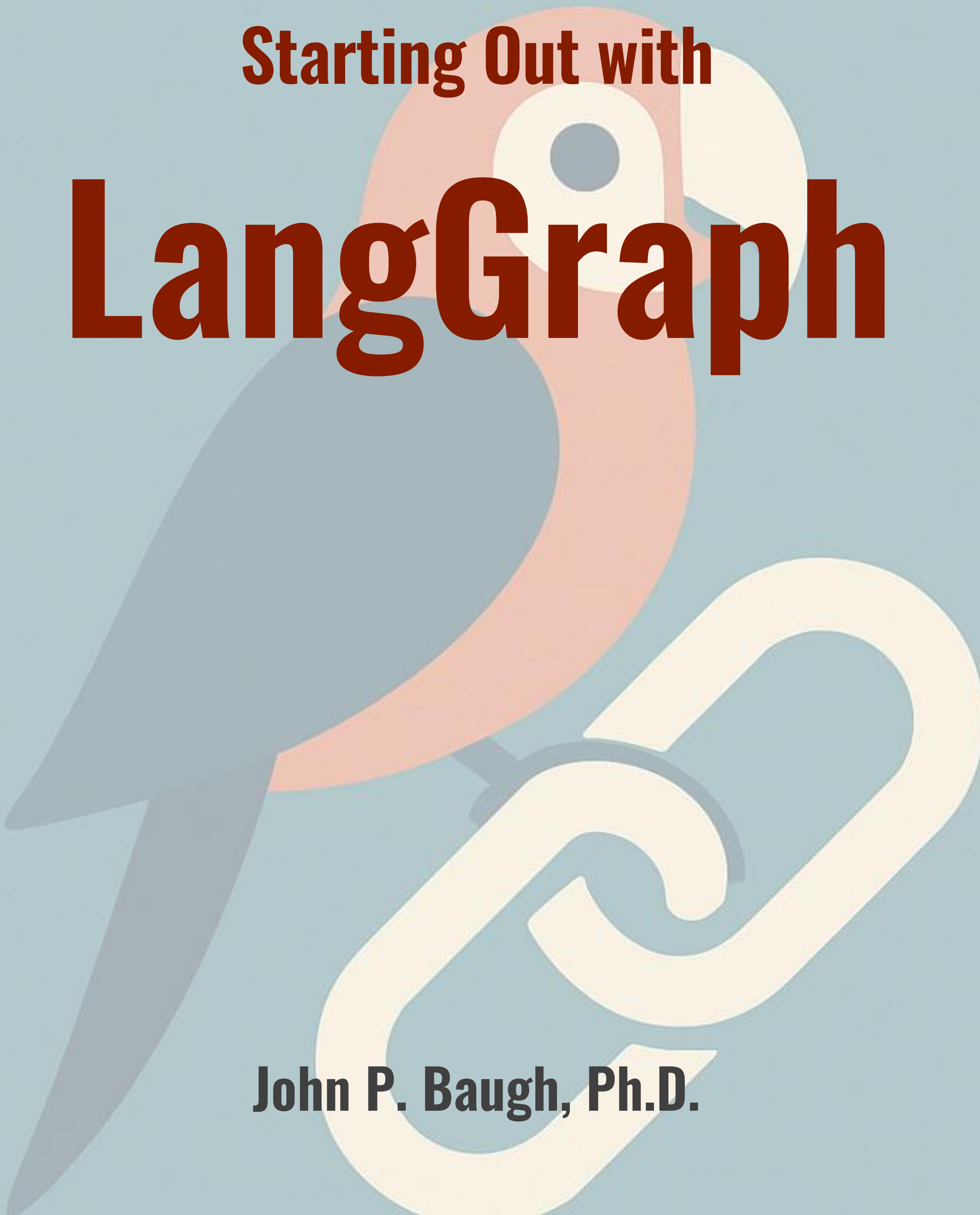


Starting Out with

LangGraph

John P. Baugh, Ph.D.

Starting Out with LangGraph

John P. Baugh, Ph.D.

v. 2025.5.7.1

Version	Date	Notes
2025.5.7.1	5/7/2025	Initial release

Table of Contents

Setup 5
  Instructions 5
Tutorial 1: Hello LangGraph 8
  Goal 8
  About the Technologies 8
  Instructions 8
  Additional Notes and Explanations 9
Tutorial 2: LangGraph with State 11
  Goal 11
  About the Technologies 11
  Instructions 11
  Additional Notes and Explanations 14
Tutorial 3: ReAct in LangGraph 15
  Goal 15
  About the Technologies 15
  Instructions 15
  Additional Notes and Explanations 18
    Agents and Modular Design in LangGraph 19
Tutorial 4: Multi-Turn Chat with a Loop 21
  Goal 21
  About the Technologies 21
  Instructions 21
  Additional Notes and Explanations 24
Tutorial 5: Creating a Reusable ReAct Agent Class 25
  Goal 25
  About the Technologies 25
  Instructions 25
  Additional Notes and Explanations 28
    Why Modularize Into a Class? 28
    Real-World Benefit 28
Tutorial 6: Routing Between Agents 29
  Goal 29

About the Technologies 29
Instructions 29
Additional Notes and Explanations 32
Why Use Keyword Routing? 32
Recap: Tutorials 1-6 33
Summary of Tutorials 33
Summary of LangChain Classes and Components 33
Tutorial 7: Adding a Retrieval-Augmented Generation Tool to InfoAgent 35
Goal 35
About the Technologies 35
Instructions 35
Additional Notes and Explanations 38
Why Add RAG to InfoAgent? 39
Prompt Engineering Reminder 39
Tutorial 8: Error Handling and Fallbacks in LangGraph Agents 40
Goal 40
About the Technologies 40
Instructions 40
Additional Notes and Explanations 42
Why Flexible Parsing? 42
Why Validate Tool Use? 43

Setup

You'll want to prepare your development environment to build LangGraph-based applications with local LLMs (e.g., LLaMA3 via Ollama). You'll install the necessary packages, verify LangGraph is working, and create a starter folder for this series.

