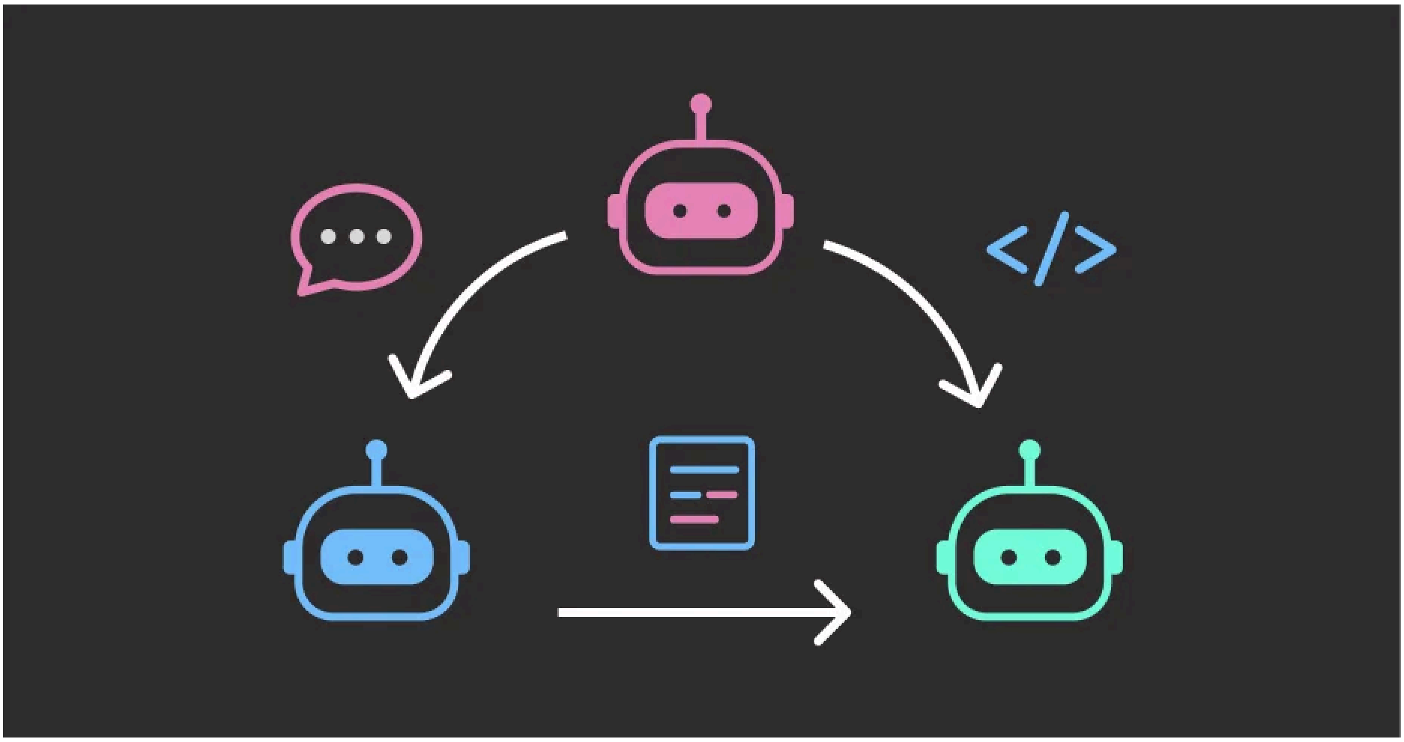
CodeCut

Search here..

Building Coordinated AI Agents with LangGraph: A Hands-On Tutorial



Building Coordinated AI Agents with LangGraph: A Hands-On Tutorial

0

Khuyen Tran Bex Tuychiev

Introduction

Have you ever noticed how a single chatbot can only analyze problems through one perspective at a time instead of weighing multiple viewpoints like humans do?

To demonstrate this, let's see how a single agent would analyze Apple's financial health:

```
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate

# Single agent approach
single_agent = ChatOpenAI(model="gpt-4")
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a financial analyst. Analyze the company's he
    ("user", "Analyze Apple's financial health considering growth, ri
])

# The agent tries to be both optimistic and pessimistic at once
response = single_agent.invoke(prompt.format_messages())
print(response.content)
```

The output might look like this:

```
Apple demonstrates a healthy financial status with consistent
```

There are several issues with this response:

1. Confirmation Bias: The response tends to be one-sided (in this case, overly positive) because there's no counter-balancing perspective. It's like having only one voice in a debate.
2. No Decision Making: The single bot only provides analysis without making any concrete decisions or recommendations.

