ChatGPT

LangGraph Mastery: A Comprehensive Course from Basics to Production

Module 1: Introduction to LangGraph

Overview: In this module, we cover what LangGraph is and its core concepts. You'll learn how to define the state, add nodes and edges, and build your first simple graph-based LLM application (e.g. a chatbot). The lessons include explanations, Python code snippets, and a mini-project to reinforce learning.

Lesson 1.1: What is LangGraph?

LangGraph is an **open-source, stateful orchestration framework** (built on LangChain) for coordinating large language model (LLM) workflows. In LangGraph, you design your application as a directed graph where **nodes** are steps (LLM calls, tools, or code functions) and **edges** control the flow of execution [1] [2]. Unlike a simple chain of LLM calls, LangGraph lets you build multi-turn, **multi-agent applications** with shared state and conditional logic. Each agent or step in the system is a node, and they communicate via a shared state (e.g. a list of messages or data). This graph-based design gives you fine-grained control and makes complex flows (like chatbots calling tools or multiple agents collaborating) more reliable [1] [2].

Key Points:
- **Stateful Graph:** LangGraph maintains a global *state* (often a Pydantic model or TypedDict) that is passed between nodes [3]. The state holds things like conversation history, retrieved documents, or intermediate results.
- **Nodes:** Each node is a Python function or object that takes the current state and returns an update (often a dictionary of modified state fields) [4]. For example, a "chatbot" node might take a list of past messages and return a new assistant message.
- **Edges:** Edges connect nodes and control the execution order. You can add static edges (fixed flow) or *conditional edges* (branching logic) [5]. There are also special START and END nodes to mark entry/exit points.
- **Multi-Agent:** Every agent can be a node in the graph. Agents share the state (e.g. a message log) or invoke each other via tool calls [6] [2]. LangGraph supports explicit flows (nodes arranged in a fixed sequence) and dynamic flows (using a `Command` object to let an LLM decide what to do next) [7] [6].

> *In summary, LangGraph lets you design AI workflows as graphs of LLM agents and tools, with full control over state and flow. This structure is ideal for chatbots, complex agents, and data-processing pipelines that need memory and decision logic.*

Lesson 1.2: Setting Up Your LangGraph Environment

Before coding, install LangGraph and its dependencies:

```
pip install langchain langgraph
```

You may also install additional packages for visualization or tools, e.g.:
- `pip install pygraphviz` (for graphviz drawings) [8]
- API client libraries (e.g. for web search tools) as needed.

Ensure you have access to an LLM (OpenAI, Anthropic, etc.) by setting up API keys (e.g. `OPENAI_API_KEY`). If you plan to use LangChain's `init_chat_model` or other integrations, have those credentials ready. After installation, you can verify by importing in Python:

```
from langgraph.graph import StateGraph, START, END
```

Lesson 1.3: Core Concepts – State, Nodes, and Edges

State (Schema): In LangGraph, the state is typically defined as a Pydantic or TypedDict model that lists the fields (channels) you want to track. For a chatbot, you might include a `messages` list. For a data pipeline, you might include fields like `question`, `documents`, `summary`, etc [9]. You can annotate fields to automatically append to lists (e.g. conversation history). For example:

```
from typing import Annotated, List
from typing_extensions import TypedDict
from langgraph.graph.message import add_messages

class ChatState(TypedDict):
    # `messages` will accumulate conversation history automatically
    messages: Annotated[List, add_messages]
```

Here we use `add_messages` to tell LangGraph to append any new messages to the `messages` list [10].

Nodes: A node is defined by giving the graph a name and a callable (function or object). The callable receives the current state and returns a **dictionary** of new state values. For example, a simple node that sends a greeting:

```
def greet_node(state: ChatState):
    return {"messages": [{"role": "assistant", "content": "Hello! How can I help you?"}]}
graph_builder.add_node("greet", greet_node)
```

After the node runs, its output dict is merged into the graph state (so `messages` would now include the new assistant message). You can have nodes written as async or sync functions, and they can perform any logic or LLM calls inside.