LangGraph is a graph-based framework built on top of **LangChain**. It lets you structure flows of logic and tool use using **state machines** and **directed graphs**, which are ideal for:

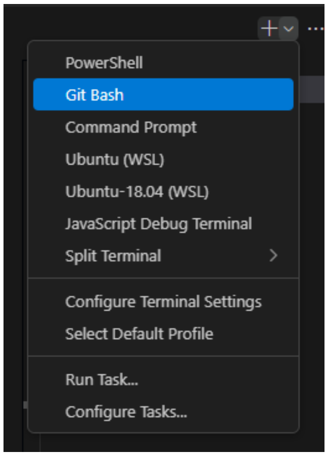
- Tool orchestration
- Multi-step reasoning
- Dynamic workflows
- Agent systems with branches or memory

Key concepts include:

- `StateGraph`: the structure of your nodes and edges
- `add_node()`, `add_edge()`: define flow between steps
- `compile()`: finalize the graph
- `invoke()`: run the graph with an input

Instructions

1. Create a folder for the tutorials
    a. E.g., **langgraph-tutorials**
2. Open that folder from VS Code
    a. File → Open Folder
    b. Find the **langgraph-tutorials** folder
    c. Select it/Open it
3. Open a Terminal window
    a. Terminal → new Terminal (in VS Code)
    b. Change the shell type to a Unix-style shell, such as Bash



4. Create and activate a virtual environment
    a. E.g., in the Terminal, type:

```
python -m venv venv
```

5. Activate the virtual environment

Type:

```
source venv\Scripts\activate
```

Windows

Or

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

macOS or Linux

6. Install required packages

```
pip install langgraph langchain langchain-core langchain-community langchain-ollama
faiss-cpu langchain-text-splitters
```

7. Install Ollama if not already installed
    a. Go to https://ollama.com/download
    b. Download the installer
    c. Install it
8. Test to make sure it is running
    a. Go into the VS Code Terminal again

```
ollama list
```

That will show installed models. If you do not see llama3 installed, you should pull it down using:

```
ollama pull llama3
```

9. If you see everything working well, you may want to restart VS Code just to make sure everything is properly registered and configured before continuing

10. In VS Code, create a test file, **test_langgraph.py**

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

# 1. Define the shape of the state using TypedDict
class EchoState(TypedDict):
    input: str
    output: str

# 2. Define a simple echo node
def echo_node(state: EchoState) -> EchoState:
    print("Inside echo_node")
    return {"output": f"You said: {state['input']}", "input": state["input"]}

# 3. Build the graph with the state schema
builder = StateGraph(EchoState)
builder.add_node("echo", echo_node)
builder.set_entry_point("echo")
builder.set_finish_point("echo")  # Required in newer LangGraph versions
graph = builder.compile()
```

```
# 4. Run the graph
result = graph.invoke({"input": "Hello, LangGraph!"})
print(result)
```

11. Run the program.

```
python test_langgraph.py
```

To be clear, what do we mean by *shape of the state* defined by `TypedDict`?

This means, "What keys and value types should the state dictionary contain as it flows through the graph?"

In LangGraph, state is *always* a dictionary. But to help validate and structure the flow, LangGraph encourages you to define the expected keys and types using `TypedDict`.

`TypedDict` is a special class from the typing module in Python. It's a way to *describe the expected structure of a dictionary* — like a lightweight schema.

LangGraph ***requires*** you to either:

- provide a `state_schema` (like a `TypedDict` or `BaseModel`), or
- define `input_keys` and `output_keys` manually.

Tutorial 1: Hello LangGraph

Goal

Build and run your first LangGraph — a basic two-node state machine that takes user input and generates a response using a local LLM (e.g., LLaMA3 via Ollama).

This is the "Hello World" of LangGraph.

About the Technologies

LangGraph is a library that lets you build **stateful graphs** of LLM-augmented logic. Unlike simple chains, LangGraph allows:

- branching
- memory across steps
- conditional routing
- retries and fallbacks
- tool orchestration

In this tutorial, you'll use:

Component	Description
`StateGraph`	Defines the structure of your node-based workflow
`TypedDict`	Defines the "shape" of the state (what keys/values flow between nodes)
`add_node()`	Adds a step to your graph
`set_entry_point()`	Defines where the flow starts
`set_finish_point()`	Marks the terminal step
`ChatOllama`	LangChain wrapper around LLaMA3 running locally via Ollama

Instructions

This graph will just:

1. Accept a string from the user
2. Send it to the LLM
3. Output the model's response

1. Create a new file, **hello_langgraph.py**
2. Write the code

```python
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama

# 1. Define the shape of the state using TypedDict
class EchoState(TypedDict):
    input: str
    output: str

# 2. Load the local LLaMA3 model
llm = ChatOllama(model="llama3")

# 3. Define a node that sends the input to the LLM
def ask_llm(state: EchoState) -> EchoState:
    user_input = state["input"]
    response = llm.invoke(f"Respond to this input: {user_input}").content
    return {"input": user_input, "output": response}

# 4. Build the graph
builder = StateGraph(EchoState)
builder.add_node("talk_to_llm", ask_llm)
builder.set_entry_point("talk_to_llm")
builder.set_finish_point("talk_to_llm")  # Required to finalize
graph = builder.compile()

# 5. Run it
user_input = input("Type something to the LLM: ")
result = graph.invoke({"input": user_input})
print("\nModel said:\n" + result["output"])
```

3. Run it

```
python hello_langgraph.py
```

Try it with something like the following:

```
Type something to the LLM: What's a fun fact about jellyfish?
```

Expected output might be something like:

```
Model said:
Jellyfish have been around for over 500 million years, making them older than
dinosaurs!
```

Additional Notes and Explanations

- **TypedDict: EchoState**
    - This defines what data flows through the graph (i.e., input and output)
    - LangGraph uses this to check validity when compiling your graph
- **ask_llm() Node**

    - This function receives the state (with "input")
    - It uses llm.invoke() to get a response
    - It returns a **new state** with both "input" and "output"
- **StateGraph**
    - add_node("name", fn): Adds a function as a node
    - set_entry_point(...): Tells LangGraph where to begin
    - set_finish_point(...): Required to mark an exit node
- **Note on Ollama**
    - Make sure Ollama is **running** in the background. If not, you'll get a connection error when the model tries to respond.

Tutorial 2: LangGraph with State

Goal

Build a graph that **tracks and updates internal state** across steps. We'll simulate a simple multi-step process where:

- The user provides an input
- The graph stores a running counter of how many interactions have occurred
- Each time you run it, the graph increments the counter and returns it along with the LLM response

About the Technologies

This tutorial introduces **mutable state flow** — a key strength of LangGraph.

Instead of just passing a single input → output, the graph now carries extra **persistent keys** (like a counter, memory, or flags) between nodes.

You'll use:

Component	Description
`TypedDict`	Now includes input, output, and turn (number of steps taken)
State mutation	Nodes read from and write to keys in the shared state
Multiple nodes	Graph flows through two steps: LLM → counter update

This structure models real-world cases like:

- Conversation turn counting
- Context window tracking
- Memory state updates

Instructions

1. Create a new file, **langgraph_with_state.py**
2. Now, write the code

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

# 1. Define state shape
class ChatState(TypedDict):
    input: str
    output: str
    turn: int  # Keep track of how many interactions have occurred

# 2. Load local model
```

```
llm = ChatOllama(model="llama3", temperature=0)

# 3. Node: Ask the LLM
def llm_node(state: ChatState) -> ChatState:
    input_text = state["input"]
    response = llm.invoke(f"A person says: {input_text}. How would you
respond?").content

    return {
        "input": input_text,
        "output": response,
        "turn": state["turn"],  # Keep turn unchanged here
    }

# 4. Node: Increment turn counter
def increment_turn(state: ChatState) -> ChatState:
    # This node keeps the same input/output but adds +1 to turn count
    return {
        "input": state["input"],
        "output": state["output"],
        "turn": state["turn"] + 1,
    }

# 5. Build the graph
builder = StateGraph(ChatState)
builder.add_node("ask_llm", llm_node)
builder.add_node("increment_turn", increment_turn)
builder.set_entry_point("ask_llm")
builder.add_edge("ask_llm", "increment_turn")
builder.set_finish_point("increment_turn")
graph = builder.compile()

# 6. Run the graph
initial_turn = 0
while True:
    user_input = input("\nType something (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = graph.invoke({"input": user_input, "turn": initial_turn, "output": ""})
    print(f"\n[Model]: {result['output']}")
    print(f"[Turn Count]: {result['turn']}")

    initial_turn = result["turn"]  # Carry turn forward for next round
```

3. Run the program

```
python langgraph_with_state.py
```

Example session:

```
Type something (or 'exit'): Hello there!

[Model]: You said: Hello there!
```

```
[Turn Count]: 1

Type something (or 'exit'): Tell me a fact.

[Model]: You said: Tell me a fact.
[Turn Count]: 2
```

In our current graph, **state** is the *dictionary* that *flows from node to node* and accumulates or updates values.

We're tracking three keys:

```
class ChatState(TypedDict):
    input: str
    output: str
    turn: int
```

Every time you type a new input:

- The **initial state** is passed in with a turn value.
- The graph executes `increment_turn()` which adds 1.
- The **returned state** includes the updated turn count.

So when you see:

```
[Turn Count]: 1
[Turn Count]: 2
```

This demonstrates that `turn` is being preserved and incremented correctly across runs.

Additionally, input and output Persist Across Nodes

Each node receives the **full state**, including input and output.

- `llm_node()` writes to output based on input
- `increment_turn()` keeps output and input the same, only mutating turn

You print:

```
print(f"[Model]: {result['output']}")
print(f"[Turn Count]: {result['turn']}")
```

All parts of the state are being passed and updated correctly.

We can also see that state carries across invocations. Even though LangGraph graphs are stateless between .invoke() calls (unless you persist it manually), you're doing this:

```
initial_turn = result["turn"]
```

So you're carrying the previous state's value back into the `next .invoke()` call manually.

This proves that you're treating LangGraph as a stateful system by managing state at the app level.

Additional Notes and Explanations

Component	Role
`turn: int`	A persistent counter showing how many times the graph has run
`increment_turn`	Modifies the state directly by incrementing turn
`add_edge()`	Connects ask_llm → increment_turn
`builder.set_finish_point(...)`	Marks the end of the graph — must be called after final node

So what's Happening Under the Hood?

1. The state dictionary (with input, turn, and output) is passed to `ask_llm`
2. That node updates the output, leaves turn alone
3. The updated state flows to `increment_turn`, which bumps the turn
4. Final state is returned to the caller
5. The CLI loop carries the new turn value forward

This tutorial is the first glimpse of true **stateful orchestration**. From here, you can:

- Track multiple variables (e.g., goals, flags, memory)
- Make conditional branches based on state
- Add history or session identifiers

Tutorial 3: ReAct in LangGraph

Goal

Implement the **ReAct** pattern (Reasoning + Acting) using multiple LangGraph nodes to mimic an agent's step-by-step behavior:

1. **Thought** — the model reasons about the user's request
2. **Action** — the model chooses a tool
3. **Observation** — the tool runs and returns output
4. **Final Answer** — the model uses the observation to produce a conclusion

This is foundational for building tool-using agents.

About the Technologies

You'll use:

Component	Description
`StateGraph`	Orchestrates the ReAct steps
`TypedDict`	Defines structured state with all four ReAct stages
LLM Node	Thinks and decides on Action
Tool Executor Node	Parses and runs the tool
Final Answer Node	Uses the observation to generate a final answer

You'll also use **regex** parsing to extract tool names and parameters — just like in your LangChain agent examples.

