**AI Career Coaching Assistant – Final Project Report**

By Anjali Chennupati

**1. Overview**

This project is a fully interactive, AI-powered career guidance system built using LangGraph and Google Gemini 1.5 Flash. The assistant allows users to ask questions about their LinkedIn profile, job descriptions, resume writing, and broader career guidance — all through a natural conversation loop. Unlike traditional chatbot implementations, this system supports LLM-based routing, agentic workflows, and persistent chat state with seamless human interaction via interrupt and resume.

The agent was designed to mimic the experience of a real career coach, capable of dynamically switching contexts between analyzing a user’s profile, evaluating job fits, improving resumes, or answering general career questions — with memory and logic preserved across turns.

**2. Approach and Design Strategy**

**a. Initial Exploration and Model Setup**

The initial focus was to find an LLM with fast response time, low token latency, and accurate classification abilities. I selected Google Gemini 1.5 Flash because of its high performance in structured reasoning and cost-effective inference for routing decisions and long prompt contexts.

I started with simple standalone LLM prompts (Gemini only) before realizing that a purely prompt-driven flow couldn't handle iterative user input well. This motivated the transition to LangGraph, which supports stateful multi-agent interactions using a graph-like topology, exactly what this project needed.

**b. Data Source Strategy and LinkedIn Profile Ingestion**

To emulate a real-world usage pattern, I enabled live ingestion of a user's LinkedIn profile using Apify’s LinkedIn Actor API. This allowed scraping relevant profile sections in real-time and converting them into structured data for analysis or rewriting. The scraped data was stored in the LangGraph state and shared across downstream nodes.

The data fields extracted typically include: About, Experience, Projects, Education, and Skills. I built a lightweight cleaning utility (scraper\_utils.py) to preprocess and normalize the extracted HTML and raw text into structured Python dictionaries.

**3. Functional Breakdown and Multi-Agent Architecture**

The system is structured as a multi-node LangGraph where each node is a task-specific agent. The routing of user queries is dynamic and based entirely on LLM intent classification.

**1. career\_qa\_router (Router Node)**

This is the central dispatcher that receives every user query and decides where to route it. It performs the following checks:

* If the user provides a LinkedIn URL → scrape it.
* If the user provides a job description → store it.
* Otherwise → use Gemini to decide between:
  + analyze\_profile
  + job\_fit\_agent
  + enhance\_profile
  + general\_qa

The LLM routing prompt is tightly structured with hardcoded decision labels and rule-based examples to improve classification precision. Routing decisions are executed using the Command(goto=...) method in LangGraph.

**2. analyze\_profile (Profile Review Agent)**

This agent is invoked when the user requests feedback or audit on their LinkedIn or resume. It consumes the profile\_data from state, applies a high-quality Gemini prompt, and returns:

* Top strengths
* Top weaknesses
* Section-by-section ratings
* Three most urgent improvements

This node is useful for debugging what parts of the user’s profile are weak or incomplete and is typically the first step in a career optimization pipeline.

**3. job\_fit\_agent (Job Matching Evaluator)**

When the user provides a job description and asks if they match the role, this agent compares their profile\_data against the stored current\_job\_description. It returns:

* Match score out of 100
* 3 key strengths
* 3 key weaknesses
* 3 specific improvements to raise the match score
* A verdict ("Strong Match", "Needs Improvement", etc.)

This was particularly useful in showing gaps between where a user is and what a specific role demands, especially in misaligned domains (e.g., AI vs Sales).

**4. enhance\_profile (Resume Rewriting Agent)**

When the user wants to improve their resume, rewrite sections, or tailor their About or Experience, this node is activated.

It uses:

* Gemini to rewrite specific sections
* A strict format: 4 bullets max per section, ≤ 25 words per bullet
* Role-aligned enhancements using the job description, if provided
* Markdown-style formatting for readability

The agent keeps hallucination minimal by grounding its rewrite on the actual profile\_data.

5. general\_qa\_node (Out-of-Scope Query Handler)

* Any question not matching the above — such as:
* “What certifications should I get?”
* “How do I get into startups?”
* “What are top AI companies hiring freshers?”

