clear task like:
   * "Create a login API"
   * "Validate user credentials"
   * "Connect to database"
2. 👨‍💻 **Dev Agent 1**: Codes the login API
3. 👨‍💻 **Dev Agent 2**: Writes code to validate users
4. 🔍 **Code Review Agent**: Checks if code is clean, secure
5. 🧪 **Testing Agent**: Tests if login works
6. 📬 **Feedback Agent**: Reports bugs if any
7. 🔄 **Agents communicate** and make improvements

This is what **Agentic AI** does:  
**Multiple intelligent AI agents** working **together** to complete complex tasks.

**💡 Realizations:**

* You are **not replacing all humans** — the system still needs some **human feedback** to stay accurate.
* This is called **Human-in-the-Loop** — a topic you’ll learn in future videos.

**🧠 One More Cool Use Case: Blog Generation AI System**

Imagine generating a blog post using Agentic AI:

| **Task** | **Agent** |
| --- | --- |
| Create blog tit%le | 🧠 Agent 1 |
| Write the content | 📝 Agent 2 |
| Generate thumbnail | 🖼️ Agent 3 |
| Check grammar & SEO | 🔍 Agent 4 |

All these agents **collaborate** to publish a blog — without needing human effort in each step.

**🔚 In Short:**

**🧍‍♂️ AI Agent:**

* Single task
* Acts alone (like a chatbot)

**🤖 Agentic AI:**

* Team of agents
* Work together like a smart company team
* Can build software, write blogs, control smart homes, and more

**🧠 What is LangGraph? (In Simple Words)**

LangGraph is a **Python library** that helps you build **AI agents** which can talk to each other and **work as a team** to complete a complex task.

Imagine you're making a chatbot that:

* Understands your question
* Looks up Google if it doesn't know
* Calls an API if needed
* Asks a human if confused
* Gives the final answer

To make this work smoothly, we **draw it like a graph** — with **steps (nodes)** and **connections (edges)** between them.

That’s exactly what LangGraph helps you do! 🎯

**🕸️ What Does “Graph” Mean Here?**

In programming, a **graph** is just a way to represent connected steps.

**For example:**

[Start] → [Chatbot] → [Weather API] → [End]

* Each **box** (like Chatbot) is a **Node**.
* Each **arrow** between them is an **Edge**.
* The arrows only move **forward** — that’s why it's called a **DAG (Directed Acyclic Graph)**.

LangGraph lets you create this exact flow for your **AI agents**.

**🤖 What Are AI Agents?**

AI Agents = **LLMs (like GPT)** + **Tools + Memory + Reasoning**

They:

* Understand input (e.g., a question or task)
* Decide what to do next
* Use tools like APIs, databases
* Remember what happened earlier
* Communicate with other agents

LangGraph allows you to **organize** and **control** these agents properly.

**🚦 Why Use LangGraph? What's Special?**

LangGraph is great because it helps you handle **complex flows** with AI.  
Other tools (like plain LangChain) can do simple tasks, but LangGraph is built for:

| **Feature** | **Why It Matters** |
| --- | --- |
| ✅ **Stateful execution** | Each step remembers what happened before |
| 🔁 **Memory** | Agents can remember conversation history |
| 🧑‍⚖️ **Human-in-the-loop** | You (a human) can jump in and correct or approve steps |
| 🛠️ **Fine control** | You decide exactly how each step behaves |
| 🚀 **Production-ready** | Used by companies like Uber, LinkedIn, Klarna |

**🧱 What Are Nodes and Edges?**

* **Node** = A task or step in the process (e.g., get weather, call API)
* **Edge** = The direction from one step to another

So if your agent needs to:

1. Get user question
2. Check if LLM knows the answer
3. If not → use Google Search API
4. Return final answer

You’ll define it like:

[Start] → [LLM Agent] → [Search API] → [Answer] → [End]

This is your **Agentic Workflow**. LangGraph helps you write this like a **flowchart**, but in Python!