This limited perspective makes it particularly challenging for complex tasks like investment analysis that require balancing different factors and perspectives.

LangGraph solves this by providing a framework for building coordinated multi-agent systems with specialized agents working together through structured communication.

Here's how the same analysis would look with a multi-agent system:

```
Key bull arguments centered on Apple's strong financials (record Q1 2

The bears countered with Apple's slowing annual growth, overreliance

The chairman ultimately decided on a HOLD/RESEARCH MORE position, bal
```

This example illustrates the structured debate process between specialized agents:

- The bull agent starts by making its investment case
- The bear agent responds by highlighting key risks
- The chairman carefully weighs both perspectives and makes a final decision

This multi-agent approach produces a more thorough and balanced analysis compared to a single agent trying to consider all angles at once.

In this article, we will explore how to build a multi-agent system with LangGraph. We will start with a simple example of an investment committee and then build a more complex system with multiple agent

Getting Started With LangGraph

Environment Setup

First, you should setup your environment with the following packages:

```
pip install langgraph langgraph-supervisor langchain langchain-core l
```

These packages are necessary for our investment committee system:

- `langgraph`: Core framework for building multi-agent systems with state management
- `langgraph-supervisor`: Provides the supervisor pattern for agent coordination
- `langchain` and `langchain-core`: Foundational components for LLM applications
- `langchain-tavily`: Integration with Tavily search API for market research
- `langchain-openai`: OpenAI model integrations
- `python-dotenv`: Environment variable management for API keys

Next, create a `.env` file and populate it with your API keys from OpenAI and Tavily (a search engine API).

```
TAVILY_API_KEY='your-api-key-goes-here'
OPENAI_API_KEY='your-api-key-goes-here'
```

Now, let's see how to create your first agent in LangGraph.

> *Note that going forward, familiarity with LangChain basics will be helpful.*

0

Creating Agents in LangGraph

LangGraph makes it very easy to create your first agent. Let's walk through how to create a general purpose assistant with web search functionality in a few lines of code.

First, we load our environment variables containing API keys using `load_dotenv()` and import necessary components:

```
from dotenv import load_dotenv

from langgraph.prebuilt import create_react_agent
from langchain_openai import ChatOpenAI
from langchain_tavily import TavilySearch

load_dotenv()
```

Next, initialize the search tool with `TavilySearch(max_results=3)`, which will return the top three search results for any query.

```
web_search = TavilySearch(max_results=3)
```

Create the agent using `create_react_agent()`, which implements the ReAct pattern, a framework that enables agents to reason about actions, execute them with tools, observe results, and plan next steps accordingly.

```
agent = create_react_agent(
    model=ChatOpenAI(model="gpt-4o"),
    tools=[web_search],
    prompt="You are a helpful assistant that can search the web for i
)
```

We configure the agent with the three core components:

- A language model (GPT-4o in this case)
- A list of tools that allows the agent to connect to external tools and APIs like a search engine
- A system prompt that defines the agent's role and behavior

Finally, we invoke the agent with a question about today's stock market activity:

```
response = agent.invoke(
    {
        "messages": [
            {
                "role": "user",
                "content": "Find the open and close prices of Apple's
            }
        ]
    }
)

print(response['messages'][-1].content)
```

```
June 1, 2025, was a Sunday, so financial markets were closed. The nex

- Opened at: $200.28
- Closed at: $201.70

You can verify this information on trusted sources such as Yahoo Fina

- Source: Yahoo Finance (historical data page)
- Source: Macrotrends Apple stock history

If you need data for the trading day immediately before June 2, 2025,
```

The output demonstrates the ReAct pattern in action:

- The agent reasons about what information it needs (stock prices for a specific date)
- Executes the task using the web search tool to find the information
- Observes the results and reasons again about their validity
- Plans next steps accordingly (in this case, providing the data f next trading day)

Creating a Supervisor Multi-Agent System

A supervisor is a special agent that manages the workflow between multiple agents. It is responsible for:

- Routing the workflow between agents
- Managing the conversation history
- Ensuring the agents are working together to achieve the goal

