RAG System

Part 1: The "Why" - Why Do We Even Need RAG?

Before we dive into what RAG is, let's understand the problem it solves.
Imagine you're talking to a standard, off-the-shelf LLM like ChatGPT. These models are incredibly smart, but they have a few fundamental limitations:

- The Knowledge Cutoff: An LLM's knowledge is frozen in time. The model I am based on was trained on data up to a certain point. If you ask me about events that happened after that date, I won't know about them.
    - Example: If you ask, "Who won the 2025 Oscar for Best Picture?", a standard LLM trained until 2023 would have no idea.
- Hallucinations: Sometimes, when an LLM doesn't know the answer, it "hallucinates" – it makes up a plausible-sounding but completely false answer. It does this because its main goal is to predict the next most likely word, not necessarily to be truthful.
- Lack of Specificity: A general-purpose LLM doesn't know about your private or specific data. It hasn't read your company's internal documents, your personal study notes, or a niche scientific domain's latest research papers.
  So, the core problem is: How can we make an LLM answer questions using up-to-date, specific, or private information without it making things up?
  Does this initial problem make sense? This is the foundation for our entire discussion.

Part 2: The "What" - Introducing Our Solution: RAG

This is where RAG comes in.
RAG stands for Retrieval-Augmented Generation.
Let's break down that name:

- Retrieval: This means "to find and get information." Think of retrieving a book from a library.
- Augmented: This means "to enhance or add to." We are adding the information we found to something else.

- Generation: This is what LLMs do best – they generate text (words, sentences, answers).
  So, in simple terms, RAG is a technique that first retrieves relevant information from an external knowledge source and then augments (adds) that information to the user's question before asking the LLM to generate the final answer.
  Think of it like this:

> Analogy: The Open-Book Exam
> Imagine you have a very smart student (the LLM) who has to take an exam.
>
> - A standard LLM is like a student taking the exam from memory alone. They know a lot, but they might forget things, get details wrong, or not know about very specific topics.
> - A RAG system is like that same student taking an open-book exam. Before answering a question, the student can look through the official textbook (the external knowledge source) to find the exact, correct information. They then use that information to write a perfect, fact-based answer.
>   RAG gives the LLM a "textbook" to consult in real-time.

Ready to move on to how this "open-book exam" actually works behind the scenes?

Part 3: The "How" - A Step-by-Step Guide to the RAG Pipeline

This is the core of our lesson. The RAG process can be split into two main phases.
Phase A: The Preparation (Indexing the Knowledge)
This is the "studying" phase that happens before the user ever asks a question. We need to prepare our "textbook" or knowledge base so it's easy to search.

- Load Documents: First, we gather our knowledge source. This could be anything: a set of PDFs, a company's internal website, a database of customer support tickets, or a collection of medical research papers.
- Chunking: We can't give the LLM an entire 500-page book at once. It's too much information. So, we break the documents down into smaller, manageable pieces, or "chunks." These could be paragraphs, pages, or sections of a certain size.

- Create Embeddings (The Magic Step): This is a crucial concept. We need a way for the computer to understand the meaning of our text chunks. We use a special model called an Embedding Model to convert each text chunk into a list of numbers, called a vector.
  - Think of these vectors as a kind of "GPS coordinate" for meaning. Chunks of text with similar meanings will have vectors that are "close" to each other in mathematical space. For example, the vector for "How much does a car cost?" will be very close to the vector for "What is the price of an automobile?".
- Store in a Vector Database: We take all these vectors (and their corresponding text chunks) and store them in a special kind of database designed for incredibly fast searching of vectors. This is our searchable library. Popular vector databases include Pinecone, Chroma, and FAISS.
  This preparation phase is a one-time setup (though you can update it with new documents later). Our library is now ready.
  Phase B: The Real-Time Process (Answering a Question)
  This happens every time a user submits a query.
- User Query: The user asks a question, for example: "What are the new HR policies on remote work for 2025?"
- Embed the Query: We use the exact same embedding model from the preparation phase to convert the user's question into a vector.
- Search/Retrieve: Now, we take the user's query vector and use it to search our vector database. The database performs a similarity search to find the text chunk vectors that are mathematically closest to the query vector. It pulls out, say, the top 3-5 most relevant chunks.
  - In our example, it would find the chunks of text from our HR documents that talk about "remote work," "work from home," and "2025 policies."
- Augment the Prompt: This is the key "Augmented" part of RAG. We don't just send the user's question to the LLM. Instead, we construct a new, more detailed prompt. It looks something like this:

  > CONTEXT:
  >
  > - "[Chunk 1: ...the new policy for 2025 states that employees can work remotely up to 3 days a week...]"*
  > - "[Chunk 2: ...approval for remote work must be obtained from a direct manager...]"*
  > - "[Chunk 3: ...all remote work must be conducted from within the country...]"*
  >
  > QUESTION:
  > "What are the new HR policies on remote work for 2025?"
  >
  > INSTRUCTION:
  > "Based only on the context provided above, answer the user's question."

- Generate the Answer: We send this entire augmented prompt to the LLM. The LLM now has all the factual information it needs right in front of it. It doesn't need to rely on its old, internal memory. It can now generate a precise, factual answer based only on the provided context.
    - Final Answer: "According to the new HR policies for 2025, employees are permitted to work remotely for up to three days per week. This requires approval from a direct manager and must be conducted from within the country."
      Take a moment to review these steps. Does the flow from indexing the documents to generating the final, context-aware answer make sense?
      Part 4: The Payoff - Why is RAG a Big Deal?
      Let's circle back to our original problems and see how RAG solves them:
- Solves Knowledge Cutoff: You can constantly update your vector database with new documents. The LLM's knowledge is no longer frozen; it's as fresh as your data.
- Reduces Hallucinations: By instructing the LLM to answer only based on the provided context, you dramatically reduce its tendency to make things up. It's "grounded" in your documents.
- Enables Domain-Specific & Private Knowledge: This is its superpower. You can now build a chatbot for your company's internal wiki, a medical assistant that uses the latest research, or a legal tool that understands a specific set of case files.
- Provides Citations: Since you know which chunks were retrieved, you can easily tell the user where the information came from (e.g., "This answer was based on document 'HR-Policy-2025.pdf', page 4."). This builds trust and verifiability.
- Cost-Effective: Fine-tuning or retraining an entire LLM on new data is incredibly expensive and time-consuming. RAG is a much cheaper and faster way to infuse new knowledge into your system.

- Tooling: Popular frameworks like LangChain and LlamaIndex that make building RAG systems much easier.
- Evaluation: How do we measure if a RAG system is good? We look at the quality of the retrieved documents and the faithfulness of the final answer.

How To Setup the Project

```
npm i @langchain/pinecone @langchain/core @pinecone-database/pinecone
@langchain/community @google/genai @langchain/google-genai
@langchain/textsplitters dotenv pdf-parse readline-sync
```

Phase One: Storing vectors into vector DB

Step0: Get Gemini API key and Pinecone API

```
GEMINI_API_KEY=AIzaSyCrghVulwaPO0xJSPLAetCLN_FlGunvw6g PINECONE_API_KEY=
pcsk_ep4SA_AnAbYzdUFMS35YqnoTy2jsLGkThsEiqfLxXqR7FUAXTgj5dSWEentk93FyLjfLR
PINECONE_ENVIRONMENT= us-east-1 PINECONE_INDEX_NAME= rohitnegi9
```

```
import * as dotenv from 'dotenv'; dotenv.config();
```

Step 1: Load the PDF file

```
import { PDFLoader } from '@langchain/community/document_loaders/fs/pdf';
const PDF_PATH = './dsa.pdf'; const pdfLoader = new PDFLoader(PDF_PATH);
const rawDocs = await pdfLoader.load();
```

```
console.log(JSON.stringify(rawDocs, null, 2)); console.log(rawDocs.length)
```