Edges: Use `add_edge` to connect nodes. For a linear flow, connect `START` → node → `END`. For branching logic, use `add_conditional_edges`. For example:

```
graph_builder = StateGraph(ChatState)
graph_builder.add_edge(START, "greet")
graph_builder.add_edge("greet", END)
```

This means: start at “greet”, then finish.

You set entry and exit points (if not using START/END explicitly) with `set_entry_point` and `set_finish_point`. For instance, to make a node the start and also allow it to end the graph:

```
graph_builder.set_entry_point("greet")
graph_builder.set_finish_point("greet")
```

Lesson 1.4: Building Your First Chatbot Graph

Let’s build a simple LangGraph chatbot that echoes user input. We use a chat LLM from LangChain.

```
from langchain.chat_models import init_chat_model
from langgraph.graph import StateGraph
from langgraph.graph.message import add_messages
from typing_extensions import TypedDict
from typing import Annotated, List

# Define state to hold conversation
class ChatState(TypedDict):
    messages: Annotated[List, add_messages]

# Initialize graph
builder = StateGraph(ChatState)
llm = init_chat_model("anthropic:claude-3-5-sonnet-latest")  # or any model

# Node function: gets entire messages history, appends a response
def chatbot_node(state: ChatState):
    # invoke LLM on the entire conversation
    reply = llm.invoke(state["messages"])
    # return new message to append
    return {"messages": [reply]}

builder.add_node("chatbot", chatbot_node)
builder.set_entry_point("chatbot")
builder.set_finish_point("chatbot")
graph = builder.compile()
```

```
# Run the chatbot:
state: ChatState = {"messages": []}
user_message = {"role": "user", "content": "Hello, LangGraph!"}
state = graph.run({"messages": [user_message]})
print(state["messages"][-1]["content"])  # should be the LLM's reply
```

In this example, the `chatbot_node` takes the conversation history and calls the LLM. We start at the "chatbot" node and also end at it (single node graph) [4] . The LLM's response is added to the `messages` list. You can loop by feeding the state back in for multi-turn chat.

Exercise: Extend this graph so that it first greets the user and then ends. (Hint: add a greeting node and chain it to the chatbot node.)

Module 2: Integrating Tools and Complex Chatbots

Overview: Now that you can build a basic chatbot, we'll enhance it by integrating tools and enabling more complex agentic behavior. You will learn how to add tool-using nodes and conditional routing so the bot can fetch external info.

Lesson 2.1: Adding Tools to Your Chatbot

LangGraph lets your agents call external tools (web searches, calculators, APIs) via **ToolNodes** and conditional edges. For example, to let our chatbot use a web search, we:

1. **Define the tool:** Import or create a tool (e.g. using LangChain's tool wrappers).
2. **Add a ToolNode:** Wrap the tool in `ToolNode` and add it to the graph.
3. **Conditional routing:** If the LLM output indicates a tool call, route to the ToolNode. When the tool finishes, return to the chatbot.

```
from langchain_tavily import TavilySearch
from langgraph.toolchain import ToolNode, tools_condition

# 1. Create search tool
search_tool = TavilySearch(max_results=2)
tools = [search_tool]

# 2. Add nodes for chatbot and tool
builder = StateGraph(ChatState)
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools=tools))

# 3. Conditional edges: if model requests a tool, go to "tools", else end
builder.add_conditional_edges(
    "chatbot",
    tools_condition,                    # prebuilt function checks for tool calls
    {"tools": "tools", END: END}
)
# Any time the "tools" node finishes, go back to chatbot
builder.add_edge("tools", "chatbot")
# Also connect start to chatbot
builder.add_edge(START, "chatbot")

graph = builder.compile()
```

In this setup, after each chatbot response, LangGraph checks `tools_condition`. If the response includes a tool call, it routes to the **ToolNode**; the `ToolNode` runs the tool and returns results, then we flow back into the chatbot node to incorporate the result. Otherwise, if no tool is needed, the flow goes to `END` [5]. This loop continues until the chatbot decides to finish (e.g. by outputting a final answer).

