**Project Development Journal**

* **Challenges Encountered**

**1. Model Selection under Resource Constraints**

* **What Happened:**  
  Initially, I intended to use open-source LLMs from Hugging Face to build the chatbot. However, I quickly ran into **memory limitations** when trying to load these models locally. Even using quantized versions didn't help much due to the restricted RAM/VRAM available on my system.
* **What It Meant:**  
  This became a bottleneck early on, preventing experimentation with popular open models like mistral, llama, etc., without offloading the workload to cloud-based GPU runtimes.
* **How I Resolved It:**  
  I decided to switch to **ChatGroq**, a hosted LLM service offering fast inference and low-latency interaction. This allowed me to move forward without worrying about local resource limits.
* **Reflection:**  
  Using Llama-3.3 turned out to be great.

**2. Tool Invocation via Agent System**

* **What Happened:**  
  Initially, I wanted to use Langchain's **agent-based system** to let the chatbot decide when to use tools like document organization or meeting slot extraction. The idea was to give the model freedom to invoke tools when it recognized relevant queries.
* **What Went Wrong:**  
  Despite setting up tools with proper descriptions and function bindings, the agent **struggled with decision-making latency**. The model often spent too long deliberating which tool to invoke or misunderstood the context. Maybe the prompt given to agent was not sufficient thus forcing the model to think in many iterations. So learnt a lot and next time certainly would use the agent system.
* **My Solution:**

I moved to a **RAG-based approach**:

* + Created vector stores for user data and resumes.
  + Used a **single smart prompt** that instructs the model how to answer queries and what to do for tasks like scheduling or organizing files.
  + Incorporated **conditional checks** (e.g., if "organize documents" in query) in the template itself.
* **Reflection:**  
  Simplifying the architecture made the app faster and easier to debug
* **What Worked Well**
* **RAG + Unified Prompt Strategy:**  
  Replacing multi-tool agents with a single intelligent prompt (that covered answering questions, scheduling meetings, and prompting for document organization) yielded faster and efficient response. It reduced complexity and gave more predictable results.
* **Vector Store Setup:**  
  Splitting user information and resume embeddings into separate persistent directories was a smart modular approach. It also improved retrieval relevance.
* **ChatGroq Integration:**  
  Fast, reliable inference from ChatGroq made it a practical choice for real-time chatbot interactions.
* **Streamlit Callback Handlers:**  
  Using StreamlitCallbackHandler allowed live streaming responses from the model—adding polish to the user experience.
* **What Didn’t Work**
* **Langchain Agents for Small Toolsets:**  
  The model either hesitated too long or made irrelevant tool calls, making the agent system overkill for a lightweight assistant.
* **Prompt Misalignment for Tool Use:**  
  Even with a clear instruction like “mention file organization only if relevant,” the model sometimes failed to suggest the tool when the query was ambiguous (e.g., “how should I organize my files?”). The lack of keyword triggering led to missed opportunities.
* **What Could Be Improved**
* **Storing User Data in Databases:**  
  Currently, user info and files are saved in **local directories**, which is not scalable or secure. Moving to a proper **database and cloud storage** system (e.g., Firebase, Supabase, or MongoDB with S3) would improve performance, data safety, and user management.
* **Tool Routing with Stricter Agents or Custom Logic:**  
  While agents were too heavy in their default form, introducing **custom-built agents with clearer intent classification** or **rule-based dispatch** could help in handling more complex toolchains in the future.
* **Enhanced Prompt Tuning for Tool Discovery:**  
  The current prompt sometimes misses user intent for document organization unless exact keywords are used. The user has to query about the functionalities of chatbot to know what can it do .
* **Multi-turn Memory & Session Persistence:**  
  Storing session context (not just chat history) could improve the user experience by maintaining continuity over multiple queries or sessions.
* **Conclusion**

This was my **first time building a chatbot**, and I got to learn a lot—from handling tool use, prompt design, vector store management, to real-time interaction via Streamlit. The process helped me understand both the strengths and limits of modern LLM integrations.

I’m definitely **interested in adding more features like** adding making history aware chatbot and those provided in the problem statement. Overall, it was a **great learning experience** and a solid foundation for future AI assistants I plan to build.