Building Multi-Agent Systems with LangGraph

Estimated Reading Time: 15 minutes

This guide demonstrates how to implement multi-agent workflows using LangGraph, a graph-based framework for orchestrating AI agents working together to complete complex tasks.

What is LangGraph?

LangGraph structures AI workflows as directed graphs where each node represents an agent or processing step, and edges control the flow of data and execution between them. The shared state enables collaboration among all agents.

Key Benefits

- Modular design with independently testable agents
- Dynamic routing based on runtime conditions
- Shared memory accessible by all nodes
- Visual, clear workflow representation

State Management

Before implementing, you need to define a shared state type that all agents will read from and update.

```
from typing import TypedDict, List, Optional

class SalesReportState(TypedDict):
    request: str
    raw_data: Optional[dict]
    processed_data: Optional[dict]
    chart_config: Optional[dict]
    report: Optional[str]
    errors: List[str]
     next_action: str
```

Agent Nodes

Agents are functions that receive the shared state, perform their task, and return the updated state. Below are simplified placeholders to illustrate this pattern.

Agent Function Placeholders

```
def data_collector_agent(state: SalesReportState) -> SalesReportState:
    # Placeholder: collect raw data based on request
    # Update state with raw_data and set next_action
    return state

def data_processor_agent(state: SalesReportState) -> SalesReportState:
    # Placeholder: process raw_data and update processed_data
    # Set next_action to next step
    return state

def chart_generator_agent(state: SalesReportState) -> SalesReportState:
    # Placeholder: create chart configuration from processed_data
    # Update chart_config and set next_action
    return state

def report_generator_agent(state: SalesReportState) -> SalesReportState:
    # Placeholder: generate textual report using processed_data
    # Update report and set next_action to complete
    return state

def error_handler_agent(state: SalesReportState) -> SalesReportState:
    # Placeholder: handle errors, prepare error messages in report
    # Set next_action to complete
    return state
```

Routing Logic

The workflow requires a router to decide which agent to run next based on the current state.

Routing Function Example

```
def route_next_step(state: SalesReportState) -> str:
    routing = {
        "collect": "data_collector",
        "process": "data_processor",
        "visualize": "chart_generator",
        "report": "report_generator",
        "error": "error_handler",
        "complete": "END"
    }
     return routing.get(state.get("next_action", "collect"), "END")
```

Building and Compiling the Workflow Graph

Using LangGraph's `StateGraph`, you add nodes for each agent, define conditional edges based on the routing function, and set the entry point.

Workflow Construction Example

```
from langgraph.graph import StateGraph, END

def create_sales_report_workflow():
    workflow = StateGraph(SalesReportState)

    workflow.add_node("data_collector", data_collector_agent)
    workflow.add_node("data_processor", data_processor_agent)
    workflow.add_node("chart_generator", chart_generator_agent)
    workflow.add_node("report_generator", report_generator_agent)
     workflow.add_node("error_handler", error_handler_agent)

     workflow.add_conditional_edges("data_collector",
  route_next_step, {
         "data_processor": "data_processor", "error_handler":
  "error_handler", END: END
     })
```

```
    workflow.add_conditional_edges("data_processor",
route_next_step, {
        "chart_generator": "chart_generator", "error_handler":
"error_handler", END: END
    })
    workflow.add_conditional_edges("chart_generator",
route_next_step, {
        "report_generator": "report_generator", "error_handler":
"error_handler", END: END
    })
    workflow.add_conditional_edges("report_generator",
route_next_step, {
        "error_handler": "error_handler", END: END
    })
    workflow.add_conditional_edges("error_handler",
route_next_step, {END: END})

    workflow.set_entry_point("data_collector")
    return workflow.compile()
```

Running the Workflow

Once compiled, the workflow can be invoked with an initial state. This runs the agents in order, respecting the routing logic.

