**SESSION:**

* + **Retrieval-Augmented Generation (RAG)**
    - Why we need RAG
    - How RAG Works.
    - RAG vs Fine-Tuning
    - What are Indexes
    - Embedding Models
    - Documents Loaders
    - Text Splitters
    - Vector Stores

**What is RAG?**

**Retrieval-Augmented Generation (RAG)** is a technique that combines **retrieval of external knowledge** with the **generation ability of Large Language Models (LLMs).**

Instead of relying only on what the model has been trained on (which may be outdated or incomplete), RAG allows the model to **fetch relevant, up-to-date, or domain-specific information** from a knowledge source (like a database, documents, or a vector store) and use it while generating responses.

When building RAG apps, we follow a pipeline:

**Why do we need RAG?**

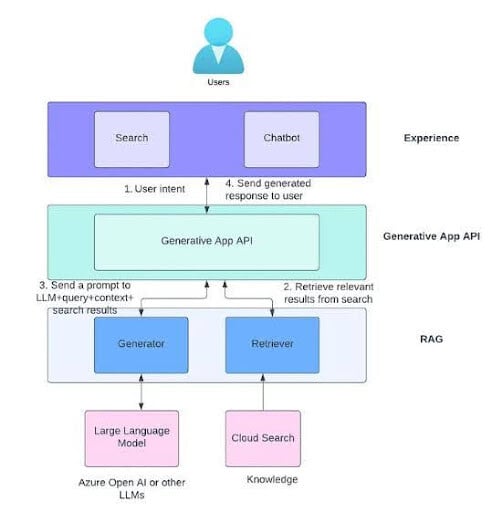
* LLMs have a **knowledge cutoff** (they don’t know everything).
* They can **hallucinate** answers (make up things).
* Businesses often want to use **their private data** (PDFs, docs, manuals, support tickets).

RAG solves this by **retrieving trusted information** first, then using it to guide the generation.

**How RAG Works (Steps)**

1. **User Query:** A user asks a question.
2. **Retrieval:** Search relevant documents from a knowledge base (using embeddings + vector databases like FAISS, Pinecone, Weaviate, Chroma).
3. **Augmentation:** Add the retrieved context to the original **prompt**.
4. **Generation:** The LLM generates an answer using both the user’s question and the retrieved information.

**Examples……………..**



**Key Components of RAG**

* **LLM** (for generation) – e.g., Gemini, GPT, LLaMA
* **Retriever** (for searching data) – e.g., FAISS, Pinecone
* **Vector Embeddings** (for semantic search) – e.g., OpenAI embeddings, Hugging Face sentence transformers
* **Generator (Chain** – combines retrieval + generation in LangChain**)**

**Indexes and Embeddings**

**1. Embeddings**

* An **embedding** is just a way of turning text into numbers.
* These numbers capture the **meaning** of the text, not just the words.
* Once text is turned into numbers, we can compare meanings using **math** (similarity search).

**Analogy:**  
Think of embeddings as **GPS coordinates for sentences**.

* “King” and “Queen” will be close to each other.
* “Apple (fruit)” and “Banana” will be close.
* “Apple (company)” will be closer to “Microsoft” than “Banana.”

**2. Indexes**

* An **Index** is like the **index at the back of a book** — it helps you **find information faster**.
* In RAG, indexes are built on embeddings so we can quickly find which chunks of text are most relevant to a query.

**Types in RAG Context:**

1. **Document Index** → Simple keyword search (like Ctrl+F).
2. **Vector Index** → Stores embeddings, allows semantic search (meaning-based, not just words).

**Analogy:**

* If you ask “What is AI?” → A keyword index looks for the word “AI.”
* A vector index understands that “Artificial Intelligence” and “AI” mean the same thing.

Together, they make **RAG possible**:  
User Query → Embedding → Index Search → Relevant Docs → LLM Answer

|  |  |  |
| --- | --- | --- |
| Feature | RAG (Retrieval-Augmented Generation) | Fine-Tuning |
| How it works | Model **fetches external info** from a knowledge base/vector store and uses it in responses. | Model is **trained again** with new data so it learns patterns permanently. |
| Analogy | Student **using Google** during exam. | Student **memorizing notes** before exam. |
| Use Case | When knowledge changes often (e.g., news, product catalogs, research papers). | When tasks are fixed (e.g., customer support FAQs, sentiment analysis). |
| Update | Easy → just update the knowledge base. | Hard → need to retrain the model. |
| Cost | Cheaper (no retraining, just storage). | Expensive (compute + training time). |
| Accuracy | Relies on retrieval quality. | Very accurate for trained tasks. |
| Flexibility | Works with many topics, adaptable. | Narrow focus on trained data. |

**Documents → Chunking → Embeddings → Vector Store → Retrieval → LLM → Output**

**Document Loaders**

* **These are connectors that help you** load data from different sources into LangChain.
* Instead of manually copying text, loaders automate pulling content.

**Text Splitters**

* Large documents are too big for LLM context windows.
* Text Splitters break them into smaller, overlapping chunks for better retrieval.
* Overlap ensures that important context isn’t lost between chunks.

**Vector Stores**

* After splitting, chunks are converted into embeddings (vectors).
* These embeddings are stored in a Vector Store.
* When a user asks a query → it is embedded → Vector Store finds the most similar chunks.

**Popular Vector Stores:**

* FAISS (local, lightweight)
* Pinecone (cloud, scalable)
* Chroma (open-source, developer-friendly)
* Weaviate, Milvus, Qdrant (production-grade)
* PostgreSQL with pgvector (hybrid option)

**Hamara Wala Kaam - Technical Workflow : )**

* + Document Ingestion
  + Text Chunking
  + Embeddings Generation (Embedding Models)
  + Storage in Vector Stores
  + Retrieval