FEBRUARY 19, 2026 / #AI

How to Develop AI Agents Using LangGraph: A Practical Guide

MA Manoj Aggarwal

AI agents are all the rage these days. They're like traditional chatbots, but they have the ability to utilize a plethora of tools in the background. They can also decide which tool to use and when to use it to answer your questions.

In this tutorial, I'll show you how to build this type of agent using `LangGraph`. We'll dig into real code from my personal project FinanceGPT, an open-source financial assistant I created to help me with my finances.

You'll walk away understanding how AI agents actually work under the hood, and you'll be able to build your own agent for whatever

What I'll Cover:

Prerequisites

Before diving in, you should be comfortable with the following:

Python knowledge: You should know how to write Python functions, work with async/await syntax, and understand decorators. The code examples use all three extensively.

Basic LLM/chatbot familiarity: You don't need to be an expert, but knowing what a large language model is and having some experience calling one (via OpenAI's API or similar) will help you follow along.

LangChain basics: We'll be using LangGraph, which is built on top of LangChain. If you've never used LangChain before, it's worth skimming their quickstart guide first.

You'll also need the following tools installed:

- Python 3.10+
- An OpenAI API key (the examples use `gpt-4-turbo-preview`)
- The following packages, installable via pip:

If you're planning to follow along with the full FinanceGPT project rather than just the code snippets, you'll also want a PostgreSQL database set up, but that's optional for understanding the core concepts covered here.

What Are AI Agents?

Think of AI agents as traditional chatbots that can answer user questions. But they specialize in figuring out what tools they need and can chain multiple actions together to get an answer.

Here's an example conversation with my FinanceGPT AI agent:

```
User: "How much did I spend on groceries this month?"

Agent: [Thinks: I need transaction data filtered by category]

Agent: [Calls search_transactions(category="Groceries")]

Agent: [Gets back: $1,245.67 across 23 transactions]

Agent: "You spent $1,245.67 on groceries this month."
```

The agent broke down the problem, picked the right tool to use, and generated the answer. This matters a lot when you're working with messy real world problems where:

- Questions don't fit into specific categories
- You need to pull data from multiple sources
- Users want to ask followup questions

What is LangGraph?

`LangGraph` is an open sourced extension of `LangChain` that's useful for creating stateful AI agents by modeling workflows as nodes and edges in a graph. You can think of your agent's logic as a flowchart where:

- **Edges** are the arrows (what happens next)
- **State** is the information passed around

LangGraph is especially good at providing the following benefits:

1. **Flow control**: You define exactly what happens when.
2. **Stateful**: The framework preserves conversation history for you.
3. **Easy to use**: Just adding a decorator to an existing Python function makes it a tool.
4. **Production-ready**: It has built-in error handling and retries.

Core Concept 1: Tools

Think of tools as just Python functions your AI agent can call. The LLM utilizes the function name, docstring, parameters, and return value to know what the functions are doing and when to use them.

`LangChain` has a `@tool` decorator that can convert any function into a tool, for example:

```
from langchain_core.tools import tool

@tool
def get_current_weather(location: str) -> str:
    """Get the current weather for a location.

    Use this when the user asks about weather conditions.

    Args:
        location: City name (e.g., "San Francisco", "New York")

    Returns:
        Weather description string
    """
    # In real life, you'd call a weather API here
    return f"The weather in {location} is sunny, 72°F"
```

