**Building a Multi-Agent RAG System with LangGraph**

**A Project-Based Course for Financial News Analysis**

**Author:** Gemini, a helpful AI assistant built by Google **Date:** June 7, 2025

**Table of Contents**

1. **Course Overview**
   * Learning Objectives
   * The Project: Financial News Analysis and Stock Idea Generator
2. **Getting Started: Workspace Setup**
   * Prerequisites
   * Project Setup Steps
3. **Module 1: Foundations**
   * Introduction to Key Concepts (RAG, Multi-Agent Systems, LangGraph)
   * Installing Required Libraries
   * Setting Up Your OpenAI API Key
   * Your First "Hello, LangGraph!" Example
4. **Module 2: Building the Agent Tools**
   * Tool 1: The News Researcher
5. **Module 3 & 4: Constructing the Graph with Model Context Protocol (MCP)**
   * Defining the State
   * Defining Tools and the MCP Data Model
   * Defining the Agent Nodes (Researcher, Analyst, Trader)
   * Defining Conditional Logic
   * Building the Graph
6. **Module 5: Finalizing and Running Your System**
   * Executing the Stock Analysis
   * Final Output and Visualization
7. **Complete Source Code**
   * A single, copy-pastable script for your Jupyter Notebook.

**1. Course Overview**

This course guides you through building a multi-agent RAG system for stock analysis using LangGraph. It is designed as a hands-on, project-based learning experience. We will start with foundational concepts and progressively build a practical tool that can analyze financial news and generate stock ideas.

**Learning Objectives:**

* Understand the fundamentals of Retrieval-Augmented Generation (RAG) and multi-agent systems.
* Master the core components of LangGraph, including state, nodes, and edges.
* Develop a multi-agent RAG system for a real-world application.
* Integrate external tools, such as a financial news API, into your system.
* Implement a Model Context Protocol (MCP) interaction for structured agent communication.
* Gain proficiency in building and debugging AI applications in a Jupyter Notebook environment.

**The Project: A Financial News Analysis and Stock Idea Generator**

We will build a multi-agent system that takes a stock ticker as input and outputs a concise analysis and a potential trading idea. This project will involve three specialized agents working together:

1. **The News Researcher Agent:** Retrieves the latest financial news for a given stock. This is the "Retrieval" part of our RAG system.
2. **The Data Analyst Agent:** Takes the news articles, analyzes the sentiment, and summarizes the key information into a structured format.
3. **The Trading Idea Generator Agent:** Receives the structured analysis and generates a brief, actionable trading idea.

**2. Getting Started: Workspace Setup**

**Prerequisites:**

* **VS Code:** [Download here](https://code.visualstudio.com/).
* **Python:** [Download here](https://www.python.org/downloads/).
* **VS Code Extensions:** The official **Python** and **Jupyter** extensions from Microsoft.

**Project Setup Steps:**

1. **Create a Project Folder:** Name it LangGraph-Stock-Analyzer.
2. **Open in VS Code:** Use File > Open Folder... to open your new project folder.
3. **Create a Virtual Environment:**
   * Open the VS Code terminal (View > Terminal).
   * Run python -m venv .venv to create a virtual environment.
   * Activate it. VS Code will likely prompt you to select this interpreter. If not:
     + **Windows:** .\.venv\Scripts\activate
     + **macOS/Linux:** source .venv/bin/activate
4. **Create Your Notebook:** In VS Code, create a new file named stock\_analyzer.ipynb. This will be your workspace for the entire project.

**3. Module 1: Foundations**

**Introduction to Key Concepts**

* **Retrieval-Augmented Generation (RAG):** Imagine an analyst who doesn't work from memory. They first *retrieve* relevant documents (news, reports) and then use that information to *generate* their analysis. RAG systems do the same: fetch data first, then generate an informed response.
* **Multi-Agent Systems:** Instead of one monolithic program, we build a team of specialized "agents." For our project, a "Researcher," an "Analyst," and an "Idea Generator" collaborate, each handling one part of the task.
* **LangGraph:** This is our project manager for the AI agents. It lets us define a workflow, or "graph," that dictates who does what and in what order, ensuring they work together smoothly.