For example, the chatbot might say: *"Here is the plan... (search_tool_call)"*. The `tools_condition` sees the tool call, takes the flow to the "tools" node, which runs the search and appends the results to state. Control then returns to the chatbot node to produce a final answer [5] [4].

Key Point: LangGraph's conditional edges and `ToolNode` let you build agent loops (LLM → tool → LLM → ...) cleanly [5].

Lesson 2.2: Maintaining Conversation State

Real chatbots need to remember past messages. We already used `add_messages` in **Module 1** to accumulate the `messages` history [10]. This built-in support means each time a node returns `{"messages": [reply]}`, that message gets appended to the history automatically. To continue the conversation, always include `messages` in your state schema and let the flow loop through the chatbot node.

For example, to have the chatbot ask follow-up questions, you might do:

```
# State already defined with 'messages' and 'add_messages' annotation

while True:
    user_input = input("User: ")
    if user_input.lower() in ("quit", "exit"):
        break
    state = graph.run({"messages": [{"role": "user", "content": user_input}]}, state)
    bot_response = state["messages"][-1]["content"]
    print("Bot:", bot_response)
```

Each call to `graph.run` uses the updated state from the previous turn, so the history grows [10]. This makes it easy to build multi-turn chatbots.

Lesson 2.3: Building an Agentic Chatbot (Example)

Let's combine what we've learned into a more capable chatbot that can use tools. The flow is: user query → chatbot LLM (may call tool) → tool lookup if needed → chatbot LLM again → final answer. Pseudocode:

```
# 1. Define state schema (with messages and tool results)
class BotState(TypedDict):
    messages: Annotated[List, add_messages]

# 2. Initialize graph and tools
builder = StateGraph(BotState)
builder.add_node("chatbot", chatbot_node)
builder.add_node("tools", ToolNode(tools=[search_tool]))
builder.add_conditional_edges("chatbot", tools_condition, {"tools": "tools", END: END})
builder.add_edge("tools", "chatbot")
builder.add_edge(START, "chatbot")
graph = builder.compile()

# 3. Chat loop (like before)
```

In this graph, whenever the LLM requests a search, the pipeline automatically detours through the tool and back to the LLM [5]. Otherwise it ends. You can test and refine the prompt so the LLM knows how to ask for tools.

Exercise: Modify the chatbot so that if no tool is needed, it still continues the conversation (instead of ending). (Hint: route the "END" back to "chatbot" or simply omit the END in conditional edges.)

Module 3: Multi-Agent Workflows

Overview: LangGraph excels at structuring multi-agent systems. In this module, we explore how to connect multiple agents (each with its own LLM, prompt, or tools) into one workflow. We'll cover explicit sequential flows, supervisor patterns, and communication strategies.

Lesson 3.1: Agents as Graph Nodes

In LangGraph, each **agent** can simply be a node or even a subgraph. For example, suppose you have two agents: a "Translator" and a "Summarizer." You can make each a node function and then chain them:

```
def translator_node(state: State):
    # Translate text in state["text"]
    return {"text": translated_text}

def summarizer_node(state: State):
    # Summarize state["text"]
    return {"summary": summary}
```

```
builder = StateGraph(State)
builder.add_node("translate", translator_node)
builder.add_node("summarize", summarizer_node)
builder.add_edge(START, "translate")
builder.add_edge("translate", "summarize")
builder.add_edge("summarize", END)
```

Here, control flows from Translator to Summarizer. Each agent sees the state passed from the previous one and appends results. This is **explicit control flow**. As LangChain's blog explains, each agent is a node and connections are edges [2]. This pattern ensures a predictable pipeline of agents.

Lesson 3.2: Sequential vs. Dynamic Multi-Agent Flows

You can define **static sequences** (as above) or **dynamic routing** where an agent (often called a *supervisor*) decides who goes next. For static flow, just chain edges in order. For dynamic flows, use the `Command` object: have one node return `Command(goto="agentX")` based on its reasoning [7].