Notice that the docstring is self-explanatory, as that's how the LLM decides whether this tool is the right choice or not.

```
m langchain_core.tools import tool
m sqlalchemy.ext.asyncio import AsyncSession
m sqlalchemy import select

 create_search_transactions_tool(search_space_id: int, db_session: AsyncSes
 """
 Factory function that creates a search tool with database access.

 This pattern lets you inject dependencies (database, user context)
 while keeping the tool signature clean for the LLM.
 """

 @tool
 async def search_transactions(
     keywords: str | None = None,
     category: str | None = None
 ) -> dict:
     """Search financial transactions by merchant or category.

     Use when users ask about:
     - Spending at specific merchants ("How much at Starbucks?")
     - Spending in categories ("How much on groceries?")
     - Both combined ("Show me restaurant spending at McDonald's")

     Args:
         keywords: Merchant name to search for
         category: Spending category (e.g., "Groceries", "Gas")

     Returns:
         Dictionary with transactions, total amount, and count
     """
     # Query the database
     query = select(Document.document_metadata).where(
         Document.search_space_id == search_space_id
     )
     result = await db_session.execute(query)
     documents = result.all()

     # Filter transactions based on criteria
     all_transactions = []
     for (doc_metadata,) in documents:
         transactions = doc_metadata.get("financial_data", {}).get("transac

         for txn in transactions:
             # Apply filters
             if category and category.lower() not in str(txn.get("category"
                 continue
             if keywords and keywords.lower() not in txn.get("description",
                 continue

             # Include matching transaction
             all_transactions.append({
                 "date": txn.get("date"),
                 "description": txn.get("description"),
```

```
    # Calculate total and return
    total = sum(abs(t["amount"]) for t in all_transactions if t["amount"] <

    return {
        "transactions": all_transactions[:20],  # Limit results
        "total_amount": total,
        "count": len(all_transactions),
        "summary": f"Found {len(all_transactions)} transactions totaling $
    }

return search_transactions
```

Let's dive into what this code is doing.

The factory function pattern: The tool only takes parameters the LLM can provide (a keyword and category), but it also needs a database session and `search_space_id` to know whose data to query. The factory function solves this by capturing those dependencies in a closure, so the LLM sees a clean interface while the database wiring stays hidden.

The filtering logic: We loop through all transactions and apply the optional filters. If `category` is provided, it must appear in the transaction's category field. If `keywords` is provided, it must appear in the merchant description. Both can be used together, letting the LLM handle questions like "How much did I spend at McDonald's in the Restaurants category?"

The return value: Instead of a raw list, the tool returns a structured dict with a capped result set, a pre-calculated total, and a plain-English summary string. The summary means the LLM can read `"Found 23 transactions totaling $1,245.67"` and immediately know what to say, rather than parsing the raw data itself.

Key Tool Design Principles

These are the principles that differentiate a good tool from a great tool:

1. **Docstrings:** Instead of vague descriptions, you need to be thorough with the explanation of the tool in the docstring.

2. **Clean signature:** The tool should only take the parameters that the LLM has access to and can provide. If the tool needs user ids, or database connections (and so on), you can hide those in factory functions using closures.
3. **Return both data and summaries:** Instead of just the raw data, if you include a summary field, the agent can just use that to understand the output better. Here's an example:

```
{
    "transactions": [...],           # For detailed analysis
    "total_amount": 1245.67,         # Pre-calculated
    "summary": "Found 23 transactions..."  # Ready to send to u
}
```

4. **Limited context window:** Capping results to a finite amount like 20-50 items depending on the use case will make sure your LLM doesn't choke or hit context limits.

Core Concept 2: Agent State

Your agent carries around information as it works. This is called the agent's state. For a chatbot, it's usually the conversation history.

In `LangGraph`, state is defined with a `TypeDict`:

```
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage

class AgentState(TypedDict):
    """
    This is what flows through your agent.

    Messages is a list that keeps growing:
    - User questions
    - Agent responses
    - Tool results
    """
    messages: Annotated[Sequence[BaseMessage], "The conversation history"
```

```
class FancierState(TypedDict):
    messages: Sequence[BaseMessage]
    user_id: str
    retry_count: int
    last_tool_used: str | None
```

This matters more than it might look. Each field here has a real purpose in a sophisticated production-grade agent. `user_id` tells every node whose data to fetch without you having to pass it around manually. `retry_count` helps agent detect when its stuck in a loop so it can bail out gracefully. `last_tool_used` helps the agent avoid redundant calls.