**🧠 What Is a Stateful Application?**

It means:  
Each time the agent runs, it **remembers** what happened before. So:

* It can **continue** from where it stopped.
* If there's a mistake, a human can **fix it**, and the agent continues.

Example:  
You’re writing a long email with an AI. It gets stuck.  
You jump in, fix the issue → AI continues.  
That’s **human-in-the-loop** and **stateful**!

**🧰 Tools You’ll Learn in This Course**

| **Tool** | **Purpose** |
| --- | --- |
| ⚙️ LangGraph | Create agent workflows |
| 🧠 LangChain | (optional) add tools, chains |
| 🧪 LangSmith | Test, debug, monitor your agents |
| 🧲 Vector DBs | For RAG (retrieve documents) |
| 🌐 API Tools | Use Google Search, Weather API, etc. |

**📦 Real-World Use Case Example**

**🧠 Build a Blog Generator AI:**

* Agent 1 → Creates blog title
* Agent 2 → Writes the blog content
* Agent 3 → Creates a thumbnail image
* Agent 4 → Publishes to website

LangGraph connects them like this:

[Start] → [Title Agent] → [Content Agent] → [Image Agent] → [Publish Agent] → [End]

If one step fails, a human can jump in and correct — that's **flexible and powerful!**

**🔚 Summary**

| **Concept** | **Meaning** |
| --- | --- |
| LangGraph | Python library to build smart AI workflows |
| Node | A single step/task in the process |
| Edge | A connection (direction) from one step to another |
| DAG | Directed flow without loops (step-by-step) |
| Stateful | Remembers what's happening at each step |
| Multi-Agent | Many AIs working together like a team |
| Human-in-the-loop | You can review/fix steps if needed |

**1. State Schema Basics**

* **Purpose**: A state schema defines the structure of the "state" object passed between nodes in a graph. It holds variables (e.g., name, game) that get updated as nodes execute.
* **Previous Context**: Earlier videos covered creating simple graphs, chatbots using graphs, and basic state schemas with TypedDict.

**2. Two Ways to Define State Schemas**

**A. Using**TypedDict

* **Definition**:

python

from typing import TypedDict, Literal

class TypedDictState(TypedDict):

name: str

game: Literal["cricket", "badminton"] *# Only these two values allowed*

* **Key Points**:
  + TypedDict defines a dictionary with fixed keys and type hints for values.
  + **Type Hints**: Not enforced at runtime (e.g., name: str won’t reject 123 unless operations fail).
  + **Example**:

python

state = {"name": "Krish", "game": "cricket"} *# Valid*

state = {"name": 123, "game": "chess"} *# No runtime error (but may fail later)*

**B. Using Data Classes**

* **Definition**:

python

from dataclasses import dataclass

from typing import Literal

@dataclass

class DataClassState:

name: str

game: Literal["cricket", "badminton"]

* **Key Points**:
  + More concise syntax for classes primarily storing data.
  + Access fields directly (e.g., state.name instead of state["name"]).
  + **Type Hints**: Like TypedDict, not enforced at runtime by default.

**3. Example: Graph with State Schema**

**Nodes and Logic**

* **Nodes**:
  + play\_game: Prints a message and returns updated state (e.g., "Krish wants to play cricket").
  + cricket/badminton: Conditional nodes triggered by decide\_play (random choice).
* **Graph Flow**:

python

builder.add\_edge("start", "play\_game")

builder.add\_conditional\_edge("play\_game", decide\_play, {"cricket": "cricket", "badminton": "badminton"})

builder.add\_edge("cricket", "end")

builder.add\_edge("badminton", "end")

**Invocation**

* TypedDict:

python

graph.invoke({"name": "Krish", "game": "cricket"}) *# Passed as dict*

* **Data Class**:

python

graph.invoke(DataClassState(name="Krish", game="cricket")) *# Passed as object*

**4. Limitations of**TypedDict**and Data Classes**

* **No Runtime Validation**: Type hints (e.g., name: str) are ignored at runtime. Example:

python

graph.invoke({"name": 123, "game": "chess"}) *# No error until operations fail*

* **Solution**: Use **Pydantic** (next topic) to enforce runtime validation.

