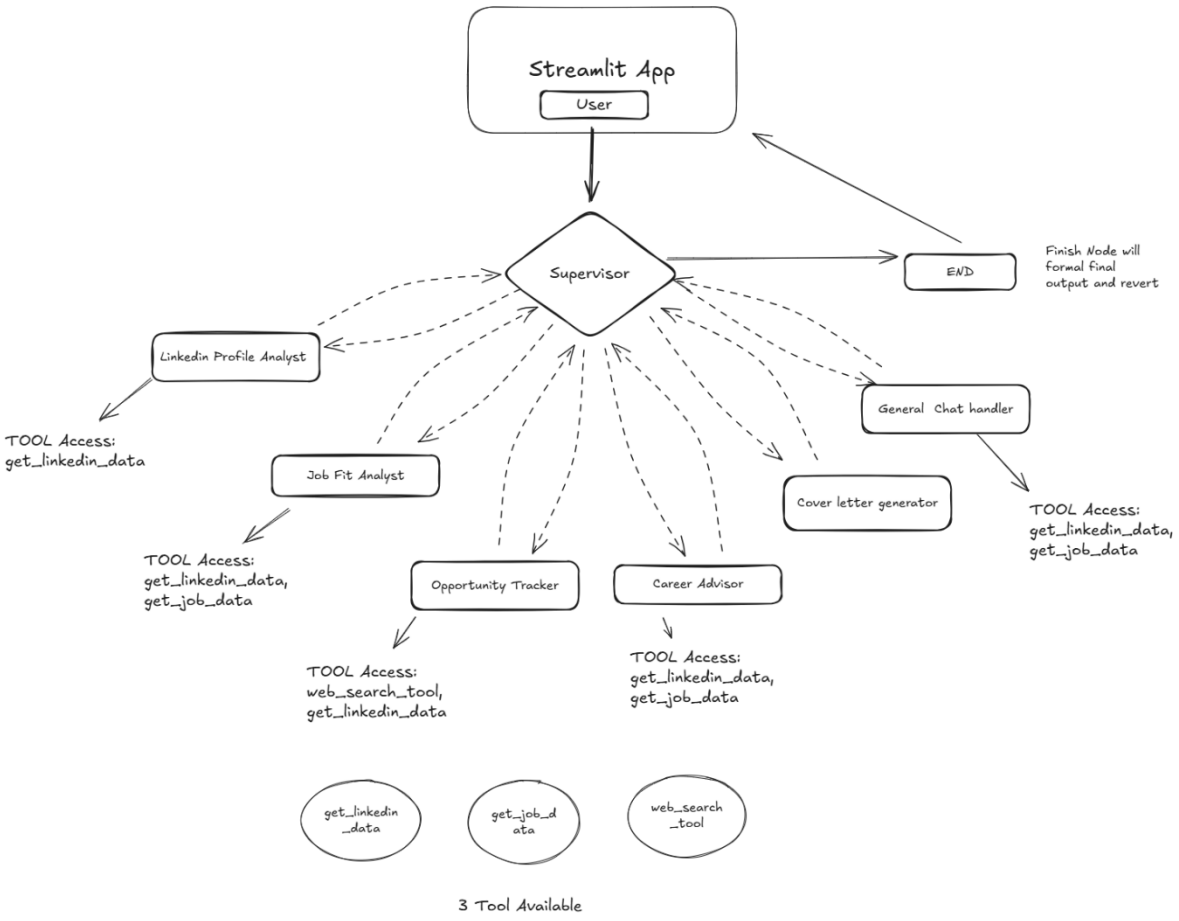
Documentation: Multi-Agent Chatbot for LinkedIn & Career Optimization Using LangGraph

Approach Overview

The architecture uses **LangGraph** framework to orchestrate interactions between multiple AI agents. A central **Supervisor** routes user queries to relevant agents, ensuring context-aware processing. Agents collaborate via a shared state (GraphState) to retain conversation history and outputs for cross-agent context sharing and give final formatted response.

Backstory: Initially I was using the Crew AI framework, but after adding all the functionality/agents, I ran into a few issues where it failed in some cases and got stuck in a loop. I chose this framework because I had limited time to complete the assignment, and it was simpler to use. While it works well for sequential agent flows, But it does not have enough flexibility for autonomous decision-making and proper state management so I moved to langgraph.

Architecture:



Architecture & Agent Roles:

Multi-Agent Management

Supervisor Agent:

- Acts as the main router.
- Receives the user query.
- Decides which specialized agent(s) should process the query.
- Collects and updates results from each agent in a shared state.
- Determines if further agent routing is needed or if the conversation can be finished.

