multi_agent_prebuilt

September 5, 2025

1 Multi-Agent System with LangGraph

This notebook provides a detailed walkthrough of the `multi_agent_prebuilt.py` script. We will explore how to build multi-agent systems using LangGraph, specifically focusing on Supervisor and Swarm patterns, as well as custom handoffs between agents.

Overview of the `multi_agent_prebuilt.py` script:

The script demonstrates three different approaches to building multi-agent systems: 1. **Supervisor Pattern:** A central supervisor agent delegates tasks to specialized agents. 2. **Swarm Pattern:** Agents can hand off tasks to each other directly. 3. **Custom Handoffs:** A more granular implementation of agent-to-agent task handoffs.

1.1 Section 1: Setup and Imports

This section handles the initial setup, including loading environment variables for API keys and importing necessary libraries. The `dotenv` library is used to load the `OPENAI_API_KEY` from a `.env` file. `rich` is used for pretty-printing outputs.

[ ]:
```
# Import the os module to interact with the operating system, used here for␣
↪environment variables.
import os
# Import load_dotenv from the dotenv library to load environment variables from␣
↪a .env file.
from dotenv import load_dotenv
# Import the print function from the rich library for enhanced, pretty-printed␣
↪output.
from rich import print
# Import ChatOpenAI, the LangChain wrapper for OpenAI's chat models.
from langchain_openai import ChatOpenAI
# Import create_react_agent, a prebuilt function to create a ReAct-style agent.
from langgraph.prebuilt import create_react_agent
# Import create_supervisor, a function to create a supervisor agent that␣
↪manages other agents.
from langgraph_supervisor import create_supervisor

# Load environment variables from a .env file into the environment.
load_dotenv()
```

```
# This line is a placeholder to remind the user to set their OpenAI API key.
# The key is required for the ChatOpenAI model to authenticate with the OpenAI␣
 ↪API.
# Make sure to set your OPENAI_API_KEY in a .env file or as an environment␣
 ↪variable
```

1.2 Section 2: Supervisor Multi-Agent System

This example demonstrates a supervisor-worker architecture. A `supervisor` agent receives a user request and routes it to the appropriate specialized agent (`flight_assistant` or `hotel_assistant`).

Code Logic: 1. **Tool Definition:** Simple functions `book_hotel` and `book_flight` are defined as tools for the agents. 2. **Agent Creation:** Two `create_react_agent` instances are created. Each agent is specialized for a single task (booking flights or hotels) and is given access to the relevant tool. 3. **Supervisor Creation:** `create_supervisor` is used to create a managing agent that orchestrates the two specialized agents. 4. **Execution:** The system is run by streaming a user request through the compiled supervisor graph.

[ ]:
```
# Define a tool function for booking a hotel.
# It takes the hotel name as input and returns a confirmation message.
def book_hotel(hotel_name: str):
    """Book a hotel"""
    # This is a mock function; in a real application, this would
    # interact with a hotel booking API.
    return f'Successfully booked a stay at {hotel_name}.'


# Define a tool function for booking a flight.
# It takes departure and arrival airports and returns a confirmation.
def book_flight(from_airport: str, to_airport: str):
    """Book a flight"""
    # This is a mock function; in a real application, this would
    # interact with a flight booking API.
    return f'Successfully booked a flight from {from_airport} to {to_airport}.'
```

[ ]:
```
# Create the first specialized agent for booking flights.
# The create_react_agent function initializes an agent that can reason and act.
flight_assistant = create_react_agent(
    # The language model that powers the agent's reasoning.
    model=ChatOpenAI(model='gpt-4o-mini'),
    # The list of tools the agent can use. This agent can only book flights.
    tools=[book_flight],
    # The system prompt that defines the agent's role and personality.
    prompt='You are a flight booking assistant. Help users book flights.',
    # A unique name for the agent, used for routing and identification.
    name='flight_assistant',
)
```

```
# Create the second specialized agent for booking hotels.
hotel_assistant = create_react_agent(
    # The language model for this agent.
    model=ChatOpenAI(model='gpt-4o-mini'),
    # This agent can only use the book_hotel tool.
    tools=[book_hotel],
    # The specific prompt for the hotel assistant.
    prompt='You are a hotel booking assistant. Help users book hotels.',
    # A unique name for the hotel agent.
    name='hotel_assistant',
)
```