As the agent grows in complexity, state becomes the single source of truth that keeps every node coordinated.

Why State Matters

State is what separates an agent which is conversational from an API call that is stateless. Without it, every message would be processed in isolation and the agent would have no recollection of what was asked earlier, what tools it already used, and what data it retrieved already.

With state, the full conversation history is passed through each step of the agent's execution.

Here's what that looks like in practice for our grocery spending example:

```
When the conversation starts:
{
    "messages": []
}

User asks something:
{
    "messages": [
        HumanMessage("How much did I spend on groceries?")
    ]
}

Agent decides to use a tool:
```

```
        ToolMessage({"total_amount": 1245.67, ...}),
    ]
}

Agent responds with the answer:
{
    "messages": [
        HumanMessage("How much did I spend on groceries?"),
        AIMessage(tool_calls=[...]),
        ToolMessage({...}),
        AIMessage("You spent $1,245.67 on groceries this month.")
    ]
}
```

Notice that the state is always growing with every tool call and every result. This means that when user has a followup like “How does that compare to last month?”, the agent can just look back and know what “that” refers to.

Core Concept 3: The Agent Graph

The graph is the backbone of your agent. Think of it as a collection of tools and an LLM, combined together to reason, act and respond in a structured way. Specifically, it determines the order of operations – that is, what runs first, what happens next, and what conditions determine which path to take.

Without a graph, you would have to manually orchestrate the workflow: calling the LLM, then checking whether it wants to use a tool, executing the tool, and then feeding the result back to it and deciding when to stop. The graph encodes this logic explicitly so that your agent figures out the right sequence.

Each node in the graph is an action like “ask the LLM” or “run a tool” and each edge is a connection between those actions.

With that in mind, let's build one step by step.

Step 1: Create the Agent Node

The agent node is where the LLM makes a decision like “Should I use a tool?” or “Which tool to use?”. Let’s take an example:

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholde

# Create the LLM with tools
llm = ChatOpenAI(model="gpt-4-turbo-preview", temperature=0)

# Create your tools
tools = [
    create_search_transactions_tool(search_space_id, db_session),
    # ... other tools
]

# Bind tools to the LLM so it knows what's available
llm_with_tools = llm.bind_tools(tools)

# Create the system prompt
system_prompt = """You are a helpful AI financial assistant.

Your capabilities:
- Search transactions by merchant, category, or date
- Analyze portfolio performance
- Find tax optimization opportunities

Guidelines:
- Be concise and cite specific data
- Format currency as $X,XXX.XX
- Remind users to consult professionals for tax/investment advice"""

prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    MessagesPlaceholder(variable_name="messages"),
])

# Define the agent node function
async def call_agent(state: AgentState):
    """
    The agent node calls the LLM to decide the next action.

    The LLM can:
    1. Call one or more tools
    2. Generate a text response
    3. Both
    """
    messages = state["messages"]

    # Format messages with system prompt
    formatted = prompt.format_messages(messages=messages)

    # Call the LLM
    response = await llm_with_tools.ainvoke(formatted)

    # Return state update (add the LLM's response)
    return {"messages": [response]}
```

model deterministic and consistent. This is important for an agent that needs to make reliable decisions rather than creative ones.

Next, we call `llm.bind_tools(tools)`. It tells the LLM what tools are available by passing along their names, descriptions, and parameter schemas. Without this, the LLM would have no idea it could call any tools at all. With it, the LLM can look at a user's question and decide both whether a tool is needed and which one to use.

The prompt is built using `ChatPromptTemplate`, which combines a static system prompt with a `MessagesPlaceholder`. The placeholder is where the full conversation history gets inserted at runtime, meaning the LLM always has the complete context of the conversation when making its decision.

Last, `call_agent` is the actual node function. It pulls the current messages from state, formats them with the prompt, calls the LLM, and returns the response to be appended to state. This is the function LangGraph will call every time execution reaches the agent node.

Step 2: Create the Tool Node

`LangGraph` has a pre-built `ToolNode` that executes tools:

```
from langgraph.prebuilt import ToolNode

# This node automatically executes any tools the LLM requested
tool_node = ToolNode(tools)
```

When the LLM includes tool calls in its response, `ToolNode` will:

1. extract the tool calls,
2. execute each tool with specific params, and
3. add `ToolMessage` object with the result to state

Step 3: Define Control Flow

```
from langgraph.graph import END

def should_continue(state: AgentState):
    """
    Router function that determines the next step.