**5. Key Takeaways**

* TypedDict**vs. Data Classes**:
  + TypedDict: Dictionary-like access, good for JSON-like states.
  + Data Classes: Object-oriented access, cleaner syntax.
* **Type Hints**: Improve code clarity but don’t enforce runtime checks.
* **Next Step**: Pydantic will enforce runtime validation (e.g., reject name=123).

**Why This Matters**

* **State Management**: Critical for graphs where nodes modify shared state.
* **Flexibility**: Different schemas suit different use cases (e.g., APIs vs. internal logic).
* **Robustness**: Pydantic (next video) ensures data integrity at runtime.

**1. Problem: Runtime Validation**

* **Previous Video**: Discussed TypedDict and dataclass for defining state schemas.
  + **Issue**: Type hints (e.g., name: str) are ignored at runtime. Example:

python

graph.invoke({"name": 123}) *# No error despite `name: str` hint!*

* **Need**: Enforce type checks **during runtime** (e.g., reject name=123).

**2. Solution: Pydantic**

* **What is Pydantic?**
  + A data validation library for Python.
  + Enforces type hints at runtime using BaseModel.
* **Key Feature**:
  + If data doesn’t match the schema (e.g., str vs int), Pydantic raises a ValidationError.

**3. Example: State Schema with Pydantic**

**Step 1: Define a Pydantic Model**

python

from pydantic import BaseModel

class State(BaseModel):

name: str *# Enforced at runtime!*

* Inherit from BaseModel to enable validation.
* name: str now **must** be a string during runtime.

**Step 2: Build a Graph with Validated State**

python

from langgraph.graph import StateGraph

*# Define a node*

def example\_node(state: State):

return {"message": f"Hello, {state.name}"}

*# Build the graph*

builder = StateGraph(State) *# Use Pydantic model as state schema*

builder.add\_node("example\_node", example\_node)

builder.add\_edge("start", "example\_node")

builder.add\_edge("example\_node", "end")

graph = builder.compile()

**Step 3: Invoke the Graph**

* **Valid Input** (Works):

python

graph.invoke({"name": "Krish"}) *# Output: {"message": "Hello, Krish"}*

* **Invalid Input** (Fails):

python

graph.invoke({"name": 123}) *# Raises ValidationError: "name" must be a string*

**4. Why Pydantic?**

* **Runtime Safety**: Catches invalid data early (e.g., int instead of str).
* **Integration**: Works seamlessly with TypedDict/dataclass-like syntax.
* **Use Case**: Critical for applications where data integrity matters (e.g., chatbots, APIs).

**5. Key Takeaways**

* **Pydantic** enforces type hints **at runtime**, unlike TypedDict/dataclass.
* **Error Handling**: Fails fast with ValidationError for invalid data.
* **Next Steps**: Use Pydantic for all state schemas in graph-based apps to ensure robustness.

**Why This Matters**

* **Reliability**: Prevents bugs caused by incorrect data types.
* **Clarity**: Explicit schemas make code easier to debug and maintain.
* **Production-Ready**: Essential for deploying stable applications.

**1. Introduction to Chains in LangGraph**

* **What is a Chain?**  
  A sequence of nodes connected in order (e.g., Node 1 → Node 2 → ... → Node N). Chains can also loop back (e.g., Node 2 → Node 1) for complex workflows.
* **Previous Topics Covered**:
  + Simple graphs with nodes/edges.
  + Conditional edges (dynamic routing between nodes).
* **Goal of This Video**:  
  Combine four concepts to build advanced chains:
  1. **Chat Messages**
  2. **Chat Models**
  3. **Binding Tools**
  4. **Executing Tool Calls**