Step 2: Create the chunk of the PDF

```
import { RecursiveCharacterTextSplitter } from '@langchain/textsplitters';
const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize:
CHUNK_SIZE, chunkOverlap: CHUNK_OVERLAP, }); const chunkedDocs = await
textSplitter.splitDocuments(rawDocs);
```

```
console.log(JSON.stringify(chunkedDocs.slice(0, 2), null, 2));
```

Step 3: Initializing the Embedding model

```
import { GoogleGenerativeAIEmbeddings } from '@langchain/google-genai';
const embeddings = new GoogleGenerativeAIEmbeddings({ apiKey:
process.env.GEMINI_API_KEY, model: 'text-embedding-004', });
```

Step4: Initialize Pinecone Client

```
import { Pinecone } from '@pinecone-database/pinecone'; const pinecone =
new Pinecone(); const pineconeIndex =
pinecone.Index(process.env.PINECONE_INDEX_NAME);
```

Step 5: Embed Chunks and Upload to Pinecone

```
import { PineconeStore } from '@langchain/pinecone'; await
PineconeStore.fromDocuments(chunkedDocs, embeddings, { pineconeIndex,
maxConcurrency: 5, });
```

Phase 2: Query Resolving phase

Step0: Take the user Input from terminal

```
import readlineSync from 'readline-sync'; async function main(){ const
userProblem = readlineSync.question("Ask me anything--> "); await
chatting(userProblem); main(); } main();
```

Step 1: Convert the user query into embedding(vector)

```
import { GoogleGenerativeAIEmbeddings } from '@langchain/google-genai';
const embeddings = new GoogleGenerativeAIEmbeddings({ apiKey:
process.env.GEMINI_API_KEY, model: 'text-embedding-004', }); const
queryVector = await embeddings.embedQuery(question);
```

Step 2: Search Relevant document into vector DB

```
import { Pinecone } from '@pinecone-database/pinecone'; const pinecone =
new Pinecone(); const pineconeIndex =
pinecone.Index(process.env.PINECONE_INDEX_NAME); const searchResults =
await pineconeIndex.query({ topK: 10, vector: queryVector,
includeMetadata: true, }); const context = searchResults.matches
.map(match => match.metadata.text) .join("\n\n---\n\n");
```

Step 3: Query + Context to LLM

```
import { GoogleGenAI } from "@google/genai"; const ai = new
GoogleGenAI({}); const History = [] History.push({ role:'user', parts:
[{text:question}] }) const response = await ai.models.generateContent({
model: "gemini-2.0-flash", contents: History, config: { systemInstruction:
`You have to behave like a Data Structure and Algorithm Expert. You will
be given a context of relevant information and a user question. Your task
is to answer the user's question based ONLY on the provided context. If
the answer is not in the context, you must say "I could not find the
answer in the provided document." Keep your answers clear, concise, and
educational. Context: ${context} `, }, }); History.push({ role:'model',
parts:[{text:response.text}] }) console.log("\n");
console.log(response.text);
```

How to Enhance our user Query

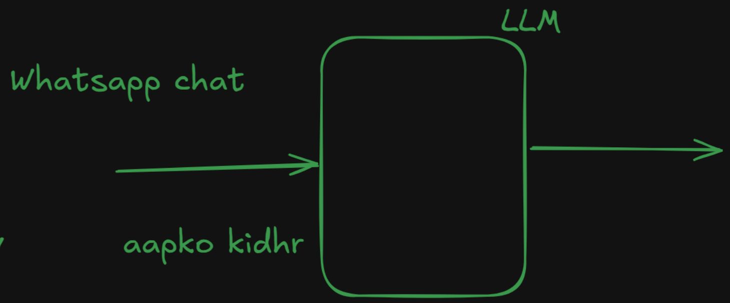
```
async function transformQuery(question){ History.push({ role:'user',
parts:[{text:question}] }) const response = await
ai.models.generateContent({ model: "gemini-2.0-flash", contents: History,
config: { systemInstruction: `You are a query rewriting expert. Based on
the provided chat history, rephrase the "Follow Up user Question" into a
complete, standalone question that can be understood without the chat
history. Only output the rewritten question and nothing else. `, }, });
History.pop() return response.text }
```