[ ]:
```
# Create the supervisor agent that will manage the specialized agents.
# The create_supervisor function wires together the agents and a routing␣
 ↪mechanism.
supervisor = create_supervisor(
    # A list of the worker agents that the supervisor can delegate tasks to.
    agents=[flight_assistant, hotel_assistant],
    # The language model for the supervisor's decision-making.
    model=ChatOpenAI(model='gpt-4o-mini'),
    # The system prompt for the supervisor, defining its role as a manager.
    prompt='You manage hotel and flight booking assistants. Assign work to them.
 ↪',
# Compile the supervisor graph into a runnable LangGraph object.
).compile()
```

[ ]:
```
# Print a header to indicate that the graph structure is being displayed.
print('Subgraph structure:')
# This block attempts to visualize the compiled graph as a Mermaid diagram.
# It's wrapped in a try-except block to handle environments where
# visualization dependencies might be missing (e.g., non-Jupyter environments).
try:
    # Import Image and display from IPython.display for rendering images in␣
 ↪notebooks.
    from IPython.display import Image, display

    # Generate a PNG image of the graph's structure and display it.
    display(Image(supervisor.get_graph().draw_mermaid_png()))
except Exception:
    # If an error occurs (e.g., missing libraries), just pass and continue.
    pass
```

[ ]:
```
# Print a header for the supervisor example output.
print('=== SUPERVISOR EXAMPLE ===')
```

```
# Stream the output from the supervisor graph for a given user request.
# The `stream` method allows processing the output as it's generated.
# `debug=True` provides verbose logging of the internal steps.
for chunk in supervisor.stream(
    # The input to the graph is a dictionary with a 'messages' key.
    {'messages': [{'role': 'user', 'content': 'Book a flight from LAX to␣
 ↪NYC'}]},
    # Enable debug mode for detailed output of the graph's execution.
    debug=True
):
    # Check if the chunk contains any data before printing.
    if chunk:
        # Print a separator for readability.
        print('-'*60)
        # Print the content of the current chunk.
        print(f'Chunk: {chunk}')
        # The original print(chunk) is commented out, but can be used for raw␣
 ↪output.
        # print(chunk)
```

1.3 Section 3: Swarm Multi-Agent System

This example showcases a swarm architecture where agents can directly hand off tasks to one another. This is useful for more collaborative workflows.

Code Logic: 1. **Tool Definition:** Similar to the supervisor example, `book_hotel_swarm` and `book_flight_swarm` are defined. 2. **Handoff Tool Creation:** `create_handoff_tool` is used to create special tools that allow one agent to transfer control to another. 3. **Agent Creation:** Agents are created with both their primary tool and the handoff tools. 4. **Swarm Creation:** `create_swarm` assembles the agents into a collaborative system. A `default_active_agent` is specified to handle the initial user request. 5. **Execution:** The user request is streamed through the swarm.

[ ]:
```
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent
from langgraph_swarm import create_swarm, create_handoff_tool
```

[ ]:
```
# Define a tool for booking a hotel in the swarm example.
def book_hotel_swarm(hotel_name: str):
    """Book a hotel"""
    return f'Hotel booked: {hotel_name}'


# Define a tool for booking a flight in the swarm example.
def book_flight_swarm(from_airport: str, to_airport: str):
    """Book a flight"""
    return f'Flight booked: {from_airport} to {to_airport}'
```