For example, a supervisor agent might look at some shared state and then direct to team A or B:

```
from langgraph.graph import Command

def supervisor_node(state: State) -> Command:
    # Decide next agent based on state
    if some_condition:
        return Command(goto="team_A_agent")
    else:
        return Command(goto="team_B_agent")

builder = StateGraph(State)
builder.add_node("supervisor", supervisor_node)
builder.add_node("team_A_agent", agent_A_node)
builder.add_node("team_B_agent", agent_B_node)
builder.add_edge(START, "supervisor")
builder.add_edge("team_A_agent", "supervisor")
builder.add_edge("team_B_agent", "supervisor")
```

Here, after each agent finishes, control returns to the supervisor, who then chooses the next agent or ends by returning `END` [11]. This allows flexible workflows where the LLM logic can route tasks at runtime.

Key Point: LangGraph's `Command` type (and tool-calling supervisor patterns) let your graph flow be decided on the fly by LLM output [7]. This is useful for complex scenarios where the "path" isn't fixed.

Lesson 3.3: Communication Patterns Between Agents

When multiple agents run, how do they share information? LangGraph supports two main paradigms [6]:

- **Graph State (Shared Memory):** All agents read and write a common state (e.g. a shared `messages` list). Each agent node gets the entire state and can append or modify it [6]. This is straightforward: simply use one state schema for the whole graph (like we did with chat history). Agents can then inspect previous agents' outputs in that shared state.
- **Tool Calls:** Alternatively, an agent can invoke another agent as if it were a tool, passing only specific arguments. In this case, the called agent only receives the inputs (not the full state) and returns outputs. This lets one agent "filter" what the next sees. The supervisor example above uses this idea: the supervisor calls sub-agents via `Command` with specific arguments.

LangGraph's documentation discusses that agents often share a message log ("scratchpad") [12]. You can choose to share the full conversation history (so every agent sees all previous thoughts) or only final results (for simpler interfaces) [12] [13]. For instance, in a collaborative team of agents, you might let each agent add its reasoning to `state["messages"]` so others can read it [2] [12]. Alternatively, if agents operate in separate channels, you can design subgraphs with distinct state schemas and convert between them [14].

Lesson 3.4: Example – Two-Agent Collaboration

As an illustrative example, consider two simple agents that answer a user query together:

1. **Agent A (Researcher):** Searches a database for facts related to the user's question.
2. **Agent B (Responder):** Uses those facts to write a final answer.

A possible LangGraph design: share a state with `question`, `facts`, and `answer`.

```
class QAState(BaseModel):
    question: str
    facts: List[str] = []
    answer: Optional[str] = None

def researcher(state: QAState):
    results = web_search(state.question)
    state.facts.extend(results)
    return {"facts": state.facts}

def responder(state: QAState):
    prompt = f"Question: {state.question}\nFacts:\n" + "\n".join(state.facts)
    answer = llm.invoke(prompt)
    return {"answer": answer}

builder = StateGraph(QAState)
builder.add_node("researcher", researcher)
builder.add_node("responder", responder)
builder.add_edge(START, "researcher")
builder.add_edge("researcher", "responder")
builder.add_edge("responder", END)
graph = builder.compile()
```

In this setup, Agent A populates `state.facts`, then Agent B reads them to craft `state.answer`. Both run in sequence, and all data lives in the shared state.

> *Tip:* LangChain's LangGraph Academy provides examples of multi-agent architectures (shared scratchpad, supervisor teams, hierarchical agents) [15] [16]. In practice, you'll pick the pattern that fits your use case.

Module 3 Project: Multi-Agent Task

Build a multi-agent system using LangGraph:

- **Scenario:** A user asks a science question. One agent fetches relevant information (e.g., from Wikipedia), another agent formulates the answer.
- **Tasks:** Define the shared state model (question, docs, answer), implement two agent nodes (search and answer), connect them with edges, and test the pipeline end-to-end. Try altering the communication (e.g. have agents use a supervisor instead of shared state).

This project reinforces how to coordinate multiple LLM-driven components in one graph.

Module 4: LangGraph for Data Pipelines

Overview: Beyond chat, LangGraph can orchestrate complex data-processing pipelines that mix LLM calls, retrieval, analysis, and more. This module shows how to build pipelines like retrieval-augmented generation (RAG) and extend them with advanced features.