RAG --> Retrieval-Augmented Generation

Problem Statement

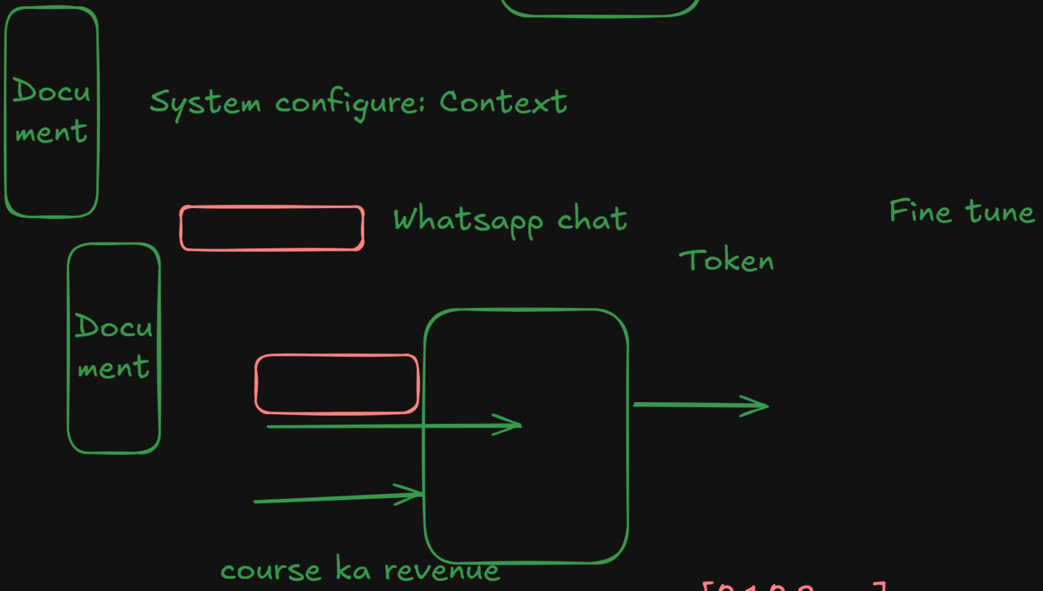
Hallcuniation

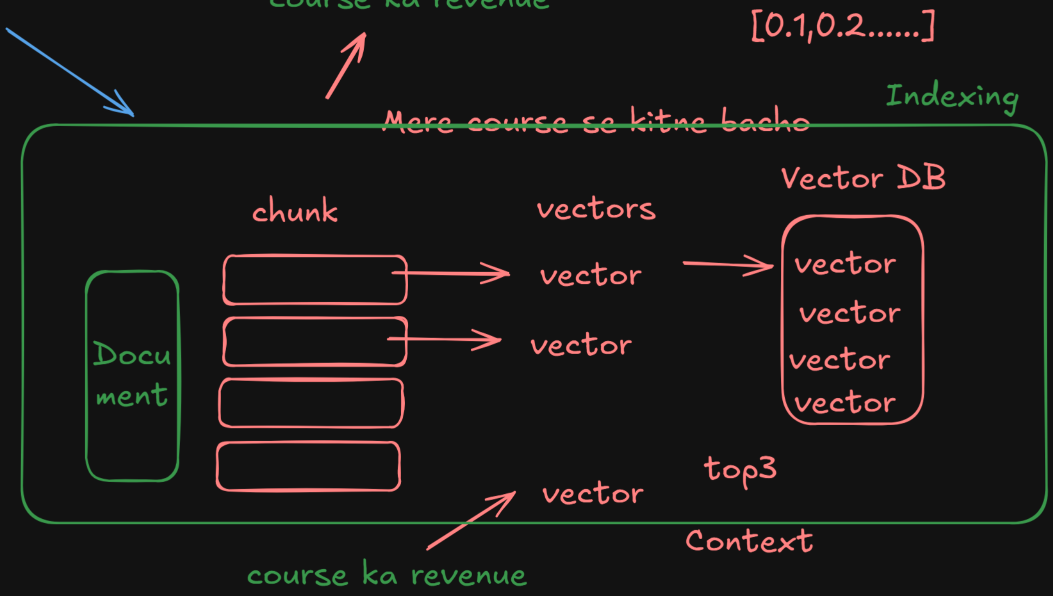
Fine tune

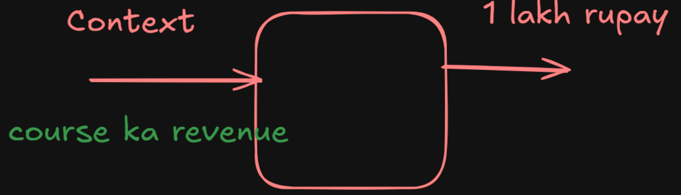


Coder Army

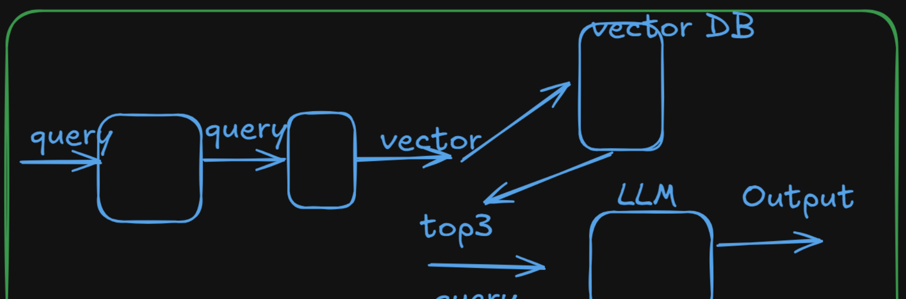
System configure: Context







query RAG System



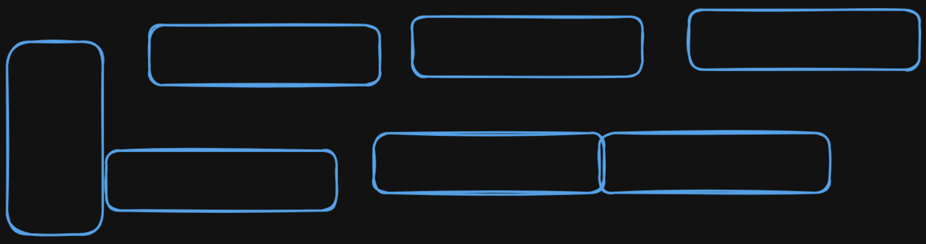