    Returns:
        "tools" - if the LLM wants to use tools
        END - if the LLM is done (just text response)
    """
    last_message = state["messages"][-1]

    # Check if the LLM included tool calls
    if hasattr(last_message, "tool_calls") and last_message.tool_calls:
        return "tools"

    # No tool calls means we're done
    return END
```

This tiny function is the decision-maker of your entire agent. After the LLM responds, LangGraph calls `should_continue` to figure out what to do next. It works by inspecting the last message in state: the LLM's most recent response. If that response contains tool calls, it means the LLM has decided it needs more data before it can answer, so we return `"tools"` to route execution to the tool node. If there are no tool calls, the LLM has produced a final answer and we return `END` to stop execution.

This is the mechanism that makes the agent loop. The agent doesn't just call one tool and stop, but it can call a tool, see the result, decide it needs another tool, call that one too, and only stop when it has everything it needs to respond.

Step 4: Assemble the Graph

Now, we can connect everything:

```
from langgraph.graph import StateGraph

# Create the graph
workflow = StateGraph(AgentState)
```

```
# Set entry point
workflow.set_entry_point("agent")

# Add conditional edge from agent
workflow.add_conditional_edges(
    "agent",           # From this node
    should_continue,   # Use this function to decide
    {
        "tools": "tools",  # If "tools" is returned, go to tools node
        END: END           # If END is returned, finish
    }
)

# After tools execute, go back to agent
workflow.add_edge("tools", "agent")

# Compile into a runnable agent
agent = workflow.compile()
```

This is where everything gets wired together. We start by creating a `StateGraph` and passing it our `AgentState` type. This tells LangGraph what shape the state will take as it flows through the graph.

We then register our two nodes with `add_node`. The string name we give each node ("agent" and "tools") is what we'll use to reference them when defining edges. `set_entry_point` tells LangGraph where execution should begin which in our case is the agent node.

The conditional edge is where the routing logic plugs in. We're telling LangGraph: "After the agent node runs, call `should_continue` to decide what happens next, then use this mapping to translate that decision into the next node." If `should_continue` returns `"tools"`, go to the tools node. If it returns `END`, stop.

Finally, `add_edge("tools", "agent")` creates an unconditional edge: after the tools node runs, always go back to the agent node. This is what creates the loop, letting the agent review the tool results and decide whether it's done or needs to keep going. Calling `workflow.compile()` locks everything in and returns a runnable agent.

Understanding the Flow

Here's what happens when you run the agent:

```
    ↓
[AGENT NODE]
    ↓
[SHOULD_CONTINUE]
    ↓
  Tools needed?
    ↓ YES   ↓ NO
[TOOLS]    [END]
    ↓
[AGENT NODE]
    ↓
[SHOULD_CONTINUE]
    ↓
    ...
```

The loop above allows the agent to:

1. Use a tool
2. See the results
3. Decide if more tools are needed
4. Use more tools or generate final answer

How to Put it All Together

Let's see the complete agent in one place:

```python
from typing import Annotated, Sequence, TypedDict
from langchain_core.messages import BaseMessage, HumanMessage
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholde
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode

# 1. Define State
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], "Conversation history"]

# 2. Create Agent Function
def create_agent(tools):
    # Set up LLM
    llm = ChatOpenAI(model="gpt-4-turbo-preview", temperature=0)
    llm_with_tools = llm.bind_tools(tools)

