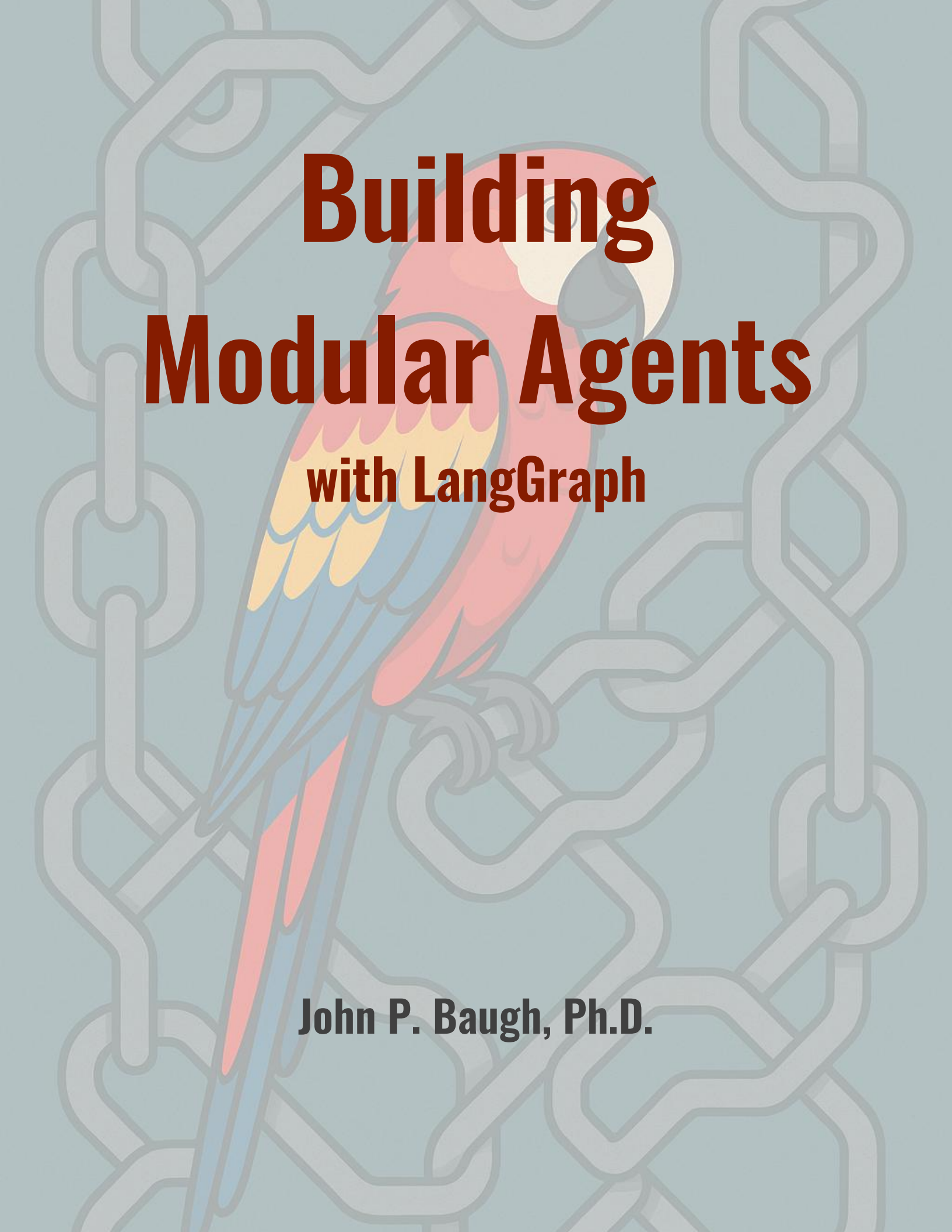


Building Modular Agents

with LangGraph

John P. Baugh, Ph.D.

Building Modular Agents with LangGraph

John P. Baugh, Ph.D.

v. 2025.5.8.1

Version	Date	Notes
2025.5.8.1	5/8/2025	Initial release

Table of Contents

Module 1: The General Format of Agents in LangGraph

In this module, we'll explore the general requirements of most Agents in LangGraph. In the next couple modules we'll do *actual* implementations of Agents using the general format for specific implementations, and how to test them out.

What Every Agent Should Include

In practically every agent you'll build, you'll need the following.

