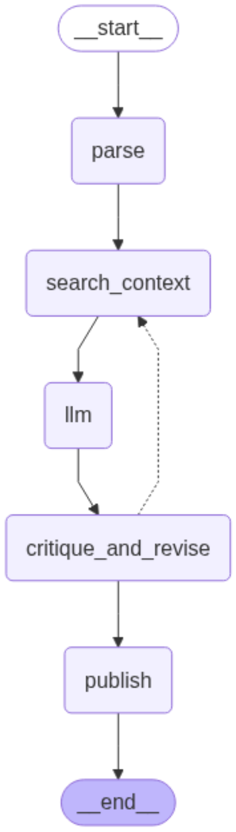
Research Conducted

I took the LangGraph Crash Course provided in the guidelines, which helped me understand how to model LLM workflows using a state machine abstraction. I explored Tavily's Search API, examined its parameters like search_depth, include_raw_content and include_answer, and tested its behavior on basic and advanced queries. I also experimented with formatting LLM input and output to ensure consistency and accuracy during RAG cycles.



Project Flow

The system is built as a LangGraph pipeline that takes a complex query, splits it (when it can) into subqueries, searches for relevant information using Tavily search, then generates a response using Cohere's Command-R model. In addition, the pipeline includes the following:

- critique_and_revise: A node that critiques and revises the LLM output.
- A conditional feedback loop: If the critique detects an issue with the response from the LLM, it re-triggers the search.
- publish: A final node that wraps and prints the output in a user-friendly format.

Technical Decisions

I considered integrating additional Tavily endpoints, such as crawl, extract, or combining multiple sources into a deeper retrieval pipeline. However, due to the assignment deadline (and other home-assignments I worked on concurrently), I focused on refining the core search and response cycle.

Another case is the decision to remove streaming of output tokens, which could be done via less structured code (removing LLM class must be done because the receiving of the streaming is done with self([HumanMessage(content=prompt)], stream=True) which requires the output to be generated in the last node, avoiding the usage of the output node and the conditional edge). As this was not a requirement as defining a good structure, This tradeoff made me decide with going strict with the assignment requirements.