[ ]:
```
# Create a handoff tool that allows transferring a task to the hotel assistant.
# The `create_handoff_tool` function generates a tool that agents can call.
transfer_to_hotel = create_handoff_tool(
    # The name of the agent to hand off to.
    agent_name='hotel_assistant',
    # The description of the tool, which the agent's LLM will use to decide␣
 ↪when to use it.
    description='Transfer to hotel booking assistant.',
)


# Create a similar handoff tool for transferring to the flight assistant.
transfer_to_flight = create_handoff_tool(
    agent_name='flight_assistant',
    description='Transfer to flight booking assistant.',
)
```

[ ]:
```
# Create the flight assistant for the swarm.
flight_assistant_swarm = create_react_agent(
    model=ChatOpenAI(model='gpt-4o-mini'),
    # This agent has its primary tool (book_flight_swarm) and a handoff tool.
    tools=[book_flight_swarm, transfer_to_hotel],
    prompt='You are a flight booking assistant.',
    name='flight_assistant',
)


# Create the hotel assistant for the swarm.
hotel_assistant_swarm = create_react_agent(
    model=ChatOpenAI(model='gpt-4o-mini'),
    # This agent can book hotels and hand off to the flight assistant.
    tools=[book_hotel_swarm, transfer_to_flight],
    prompt='You are a hotel booking assistant.',
    name='hotel_assistant',
)
```

[ ]:
```
print('Subgraph structure:')
# Optional: Display a visualization of the graph's structure.
try:
    from IPython.display import Image, display

    display(Image(hotel_assistant_swarm.get_graph().draw_mermaid_png()))
except Exception:
    pass
```

[ ]:
```
# Create the swarm by passing the list of agents.
# `create_swarm` wires them together so they can hand off tasks to each other.
```

```
# `default_active_agent` specifies which agent should receive the initial␣
 ↪request.
swarm = create_swarm(agents=[flight_assistant_swarm, hotel_assistant_swarm],␣
 ↪default_active_agent='flight_assistant').compile()
```

[ ]:
```
# Print a header for the swarm example.
print('\n=== SWARM EXAMPLE ===')

# Stream the output from the swarm.
# The initial request is to book a hotel, which will go to the default␣
 ↪'flight_assistant'.
# The flight assistant should then use the `transfer_to_hotel` tool.
for chunk in swarm.stream({'messages': [{'role': 'user', 'content': 'Book a␣
 ↪hotel at Marriott'}]}, debug=True):
    if chunk:
        print('-'*60)
        print(f'Chunk: {chunk}')
        # print(chunk)
```

1.4 Section 4: Custom Handoffs Implementation

This section provides a from-scratch implementation of handoffs, giving you more control over the agent interaction logic. It uses a `StateGraph` to define the workflow explicitly.

Code Logic: 1. **Tool Definition:** `book_hotel_custom` and `book_flight_custom` are defined. 2. **Custom Handoff Tool Factory:** A function `create_handoff_tool` is created. This factory generates a tool that, when called, returns a `Command` object. This `Command` instructs the graph to transition to a different agent node. 3. **Agent Definition:** Agents are created with their respective tools, including the custom handoff tools. 4. **Graph Building:** A `StateGraph` is constructed. The agents are added as nodes, and an entry point (`START`) is defined, directing initial requests to the `flight_assistant`.

[ ]:
```
from typing import Annotated
from langchain_core.tools import tool, InjectedToolCallId
from langchain_openai import ChatOpenAI
from langgraph.prebuilt import create_react_agent, InjectedState
from langgraph.graph import StateGraph, START, MessagesState
from langgraph.types import Command
```

[ ]:
```
# Define a custom tool for booking a hotel.
def book_hotel_custom(hotel_name: str):
    """Book a hotel"""
    return f'Custom hotel booking: {hotel_name}'


# Define a custom tool for booking a flight.
def book_flight_custom(from_airport: str, to_airport: str):
    """Book a flight"""
```

```
    return f'Custom flight booking: {from_airport} to {to_airport}'
```

