RAG (Retrieval Augmented Generation)

useful starting links

- https://www.youtube.com/watch?v=ZaPbP9DwBOE (just good for a speedy run down on ai concepts)
- https://github.com/griminir/rag-learning (repo)

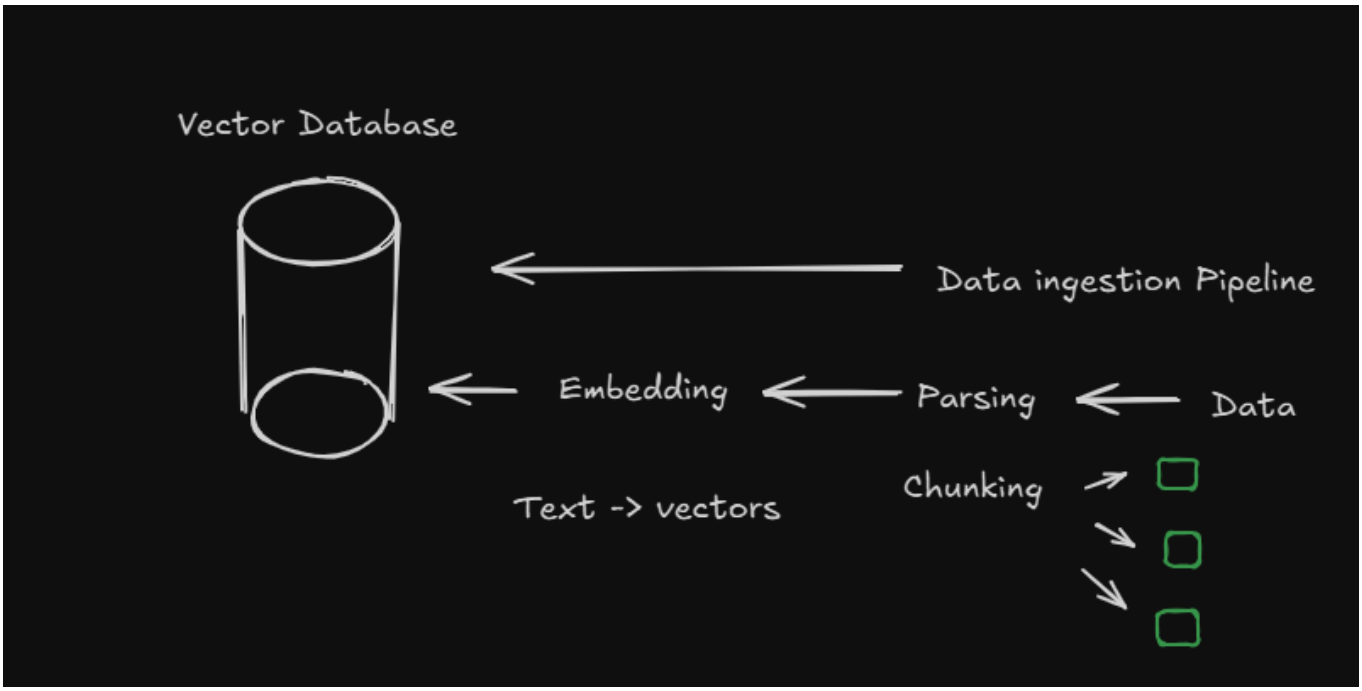
why RAG

- training an AI model is expensive
- RAG lets us add onto the LLM knowledge base after it has been trained
- This lets us use a LLM as the base and then add our own knowledge bases on top of the LLM
- used to retrieve specific facts
- users need auditability, traceability and minimizing hallucinations

why not to use RAG

- Small knowledge bases and static knowledge bases think <50 pages
- Queries that mostly requires reasoning, not looking up facts
  - multi-step reasoning
  - planning
  - mathematical logic
  - code generation from scratch
- Context that can be gleamed from a small set of summaries
- if user tasks are predominantly reasoning and not look up/summarize RAG adds complexity without performance gain
- documents are unstructured and you cannot fix it
  - no consistent structure
  - mixed media (complex spreadsheets, diagrams)
  - contains implicit knowledge not written in text
  - if data is not text retrievable rag will not work reliably and can lead to hallucinations
- the RAG pipeline takes time so if your latency requirements are low its not a good fit
- it needs to be maintained

