LangGraph Ultimate Cheat Sheet

Stateful, Multi-Actor Applications with LLMs

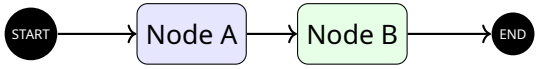
1 Core Concepts

StateGraph
The core container. It explicitly defines the state schema that all nodes access and modify.

Nodes
Python functions that perform work (LLM calls, tool execution). They receive the current state and return an update.

Edges
Define the control flow.

- **Normal**: Fixed transition (A → B).
- **Conditional**: Dynamic transition based on state (Router).



2 Defining State

State is usually a `TypedDict` *or Pydantic model.*

Basic State

```python
from typing import TypedDict

class State(TypedDict):
    # Overwrites value on update
    input: str
    output: str
```

Message State (Chat) *Use* `add_messages` *reducer to append history rather than overwrite.*

```python
from typing import Annotated
from langgraph.graph.message import \
    add_messages

class AgentState(TypedDict):
    # Appends new messages to list
    messages: Annotated[list, add_messages]
```

3 Building Graphs

1. Define Nodes

```python
def chatbot(state: AgentState):
    return {"messages": ["Hello!"]}

def tool_node(state: AgentState):
    # Execute tool logic...
    return {"messages": ["Tool Result"]}
```

2. Construct Graph

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

builder = StateGraph(AgentState)

# Add Nodes
builder.add_node("bot", chatbot)
builder.add_node("tools", tool_node)

# Add Edges (Linear)
builder.add_edge(START, "bot")
builder.add_edge("tools", "bot")
# Cycle: tools -> bot
```

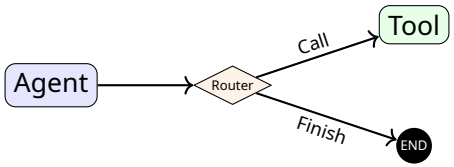
3. Compile

```python
graph = builder.compile()

# Invoke
res = graph.invoke(
    {"messages": [("user", "Hi")]}
)
```

4 Conditional Edges

Dynamic routing (e.g., "Should I call a tool or end?").



The Router Function

```python
def route_tools(state: AgentState):
    last_msg = state["messages"][-1]
    # Check if LLM wants to use tool
    if last_msg.tool_calls:
        return "tools"
    return END
```

Adding Conditional Edge

```python
builder.add_conditional_edges(
    "bot",          # Source node
    route_tools,    # Decision function
    # Path map (Optional if names match)
    {"tools": "tools", END: END}
)
```

5 Persistence (Memory)

Allows the graph to remember state across interactions (Checkpointing).

Setup Checkpointer

```python
from langgraph.checkpoint.memory \
    import MemorySaver

memory = MemorySaver()

# Compile with checkpointer
graph = builder.compile(checkpointer=memory)
```

Threaded Execution *Use* `thread_id` *to isolate user sessions.*

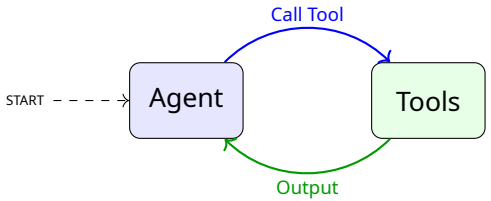
```python
config = {"configurable":
          {"thread_id": "user_1"}}

# First turn
graph.invoke(
    {"messages": [("user", "Hi")]},
    config=config
)

# Second turn (remembers context)
graph.invoke(
    {"messages": [("user", "My name is...")]},
    config=config
)
```

6 Prebuilt Agents

LangGraph provides a prebuilt ReAct agent pattern.



create_react_agent

```python
from langgraph.prebuilt import \
    create_react_agent
from langchain_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4o")
tools = [search_tool, math_tool]

# Creates node/edges automatically
agent = create_react_agent(
    model,
    tools=tools,
    checkpointer=memory
)

agent.invoke(
```

```
    {"messages": [("user", "Weather?")]},
    config=config
)
```

7 Human-in-the-loop

Pause execution for approval or input.

Interrupt Before

```
# Pause before entering 'tools' node
graph = builder.compile(
    checkpointer=memory,
    interrupt_before=["tools"]
)
```

Resuming Execution

```
# 1. Run until interrupt
graph.invoke(inputs, config=config)

# 2. Inspect State (Snapshot)
snapshot = graph.get_state(config)
print(snapshot.next) # ('tools',)

# 3. Resume (None = proceed as planned)
graph.invoke(None, config=config)
# OR Update state before resuming
# graph.update_state(config, ...)
```

8 Visualization

Generate an image of the graph structure.

```
from IPython.display import Image, display

display(
  Image(
    graph.get_graph().draw_mermaid_png()
  )
)
```