Multi-Agent System with Tool Calling

A simple demonstration of building a multi-agent system using **LangChain**, **LangGraph**, and **Pydantic** with **ChatGPT** as the underlying language model.

🚀 Features

- **Multi-Agent Coordination**: Research and Writing agents working together
- **Tool Calling**: Custom tools for web search, fact-checking, and text analysis
- **Structured Data**: Pydantic models for type-safe data handling
- **Workflow Orchestration**: LangGraph for managing agent interactions
- **State Management**: Shared state between agents with checkpointing

🏗️ Architecture

Agents

1. **Research Agent** - Gathers and analyzes information
2. **Writing Agent** - Creates content based on research

Tools

- `web_search`: Mock web search functionality
- `fact_checker`: Validates claims and information
- `word_counter`: Counts words in text

Data Models (Pydantic)

- `ResearchResult`: Structured research findings

- `ArticleContent`: Structured article format
- `AgentState`: Shared state between agents

📁 Project Structure

```
/workspace/
├── multi_agent_system.py    # Main system implementation
├── example_usage.py         # Demonstration script
├── .env.example            # Environment variables template
└── README.md               # This file
```

🛠️ Setup

1. **Install Dependencies**
   `bash uv add langchain langchain-openai langgraph pydantic python-dotenv`
2. **Set up OpenAI API Key** (Optional - works with mock data without key)
   `bash cp .env.example .env # Edit .env and add your OpenAI API key`
3. **Run the Demo**
   `bash python example_usage.py`

💡 Usage Examples

Basic Usage with Mock Data

```
from example_usage import run_demo_with_mock_data

# Run demonstration without API key
research, article = run_demo_with_mock_data()
print(f"Research Topic: {research.topic}")
print(f"Article Title: {article.title}")
```

Full System with OpenAI API

```
from multi_agent_system import MultiAgentSystem
import os

# Initialize system with API key
system = MultiAgentSystem(os.getenv("OPENAI_API_KEY"))

# Run workflow
result = system.run_workflow("artificial intelligence")

# Access results
if result.get("article_content"):
    article = ArticleContent(**result["article_content"])
    print(f"Created: {article.title}")
```

Individual Tool Usage

```
from multi_agent_system import web_search, fact_checker,
word_counter

# Use tools directly
search_result = web_search.invoke({"query": "climate change"})
fact_result = fact_checker.invoke({"claim": "AI improves
efficiency"})
word_count = word_counter.invoke({"text": "Hello world"})
```

🔧 Key Technologies

- **LangChain**: Framework for building applications with LLMs
- **LangGraph**: Library for building stateful, multi-actor applications
- **Pydantic**: Data validation using Python type annotations
- **OpenAI GPT**: Large language model for intelligent responses

🎯 Workflow Process

1. **Research Phase**
   - Research agent receives a topic
   - Uses web search tool to gather information
   - Structures findings into ResearchResult model
2. **Writing Phase**
   - Writing agent receives research results
   - Creates structured article content
   - Uses word counter tool for metrics

3. **State Management**
   - LangGraph manages workflow transitions
   - Shared state preserves data between agents
   - Memory checkpointing enables workflow resumption

🚦 Demo Output

When you run the demo, you'll see:

```
🎉 Multi-Agent System Demonstration
============================================================

🔄 No API key - running mock demonstration
🤖 Running Multi-Agent System Demo with Mock Data
============================================================
🔍 Research Phase Complete:
   Topic: Artificial Intelligence in Healthcare
   Key Findings: 4 insights discovered
   Sources: 3 references analyzed

✍️  Writing Phase Complete:
   Title: The Revolutionary Impact of AI in Modern Healthcare
   Sections: 4 main content areas
   Word Count: 156 words
```

🛡️ Error Handling

The system includes:
- Graceful fallbacks when API keys are missing
- JSON parsing with fallback data structures
- Tool error handling and recovery
- State validation with Pydantic models

🎨 Customization

Adding New Agents

```
class AnalysisAgent:
    def __init__(self, llm):
        self.llm = llm
        self.tools = [custom_tool]

    def analyze_data(self, data):
        # Implementation here
        pass
```

Adding New Tools

```
@tool
def custom_tool(input_text: str) -> str:
    """Description of what the tool does"""
    # Tool implementation
    return result
```

Extending Workflow

```
# Add new nodes to the workflow
workflow.add_node("analysis", analysis_step)
workflow.add_conditional_edges("analysis", should_continue)
```

📚 Learning Resources

- LangChain Documentation

- LangGraph Documentation
- Pydantic Documentation
- OpenAI API Documentation

🤝 Contributing

This is a demonstration project. Feel free to:
- Extend the agents with new capabilities
- Add more sophisticated tools
- Implement real API integrations
- Enhance the workflow with additional steps

📄 License

Open source - feel free to use and modify for your own projects!