Instructions

1. Create a new file, **react_langgraph.py**
2. Write the code:

```
import re
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama

# 1. Define the full ReAct state structure
class ReActState(TypedDict):
    input: str
    thought: str
    action: str
    observation: str
    final_answer: str
```

```
# 2. Define a simple calculator tool
def calculator(expression: str) -> str:
    try:
        return str(eval(expression))
    except Exception as e:
        return f"Error: {e}"

# 3. Load the model
llm = ChatOllama(model="llama3", temperature=0)

# 4. Node: Ask for Thought and Action
def planner_node(state: ReActState) -> ReActState:
    prompt = f"""You are a helpful assistant using the ReAct pattern.
Always respond with the following format — use these labels **exactly**:

Thought: <your reasoning here>
Action: calculator[math expression here]

For example:
Thought: I need to add 2 and 2.
Action: calculator[2 + 2]

Now respond to this question:
Question: {state['input']}
"""

    response = llm.invoke(prompt).content
    thought_match = re.search(r"Thought:\s*(.*)", response)
    action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)

    return {
        "input": state["input"],
        "thought": thought_match.group(1).strip() if thought_match else "",
        "action": action_match.group(0).strip() if action_match else "",
        "observation": "",
        "final_answer": "",
    }

# 5. Node: Execute the tool
def action_node(state: ReActState) -> ReActState:
    match = re.search(r"(\w+)\[(.*?)\]", state["action"])
    if match:
        tool, param = match.groups()
        if tool == "calculator":
            result = calculator(param)
        else:
            result = f"Unknown tool: {tool}"
    else:
        result = "Invalid action format."

    return {
```

```
        **state,
        "observation": result
    }

# 6. Node: Generate the final answer
def final_node(state: ReActState) -> ReActState:
    prompt = f"""You previously reasoned:

{state['thought']}

You used this tool:
{state['action']}

And got this observation:
{state['observation']}

Now respond with your final answer.
Final Answer:"""

    result = llm.invoke(prompt).content
    return {
        **state,
        "final_answer": result.strip()
    }

# 7. Build the graph
builder = StateGraph(ReActState)
builder.add_node("plan", planner_node)
builder.add_node("act", action_node)
builder.add_node("final", final_node)
builder.set_entry_point("plan")
builder.add_edge("plan", "act")
builder.add_edge("act", "final")
builder.set_finish_point("final")
graph = builder.compile()

# 8. Run it
while True:
    user_input = input("\nAsk a math question (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = graph.invoke({
        "input": user_input,
        "thought": "",
        "action": "",
        "observation": "",
        "final_answer": ""
    })

    print("\n--- ReAct Breakdown ---")
    print("Thought:", result["thought"])
```

```
    print("Action:", result["action"])
    print("Observation:", result["observation"])
    print("Final Answer:", result["final_answer"])
```

3. Run the program

```
python react_langgraph.py
```

Example output/interaction:

```
Ask a math question (or 'exit'): What is (25 + 5) * 3?

--- ReAct Breakdown ---
Thought: To evaluate the expression, I'll follow the order of operations
(PEMDAS). First, I'll calculate the sum inside the parentheses, then multiply
by 3.
Action: Action: calculator[(25 + 5) * 3]
Observation: 90
Final Answer: I'm glad you asked!

Based on my previous calculation using PEMDAS:

1. Evaluate the expression inside the parentheses: 25 + 5 = 30
2. Multiply the result by 3: 30 * 3 = 90

So, my final answer is:

Final Answer: 90
```

Note: The model sometimes echoes the label "Action:" redundantly. This does not affect functionality but may be cleaned in post-processing.

Additional Notes and Explanations

- **planner_node**
  - Generates both Thought: and Action:
  - Uses regex to extract both pieces cleanly
- **action_node**
  - Runs the tool using regex to pull tool[param]
  - Right now supports only calculator[...]
- **final_node**
  - Reconstructs full reasoning trail
  - Prompts the model to conclude based on observation

Node	Purpose

`plan`	Think + Decide on tool
`act`	Execute tool
`final`	Conclude

This mimics real agent frameworks and lets you **plug in different tools, fallback paths, or error handling** in the future.

The planner → tool executor → summarizer/finalizer is a common graph pattern, especially for agent-style tasks. This is widely used since it mirrors human logic:

1. **Think** (what do I need to do?)
2. **Do** (run the action/tool)
3. **Report** (summarize what happened)

This is often seen in:

- ReAct agents
- Tool-using chains
- LLM + memory + tool workflows

But LangGraph is *fully flexible*, so, you don't *have* to use this format — you define the nodes based on your use case and what state you want to evolve.

Agents and Modular Design in LangGraph

LangGraph *itself* does not provide a built-in Agent class like LangChain's `AgentExecutor`. Instead, it encourages you to define agentic behavior using node flows, where each node represents part of a larger reasoning or action pipeline.

In this tutorial, the nodes plan, act, and final collectively form an agent-like behavior, even though there's no formal Agent object involved. This pattern mirrors how many ReAct agents function internally — they reason, choose a tool, use it, and generate a final answer — and LangGraph gives you full control over this flow.

It is ***common practice*** for developers to encapsulate node logic and subgraph construction into a custom class or function, often called something like `MathAgent`, `SearchAgent`, or `ToolAgent`. These encapsulations make it easier to:

- reuse agent flows across projects,
- plug one graph into another as a subgraph or node,
- test and maintain individual agent behaviors cleanly.

For example, you might wrap the logic in this tutorial into a `ReActAgent` class with a `.run()` method that invokes the graph. That class becomes your conceptual "agent," even though it's just organizing the LangGraph machinery under the hood.

The bottom line is this: In LangGraph, agents are *a pattern*, not a class. You implement agentic logic using nodes, and you can modularize it into an "agent" any way that suits your architecture.

This design offers more flexibility than traditional agent frameworks, letting you create highly customized agent flows tailored to your application's state, tools, and routing needs.

Tutorial 4: Multi-Turn Chat with a Loop

Goal

Refactor tool logic (e.g., calculator) into a centralized, reusable **Tool Executor Node**, separating tool parsing and execution from your agent's reasoning logic. This will:

- Make your graphs cleaner and more scalable
- Allow new tools to be added easily
- Mirror real agent frameworks like AgentCore

About the Technologies

In this tutorial, you'll improve your ReAct graph by creating a **tool execution node** that can:

- Parse the Action: tool[param] output
- Check for tool existence
- Run the correct function or return an error

Component	Description
Central tool registry	A dictionary mapping tool names to functions
Tool executor node	Handles parsing and execution
`StateGraph`	Same 3-stage ReAct graph, but with cleaner separation
`calculator()`	Your sample tool (same as before)

This will help prep for multi-agent graphs and complex orchestration later.