RAG Pipeline



Data:

- can can be any format but should be able to be text based

Parsing:

- this is where chunking happens where we split up the data into smaller pieces for retrieval accuracy and performance

Chunking

- chunking is more about fine tuning things that require larger context should have larger chunks, like a technical manual
- but things like an FAQ can have smaller chunks
  Typical chunking process

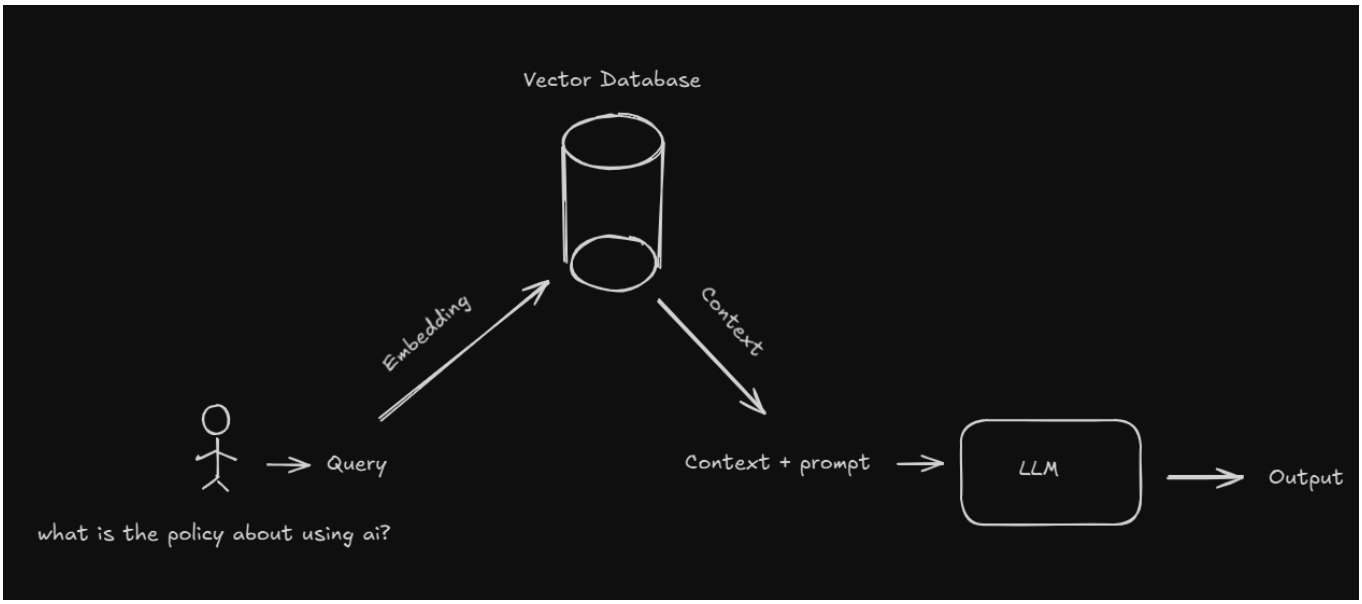
1. normalize and clean the input
   a. remove boilerplate text
   b. standardize formatting
   c. extract readable content from pdf, html and so on
2. apply chunking strategy
   a. token based chunking (most common in RAG)
   b. sentence-based chucking
   c. paragraph-based chunking
   d. semantic chucking using embeddings to detect topic shifts
   example: 500-tokens chunk with 50-tokens overlap
3. store meta data

- Each chunk gets metadata (document ID, page number, section, timestamps, etc.)