RAG

1-1000
801-1800

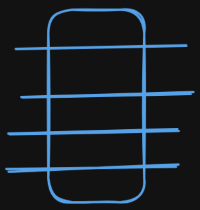
Langchain(utility function)

Phase 1:

PDF --> Load karke

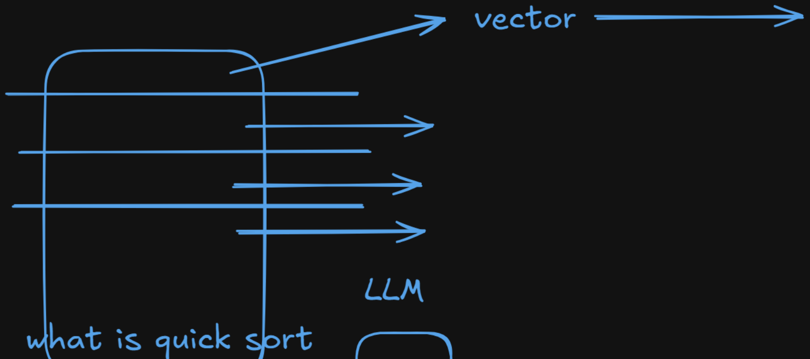


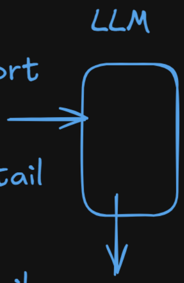
Phase 2: Chunking



phase 3: Convert each chunk into vector

phase 4: Store them into vectorDB (pinecone)