Component	Description
`State (TypedDict or BaseModel)`	Defines what data flows through the graph
`__init__()`	Initializes and stores the graph
`_build_graph()`	Wires up the graph using `StateGraph`
Node function(s)	One or more functions that modify state
`.run()`	External entry point that accepts input and invokes the graph

To add a little more clarity to the State, let's clarify its purpose.

- The **state** is a shared dictionary (typically, `TypedDict`) that carries all your data through your agent's internal graph.
- Think of it as a "backpack" each node reads from and writes to.
- Nodes never talk to each other directly — they only mutate the ***shared*** state

Template Agent Structure

This is a *template* agent structure. You *don't really need* to try to code this (you'll have concrete examples in just a bit). Just observe what the different parts do, based on the descriptions in the table above.

```
from typing import TypedDict
from langgraph.graph import StateGraph

class MyAgentState(TypedDict):
    input: str
    output: str

class MyAgent:
    def __init__(self):
        self.graph = self._build_graph()
```

```
    def _build_graph(self):
        builder = StateGraph(MyAgentState)
        builder.add_node("step", self._step)
        builder.set_entry_point("step")
        builder.set_finish_point("step")
        return builder.compile()

    def _step(self, state: MyAgentState) -> MyAgentState:
        return {**state, "output": f"You said: {state['input']}"}

    def run(self, user_input: str) -> MyAgentState:
        return self.graph.invoke({"input": user_input, "output": ""})
```

Before moving on…

Once you go to the next tutorial, before you jump in to coding right away, make sure you have it set up properly.

- In VS Code, make sure to:
    - Select the folder you want to work in
    - Create the directory structure (see the specific tutorials)
    - Create a virtual environment

```
python -m venv venv
```

    - Activate the virtual environment

macOS/Linux:

```
source venv/bin/activate
```

OR:

Windows (Bash terminal):

```
source venv/Scripts/activate
```

    - Install necessary libraries

```
pip install langchain langgraph langchain-core langchain-ollama
```

Module 2: EchoAgent (One-Node Agent)

This Agent is actually LLM-free, for the purpose of clarity and simplicity. It demonstrates core **LangGraph** mechanics (state passing, node logic, modular design) without requiring any model inference.

This program will simply echo what the user inputs.

Important Preliminary Information

Recall the 5 components from Module 1 (the General Format), and let's see what we'll do in the EchoAgent we're about to build.

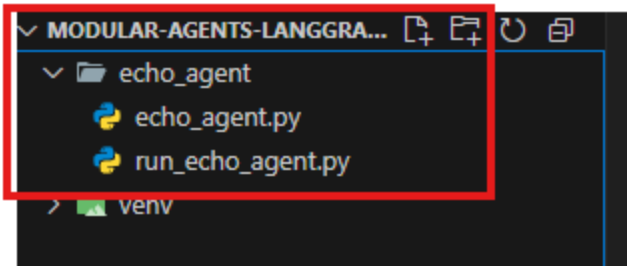
Required Part	In EchoAgent	Purpose
`State`	EchoState	Defines `input` and `output` as the two keys flowing between nodes
`__init__()`	Yes	Calls `_build_graph()` once when the agent is created
`_build_graph()`	Yes	Defines and compiles a one-node LangGraph
Node function(s)	`_echo_node()`	Reads from state and adds `output`
`.run()`	Yes	Starts execution with initial state dict and returns the final state

Additional important functions that we'll use:

- **`add_node("echo", self._echo_node)`**
  - This tells LangGraph: "We have a node named echo, and it runs the function _echo_node()."
  - That function receives the entire state, modifies it, and returns a new version.
  - This is the *core of LangGraph* — building flows of logic based on discrete named steps (nodes).
- **`set_entry_point("echo") + set_finish_point("echo")`**
  - Marks the graph's start and end.
  - Since there's only one node, this is both the first and last step.