embedding

- this stores the parsed chunks into the vector database
- there are lots of different models for this they have different implementations

what happens when a query is sent to a RAG system



picture of traditional retrieval pipeline

- Perplexity is a RAG systems it's connected to tools, retrievers, web search

1. user query
2. retriever takes the embedding of the query and to the vector DB and gets the context and feeds it to the LLM
3. then with the context and the prompt the LLM generates and output

Document structure

this can be down with a library called Langchain (python) or Langchain.js (JavaScript/typescript)
Langchain.js is under the MIT license

has 2 core components

- page_content
  - this is the text of whatever we want
  - string
- metadata
  - additional information about the file

- things like filename, page count, timestamp and so on (this can be changed as you see fit)
- object
- this helps us can help us narrow down queries later when we retrieve the data, think of it like tags

Data ingestion pipeline

Data Ingestion

you might have a lot of different files:

- pdf
- html
- excel
- Database
- csv
  how do you read the file?
  how do you parse the data?
  how to convert this into a document structure?

all of this is done by using the correct loader (Langchain.js has multiple different loaders)
[Document loaders - Docs by LangChain]

code example:

```
import { readFile } from "fs/promises";
import { Document } from "@langchain/core/documents";

// Step 1: Load documents
export async function loadDocuments(filePath: string) {
  console.log("Loading documents...");
  const text = await readFile(filePath, "utf-8");
  const docs = [new Document({ pageContent: text, metadata: { source: filePath
} })];

  console.log(`Loaded ${docs.length} document(s)`);
  console.log(docs);
  return docs;
}
```

Chunking

we have to divide the documents we have into smaller parts
this is because there is a fixed context size for the embedding process and later for the LLM model

this is just 1 way of chunking
all type of chunking needs overlap this helps keep context whole when to feed the ai
chunks need to be smaller to get embedded think of the overlap not as wasted extra space but as page numbers that tells you the order to put the story together

code example:

```
import { RecursiveCharacterTextSplitter } from "@langchain/textsplitters";
import { Document } from "@langchain/core/documents";

// Step 2: Split documents into chunks
export async function splitDocuments(docs: Document[]) {
  console.log("\nSplitting documents into chunks...");
  const splitter = new RecursiveCharacterTextSplitter({
    chunkSize: 50,
    chunkOverlap: 10,
  });

  const chunks = await splitter.splitDocuments(docs);

  console.log(`Created ${chunks.length} chunks`);

  chunks.forEach((chunk, i) => {
    console.log(`\n--- Chunk ${i + 1} (${chunk.pageContent.length} chars) ---
`);
    console.log(chunk.pageContent);

    // Show overlap with next chunk
    if (i < chunks.length - 1) {
      const nextChunk = chunks[i + 1].pageContent;
      const currentEnd = chunk.pageContent;
      let overlap = "";
      for (let len = Math.min(currentEnd.length, nextChunk.length); len > 0;
len--) {
        const currentSuffix = currentEnd.slice(-len);
        const nextPrefix = nextChunk.slice(0, len);

        if (currentSuffix === nextPrefix) {
          overlap = currentSuffix;
          break;
        }
```

```
      }

      if (overlap) {
        console.log(`↓ OVERLAP (${overlap.length} chars): "${overlap}"`);
      }
    }
  });

  console.log(chunks);
  return chunks;
}
```

Embedding

this takes the chunks and turns the text into a vector some where in the database with often multiple dimensions instead of just a x and y axis.
there are a lot of different embedding models, but its important to use the same one for retrieving and creating embeddings (it is possible to make a bridge but its costly)