…gets routed here.

This agent has a dedicated Gemini prompt that:

* Leverages profile\_data and job context
* Maintains a helpful, professional tone
* Responds with bullet points or tight paragraphs
* This node ensured graceful fallback behavior for questions that didn’t fit the narrow task buckets.

**4. LangGraph Implementation Details**

**a. Agent State**

I defined a custom AgentState using Python TypedDict:

class AgentState(TypedDict, total=False):

messages: Annotated[list[BaseMessage], operator.add]

profile\_data: Annotated[dict, lambda \_, x: x]

current\_job\_description: Annotated[Optional[str], lambda \_, x: x]

The operator.add annotation ensures that messages accumulate across turns (critical for chat memory).

Each field is mutable and updated in-place.

**b. Interrupt & Resume (Human-in-the-loop)**

Using interrupt("continue\_chat"), I was able to:

* Pause node execution after each response
* Wait for new user input
* Automatically resume from the last state

This enabled a real conversation loop without needing to recompile the graph or reset state. It's what made this "chatbot" feel fluid and responsive.

c. Command-Based Routing

The router uses Command(goto=node\_name) when the LLM decides a specific agent is needed. This allows the graph to jump non-linearly to different nodes without relying on if-else based logic or hardcoded edges.

**d. Circular Graph Flow**

All nodes (agents) return back to the career\_qa\_router, forming a cyclical interaction graph. This ensures continuous conversation and dynamic turn-by-turn context retention.

**5. Challenges and Solutions**

**1. State Not Updating Correctly Across Nodes**

**Problem**: Initially, I returned (interrupt("continue\_chat"), state) which silently failed — LangGraph discarded the state dict in this tuple.

**Solution**: I discovered that nodes must mutate the state dictionary in-place and return only interrupt(...) (without a tuple). Once I stopped using the update= argument with interrupt and instead modified the state directly, memory persisted properly.

**2. Routing to General QA Was Too Generic**

**Problem**: I originally handled out-of-scope questions inside the router by replying with a generic message. This led to non-dynamic, repetitive replies.

**Solution**: I built a separate general\_qa\_node and routed any general\_qa decisions there using Command(goto="general\_qa\_node"). The dedicated prompt here made answers dynamic and natural.

**3. Gemini Returned Invalid JSON / Strings**

**Problem**: I experimented with JSON-based LLM routing outputs and encountered decode errors when Gemini returned malformed JSON or extra whitespace.

**Solution**: I shifted to a strict instruction-based prompt asking Gemini to return one of four words only — no punctuation, no explanations. Then I used str.strip().lower() to safely parse the raw decision.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**4. Profile Data or JD Getting Lost Between Turns**

**Problem**: Any update to profile\_data or current\_job\_description would get overwritten if I didn’t include them explicitly in the update dict.

**Solution**: I standardized all in-place state["messages"] = messages and state["profile\_data"] = scraped, etc., ensuring values persist in memory without needing explicit merging.

**5. Handling Bad Inputs Gracefully**

**Problem**: Users sometimes gave invalid LinkedIn URLs or job descriptions mid-convo, and the graph crashed or behaved unpredictably.

**Solution**: I added robust try/except blocks and fallback messages in both the router and scraper nodes. If scraping failed or the JD wasn’t found, the user is prompted clearly and the graph resumes.

**6. Debugging LangGraph’s Internal Flow**

**Problem**: Understanding why nodes weren’t executing or resuming correctly was initially confusing.

**Solution**: I added debug print() statements and state inspection between graph turns. I also referred heavily to LangGraph’s documentation on Command, interrupt, and StateGraph flows.

**6. Technical Highlights**

* Modular, agentic design using LangGraph, allowing for dynamic LLM-in-the-loop workflows.
* Live profile ingestion using Apify, enabling real-time LinkedIn analysis.
* Natural routing using Gemini prompts, no keywords or classification rules.
* Stateful memory through custom agent state + LangGraph’s operator.add merge strategy.
* Graceful fallbacks and human-like interaction, including clarifying questions, tone matching, and iterative conversation.