    # Create prompt
    prompt = ChatPromptTemplate.from_messages([
        ("system", "You are a helpful AI assistant."),
```

```
    async def call_agent(state: AgentState):
        formatted = prompt.format_messages(messages=state["messages"])
        response = await llm_with_tools.ainvoke(formatted)
        return {"messages": [response]}

    def should_continue(state: AgentState):
        last_message = state["messages"][-1]
        if hasattr(last_message, "tool_calls") and last_message.tool_call
            return "tools"
        return END

    # Build graph
    workflow = StateGraph(AgentState)
    workflow.add_node("agent", call_agent)
    workflow.add_node("tools", ToolNode(tools))
    workflow.set_entry_point("agent")
    workflow.add_conditional_edges("agent", should_continue, {"tools": "t
    workflow.add_edge("tools", "agent")

    return workflow.compile()

# 3. Use the Agent
async def main():
    # Create tools (simplified example)
    tools = [create_search_transactions_tool(user_id=1, db_session=sessic

    # Create agent
    agent = create_agent(tools)

    # Run agent
    result = await agent.ainvoke({
        "messages": [HumanMessage(content="How much did I spend on grocer
    })

    # Get final response
    final_response = result["messages"][-1].content
    print(final_response)
```

How the Agent Thinks

Let’s use an example to see how the agent reasons.

Example: “How much did I spend on groceries this month?”

Step 1: User Input

```
State: {
    "messages": [HumanMessage("How much did I spend on groceries this mor
```

Step 2: Agent Node

The LLM gets:

- A system prompt, like the one we defined above
- User question: "How much did I spend on groceries this month?"
- List of available tools: `search_transactions(keywords, category)`

The LLM reasons that this is about spending in a specific category and decides that it should use `search_transactions` with `category='groceries'`. It responds with a tool call:

```
AIMessage(
    content="",
    tool_calls=[{
        "name": "search_transactions",
        "args": {"category": "Groceries"},
        "id": "call_123"
    }]
)
```

Step 3: Should Continue

The router sees tool calls and returns "tools".

Step 4: Tools Node

It executes `search_transactions(category="Groceries")` and gets:

```
{
    "transactions": [...],
    "total_amount": 1245.67,
    "count": 23,
    "summary": "Found 23 transactions totaling $1,245.67"
}
```

```
ToolMessage(
    content='{"transactions": [...], "total_amount": 1245.67, ...}',
    tool_call_id="call_123"
)
```

Step 5: Agent Node Again

The LLM now sees the user question, its previous tool, and the results. The LLM thinks: “I now have the data, the user spent $1245.67 on groceries. I can answer now.” And the LLM responds with:

```
AIMessage(content="You spent $1,245.67 on groceries this month across 23
```

Step 6: Should Continue

No tool calls this time, so returns END.

Final State:

```
{
    "messages": [
        HumanMessage("How much did I spend on groceries this month?"),
        AIMessage("", tool_calls=[...]),
        ToolMessage('{"total_amount": 1245.67, ...}'),
        AIMessage("You spent $1,245.67 on groceries this month across 23
    ]
}
```

The user receives: "You spent $1245.67 on groceries this month across 23 transactions."

Conclusion

Building an AI agent boils down to three ideas:

1. Tools

LangGraph gives you control, so you are not left hoping that the agent does the right thing – instead, you're explicitly defining what the "right thing" is.

The FinanceGPT example shows how this works in a real application. By learning these concepts, now you can build specialized agents for different jobs.

Resources Worth Checking Out

These helped me learn LangGraph:

- Official LangGraph docs: Start here
- LangGraph conceptual guide: Deeper theory
- LangChain agent patterns: Alternative approaches

Check Out FinanceGPT

All the code examples here came from FinanceGPT. If you want to see these patterns in a complete app, poke around the repo. It's got document processing, portfolio tracking, tax optimization – all built with LangGraph.

If you find this helpful, give the project a star on GitHub – it helps other developers discover it.

MA

Manoj Aggarwal

Read more posts.