Running Example

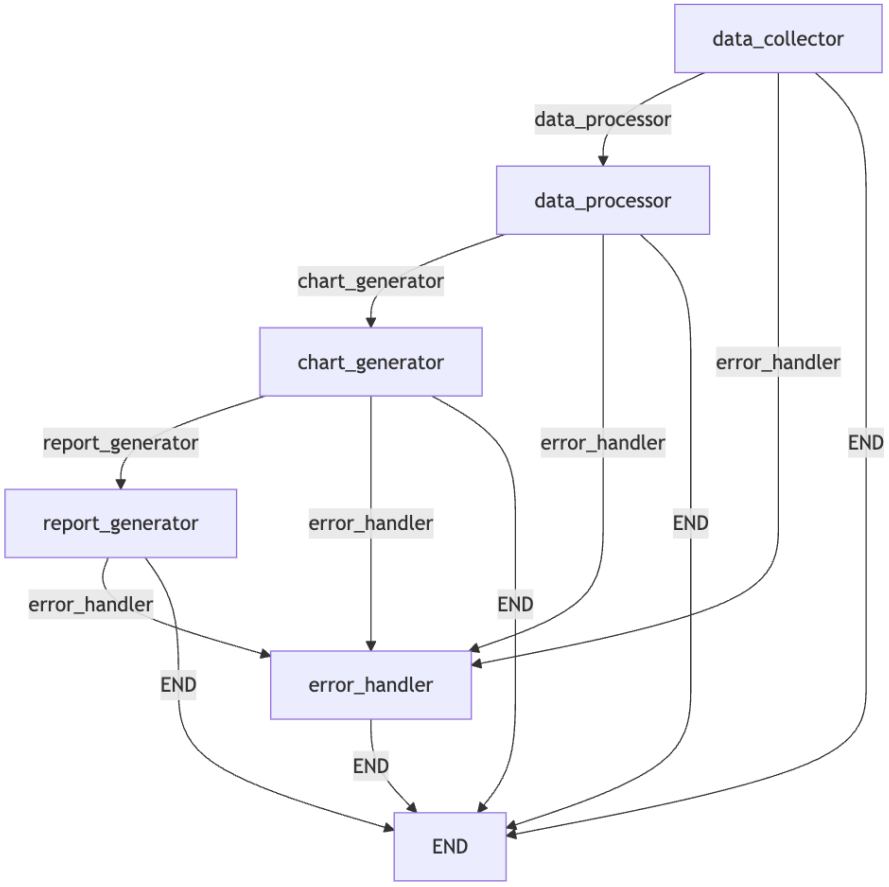
```
def run_sales_report_workflow():
    app = create_sales_report_workflow()
    initial_state = SalesReportState(
        request="Q1-Q2 2024 Sales Analysis",
        raw_data=None,
        processed_data=None,
        chart_config=None,
        report=None,
        errors=[],
          next_action="collect"
    )
    print("Starting workflow...\n")
    final_state = app.invoke(initial_state)
    print("\nWorkflow Complete\n")
    if final_state["errors"]:
        print("Errors:")
        for err in final_state["errors"]:
            print(f"- {err}")
    print("\nFinal Report:\n", final_state["report"])
```

```
    return final_state

if __name__ == "__main__":
    run_sales_report_workflow()
```

Example Visualization: Multi-Agent Workflow Graph

This diagram illustrates the multi-agent workflow constructed using LangGraph for the sales report generation system.



Nodes: Each rectangle represents an individual agent responsible for a specific task:

- `data_collector`: Gathers the raw sales data based on the user request.
- `data_processor`: Analyzes and processes the collected data.
- `chart_generator`: Prepares chart configurations for data visualization.
- `report_generator`: Produces the final textual sales report.
- `error_handler`: Manages errors and generates error reports if any step fails.
- `END`: Represents the termination of the workflow.

Edges: Directed arrows show the flow of control between agents based on routing logic:

- From each agent, the workflow can proceed to the next logical agent if successful.
- If an error occurs, the workflow routes to the `error_handler` agent for recovery.
- The workflow terminates by reaching the `END` node.

Dynamic Routing: The routing decisions are based on the current state's `next_action` value, enabling flexible and conditional progression through the agents.

Error Handling: Multiple agents can trigger the error handler, which centralizes error management and ensures graceful workflow completion.

This visualization captures the modular and scalable nature of LangGraph multi-agent workflows, highlighting how individual agents collaborate through shared state and conditional transitions to accomplish complex tasks reliably.

This example demonstrated a simple multi-agent system using LangGraph with:

- Separate agents handling data collection, processing, visualization, and reporting.
- Dynamic routing based on agent outcomes.
- Shared state passed and updated by each agent.
- A clean, maintainable workflow that can be extended with error handling or parallelism.

Author(s)

Faranak Heidari

Other Contributors

Karan Goswami