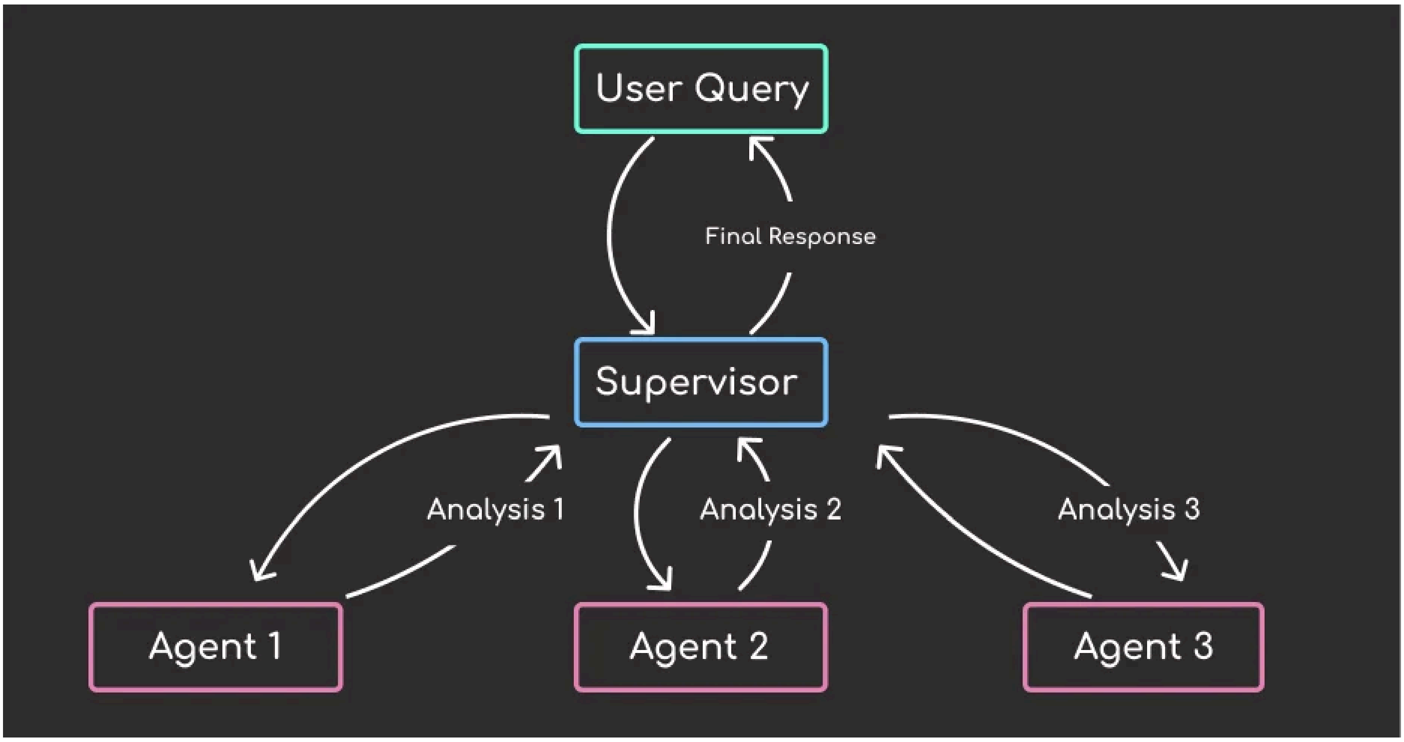
Let's create a supervisor multi-agent system with three agents:

```
from langgraph_supervisor import create_supervisor

# Define domain-specific agents with their own tools and prompts
agent1 = create_react_agent(...)
agent2 = create_react_agent(...)
agent3 = create_react_agent(...)

# Create a memory checkpointer to persist conversation history for th
memory = MemorySaver()

supervisor = create_supervisor(
    model=ChatOpenAI(model="o3"),
    agents=[agent1, agent2, agent3],
    prompt="Detailed system prompt instructing the model how to route
).compile(checkpointer=memory)

# Call the supervisor with a user query
response = supervisor.invoke({
    "messages": [
        {
            "role": "user",
            "content": "Your question here..."
        }
    ]
})
```

0

The supervisor multi-agent system follows a structured workflow for processing user queries. Here's how it works:

- User submits a query to the supervisor
- Supervisor routes the query to appropriate agents based on the system prompt
- Agents analyze the information independently and return their findings back to the supervisor
- Supervisor aggregates the findings and generates a final response
- The final response is returned to the user



Building The Investment Board Of Agents

Now that we've covered the basics of LangGraph, let's move on to explore the investment board of agents application.