Embedding models - Docs by LangChain

example code:

```
import { Document } from "@langchain/core/documents";
import { pipeline as transformersPipeline } from "@xenova/transformers";

// Step 3: Generate embeddings for chunks (only for new chunks)
export async function embedChunks(chunks: Document[]) {
  if (chunks.length === 0) {
    console.log("\n✓ No new chunks to embed (all exist in database)");
    return [];
  }

  console.log("\nLoading embedding model...");

  const extractor = await transformersPipeline('feature-extraction',
'Xenova/all-MiniLM-L6-v2');

  console.log(`Generating embeddings for ${chunks.length} new chunks...`);

  const vectors = await Promise.all(
    chunks.map(async (c) => {
      const output = await extractor(c.pageContent, { pooling: 'mean',
normalize: true });
      return Array.from(output.data);
```

```
    })
  );

  const embedded = chunks.map((chunk, i) => ({
    metadata: chunk.metadata,
    text: chunk.pageContent,
    embedding: vectors[i],
  }));

  console.log(`✓ Created ${embedded.length} new embeddings, each with
${embedded[0].embedding.length} dimensions`);

  return embedded;

}
}```


### Vector store

now we need to store our embeddings
this is where we add our vectors we create from the documents and also where
we will retrieve them from
simple docker file:
```yml
# Docker Compose for ChromaDB
services:
  chromadb:
    image: chromadb/chroma:latest
    ports:
      - "8000:8000"
    volumes:
      - ./data/vector_store:/data
    environment:
      - IS_PERSISTENT=TRUE
      - ANONYMIZED_TELEMETRY=FALSE
```

code example for initializing the store:

```
import { ChromaClient } from "chromadb";

const COLLECTION_NAME = "test_collection";
// Initialize ChromaDB vector store
export async function initVectorStore() {

  console.log(`Initializing ChromaDB...`);
```
```

```
  // ChromaDB requires a running server
  // To start: chroma run --path ./data/vector_store --port 8000
  const client = new ChromaClient({
    path: "http://localhost:8000"
  });



  // Get or create collection
  let collection;

  try {
    collection = await client.getOrCreateCollection({
      name: COLLECTION_NAME,
      metadata: { "hnsw:space": "cosine" }, // Use cosine similarity
    });
    console.log(`✓ Collection "${COLLECTION_NAME}" ready`);
  } catch (error) {
    console.error("✗ ChromaDB server not running at http://localhost:8000");
    console.error("Start with: chroma run --path ./data/vector_store --port
8000");
    throw error;
  }
  return { client, collection };
}
```

metadata: { "hnsw:space": "cosine" } — configures the index:

- “hnsw” indicates using the HNSW approximate nearest-neighbor index.
- "space": "cosine" tells the index to use cosine similarity (Chroma stores this as a distance internally).

simple test for if the docker ChromaDB works:

```
import { initVectorStore } from "./initVectorStore";

async function testVectorStore() {
  try {
    // Initialize the vector store
    const { client, collection } = await initVectorStore();

    // Check collection info
    const count = await collection.count();
    console.log(`\nCollection has ${count} items`);
```

```
    // Test adding a simple vector
    await collection.add({
      ids: ["test-2"], // Unique ID for the vector should be uuid for real db
      embeddings: [[0.1, 0.2, 0.3, 0.4]], // Simple 4D vector
      documents: ["This is a test document"],
      metadatas: [{ source: "test" }],
    });
    console.log("✓ Successfully added test vector");

    // Verify it was added
    const newCount = await collection.count();
    console.log(`Collection now has ${newCount} items`);

    // Query the vector
    const results = await collection.query({
      queryEmbeddings: [[0.1, 0.2, 0.3, 0.5]], // play with this
      nResults: 10, // Return up to 10 results
    });

    console.log("\nQuery results:");
    console.log(`Found ${results.ids[0].length} matches:`);
    console.log(results);
    console.log("\n✓ Vector store is working correctly!");
  } catch (error) {
    console.error("✗ Test failed:", error);
  }
}


testVectorStore();
```

Cosine Similarity (the mathematical concept)

- 1 = identical (pointing same direction)
- 0 = orthogonal (unrelated)
- -1 = opposite
  ChromaDB converts similarity to distance so that "closer to 0 = more similar"
  is consistent with other distance metrics like Euclidean distance.
  so in chromaDB, distance = 1 - cosine_similarity