**Installing Required Libraries**

Python

# Run this in a cell in your Jupyter Notebook

!pip install langgraph langchain langchain\_openai beautifulsoup4 yfinance duckduckgo-search

**Setting Up Your OpenAI API Key**

1. Get a key from the [OpenAI API keys page](https://platform.openai.com/api-keys).
2. In your VS Code terminal (with the .venv activated), set the key as an environment variable to keep it secure. **You must restart the terminal after running this command.**
   * **macOS/Linux:** export OPENAI\_API\_KEY='your-key-here'
   * **Windows:** set OPENAI\_API\_KEY='your-key-here'

**Your First "Hello, LangGraph!" Example**

This simple graph demonstrates the core concepts of State, Nodes, and Edges.

Python

import os

from typing import TypedDict, Annotated

from langchain\_openai import ChatOpenAI

from langgraph.graph import StateGraph, END

from IPython.display import Image, display

# Define the state: the data that flows through the graph

class SimpleState(TypedDict):

input: str

result: str

# Initialize our LLM

llm = ChatOpenAI(model="gpt-4o", temperature=0)

# Define our nodes (the "workers" in the graph)

def node\_one(state: SimpleState):

print("---EXECUTING NODE ONE---")

state['result'] = "Hello"

return state

def node\_two(state: SimpleState):

print("---EXECUTING NODE TWO---")

state['result'] = state['result'] + ", LangGraph!"

return state

# Build the graph

workflow = StateGraph(SimpleState)

workflow.add\_node("node\_one", node\_one)

workflow.add\_node("node\_two", node\_two)

workflow.set\_entry\_point("node\_one")

workflow.add\_edge("node\_one", "node\_two")

workflow.add\_edge("node\_two", END)

# Compile and run

app = workflow.compile()

final\_state = app.invoke({"input": "start", "result": ""})

print(f"\n---FINAL RESULT---\n{final\_state['result']}")

# Visualize the graph

print("\n---GRAPH VISUALIZATION---")

display(Image(app.get\_graph().draw\_png()))

**4, 5, & 6. The Complete Multi-Agent RAG System**

This section contains the full code for the final project, combining modules 2, 3, 4, and 5. The code is heavily commented to explain each part, from defining tools and the state, to building the graph with conditional logic and our Model Context Protocol.

You can copy and paste this entire block into a single cell in your Jupyter Notebook.

**7. Complete Source Code**

Python

# --- Required Imports ---

import os

import json

import yfinance as yf

from typing import TypedDict, Annotated, List

from langchain.tools import DuckDuckGoSearchRun

from langchain\_core.pydantic\_v1 import BaseModel, Field

from langchain\_openai import ChatOpenAI

from langgraph.graph import StateGraph, END

from langgraph.graph.message import add\_messages

from IPython.display import Image, display

# --- AGENT TOOLS ---

def get\_stock\_news(ticker: str) -> list:

"""Gets recent news for a stock ticker using yfinance."""

print(f"---RESEARCHER: Searching yfinance news for {ticker}---")

stock = yf.Ticker(ticker)

return stock.news or []

def general\_search(query: str) -> str:

"""Performs a general web search using DuckDuckGo for fallback."""

print(f"---RESEARCHER: Performing general web search for '{query}'---")

search = DuckDuckGoSearchRun()

return search.run(query)

# --- MODEL CONTEXT PROTOCOL (MCP) DEFINITION ---

# The Analyst Agent MUST return data in this Pydantic format.

class Analysis(BaseModel):

"""The result of a detailed financial news analysis."""

summary: str = Field(description="A concise summary of the key news points and their implications.")

sentiment\_score: float = Field(description="A sentiment score from -1.0 (very negative) to 1.0 (very positive) based on the news.")