[ ]:
```
# This function is a factory for creating custom handoff tools.
def create_handoff_tool_custom(*, agent_name: str, description: str):
    # Generate a unique name for the tool based on the target agent.
    name = f'transfer_to_{agent_name}'

    # Use the @tool decorator to define the handoff tool.
    @tool(name, description=description)
    def handoff_tool(
        # Inject the current graph state into the tool call.
        state: Annotated[MessagesState, InjectedState],
        # Inject the ID of the tool call that triggered this.
        tool_call_id: Annotated[str, InjectedToolCallId],
    ) -> Command:
        # Create a tool message to record the handoff event.
        tool_message = {
            'role': 'tool',
            'content': f'Transferred to {agent_name}',
            'name': name,
            'tool_call_id': tool_call_id,
        }
        # Return a Command object to instruct the graph to transition.
        return Command(
            # The `goto` field specifies the next node to execute.
            goto=agent_name,
            # The `update` field modifies the state, adding the tool message.
            update={'messages': state['messages'] + [tool_message]},
            # `graph=Command.PARENT` indicates the command applies to the␣
 ↪parent graph.
            graph=Command.PARENT,
        )

    return handoff_tool
```

[ ]:
```
# Create the custom handoff tools using the factory function.
transfer_to_hotel_custom = create_handoff_tool_custom(
    agent_name='hotel_assistant',
    description='Transfer to hotel assistant.',
)

transfer_to_flight_custom = create_handoff_tool_custom(
    agent_name='flight_assistant',
    description='Transfer to flight assistant.',
)
```

[ ]:
```
# Define the agents using the custom tools.
flight_assistant_custom = create_react_agent(
    model=ChatOpenAI(model='gpt-4o-mini'),
    tools=[book_flight_custom, transfer_to_hotel_custom],
    prompt='You are a flight booking assistant.',
    name='flight_assistant',
)

hotel_assistant_custom = create_react_agent(
    model=ChatOpenAI(model='gpt-4o-mini'),
    tools=[book_hotel_custom, transfer_to_flight_custom],
    prompt='You are a hotel booking assistant.',
    name='hotel_assistant',
)
```

[ ]:
```
# Build the state graph manually.
multi_agent_graph = (
    # Initialize a StateGraph with the MessagesState schema.
    StateGraph(MessagesState)
    # Add the flight assistant as a node in the graph.
    .add_node('flight_assistant', flight_assistant_custom)
    # Add the hotel assistant as another node.
    .add_node('hotel_assistant', hotel_assistant_custom)
    # Define the entry point of the graph. All requests will start at the␣
 ↪flight assistant.
    .add_edge(START, 'flight_assistant')
    # Compile the graph into a runnable object.
    .compile()
)
```

[ ]:
```
print('Subgraph structure:')
# Optional: Display a visualization of the graph's structure.
try:
    from IPython.display import Image, display
    img = Image(multi_agent_graph.get_graph().draw_mermaid_png())
    # save to file
    # img.save('/Users/james/Library/CloudStorage/Dropbox/GitHub/YouTube/
 ↪LangGraph_101/02-Agents/19-Multi_Agent/handoffs.png')
    display(img)
except Exception:
    pass
```

[ ]:
```
# Print a header for the custom handoffs example.
print('\n=== CUSTOM HANDOFFS EXAMPLE ===')

# Stream the output from the custom graph.
```

```
for chunk in multi_agent_graph.stream({'messages': [{'role': 'user', 'content':␣
↪'Book a flight from SFO to LAX'}]}, debug=True):
    if chunk:
        print('-'*60)
        print(f'Chunk: {chunk}')
        # print(chunk)
```

1.5 Section 5: Conclusion

This notebook has demonstrated three powerful patterns for building multi-agent systems with LangGraph:

- **Supervisor:** Ideal for hierarchical task delegation where a manager oversees workers.
- **Swarm:** Suitable for collaborative environments where agents can pass tasks amongst themselves.
- **Custom Handoffs:** Offers the most flexibility for designing complex, bespoke agent interactions.

Potential Extensions:

- **Error Handling:** Implement more robust error handling within each agent.
- **Dynamic Routing:** Add more complex conditional logic for routing tasks.
- **State Management:** Enhance the state to carry more context between agent interactions.