Code example for storing the embeddings in ChromaDB:

```
import { createHash } from "crypto";
import type { Collection } from "chromadb";
```

```typescript
interface EmbeddedChunk {
  metadata: any;
  text: string;
  embedding: number[];
}

// Step 4: Store embeddings in ChromaDB with smart deduplication
export async function storeEmbeddings(
  collection: Collection,
  embedded: EmbeddedChunk[],
  clearSource?: string
) {
  console.log("\nStoring embeddings in ChromaDB...");

  // Generate deterministic IDs based on content hash
  const newIds = embedded.map(e => {
    const hash = createHash('sha256')
      .update(e.text)
      .update(e.metadata.source || '')
      .digest('hex');
    return hash.substring(0, 36);
  });

  // Find and remove stale chunks first (to check if any work is needed)
  let staleIds: string[] = [];
  if (clearSource) {
    try {
      const existing = await collection.get({
        where: { source: clearSource }
      });
      staleIds = existing.ids.filter(id => !newIds.includes(id));
    } catch (error) {
      // No existing chunks
    }
  }

  // OPTIMIZATION #4: Skip DB operations if nothing changed
  const hasNewChunks = embedded.some(e => e.embedding.length > 0); // Check if
there are new embeddings
  if (!hasNewChunks && staleIds.length === 0) {
    console.log(` Fast path: All ${embedded.length} chunks already in
database with correct data, skipping upsert`);
    return newIds;
  }
```

```
  // Extract embeddings, documents, and metadata
  const embeddings = embedded.map(e => e.embedding);
  const documents = embedded.map(e => e.text);
  // ChromaDB only accepts flat metadata (string, number, boolean)
  const metadatas = embedded.map(e => ({
    source: e.metadata.source || 'unknown'
  }));

  // Upsert chunks (updates existing, inserts new - no re-embedding needed for
unchanged chunks)
  await collection.upsert({
    ids: newIds,
    embeddings,
    documents,
    metadatas,
  });

  console.log(`√ Upserted ${embedded.length} chunks`);

  // Remove stale chunks if any were found
  if (staleIds.length > 0) {
    await collection.delete({
      ids: staleIds
    });
    console.log(`√ Removed ${staleIds.length} stale chunks`);
  } else {
    console.log(`√ No stale chunks found`);
  }

  console.log(`Sample IDs: ${newIds.slice(0, 3).join(", ")}...`);
  return newIds;
}
```

not sure if this is the optimal way to store things but it works

pipeline

code example:

```
import { loadDocuments } from "./loadDocuments";
import { splitDocuments } from "./splitDocuments";
import { embedChunks } from "./embedChunks";
import { storeEmbeddings } from "./storeEmbeddings";
import { initVectorStore } from "./initVectorStore";
import { checkExistingChunks } from "./checkExistingChunks";
import { hasFileChanged, updateFileCache, getCachedChunkCount } from
```