Lesson 4.1: Simple RAG Pipeline

A common pattern is a Retrieval-Augmented Generation pipeline: the system takes a question, retrieves documents, and generates an answer. In LangGraph, we can encode this as a graph:

1. **Define State:** Include fields for user question, retrieved documents, and LLM answer. For example:

   ```
   from langchain_core.pydantic_v1 import BaseModel

   class RAGState(BaseModel):
       question: str
       documents: List[str] = []
       answer: Optional[str] = None
   ```

   LangGraph will track this state across nodes [3].

2. **Retrieval Node:** Use a retriever (vector store, search API) to fetch documents based on `state.question`. Update `state.documents`.

```
def retrieve_node(state: RAGState):
    results = retriever.invoke(state.question)
    docs = [d.page_content for d in results]
    state.documents.extend(docs)
    return {"documents": state.documents}
```

3. **Generation Node:** Pass the question and documents to an LLM (via a LangChain chain) to get an answer. Update `state.answer`.

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser

prompt = ChatPromptTemplate.from_messages([...])  # set up prompt with {question} and {conte
llm_chain = prompt | ChatOpenAI(model='gpt-4o') | StrOutputParser()

def generate_node(state: RAGState):
    response = llm_chain.invoke({
        "question": state.question,
        "context": "\n\n".join(state.documents)
    })
    return {"answer": response}
```

4. **Build the Graph:**

```
from langgraph.graph import StateGraph, START, END

pipeline = StateGraph(RAGState)
pipeline.add_node("retrieve", retrieve_node)
pipeline.add_node("generate", generate_node)
pipeline.add_edge(START, "retrieve")
pipeline.add_edge("retrieve", "generate")
pipeline.add_edge("generate", END)
pipeline = pipeline.compile()
```

When you run `pipeline.run({"question": "..."} )`, LangGraph automatically passes state between nodes. This structure (retrieval → generation) matches the example in AIgents' tutorial [17]. In fact, the published example does exactly this: it creates a `GraphState` with question, docs, generation, and connects the retrieval and generator nodes [3] [17].

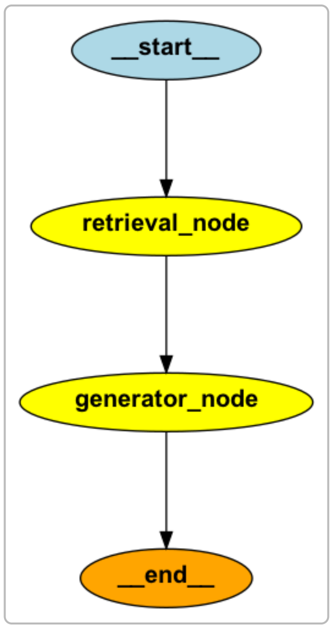

Figure: Example LangGraph pipeline (RAG) with a retrieval node followed by a generation node. State flows from start through each node to produce the final answer.

> *Tip:* The image above is a simple visualization of the pipeline. In LangGraph you can actually **draw** your graph (e.g. using `graph.get_graph().draw_png()`) to inspect it.

Lesson 4.2: Extending the Pipeline (Rewriting & Feedback)

Once the basic pipeline works, you can enhance it with extra steps:

- **Query Rewriting:** Often, the raw question can be improved before retrieval. Add a "rewriter" node that uses an LLM to reformulate the query [18]. For instance:

  ```
  def rewrite_node(state: RAGState):
      new_query = rewriter_chain.invoke({"question": state.question})
      return {"question": new_query}
  ```

  Insert this node before `retrieve` so that retrieval uses the rewritten question [18]:

  ```
  pipeline.add_node("rewrite", rewrite_node)
  pipeline.add_edge(START, "rewrite")
  pipeline.add_edge("rewrite", "retrieve")
  ```

  This follows the approach in the example blog [18], which adds a `rewritten_question` field to the state and routes through it.

- **Answer Verification:** After generation, you might check if the answer is good. You can add a "hallucination checker" or QA verifier that reads `state.documents` and `state.answer` and returns a score or feedback. Based on that, you could loop back and try again. For example:

```python
def check_node(state: RAGState):
    verdict = verifier_chain.invoke({
        "question": state.question,
        "documents": "\n\n".join(state.documents),
        "answer": state.answer
    })
    if verdict == "yes":
        return {"final": "return"}
    else:
        return Command(goto="rewrite")  # try rewriting and retrieving again
```

  This illustrates how you could use `Command` and additional nodes to create complex logic (e.g. retrying retrieval if the answer doesn't fully address the question). The AIgents article demonstrates this self-reflection loop [19] (checking for hallucinations and re-running parts of the pipeline).

By chaining more nodes and using conditional edges, you can turn a simple pipeline into a robust system with checks. LangGraph makes it easy to insert steps wherever you need in the workflow [18] [20].

Lesson 4.3: Other Data Pipelines

LangGraph can handle any multi-step data task, not just RAG. For example:
- **Text Analysis Pipeline:** Tokenize or split text, then classify sentiments, then aggregate results. Each step is a node, sharing the text or intermediate data in state [21] [3].
- **ETL Workflows:** Extract data from an API, transform it via LLM (e.g. for cleaning or summarizing), then load to a database. You can model each stage as nodes in a LangGraph.
- **Batch Processing with Loops:** You can loop over data items by feeding the state back to a node until a condition is met (e.g. using conditional edges as a "while" loop).

LangGraph's strength is in any scenario where you need to maintain state across steps and have decision points. We saw in [14] how the state model captures all needed fields.

Exercise: Build a simple multi-step text pipeline. For example, create nodes to (a) split a large document into paragraphs, (b) summarize each paragraph with an LLM, and (c) combine summaries. Track all summaries in state. Test it on a sample text.

Module 5: Advanced LangGraph Features

Overview: In this module we explore advanced control flows and features that make LangGraph powerful for production: conditional branching, loops, memory management, visualization, and debugging.

Lesson 5.1: Conditional Branching and Looping

We have seen a basic example of conditional edges (tool routing) [5]. You can write custom conditions too. For instance, to loop until a state condition, define a function that returns either `END` or the next node name:

```
from langgraph.graph import END

def loop_condition(state):
    if check_done(state):
        return END
    else:
        return "process_node"

builder.add_conditional_edges("process_node", loop_condition, {"process_node": "process_node", EN
```

This will keep invoking `process_node` until `check_done(state)` becomes true. LangGraph's docs describe using conditional edges to create loops and branches (much like a state machine) [5].

Lesson 5.2: Graphviz and Mermaid Visualization

Visualizing your graph is helpful for debugging. LangGraph supports drawing graphs via Graphviz or Mermaid. For example, to draw the pipeline we built, you could do:

```
graph = pipeline  # compiled StateGraph
from IPython.display import Image, display
display(Image(graph.get_graph().draw_png()))  # requires pygraphviz or pydot installed
```

This renders an image of the graph (Start, nodes, End). The LangGraph docs show similar usage to display pipeline diagrams [22] [23]. You can experiment with `.draw_mermaid_png()` or `.draw_ascii()` if you don't want image dependencies.

Lesson 5.3: Stateful Memory and Long-Term Context

LangGraph's state model can also represent *memory* across sessions. For a chatbot, you might store user data or preferences in the state model (e.g. `class ChatState(..., history: List[str]=[])`). The advantage over a stateless chain is that LangGraph persists state across each `run`. You can even load/save the state between program runs for long-term memory.

LangGraph Platform (LangSmith) provides features like state versioning and "time-travel" debugging [24], but at the library level you manage state yourself. Use Pydantic fields for any persistent data (conversation logs, user info, etc.).

Lesson 5.4: Debugging and Tracing

When building complex graphs, it helps to log each node's output. LangGraph doesn't include built-in printouts, but you can instrument nodes with print statements or use LangSmith tracing. For example:

```
def debug_node(state: State):
    result = llm.invoke(...)
    print(f"Debug: Node processed, result={result}")
    return {"output": result}
```

Additionally, after running the graph, you can inspect the state or use LangSmith (if available) to see the sequence of nodes executed. The LangGraph website mentions that LangGraph "enables granular control over the agent's thought process" and you can inspect the agent's actions [25] [1]. In practice, printing or logging inside nodes usually suffices.

Exercise: Take the RAG pipeline you built and add a debug node between retrieval and generation that logs how many documents were found. Ensure your graph still produces the correct final answer.

Module 6: Putting It All Together – Projects

In this final module, you'll undertake hands-on projects that combine the concepts above into complete applications. Each project outline below includes goals and suggested steps.

Project 6.1: Knowledge-Base Chatbot

Goal: Build a chatbot that can answer questions using a custom knowledge base (via RAG) and tools.
Steps:
- Prepare or load a text dataset (e.g. Wikipedia articles) into a vector store.
- Create a LangGraph with nodes: *rephrase query* → *retrieve docs* → *generate answer*.
- Add a tool (e.g. browser search) to handle unknown questions.
- Ensure state includes conversation history so the bot can maintain context.
- Test the bot with user queries, including ones requiring the knowledge base and ones needing external search.

Project 6.2: Multi-Agent Writing Assistant

Goal: Design a multi-agent system where one agent brainstorms ideas and another writes a polished paragraph.
Steps:
- Define state with fields for `topic`, `ideas`, `draft`.
- Agent A (Ideator): Given a topic, produces a list of ideas (using an LLM).
- Agent B (Writer): Takes the list of ideas and writes a full paragraph or story.
- Optionally add a supervisor: if the ideas list is empty, the supervisor prompts Agent A again; else moves to Agent B.
- Use LangGraph to orchestrate the agents (shared state or supervisor with Commands).

- Experiment by changing prompts and flow (e.g. have the writer ask for more details if the draft is unsatisfying).

Project 6.3: Data Analysis Pipeline

Goal: Use LangGraph to create a pipeline that processes input data end-to-end.
Example: A text sentiment pipeline: *Ingest text → split into sentences → analyze sentiment per sentence → summarize sentiments*.
Steps:
- Define a state model with the input text, list of sentences, list of sentiment scores, and final summary.
- Create nodes: (1) `split_text`, (2) `analyze_sentiment`, (3) `summarize`. The second node can loop over sentences if desired.
- Optionally use conditional edges: e.g. if a sentence has no emotion words, skip analysis.
- Run the graph on a sample paragraph and verify the output.

Each project is an opportunity to solidify how state flows through LangGraph, how to define nodes, and how to connect everything. Feel free to add more complexity (error handling, user prompts, iterations) as you gain confidence.

Course Summary: By completing this course, you will have a deep understanding of LangGraph's core primitives and how to apply them in building chatbots, multi-agent systems, and data pipelines. You will see that LangGraph provides a **flexible graph-based framework** for orchestrating LLMs, with features like built-in statefulness and fine-grained control flow [1] [2]. The included coding examples and projects prepare you to build production-grade applications with LangGraph.

Further Reading: Refer to the LangGraph documentation and tutorials for more examples. In particular, the LangChain Academy and official guides cover topics like human-in-the-loop, subgraphs, and LangGraph Platform integrations.

---

[1] [4] [5] [10] [23] **Learn the basics**
https://langchain-ai.github.io/langgraph/tutorials/introduction/

[2] [15] [16] **LangGraph: Multi-Agent Workflows**
https://blog.langchain.dev/langgraph-multi-agent-workflows/

[3] [9] [17] [18] [19] [20] [21] [22] **How to Build Complex LLM Pipelines with LangGraph!**
https://aigents.co/data-science-blog/coding-tutorial/how-to-build-complex-llm-pipelines-with-langgraph

[6] [7] [11] [12] [13] [14] **Multi-agent Systems**
https://langchain-ai.github.io/langgraph/concepts/multi_agent/

[8] **How to visualize your graph**
https://langchain-ai.github.io/langgraph/how-tos/visualization/

[24] [25] **LangGraph**
https://www.langchain.com/langgraph