# --- AGENT STATE DEFINITION ---

class AgentState(TypedDict):

ticker: str

news: List[dict]

analysis: str # This will hold the structured MCP data (as a JSON string)

recommendation: str

messages: Annotated[list, add\_messages] # For chat history

# --- AGENT NODES DEFINITION ---

# Initialize LLM and Tools

llm = ChatOpenAI(model="gpt-4o", temperature=0)

structured\_llm = llm.with\_structured\_output(Analysis)

search\_tool = DuckDuckGoSearchRun()

def researcher\_node(state: AgentState):

"""Node 1: Fetches news for the stock (the 'R' in RAG)."""

print("---NODE: RESEARCHER---")

ticker = state['ticker']

news\_items = get\_stock\_news(ticker)

if not news\_items:

print(f"No yfinance news for {ticker}, using DuckDuckGo as fallback.")

query = f"latest financial news and developments for {ticker} stock"

search\_results = general\_search(query)

news\_items = [{"title": "General Search Results", "summary": search\_results}]

return {"news": news\_items}

def analyst\_node(state: AgentState):

"""Node 2: Analyzes news and creates a structured JSON output (our MCP)."""

print("---NODE: ANALYST---")

news\_items = state['news']

news\_text = "\n\n".join([item.get('title', '') + ": " + item.get('summary', '') for item in news\_items if item.get('summary')]) # Ensure summary exists

prompt = f"You are a senior financial analyst. Analyze the following news for the stock ticker {state['ticker']}. Provide a concise summary and a sentiment score from -1.0 to 1.0.\n\nNews articles:\n{news\_text}"

analysis\_result = structured\_llm.invoke(prompt)

return {"analysis": analysis\_result.json()} # Store MCP data as JSON string in state

def trading\_idea\_node(state: AgentState):

"""Node 3: Generates a trading idea based on the structured analysis."""

print("---NODE: TRADING IDEA GENERATOR---")

analysis = json.loads(state['analysis']) # Load MCP data from JSON string

sentiment = analysis['sentiment\_score']

summary = analysis['summary']

prompt = f"""

You are a trading strategist providing advice for a retail investor. Based on the following analysis, generate a brief, actionable trading idea.

If sentiment is highly positive (> 0.6), suggest a bullish stance.

If sentiment is highly negative (< -0.6), suggest a bearish stance.

Otherwise, suggest a neutral or 'wait and see' approach.

Explain your reasoning in 2-3 sentences.

Analysis Summary: {summary}

Sentiment Score: {sentiment}

"""

response = llm.invoke(prompt)

return {"recommendation": response.content}

# --- CONDITIONAL LOGIC FOR THE GRAPH ---

def should\_continue(state: AgentState):

"""Determines the next step after the researcher node."""

if state.get('news'):

return "analyst"

return END

# --- GRAPH CONSTRUCTION ---

workflow = StateGraph(AgentState)

workflow.add\_node("researcher", researcher\_node)

workflow.add\_node("analyst", analyst\_node)

workflow.add\_node("trader", trading\_idea\_node)

workflow.set\_entry\_point("researcher")

workflow.add\_conditional\_edges("researcher", should\_continue, {"analyst": "analyst", END: END})

workflow.add\_edge('analyst', 'trader')

workflow.add\_edge('trader', END)

app = workflow.compile()

# --- RUNNING THE SYSTEM ---

# Define the input for our graph

ticker\_to\_analyze = "MSFT" # Try 'AAPL', 'GOOG', 'TSLA', etc.

inputs = {"ticker": ticker\_to\_analyze}

# Invoke the graph and print the final result

final\_state = app.invoke(inputs)

print("\n" + "="\*50)

print(f" FINAL ANALYSIS FOR: {final\_state['ticker']}")

print("="\*50 + "\n")

# Pretty print the final analysis and recommendation

analysis\_data = json.loads(final\_state['analysis'])

print("--- ANALYSIS ---")

print(f"Sentiment Score: {analysis\_data['sentiment\_score']:.2f}")

print(f"Summary: {analysis\_data['summary']}")

print("\n--- TRADING IDEA ---")

print(final\_state['recommendation'])

print("\n" + "="\*50 + "\n")

# Visualize the final graph structure

print("--- GRAPH VISUALIZATION ---")

try:

display(Image(app.get\_graph().draw\_png()))

except Exception as e:

print(f"Could not draw graph: {e}. You might need to install graphviz.")

*End of Document*