**2. Core Concepts**

**A. Chat Messages**

* **Purpose**: Track conversation roles (human, AI, system, tools) in a chatbot.
* **Types**:
  + HumanMessage: User input (e.g., "I want to learn coding").
  + AIMessage: LLM response (e.g., "Which programming language?").
  + SystemMessage: Instructions to the LLM (e.g., "Be concise").
  + ToolMessage: Output from external tools (e.g., API responses).
* **Example**:

python

from langchain.schema import HumanMessage, AIMessage

messages = [

AIMessage(content="How can I help?", name="AI\_Assistant"),

HumanMessage(content="I want to learn coding", name="User"),

AIMessage(content="Which programming language?", name="AI\_Assistant")

]

* + **State Management**: Messages are appended to the graph’s state using **reducers** (to avoid overwrites).

**B. Chat Models**

* **Integration**: Use LLMs (e.g., OpenAI, Grok) within graph nodes to process messages.
* **Example**:

python

from langchain\_groq import ChatGroq

llm = ChatGroq(model="mixtral-8x7b-32768")

response = llm.invoke(messages) *# Processes the conversation history*

print(response.content) *# Output: "Great choice! Python is versatile..."*

* + **Key Point**: Chat models take a **sequence of messages** as input to maintain context.

**C. Binding Tools**

* **Purpose**: Extend LLMs with external APIs/databases for real-time data (e.g., news, weather).
* **Scenario**:
  + User asks: *"What’s today’s news?"*
  + LLM lacks real-time data → Calls a **tool** (e.g., News API) → Returns fetched data.
* **Implementation** (Preview for Next Video):

python

*# Pseudocode*

llm\_with\_tools = llm.bind\_tools([NewsAPITool])

**D. Executing Tool Calls**

* **Mechanism**: LLM decides when to invoke tools based on input (e.g., "Get news for June 2025" triggers NewsAPITool).
* **Flow**:
  1. LLM detects need for external data.
  2. Calls the bound tool.
  3. Tool returns structured data (e.g., JSON).
  4. LLM formats the final response.

**3. Practical Demo**

**Step 1: Setup**

* Load environment variables (API keys for LLMs like Grok/OpenAI).

python

from dotenv import load\_dotenv

load\_dotenv()

**Step 2: Chat Messages**

* Manually create a conversation history:

python

messages = [

AIMessage(content="How can I help?", name="AI"),

HumanMessage(content="I want to learn Python", name="User")

]

**Step 3: Chat Model Integration**

* Initialize Grok LLM and pass messages:

python

from langchain\_groq import ChatGroq

llm = ChatGroq(model="mixtral-8x7b-32768")

response = llm.invoke(messages)

print(response.content) *# Output: Advice on learning Python*

**Step 4: Metadata Inspection**

* Access token usage and performance metrics:

python

print(response.response\_metadata)

*# Output: {'tokens': 521, 'model': 'mixtral-8x7b-32768', ...}*

**4. Key Takeaways**

* **Chat Messages**: Structure conversations with roles (Human/AI/Tool).
* **Chat Models**: Process message sequences in graph nodes.
* **Tools**: Extend LLMs with external APIs (next video).
* **Validation**: Pydantic ensures runtime type safety (from previous video).

**Why This Matters**

* **Real-World Use Case**: Build chatbots that remember context and fetch live data.
* **Modularity**: Chains allow reusable, composable workflows.
* **Next Steps**: Binding tools (e.g., News API) to handle queries like *"What’s today’s stock price?"*.

**Understanding Routers in LangGraph (Agent Workflows)**

This tutorial introduces **routers**, a key concept to transform LLM-powered nodes into **agents** that dynamically decide which tools/functions to execute based on input. Here’s a breakdown:

**1. Recap: Previous Workflow (Tool Binding)**

* **What We Did Earlier**:
  + Created a graph with an LLM node (e.g., ChatGroq).
  + **Bound a custom function** (e.g., add(a, b)) to the LLM using @tool.
  + Example:

python

from langchain.tools import tool

@tool

def add(a: int, b: int) -> int:

"""Adds two numbers."""

return a + b

llm\_with\_tools = llm.bind\_tools([add]) *# Bind tool to LLM*

* + **Limitation**: The LLM could only call the add tool when explicitly prompted (e.g., "Calculate 2+3").