Instructions

1. Create a new file, **react_tool_executor.py**.
2. Write the code

```
import re
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama

# 1. Define the state structure
class ReActState(TypedDict):
    input: str
    thought: str
    action: str
    observation: str
```

```
    final_answer: str

# 2. Define tool functions
def calculator(expression: str) -> str:
    try:
        return str(eval(expression))
    except Exception as e:
        return f"Error: {e}"

tools = {
    "calculator": calculator
}

# 3. Load the model
llm = ChatOllama(model="llama3", temperature=0)

# 4. Node: Plan with Thought + Action
def planner_node(state: ReActState) -> ReActState:
    prompt = f"""You are a helpful assistant using the ReAct pattern.
You must solve the entire question using a single Action.

Use this format exactly:
Thought: <your full reasoning>
Action: calculator[entire expression]

Example:
Thought: I need to divide 50 by the result of 9+ 1.
Action: calculator[50 / (9 + 1)]

Question: {state['input']}
"""

    response = llm.invoke(prompt).content
    print("DEBUG planner response:", response)

    thought_match = re.search(r"Thought:\s*(.*)", response)
    action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)

    return {
        "input": state["input"],
        "thought": thought_match.group(1).strip() if thought_match else "[No
Thought]",
        "action": action_match.group(0).strip() if action_match else "[No
Action]",
        "observation": "",
        "final_answer": "",
    }

# 5. Node: Centralized tool executor
def tool_executor_node(state: ReActState) -> ReActState:
    action_str = state["action"]
    match = re.search(r"(\w+)\[(.*?)\]", action_str)
```

```
    if not match:
        return {**state, "observation": "Invalid action format."}

    tool, param = match.groups()
    tool = tool.strip()
    param = param.strip()

    if tool not in tools:
        return {**state, "observation": f"Unknown tool: {tool}"}

    try:
        result = tools[tool](param)
        return {**state, "observation": result}
    except Exception as e:
        return {**state, "observation": f"Error executing tool: {e}"}

# 6. Node: Final summary
def final_node(state: ReActState) -> ReActState:
    prompt = f"""You previously reasoned:

{state['thought']}

You used this tool:
{state['action']}

You observed:
{state['observation']}

Now respond with your final answer.
Final Answer:"""

    result = llm.invoke(prompt).content
    cleaned = result.strip()
    if cleaned.lower().startswith("final answer:"):
        cleaned = cleaned[len("final answer:"):].strip()

    return {**state, "final_answer": cleaned}

# 7. Build the graph
builder = StateGraph(ReActState)
builder.add_node("plan", planner_node)
builder.add_node("run_tool", tool_executor_node)
builder.add_node("final", final_node)

builder.set_entry_point("plan")
builder.add_edge("plan", "run_tool")
builder.add_edge("run_tool", "final")
builder.set_finish_point("final")

graph = builder.compile()
```

```
# 8. Run it
while True:
    user_input = input("\nAsk a math question (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = graph.invoke({
        "input": user_input,
        "thought": "",
        "action": "",
        "observation": "",
        "final_answer": ""
    })

    print("\n--- ReAct Breakdown ---")
    print("Thought:", result["thought"])
    print("Action:", result["action"])
    print("Observation:", result["observation"])
    print("Final Answer:", result["final_answer"])
```

3. Run the program

```
python react_tool_executor.py
```

Additional Notes and Explanations

Component	Role
planner_node	Model generates Thought + Action
tool_executor_node	Parses and executes tools based on action
final_node	Concludes based on observation
tools dict	Stores callable functions by name

Tutorial 5: Creating a Reusable ReAct Agent Class

Goal

Encapsulate your ReAct agent into a modular, reusable class (MathAgent) that can be instantiated and invoked like a component. You'll structure the code across two files:

- math_agent.py – contains the MathAgent class
- run_math_agent.py – imports and runs the agent interactively

This mirrors real-world practice where agents are isolated, reusable components that can be plugged into larger multi-agent graphs.

About the Technologies

This tutorial applies:

- Modular software engineering
- LangGraph composition
- State-typed agents
- Custom method interfaces (.run() or .invoke())

Instructions

1. Create a new file, **math_agent.py**
2. Write the following code:

```
# math_agent.py

import re
from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama

# 1. Define shared state type
class ReActState(TypedDict):
    input: str
    thought: str
    action: str
    observation: str
    final_answer: str

# 2. Sample calculator tool
def calculator(expression: str) -> str:
    try:
        return str(eval(expression))
    except Exception as e:
        return f"Error: {e}"
```

```
# 3. Tools registry
tools = {
    "calculator": calculator
}

# 4. Reusable MathAgent class
class MathAgent:
    def __init__(self):
        self.llm = ChatOllama(model="llama3", temperature=0)
        self.graph = self._build_graph()

    def _build_graph(self):
        builder = StateGraph(ReActState)

        builder.add_node("plan", self._planner_node)
        builder.add_node("run_tool", self._tool_executor_node)
        builder.add_node("final", self._final_node)

        builder.set_entry_point("plan")
        builder.add_edge("plan", "run_tool")
        builder.add_edge("run_tool", "final")
        builder.set_finish_point("final")

        return builder.compile()

    def _planner_node(self, state: ReActState) -> ReActState:
        prompt = f"""You are a helpful assistant using the ReAct pattern.
You must solve the entire question using a single Action.

Format:
Thought: <reasoning>
Action: calculator[expression]

Example:
Thought: I need to divide 100 by the result of 4 + 1.
Action: calculator[100 / (4 + 1)]

Question: {state['input']}
"""
        response = self.llm.invoke(prompt).content
        print("DEBUG planner response:", response)

        thought_match = re.search(r"Thought:\s*(.*)", response)
        action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)

        return {
            "input": state["input"],
            "thought": thought_match.group(1).strip() if thought_match else
"[No Thought]",
            "action": action_match.group(0).strip() if action_match else "[No
Action]",
            "observation": "",
```

```
            "final_answer": "",
        }

    def _tool_executor_node(self, state: ReActState) -> ReActState:
        match = re.search(r"(\w+)\[(.*?)\]", state["action"])
        if not match:
            return {**state, "observation": "Invalid action format."}

        tool, param = match.groups()
        tool = tool.strip()
        param = param.strip()

        if tool not in tools:
            return {**state, "observation": f"Unknown tool: {tool}"}

        try:
            result = tools[tool](param)
            return {**state, "observation": result}
        except Exception as e:
            return {**state, "observation": f"Error executing tool: {e}"}

    def _final_node(self, state: ReActState) -> ReActState:
        prompt = f"""The original user question was: {state['input']}

You reasoned:
{state['thought']}

You used this tool:
{state['action']}

You observed:
{state['observation']}

Now provide your final answer.
Final Answer:"""

        result = self.llm.invoke(prompt).content.strip()

        if result.lower().startswith("final answer:"):
            result = result[len("final answer:"):].strip()

        return {**state, "final_answer": result}

    def run(self, user_input: str) -> ReActState:
        return self.graph.invoke({
            "input": user_input,
            "thought": "",
            "action": "",
            "observation": "",
            "final_answer": ""
        })
```