High-level Application Overview

Here's a high-level overview of the application:

```
investment-committee/
├── src/
│   ├── __init__.py           # Package marker
```

0

```
│   ├── config.py           # Prompts and model configuration
│   ├── tools.py            # Agent tools organized by function
│   ├── utils.py            # Utility functions for display
│   └── agents.py           # Agent and supervisor creation
├── main.py                 # Command-line interface
├── requirements.txt        # Python dependencies
├── .env.example           # Environment variables template
└── README.md              # Documentation
```

In this structure:

- `src/config.py` contains the system configuration
- `src/tools.py` contains the tools for the agents
- `src/utils.py` contains the utility functions for the application
- `src/agents.py` contains the agent and supervisor creation
- `main.py` is the entry point for the command-line interface

Let's go through each of these files in more detail.

Setting up configuration

The `config.py` file centralizes all system configuration in one location, making the application easy to customize and maintain. The file contains two main types of configuration:

- Model configuration: Specifies which language model to use across all agents, which is currently set to `"openai:gpt-4"`.
- Agent prompts: Define the personality and behavior of each agent through detailed system prompts.

Here is the supervisor prompt:

```
SUPERVISOR_PROMPT = (
    "You are a SIMPLE ROUTER with one final summary task.\n\n"
    "MANDATORY WORKFLOW (follow exactly):\n"
    "1. bull_agent: Make initial bullish case\n"
    "2. bear_agent: Make initial bearish case\n"
    "3. bull_agent: Counter the bear's specific arguments\n"
    "4. bear_agent: Counter the bull's specific arguments\n"
```

```
        "5. chairman_agent: Make final investment decision\n"
        "6. YOU: Summarize the debate outcome\n\n"
        "RULES:\n"
        "- DO NOT summarize until AFTER chairman makes decision\n"
        "- ALWAYS end with chairman_agent making the decision first\n"
        "- Route agents in the exact order above\n"
        "- After chairman decides, provide a brief summary of:\n"
        "   • Key bull arguments\n"
        "   • Key bear arguments  \n"
        "   • Chairman's final decision and reasoning\n"
        "- Keep summary concise (3-4 sentences max)"
    )
```

Defining Tools

Tools are functions that agents can use to interact with external data sources and perform specific tasks. There are six tools used in this system:

- `find_positive_news`: Searches for positive news and developments about a stock
- `calculate_growth_potential`: Calculates basic growth metrics and bullish indicators
- `find_negative_news`: Searches for negative news and risks about a stock
- `assess_market_risks`: Assesses overall market risks and bearish indicators
- `get_current_market_sentiment`: Gets overall market sentiment and recent performance
- `make_investment_decision`: Makes final investment recommendation based on bull and bear arguments

Each tool uses the Tavily search API to gather real-time market information, ensuring agents base their arguments on current data rather than outdated training information.

Here is the code for the `find_positive_news` tool:

```
def find_positive_news(stock_symbol: str):
    """Search for positive news and developments about a stock""
```

```
    query = f"{stock_symbol} stock positive news earnings growth reve
    keywords = ["profit", "growth", "upgrade", "beat", "strong", "pos
    prefix = "🐂 POSITIVE SIGNALS"
    default = "Limited positive news found, but that could mean it's
    return search_and_extract_signals(stock_symbol, query, keywords,
```

View the full code for the tools in the tools.py file.

Agents are the core components of the multi-agent system. They are responsible for analyzing the information and making decisions.

The `agents.py` file creates the agents and supervisor. There are three specialized agents:

1. Bull agent: An optimistic analyst who searches for positive indicators and growth potential
2. Bear agent: A pessimistic analyst who identifies risks and negative signals
3. Chairman agent: A neutral decision-maker who weighs both sides and makes final investment recommendations

Investment committee diagram showing bull, bear, and chairman agents with their roles in the analysis process

Each agent gets created with the `create_react_agent` function with the following parameters:

- `model`: The language model to use
- `tools`: The tools to use
- `prompt`: The system prompt to use
- `name`: The name of the agent

Here is the code for the three agents:

```
def create_bull_agent():
    """Create the bull (optimistic) investment agent"""
    return create_react_agent(
```

```
        model=MODEL_NAME,
        tools=[find_positive_news, calculate_growth_potential],
        prompt=BULL_AGENT_PROMPT,
        name="bull_agent",
    )


def create_bear_agent():
    """Create the bear (pessimistic) investment agent"""
    return create_react_agent(
        model=MODEL_NAME,
        tools=[find_negative_news, assess_market_risks],
        prompt=BEAR_AGENT_PROMPT,
        name="bear_agent",
    )


def create_chairman_agent():
    """Create the chairman (decision maker) agent"""
    return create_react_agent(
        model=MODEL_NAME,
        tools=[get_current_market_sentiment, make_investment_decision
        prompt=CHAIRMAN_AGENT_PROMPT,
        name="chairman_agent",
    )
```