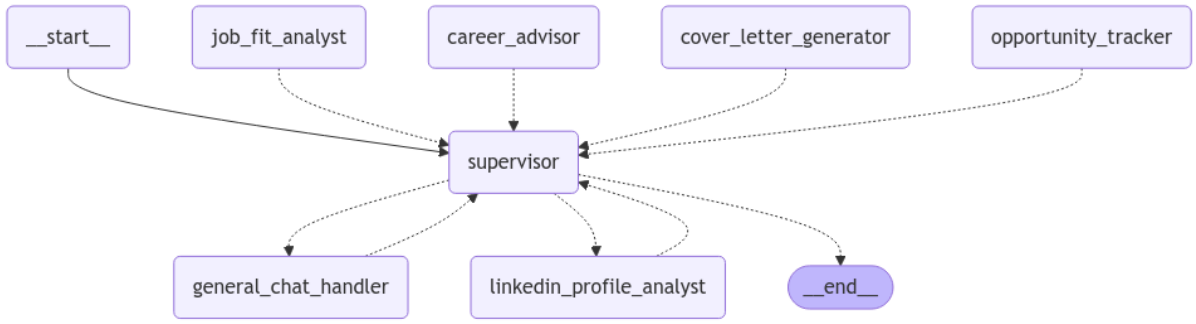
Specialized Agents

1. **General Chat Handler:**
    - Handles greetings and casual conversation.
2. **LinkedIn Profile Analyst:**
    - Extracts and evaluates profile sections (e.g., About, Experience, Skills).
    - Suggests improvements to the profile.
3. **Job Fit Analyst:**
    - Analyzes job descriptions from a provided LinkedIn job URL.
    - Compares the job description with the user's profile.
    - Generates a match score and suggests improvements for a better fit.
4. **Career Advisor:**
    - Identifies skill gaps based on the user's profile and career goals.
    - Suggests learning resources and potential career paths.
5. **Cover Letter Generator:**
    - Creates personalized cover letters using details from job descriptions.
6. **Opportunity Tracker:**
    - Retrieves job postings, tracks industry trends, and identifies networking opportunities.
7. **Finish Node:**
    - Collects the output from all agents.
    - Makes a final LLM call to format the response into a user-friendly message without exposing technical details.
    - Ends the conversation or asks for further input if needed.

Decision Logic:

- Simple queries (e.g., greetings) → general_chat_handler.
- LinkedIn profile analysis → linkedin_profile_analyst.
- Job fit analysis → job_fit_analyst.
- Cover letter requests → cover_letter_generator.
- Ends conversation if query is resolved → FINISH.

LangGraph generated Graph:



State Management and Multi-agent Collaboration:

Initial Query Handling:

- The supervisor receives the query and examines its content.
- For simple queries (e.g., greetings), it sends the query to the General Chat Handler.
- For complex queries (e.g., combining job fit analysis and cover letter generation), it routes the query sequentially to the required agents.

GraphState Maintenance:

- A central state (GraphState) stores the user query, previous chat history, and outputs from each agent.
- This state is shared among agents so that each one can use the previous outputs as context if needed.

Final Output Processing:

- Once all required agents have processed the query, the supervisor sends the accumulated output to the Finish Node.
- The Finish Node makes a final call to an LLM to merge and format the results, removing any internal technical details before sending the final output to the user.

GraphState Structure:

```python
{
  "user_query": str,
  "chat_history": List[dict],  # Previous interactions
  "outputs": dict,  # {agent_name: output}
  "profile_data": dict,  # Extracted LinkedIn data
  "job_data": dict  # Extracted job description
}
```

Example Workflow:

For a query like:
"Can you help analyze the job fit for this role and then generate a cover letter? Here is my LinkedIn profile URL and LinkedIn job URL."
The supervisor would:

1. Route the query to the Job Fit Analyst agent.
2. Update the GraphState with Agent/worker tag name in output with the job analysis results.
3. Route the query to the Cover Letter Generator agent.
4. Update the GraphState with the cover letter details.
5. Route the conversation to the Finish Node where a final LLM call formats the overall response.
6. Return the final formatted output to the user.

Challenges:

Edge Cases:

- This is in an initial phase, so many edge cases may not be handled perfectly.
- Need to give more time to make it more reliable

LinkedIn API rate limits:

- The LinkedIn scraping API has usage limits and may get exhausted, affecting the system's ability to fetch profile or job data.

Solutions:

Add Planning and Task manager to the architecture:

- Introduce a Planner Agent under supervisor that can break down complex queries into smaller, manageable steps.
- Use a Manager Agent to execute these steps using the specialized agents.
- The supervisor can determine if a query is complex and route it to the planner accordingly.
- This would help improve output accuracy and handle more complex queries effectively.

Use of Multiple LLMs:

- Use different language models for simple and complex tasks to reduce cost.

Voice input/output:

- Integrate ASR/TTS APIs (e.g., Whisper, ElevenLabs) for voice support.

Scalability:

- Design to be scale by including load balancing and efficient state management.

Directory structure:

└── **styleebender-learntube_careersync/**

├── **README.md**

├── **agents_prompts.py** – Contains all the prompt for agents and supervisor

├── **astream_events_handler.py –** To handle real time events show tool calling and thoughts

├── **config.py –** get API key for scraping

├── **data_process.py –** Filter and clean the data extracted data from API.

├── **graph.py –** end to end graph nodes, tool and state and compiled graph

├── **linkedin_scraper.py –** scraping API functions

├── **requirements.txt**

├── **sample_data.py –** sample data for test

└── **streamlit_app.py** – Main streamlit app (UI)

Note: Setup instruction in readme file.