3. Create another file, **run_math_agent.py**
4. Write the code:

```
# run_math_agent.py

from math_agent import MathAgent

agent = MathAgent()

while True:
    user_input = input("\nAsk a math question (or 'exit'): ")
    if user_input.lower() == "exit":
        break

    result = agent.run(user_input)

    print("\n--- ReAct Breakdown ---")
    print("Thought:", result["thought"])
    print("Action:", result["action"])
    print("Observation:", result["observation"])
    print("Final Answer:", result["final_answer"])
```

5. Run the program

```
python run_math_agent.py
```

Test with something like:

```
Ask a math question (or 'exit'): What is (12 + 6) * 2?
```

Additional Notes and Explanations

File	Role
math_agent.py	Contains `MathAgent` class with node logic + tool map
run_math_agent.py	Provides a clean interactive interface using the agent

Why Modularize Into a Class?

- **Encapsulation:** Everything the agent needs is inside the class
- **Reusability:** You can instantiate multiple agents (e.g., `MathAgent`, `InfoAgent`)
- **Testability:** Each method (node) is testable in isolation
- **Composition:** You can plug this into a larger LangGraph later as a single node

Real-World Benefit

This mirrors how modern LangGraph and LangChain systems are structured — not as flat files, but as modular, testable agent units that are composable in larger orchestration systems.

Tutorial 6: Routing Between Agents

Goal

Create a LangGraph that can route user questions to different agents — for example, a `MathAgent` for math queries and an `InfoAgent` for fact-style queries.

You'll use:

- A **router node** that decides which agent to invoke
- Two separate agent classes (MathAgent, InfoAgent)
- A simple **keyword-based routing strategy**

This structure mimics real-world multi-agent systems.

About the Technologies

Component	Description
Router Node	Determines which sub-agent to invoke based on input
Agent Class (`MathAgent`)	Encapsulates planning, tool execution, and finalization
Subgraph execution	Each agent's internal graph is invoked inside the router graph
LangGraph composition	Treats agent subgraphs as callable nodes or functions

Instructions

We'll be using math_agent.py from the previous tutorial (Tutorial 5). So, it must exist and work properly for this tutorial to work as well.

1. Create a new file, **info_agent.py**
2. Write the code:

```
# info_agent.py

from typing import TypedDict
from langgraph.graph import StateGraph
from langchain_ollama import ChatOllama

class InfoState(TypedDict):
    input: str
    thought: str
    final_answer: str

class InfoAgent:
    def __init__(self):
        self.llm = ChatOllama(model="llama3", temperature=0)
        self.graph = self._build_graph()
```

```
    def _build_graph(self):
        builder = StateGraph(InfoState)

        builder.add_node("think_and_answer", self._respond)
        builder.set_entry_point("think_and_answer")
        builder.set_finish_point("think_and_answer")

        return builder.compile()

    def _respond(self, state: InfoState) -> InfoState:
        prompt = f"""You are an information assistant.

Question: {state['input']}

Respond with:
Thought: <why you know this or how you'd look it up>
Final Answer: <concise, friendly fact-based answer>
"""
        response = self.llm.invoke(prompt).content
        lines = response.strip().split("Final Answer:")
        thought = lines[0].replace("Thought:", "").strip() if len(lines) > 0
else ""
        final = lines[1].strip() if len(lines) > 1 else "[No Final Answer]"

        return {
            "input": state["input"],
            "thought": thought,
            "final_answer": final
        }

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

3. Create a new file, **router_agent.py**
4. Write the code:

```
# router_agent.py

from math_agent import MathAgent
from info_agent import InfoAgent

class RouterAgent:
    def __init__(self):
        self.math_agent = MathAgent()
        self.info_agent = InfoAgent()

    def route(self, user_input: str) -> str:
        lowered = user_input.lower()
```

```
        if any(keyword in lowered for keyword in ["calculate", "+", "-", "*",
"/", "what is", "square root", "evaluate"]):
            return "math"
        else:
            return "info"

    def run(self, user_input: str) -> dict:
        route = self.route(user_input)
        print(f"[Router] Routing to: {route}")

        if route == "math":
            result = self.math_agent.run(user_input)
            return {
                "agent": "MathAgent",
                "thought": result["thought"],
                "action": result["action"],
                "observation": result["observation"],
                "final_answer": result["final_answer"]
            }
        else:
            result = self.info_agent.run(user_input)
            return {
                "agent": "InfoAgent",
                "thought": result["thought"],
                "final_answer": result["final_answer"]
            }
```

5. And, let's create another file, **run_router.py**
6. Write the code:

```
# run_router.py

from router_agent import RouterAgent

router = RouterAgent()

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

    result = router.run(user_input)

    print(f"\n--- Routed to: {result['agent']} ---")
    print("Thought:", result.get("thought", "[None]"))
    if result['agent'] == "MathAgent":
        print("Action:", result["action"])
        print("Observation:", result["observation"])
    print("Final Answer:", result["final_answer"])
```

7. Run the program

```
python run_router.py
```

Try with input like:

```
What is (12 + 4) * 2?
```

And also:

```
Tell me a fact about octopuses.
```

Additional Notes and Explanations

File	Role
math_agent.py	ReAct-based agent for calculator tasks
info_agent.py	LLM-only fact agent
router_agent.py	Routes between agents based on simple keyword logic
run_router.py	CLI frontend to interact with the system

Why Use Keyword Routing?

- It's simple to implement
- Easy to extend or debug
- Serves as a placeholder for future classification-based routing

Note also that you could:

- Replace it with embedding-based routing
- Use a dedicated **LLM classifier node**
- Build more sophisticated dispatch logic inside LangGraph itself

Recap: Tutorials 1-6

Summary of Tutorials

Tutorial #	Title	Key Concepts Covered	Skills Gained
1	Hello LangGraph	`StateGraph`, `TypedDict`, `add_node`, `invoke`	Define and run a minimal LangGraph with input/output state
2	LangGraph with State	State mutation, counters, node chaining	Maintain and mutate internal state (e.g., conversation turn count)
3	ReAct in LangGraph	ReAct pattern, regex parsing, stateful reasoning	Implement agent-like logic using plan → act → observe → respond
4	Modularizing Tool Execution	Central tool registry, reusable tool executor node	Cleanly separate planning vs tool execution in LangGraph
5	Creating a Reusable Agent Class	Class-based agent, internal graph encapsulation	Build a testable, modular `MathAgent` with internal node logic
6	Routing Between Agents	Simple router, multi-agent design, keyword dispatch	Build a router agent that delegates input to either `MathAgent` or `InfoAgent`

Summary of LangChain Classes and Components

Class / Component	Source	Purpose
`StateGraph(...)`	`langgraph.graph`	Core class for defining and compiling directed graphs of stateful nodes
`add_node(name, fn)`	`StateGraph`	Registers a node (function) by name in the graph
`add_edge("from", "to")`	`StateGraph`	Specifies a directional transition between nodes
`set_entry_point(name)`	`StateGraph`	Declares which node begins the graph
`set_finish_point(name)`	`StateGraph`	Declares which node terminates the graph
`TypedDict`	`typing`	Declares the structure of state dictionaries flowing through the graph

`ChatOllama`	`langchain_ollama`	Wraps a local Ollama-hosted model as an LLM interface for inference
`graph.invoke(input_dict)`	`CompiledGraph`	Runs the graph with a given state input and returns the final output
`re.search(...)`	Python Standard Library	Parses tool names and parameters from model outputs
`tools = {name: fn}`	User-defined	Registry for mapping tool names to their callable Python functions
`AgentClass.run()`	User-defined	Encapsulates graph invocation with pre-built node and state logic

Tutorial 7: Adding a Retrieval-Augmented Generation Tool to InfoAgent

Goal

Enhance your `InfoAgent` by adding a `doc_search` tool that performs vector-based document retrieval using **embeddings + FAISS**. This will allow your agent to answer questions based on custom documents.

You will:

- Add a document (e.g., `company_guide.txt`)
- Embed it using `nomic-embed-text` and FAISS
- Add a `doc_search[query]` tool to your `InfoAgent`

About the Technologies

Component	Description
`TextLoader`	Loads a local .txt file
`RecursiveCharacterTextSplitter`	Breaks it into chunks
`OllamaEmbeddings`	Creates vector representations (embeddings)
`FAISS`	Stores and retrieves vectors
`doc_search` tool	Performs similarity search over vectorstore

Instructions

1. If you haven't already, you may need to install some of these:

```
pip install langchain langchain-community langchain-ollama langchain-text-splitters faiss-cpu
```

2. Pull the nomic-embed-text if you haven't already:

```
ollama pull nomic-embed-text
```

3. Create a file, **company_guide.txt**

```
Section 1: Employee Benefits
Our company offers comprehensive health insurance including medical, dental,
and vision coverage.
Employees are eligible after 30 days of employment.
We also provide a 401(k) plan with a 4% employer match after 6 months.
```

```
Section 2: Vacation and PTO
Full-time employees accrue 15 days of paid vacation per year.
Unused PTO may roll over up to 5 days into the next calendar year.
Sick leave is tracked separately and accrues at 1 day per month.

Section 3: Remote Work Policy
Employees are allowed to work remotely up to 3 days per week with manager
approval.
Fully remote positions are available in engineering, design, and support teams.
Remote workers must attend mandatory quarterly on-site meetings.
```

4. Create **tools.py**

```
# tools.py

from langchain_community.document_loaders import TextLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_ollama import OllamaEmbeddings
from langchain_community.vectorstores import FAISS

# Load and embed the document
loader = TextLoader("company_guide.txt")
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk_size=300, chunk_overlap=50)
chunks = splitter.split_documents(docs)

embeddings = OllamaEmbeddings(model="nomic-embed-text")
vectorstore = FAISS.from_documents(chunks, embeddings)

# RAG-style tool function
def doc_search(query: str) -> str:
    results = vectorstore.similarity_search(query, k=1)
    return results[0].page_content if results else "No relevant information
found."
```

5. Inside **info_agent.py**, update `InfoAgent` to include `doc_search` tool:

```
from tools import doc_search
```

6. Add a tool registry in `InfoAgent.__init__()`:

```
self.tools = {
    "doc_search": doc_search
}
```

7. Replace the **_respond()** method with a ReAct-style node:

```
import re
```

```
def _respond(self, state: InfoState) -> InfoState:
    prompt = f"""You are an information assistant.