The supervisor combines all agents under coordinated workflow management. Here is the code for the supervisor:

```
def create_investment_supervisor():
    """Create the supervisor that manages the investment committee"""
    bull_agent = create_bull_agent()
    bear_agent = create_bear_agent()
    chairman_agent = create_chairman_agent()

    supervisor = create_supervisor(
        model=init_chat_model(MODEL_NAME),
```

```
        agents=[bull_agent, bear_agent, chairman_agent],
        prompt=SUPERVISOR_PROMPT,
        add_handoff_back_messages=True,
        output_mode="full_history",
    ).compile()

    return supervisor
```

In this code:

- `add_handoff_back_messages=True` ensures agents can reference each other's previous arguments, creating true conversational debate rather than isolated analysis.
- `output_mode="full_history"` provides complete conversation context, allowing you to see the full reasoning process rather than just final decisions.

Formatting the Output

To format the output, we use the `pretty_print_messages` function in `utils.py`. It takes the conversation stream updates and formats them into readable output by:

- Identifying which agent is speaking
- Processing supervisor workflow updates and agent interactions
- Converting message objects into a human-readable format

This allows us to clearly see the back-and-forth debate between agents as they analyze investment opportunities.

```
while True:
    try:
        stock_input = input("Enter stock symbol (or 'quit' to exit):

        if stock_input.lower() in ["quit", "exit", "q"]:
            print("\n👋 Goodbye! Happy investing!")
            break
```

```
        if not stock_input:
            print("Please enter a valid stock symbol.")
            continue

        analyze_stock(supervisor, stock_input)

    except KeyboardInterrupt:
        print("\n\n👋 Goodbye! Happy investing!")
        sys.exit(0)
    except Exception as e:
        print(f"\n❌ Error: {e}")
        print("Please try again with a different stock symbol.")
```

Adding a Terminal-based Chatbot Interface

Next, let's set up an interactive command-line interface that makes the investment committee system accessible to users.

In the `main.py` file, we set up the welcome message that will be displayed when the application is initialized:

```
def print_welcome():
    """Print welcome message and system description"""
    print("💼 INVESTMENT COMMITTEE SYSTEM")
    print("=" * 50)
    print("🐂 Bull Agent: Finds reasons to BUY")
    print("🐻 Bear Agent: Finds reasons to AVOID")
    print("🎯 Chairman: Makes final decision")
    print("=" * 50)
```

Next, create the `analyze_stock` function that handles the core interaction pattern:

```
def analyze_stock(supervisor, stock_symbol):
    """Analyze a stock using the investment committee"""
    print(f"\n📈 ANALYZING: {stock_symbol.upper()}")
    print("-" * 30)
```

0

```python
    user_query = f"Should I invest in {stock_symbol} stock? I want to

    for chunk in supervisor.stream(
        {
            "messages": [
                {
                    "role": "user",
                    "content": user_query,
                }
            ]
        },
    ):
        pretty_print_messages(chunk, last_message=True)
```

The function:

- Takes the stock symbol and constructs a natural language query asking for investment advice
- Uses `supervisor.stream()` to get real-time updates from each agent as they analyze the stock
- Displays agent responses incrementally using `pretty_print_messages()` to show the analysis process

Testing and Running the System

Now we are ready to test and run the system! Run the `main.py` script to start the application:

```
python main.py
```

You will be prompted to enter a stock symbol. Let's try it with `NVDA`:

```
💼 INVESTMENT COMMITTEE SYSTEM
==================================================
🐂 Bull Agent: Finds reasons to BUY
🐻 Bear Agent: Finds reasons to AVOID
```

```
🎯 Chairman: Makes final decision
==================================================
🔄 Initializing investment committee...
✅ Committee ready!

Enter stock symbol (or 'quit' to exit): NVDA
```

The output will look like this:

```
📈 ANALYZING: NVDA
------------------------------
...

================================== Ai Message ======================
Name: supervisor

**Summary:**
The bullish case for NVDA centers on historic revenue and profit grow
```

Final Thoughts

The investment committee system demonstrates the power of coordinated AI agents in making complex decisions. By combining specialized agents with different perspectives, we've created a system that can:

- Provide balanced analysis through structured debate
- Consider multiple viewpoints simultaneously
- Make informed decisions based on comprehensive data
- Adapt to new information through real-time updates

This approach can be extended to other domains where multiple

Related Posts

0