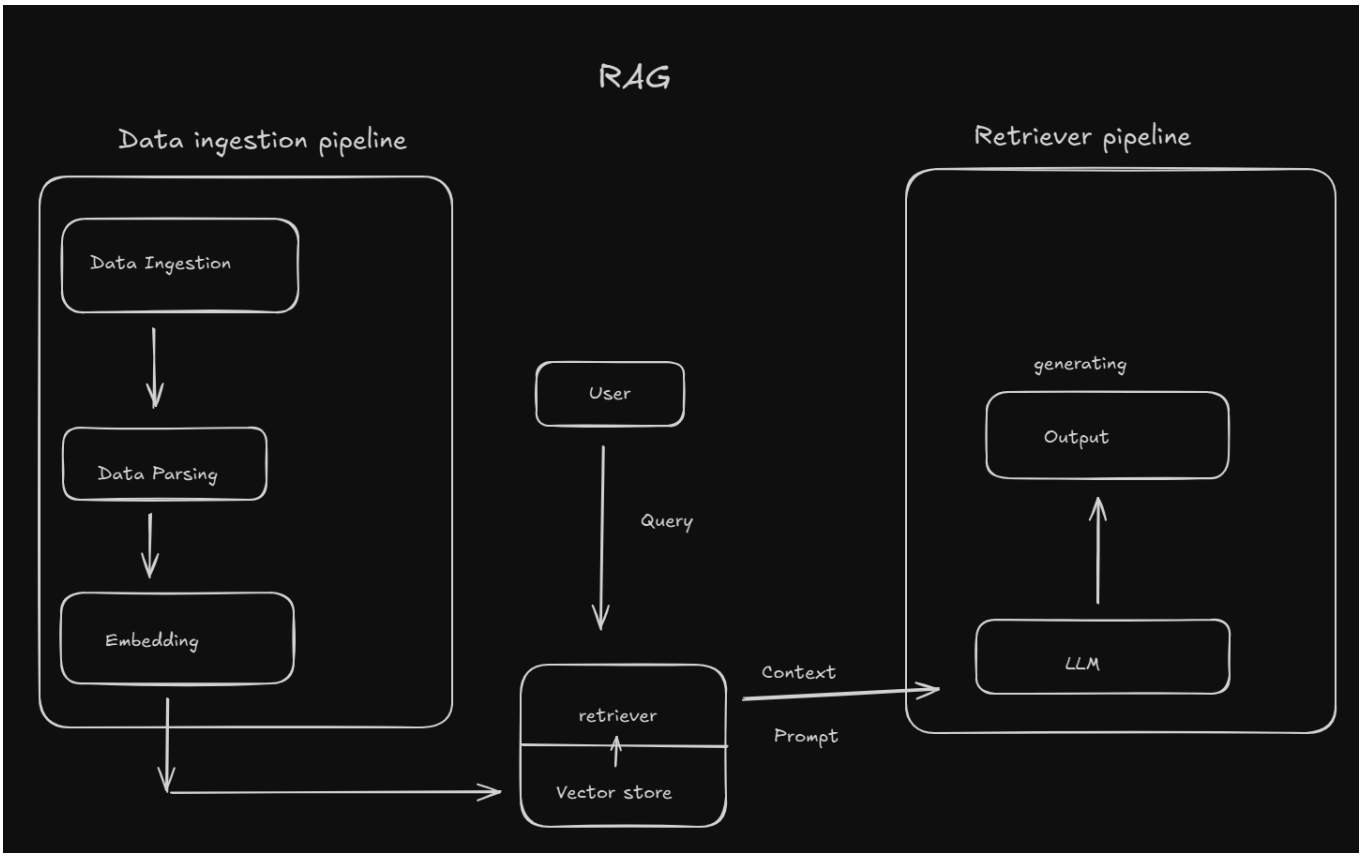
```
"./fileCache";
import { discoverFiles } from "./discoverFiles";

// Main pipeline with smart embedding optimization
export async function pipeline(inputPath: string | string[]) {
  // Discover all files to process
  const filePaths = await discoverFiles(inputPath, {
    extensions: [".txt", ".md"],
    recursive: false
  });

  if (filePaths.length === 0) {
    console.log(`\n⚠️ No files found matching: ${inputPath}`);
    return { ids: [], collection: null };
  }

  console.log(`\n📂 Found ${filePaths.length} file(s) to process:`);
  filePaths.forEach(fp => console.log(`   - ${fp}`));

  // Initialize vector store ONCE for all files
  const { collection } = await initVectorStore();
  let totalProcessed = 0;
  let totalNew = 0;
  let totalReused = 0;
  let totalSkipped = 0;

  // Process each file
  for (const filePath of filePaths) {
    console.log(`\n${'='.repeat(60)}`);
    console.log(`📄 Processing: ${filePath}`);
    console.log('='.repeat(60));

    // OPTIMIZATION #1: Check if file changed - skip everything if unchanged
    const fileChanged = await hasFileChanged(filePath);
    if (!fileChanged) {
      const cachedCount = await getCachedChunkCount(filePath);
      console.log(`⚡ Fast path: File unchanged, ${cachedCount} chunks already
in database`);
      totalSkipped++;
      totalProcessed += cachedCount || 0;
      continue;
    }

    // step 1 data ingestion
    const docs = await loadDocuments(filePath);
```

```
    // step 2 chucking
    const chunks = await splitDocuments(docs);

    // OPTIMIZATION #2: Check which chunks already exist (to avoid re-
embedding)
    const { needsEmbedding, existing } = await checkExistingChunks(collection,
chunks, filePath);

    // step 3 embedding
    // Only embed new chunks
    const newEmbeddings = await embedChunks(needsEmbedding);

    // Combine new embeddings with existing ones
    const allEmbeddings = [
      ...newEmbeddings,
      ...existing.map(e => ({
        metadata: e.chunk.metadata,
        text: e.chunk.pageContent,
        embedding: e.embedding
      }))
    ];

    // step 4 storing the embeddings
    const ids = await storeEmbeddings(collection, allEmbeddings, filePath);

    // Update cache after successful processing
    await updateFileCache(filePath, ids.length);
    totalProcessed += ids.length;
    totalNew += newEmbeddings.length;
    totalReused += existing.length;
    console.log(`✓ File complete: ${ids.length} chunks
(${newEmbeddings.length} new, ${existing.length} reused)`);
  }

  // Summary
  console.log(`\n${'='.repeat(60)}`);
  console.log(`✓ Pipeline complete!`);
  console.log(`   Files processed: ${filePaths.length -
totalSkipped}/${filePaths.length}`);
  console.log(`   Files skipped (unchanged): ${totalSkipped}`);
  console.log(`   Total chunks: ${totalProcessed}`);
  console.log(`   New embeddings: ${totalNew}`);
  console.log(`   Reused embeddings: ${totalReused}`);
  console.log('='.repeat(60));
```

```
    return { totalProcessed, totalNew, totalReused, collection };
}
```