You can answer questions using:
- doc_search[query] → for company-specific information

Use this format exactly:
Thought: ...
Action: tool_name[parameter]

Example:
Thought: I need to check the remote work policy.
Action: doc_search[remote work policy]

Question: {state['input']}
"""

    response = self.llm.invoke(prompt).content
    print("DEBUG info planner response:", response)

    thought_match = re.search(r"Thought:\s*(.*)", response)
    action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)

    if not action_match:
        return {
            "input": state["input"],
            "thought": "[No thought]",
            "final_answer": "Invalid tool format."
        }

    tool_name, param = action_match.groups()

    if tool_name not in self.tools:
        return {
            "input": state["input"],
            "thought": thought_match.group(1).strip() if thought_match else
"[No thought]",
            "final_answer": f"Unknown tool: {tool_name}"
        }

    observation = self.tools[tool_name](param)

    final_prompt = f"""You were asked: {state['input']}

You reasoned:
{thought_match.group(1).strip() if thought_match else ''}

You used:
Action: {tool_name}[{param}]
Observation: {observation}

Now provide a final answer.
```

```
Final Answer:"""

    final = self.llm.invoke(final_prompt).content.strip()
    return {
        "input": state["input"],
        "thought": thought_match.group(1).strip() if thought_match else "[No
thought]",
        "final_answer": final
    }
```

8. Update **router_agent.py**, replacing the route() method with the following:

```
def route(self, user_input: str) -> str:
    lowered = user_input.lower()

    math_keywords = ["calculate", "+", "-", "*", "/", "evaluate", "math", "solve",
"equation"]
    info_keywords = ["policy", "benefits", "pto", "company", "fact", "tell me",
"explain"]

    if any(kw in lowered for kw in math_keywords):
        return "math"
    elif any(kw in lowered for kw in info_keywords):
        return "info"
    else:
        return "info"  # default fallback
```

9. Run the program

```
python run_router.py
```

10. Try questions like:

```
What is the remote work policy?
Tell me about employee benefits.
```

Additional Notes and Explanations

Component	**Role**
`doc_search`	Tool to retrieve most relevant document chunk
FAISS	In-memory vector store to support similarity search
Embeddings	Represent text semantically for comparison
Updated `InfoAgent`	Now behaves like a tool-using agent

Why Add RAG to InfoAgent?

- Provides **knowledge grounding** from internal docs
- Simulates company-specific chatbots or assistants
- Separates public knowledge (LLM) vs. private facts (RAG)

Prompt Engineering Reminder

Make sure your ReAct prompts:

- Clearly distinguish tool usage (tool[param])
- Forbid hallucination
- Encourage full answers only *after* using the tool

Tutorial 8: Error Handling and Fallbacks in LangGraph Agents

Goal

Add robust error handling to your **LangGraph-based agents**, especially MathAgent and InfoAgent, so they can:

- Handle malformed tool calls
- Detect missing or misformatted actions
- Respond with helpful fallback messages

About the Technologies

Component / Concept	Description
`re.search(...)`	Used to flexibly parse tool and parameter combinations (even if slightly malformed)
Regex fallback parsing	Supports patterns like `Action: calculator[...]` or `Action: [calculator ...]`
Tool existence checks	Prevents calling unregistered tools and returns a human-readable error
Observation field fallback	Handles failed tool calls gracefully without crashing the graph
Final node filtering	Intercepts tool failures and prevents bad output from propagating

Instructions

1. Modify the **_planner_node** in **math_agent.py**

Find:

```
action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)
```

Replace it with:

```
action_match =
re.search(r"Action:\s*(?:([\w_]+)\[(.*?)\]|\[([\w_]+)\s+(.*?)\])", response)
```

2. Now, still inside _planner_node (inside the return object):

Find:

```
"action": action_match.group(0).strip() if action_match else "[No Action]",
```

Replace with this:

```
"action": (
    f"{(action_match.group(1) or
action_match.group(3))}[{(action_match.group(2) or
action_match.group(4)).strip()}]"
    if action_match else "[No Action]"
)
```

3. Replace the full _tool_executor_node() function with the following:

```
def _tool_executor_node(self, state: ReActState) -> ReActState:
    match = re.search(r"(\w+)\[(.*?)\]", state["action"])
    if not match:
        return {**state, "observation": "Invalid action format. Use:
tool[param]"}

    tool, param = match.groups()
    tool = tool.strip()
    param = param.strip()

    if tool not in tools:
        return {
            **state,
            "observation": f"Unknown tool: '{tool}'. Available tools: {',
'.join(tools.keys())}"
        }

    try:
        result = tools[tool](param)
        return {**state, "observation": result}
    except Exception as e:
        return {**state, "observation": f"Tool error: {e}"}
```

4. Modify **info_agent.py**, inside **_respond()**

Find:

```
action_match = re.search(r"Action:\s*(\w+)\[(.*?)\]", response)
```

Replace it with:

```
action_match =
re.search(r"Action:\s*(?:([\w_]+)\[(.*?)\]|\[([\w_]+)\s+(.*?)\])", response)
```

5. Directly below the `action_match` code, insert the following code:

```
if action_match:
    tool_name = (action_match.group(1) or action_match.group(3)).strip()
    param = (action_match.group(2) or action_match.group(4)).strip()
else:
    tool_name, param = None, None
```

6. Run the program

```
python run_router.py
```

Try with the following types of tests:

Valid Math query

```
What is (12 + 4) * 2?
```

Expected:

- Routed to `MathAgent`
- Thought, Action, Observation, Final Answer all displayed

Valid Info query

```
Tell me about the vacation policy.
```

Expected:

- Routed to `InfoAgent`
- Should trigger `doc_search[vacation policy]` and return summary

Malformed Action Format (e.g., no Action line at all)

You can simulate this in development by modifying a prompt slightly to drop the Action line or test:

```
Answer this without any tools.
```

Expected:

- Returns: Invalid action or tool format. Use: tool[param]

Unknown Tool

You can force this via:

```
Action: madeup_tool[foo]
```

Expected:

- Returns: Unknown tool: 'madeup_tool'. Available tools: calculator (or `doc_search` for `InfoAgent`)

Additional Notes and Explanations

Why Flexible Parsing?

LLMs often format output inconsistently:

- Sometimes spacing is irregular (`tool [param]`)
- Sometimes brackets or colons are misplaced
- Occasionally the model switches to Action: [tool param]

By using a more forgiving regex in `_planner_node`, your system can handle a wider range of valid responses without crashing or misrouting the flow.

Why Validate Tool Use?

It's possible for models to:

- Hallucinate nonexistent tools
- Mistype registered tool names
- Send in non-numeric or bad inputs to tools like calculator

By checking:

```
if tool not in tools:
```

and wrapping the tool call in a try/except, you ensure:

- No runtime crashes
- Informative, user-facing feedback
- A more "resilient" agent that degrades gracefully under error