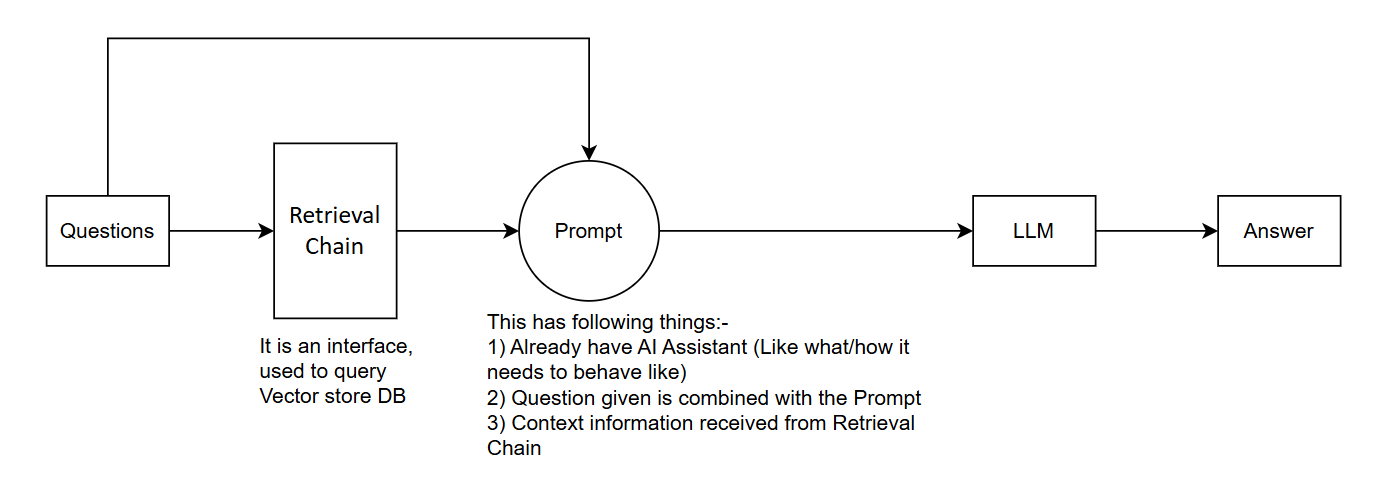
Lang-Chain:- Framework to create/develop Genai applications.

Interview – How to use langchain to develop agentic AI application.

Lang-Serve:-

Lang-Smith:- From prototype to production

Rag:- We have GenAI application and thousands of pdfs. Application answers based on the pdfs.

* Module 1Rag (Retrival Augmented Generation)
  + Data Ingestion-> Load data from Data Sourses
  + Data Transformation-> Divide data into Text/ Splitting of Data
  + Embedding-> Text embedding/ Convert text and convert it into vectors
    - We need to convert data into vectors, for cosine similarities. E.g- words like good and nice gets mapped together or close.
  + Vector Store-> Stroring of vector
    - FAISS
    - Cromoa DB
    - ASTRA
  + We can query data directly in vector DB (technique is called- **Approximate Nearest Neighbor (ANN)**). From here we get the context of data/ meaning of data.
* Module2
  + Question asked by user
  + Retrieval Chain-> Interface to query Vector store DB. Provides context information.
  + Prompt-> Predefine instructions provided to AI Assistant
  + LLM
  + Output response
  + 

2) Query processing phase

Pipeline

**🔁 RAG Pipeline: Step-by-Step**

**1. Data Ingestion**

* **Sources**:
  + PDFs
  + CSVs
  + Knowledge base articles
  + All fetched from **Cloud Storage** (e.g., GCP Cloud Storage buckets)
* **Tools/Tech**:
  + Python scripts or LangChain document loaders
  + Use GCSFileSystem or gcsfs for accessing cloud storage
  + LlamaIndex's file readers for PDF, CSV

**2. Preprocessing & Parsing**

* **Parsing**:
  + Extract text from PDF/CSV/HTML/Markdown/etc.
  + Normalize text (remove headers, footers, special characters)
* **Chunking**:
  + Split large texts into manageable chunks (e.g., 500–1000 tokens) using:
    - **LlamaIndex's TextSplitter**
    - or LangChain’s RecursiveCharacterTextSplitter
* **Metadata Attachment**:
  + Attach source metadata (file name, type, section headers)

**3. Embedding Generation**

* **Embedding Model**:
  + Use LLM embedding model via **GCP Vertex AI**, or open models like:
    - text-embedding-ada-002 (OpenAI)
    - **LLaMA 2 embeddings** or Sentence Transformers
* **Process**:
  + Convert each text chunk into a vector (embedding)
  + Store chunk-to-embedding mappings

**4. Vector Store Indexing**

* **Vector Database**:
  + Use **FAISS** to store embeddings for fast semantic similarity search
  + Index type: Flat, HNSW, or IVF depending on retrieval speed vs. accuracy tradeoff
* **Tooling**:
  + Use LlamaIndex's FAISS integration
  + Or manage FAISS directly via Python

**5. Knowledge Index Construction**

* **Index Type**:
  + LlamaIndex's VectorStoreIndex (or GPTVectorStoreIndex)
  + LangChain’s VectorStoreRetriever connected to FAISS
* **Optimization**:
  + Tune retrieval parameters like top\_k, similarity\_threshold

**6. Query Interface**

* **User Input**:
  + Query comes via FastAPI REST endpoint
* **Retrieval**:
  + Convert query to embedding
  + Retrieve top-k similar chunks using FAISS
* **Tools**:
  + LangChain RetrievalQA or LlamaIndex’s QueryEngine

**7. RAG Response Generation**

* **LLM Integration**:
  + Use **LLaMA 2** or **GCP Vertex AI LLMs** to:
    - Accept query + retrieved context
    - Generate grounded, contextual response
* **Prompt Engineering**:
  + Use templates that include:
    - User query
    - Retrieved document context
    - Instructions for tone, brevity, structure

**8. Response Delivery**

* **API Response**:
  + Deliver final answer via FastAPI
  + Include:
    - Answer
    - Source references (optional)
    - Confidence score (if available)

**9. Logging, Monitoring & Metrics**

* **Performance Tracking**:
  + Log query time, response quality
  + Track user queries in **BigQuery** for analytics
* **Monitoring**:
  + Use GCP tools like Stackdriver or Cloud Logging
* **Outcome**:
  + Achieved **65% reduction in average query resolution time**

**10. (Optional) Feedback Loop / Continuous Learning**

* Collect user feedback on responses
* Re-rank or retrain embeddings/models if needed