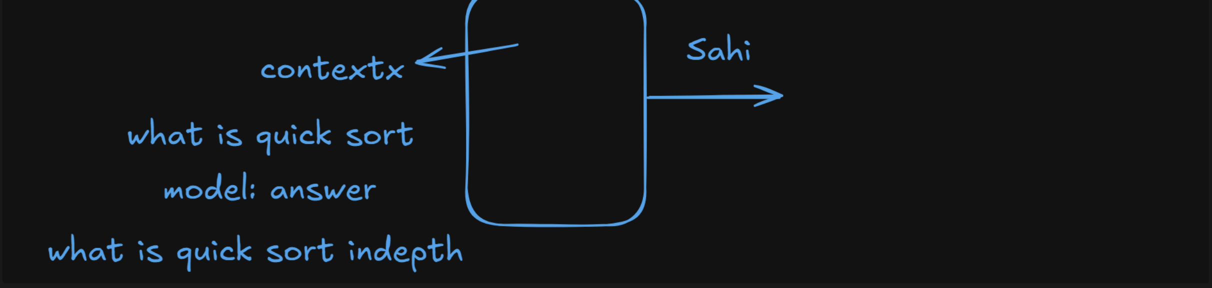
what is quick sort
model: answer
explain it in detail

contextx: Vector DB

explain it in detail

what is quick sort indepth

what is quick sort



```
// PDF load karne ka index.js file import * as dotenv from 'dotenv';
dotenv.config(); import { PDFLoader } from
'@langchain/community/document_loaders/fs/pdf'; import {
RecursiveCharacterTextSplitter } from '@langchain/textsplitters'; import {
GoogleGenerativeAIEmbeddings } from '@langchain/google-genai'; import {
Pinecone } from '@pinecone-database/pinecone'; import { PineconeStore }
from '@langchain/pinecone'; async function indexDocument() { const
PDF_PATH = './dsa.pdf'; const pdfLoader = new PDFLoader(PDF_PATH); const
rawDocs = await pdfLoader.load(); console.log("PDF loaded"); // Chunking
karo const textSplitter = new RecursiveCharacterTextSplitter({ chunkSize:
1000, chunkOverlap: 200, }); const chunkedDocs = await
textSplitter.splitDocuments(rawDocs); console.log("Chunking Completed");
// vector Embedding model const embeddings = new
GoogleGenerativeAIEmbeddings({ apiKey: process.env.GEMINI_API_KEY, model:
'text-embedding-004', }); console.log("Embedding model configured") //
Database ko bhi configure // Initialize Pinecone Client const pinecone =
new Pinecone(); const pineconeIndex =
pinecone.Index(process.env.PINECONE_INDEX_NAME); console.log("Pinecone
configured") // langchain (chunking,embedding,database) await
PineconeStore.fromDocuments(chunkedDocs, embeddings, { pineconeIndex,
maxConcurrency: 5, }); console.log("Data Stored succesfully") }
indexDocument();
```

```
import * as dotenv from 'dotenv'; dotenv.config(); import readlineSync
from 'readline-sync'; import { GoogleGenerativeAIEmbeddings } from
'@langchain/google-genai'; import { Pinecone } from '@pinecone-
database/pinecone'; import { GoogleGenAI } from "@google/genai"; const ai
= new GoogleGenAI({}); const History = [] async function
transformQuery(question){ History.push({ role:'user', parts:
[{text:question}] }) const response = await ai.models.generateContent({
model: "gemini-2.0-flash", contents: History, config: { systemInstruction:
`You are a query rewriting expert. Based on the provided chat history,
rephrase the "Follow Up user Question" into a complete, standalone
question that can be understood without the chat history. Only output the
```