Instructions

1. Create the following file structure in your project
   a. A subfolder for echo_agent or making this its own VS Code project are both ok



2. In **echo_agent.py**, write the following code:

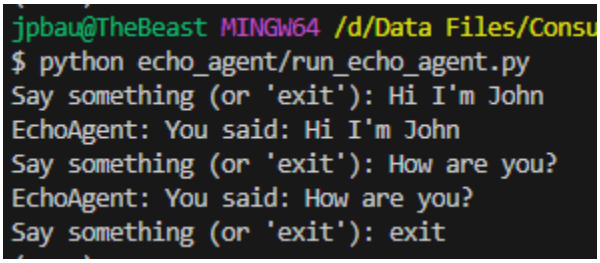
```
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain.prompts import PromptTemplate

class EchoState(TypedDict):
    input: str
    output: str

class EchoAgent:
    def __init__(self):
        self.prompt = PromptTemplate.from_template("You said: {input}")
        self.graph = self._build_graph()

    def _build_graph(self):
        builder = StateGraph(EchoState)
        builder.add_node("echo", self._echo_node)
        builder.set_entry_point("echo")
        builder.set_finish_point("echo")
        return builder.compile()

    def _echo_node(self, state: EchoState) -> EchoState:
        message = self.prompt.format(text=state["input"])
        return {**state, "output": message}

    def run(self, user_input: str) -> EchoState:
        return self.graph.invoke({"input": user_input, "output": ""})
```

3. In **run_echo_agent.py**, write the following code:

```
from echo_agent import EchoAgent

agent = EchoAgent()

while True:
    user_input = input("Say something (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = agent.run(user_input)
    print("EchoAgent:", result["output"])
```

4. Now, **run** the program:

```
python run_echo_agent.py
```



Module 3: MiniAgent (Two-Step Agent)

This module also presents an LLM-free agent, but this time with two steps.

Here's what it does:

- Converts input to uppercase (`shoutify` step)
- Wraps the result in `>>> <<<` (`finalize` step)

Important Preliminary Information

Required Part	In MiniAgent	Purpose
State	MiniAgentState	Tracks input, modified, and output across nodes
__init__()	Yes	Same pattern: calls _build_graph()
_build_graph()	Yes	Wires up two LangGraph nodes and connects them
Node function(s)	_shoutify() and _finalize()	First modifies text, second formats it
.run()	Yes	Starts graph execution with full default state

Flow Summary:

1. User input is placed into `state['input']`
2. `shoutify` node uppercases the input → saves in `state['modified']`
3. `finalize` node adds formatting → saves in `state['output']`

Other functions that are used:

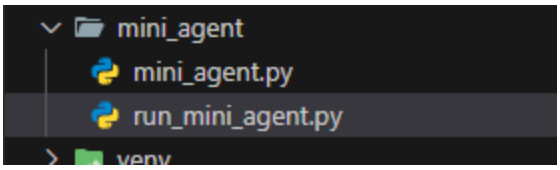
- **`add_node("shoutify", self._shoutify)`**
  - Adds the first transformation step: turns input into uppercase.
- **`add_node("finalize", self._finalize)`**
  - Adds the second step: wraps the uppercase result in decorative marks.
- **`add_edge("shoutify", "finalize")`**
  - Creates a **directed link** between nodes, like an arrow in a flowchart:

```
shoutify → finalize
```

LangGraph will pass the returned state from shoutify into finalize.

Instructions

1. Create the following structure for this project:



2. If you haven't done so yet, make sure to setup up a virtual environment (venv) and install the libraries as appropriate

3. Fill in the code for **mini_agent.py**

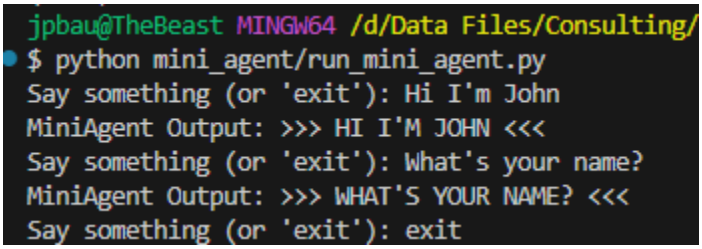
```
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain.prompts import PromptTemplate

class MiniAgentState(TypedDict):
    input: str
    modified: str
    output: str

class MiniAgent:
    def __init__(self):
        self.final_prompt = PromptTemplate.from_template(">>> {text} <<<")
        self.graph = self._build_graph()

    def _build_graph(self):
        builder = StateGraph(MiniAgentState)
        builder.add_node("shoutify", self._shoutify)
        builder.add_node("finalize", self._finalize)
        builder.set_entry_point("shoutify")
        builder.add_edge("shoutify", "finalize")
        builder.set_finish_point("finalize")
        return builder.compile()

    def _shoutify(self, state: MiniAgentState) -> MiniAgentState:
        return {**state, "modified": state["input"].upper()}

    def _finalize(self, state: MiniAgentState) -> MiniAgentState:
        output = self.final_prompt.format(text=state["modified"])
        return {**state, "output": output}

    def run(self, user_input: str) -> MiniAgentState:
        return self.graph.invoke({
            "input": user_input,
            "modified": "",
            "output": ""
        })
```