**2. Introducing Routers**

* **Purpose**:  
  A router **dynamically selects which node/tool to execute next** based on the input or conversation state.
  + Turns static nodes into **agents** (autonomous decision-makers).
  + Enables complex workflows like:

text

User Query → Router → [Tool A] or [Tool B] or [LLM] → Response

* **Why Routers?**
  + **Flexibility**: Handle diverse queries without hardcoding paths (e.g., "Book a flight" vs. "Check weather").
  + **Agent Behavior**: LLM decides the best tool for the task at runtime.

**3. Key Components**

**A. Conditional Edges (Recap)**

* Used to branch between nodes based on logic:

python

from langgraph.graph import StateGraph

builder = StateGraph()

builder.add\_conditional\_edge(

"llm\_node",

decide\_function, *# Returns "tool\_a" or "tool\_b"*

{"tool\_a": "node\_a", "tool\_b": "node\_b"}

)

**B. Router (Advanced Conditional Edge)**

* **LLM as the Decision-Maker**:  
  Instead of a fixed decide\_function, use the LLM to **choose the next step**:

python

from langchain\_core.prompts import ChatPromptTemplate

router\_prompt = ChatPromptTemplate.from\_template("""

Given the user query, select the best tool:

Query: {input}

Options:

1. add (for math)

2. search (for general info)

Your choice (return ONLY the tool name):

""")

def router(state: dict):

response = llm.invoke(router\_prompt.format(input=state["query"]))

return response.content.strip() *# Returns "add" or "search"*

**C. Agent State**

* **Shared Memory**: Pass data between nodes (e.g., user query, tool outputs).

python

from typing import TypedDict

class AgentState(TypedDict):

query: str

result: str

history: list *# Chat messages*

**4. Implementation Example**

**Step 1: Define Tools**

python

@tool

def add(a: int, b: int) -> int:

"""Adds two numbers."""

return a + b

@tool

def search(query: str) -> str:

"""Searches the web."""

return f"Results for {query}"

**Step 2: Build the Graph**

python

from langgraph.graph import StateGraph

*# Define nodes*

def llm\_node(state: AgentState):

response = llm\_with\_tools.invoke(state["query"])

return {"result": response.content}

def tool\_node(state: AgentState):

tool\_name = state["selected\_tool"]

if tool\_name == "add":

a, b = parse\_input(state["query"]) *# Extract numbers*

return {"result": add(a, b)}

else:

return {"result": search(state["query"])}

*# Router function*

def router(state: AgentState):

response = llm.invoke(router\_prompt.format(input=state["query"]))

return {"selected\_tool": response.content.strip()}

*# Build graph*

builder = StateGraph(AgentState)

builder.add\_node("llm", llm\_node)

builder.add\_node("tool", tool\_node)

builder.add\_conditional\_edges("llm", router, {"add": "tool", "search": "tool"})

builder.add\_edge("tool", "end")

graph = builder.compile()

**Step 3: Invoke the Agent**

python

output = graph.invoke({"query": "What's 5+3?"})

print(output["result"]) *# 8*

output = graph.invoke({"query": "Latest AI news"})

print(output["result"]) *# Results for Latest AI news*

**5. Key Takeaways**

* **Routers Enable Agents**:  
  LLMs dynamically route queries to the right tool (no fixed paths).
* **State Management**:  
  AgentState tracks conversation context and tool outputs.
* **Real-World Use Cases**:
  + Customer support bots (route to FAQs/order lookup).
  + Data analysis pipelines (route to SQL/visualization tools).

**Next Steps**

* **Multi-Agent Workflows**: Combine routers for collaborative agents.
* **Error Handling**: Fallback nodes for failed tool calls.

1