4. Now, fill in the code for **run_mini_agent.py**

```
from mini_agent import MiniAgent

agent = MiniAgent()

while True:
    user_input = input("Say something (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = agent.run(user_input)
    print("MiniAgent Output:", result["output"])
```

5. **Run** the program:

```
python run_mini_agent.py
```



Module 4: LLMResponderAgent (Simple LLM-Based Agent)

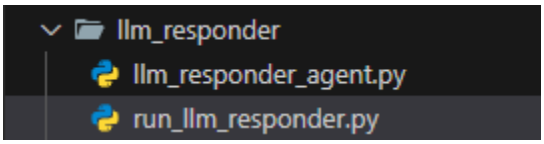
This is your first **LangGraph Agent that uses an LLM** via langchain-ollama. This builds on the format you've seen before but introduces a new pattern: using a local model (like LLaMA3) as part of a node's logic.

Important Preliminary Information

Required Part	In LLMResponderAgent	Purpose
`State`	`LLMResponderState`	Tracks the `input` and the `llm_response`
`__init__()`	Yes	Instantiates the LLM and the subgraph
`_build_graph()`	Yes	Adds and compiles the LLM node
Node function	`_respond_with_llm()`	Calls Ollama LLM with the user's input
`.run()`	Yes	Kicks off execution and returns the final result

Instructions

1. Create the following structure for this project:



2. If you haven't done so yet, make sure to setup up a virtual environment (venv) and install the libraries as appropriate

3. Fill in the code for **llm_responder_agent.py**

```
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama
from langchain.prompts import PromptTemplate

class LLMResponderState(TypedDict):
    input: str
    llm_response: str
```

```
class LLMResponderAgent:
    def __init__(self):
        self.llm = ChatOllama(model="llama3")
        self.prompt = PromptTemplate.from_template(
            "You are a helpful assistant. Respond concisely to: {query}"
        )
        self.graph = self._build_graph()

    def _build_graph(self):
        builder = StateGraph(LLMResponderState)
        builder.add_node("respond", self._respond_with_llm)
        builder.set_entry_point("respond")
        builder.set_finish_point("respond")
        return builder.compile()

    def _respond_with_llm(self, state: LLMResponderState) -> LLMResponderState:
        prompt_text = self.prompt.format(query=state["input"])
        response = self.llm.invoke(prompt_text).content
        return {**state, "llm_response": response}

    def run(self, user_input: str) -> LLMResponderState:
        return self.graph.invoke({"input": user_input, "llm_response": ""})
```

4. Write the code for **run_llm_responder.py**

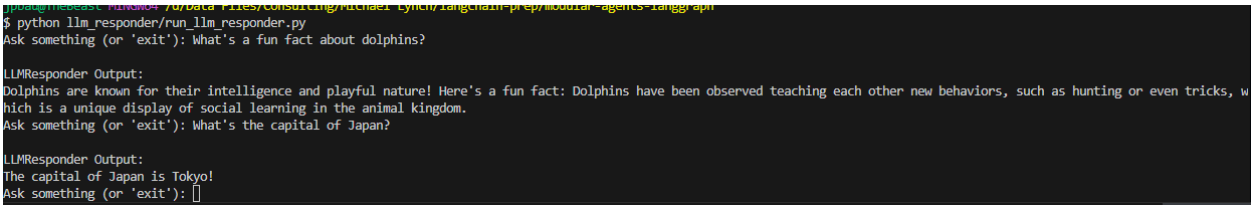
```
from llm_responder_agent import LLMResponderAgent

agent = LLMResponderAgent()

while True:
    user_input = input("Ask something (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = agent.run(user_input)
    print("\nLLMResponder Output:")
    print(result["llm_response"])
```

5. **Run** it:

```
python run_llm_responder.py
```

Try with questions like:

- What's a fun fact about dolphins?
- What is the capital of Japan?



Summary

- This module introduces the **LLM component** of agentic design.
- You still follow the modular LangGraph format:
- Define a state
- Add a node
- Set entry and finish points
- The *LLM call is **hidden** inside a node,* so from the graph's perspective, it's just another state transformation.