optimization can be found in the project repo
as well as a test pipeline file made that ai mocked up

Relationship between the 2 pipelines (Lord of the RAG: Two pipelines)



so earlier we looked at the pipeline for data ingestion now how do we use the data we added

- user sends a query "can i bring my dog into work on Friday?"
- this triggers the retriever that embeds the wording of the query into vectors
- then based on a algorithm it looks for vectors that are similar to the vectors from the query and grabs that as context and sends a prompt to the LLM

now what is a prompt in this case?

- user query: "can i bring my dog into work on Friday?"
- context: policy for animals at work
- system instructions: "only use context that is given, you love animals"

then the LLM generates an answer based on all of this:
LLM output: "Yes, you can bring your adorable little friend to work as the company has a pets

allowed policy"

different set up:

- user query: "can i bring my dog into work on Friday?"
- context: policy for animals at work
- system instructions: "only use context that is given, cite your sources, if you dont have enough information say so"

LLM output: yes as per animals at work policy you can bring your pet to work. (probably a way to maybe link to it if we want or user can go and look themselves to double check)

retriever

code example of retriever:

```
import type { Collection } from "chromadb";
import { pipeline as transformersPipeline } from "@xenova/transformers";

interface RetrieverOptions {
  k?: number; // default number of results
  modelName?: string; // embedding model name
  minScore?: number; // minimum normalized similarity [0..1] to keep a hit
(default 0) fine tune to your dataset
}

interface RetrievedDocument {
  id: string;
  score: number; // similarity score normalized to [0..1] (1 = most similar).
computed as ( (1 - distance) + 1 ) / 2
  text: string;
  metadata: any;
}

// Factory: create a retriever bound to a ChromaDB collection
export async function createRetriever(collection: Collection, options?:
RetrieverOptions) {

  const opts = Object.assign({ k: 5, modelName: "Xenova/all-MiniLM-L6-v2",
minScore: 0 }, options || {});

  // Lazy-load the embedding extractor once and reuse
  console.log("Loading embedding model for retriever...");

  const extractor = await transformersPipeline("feature-extraction",
```

```
opts.modelName);

  // Embed a single piece of text into a vector (mean pooling, normalized)
  async function embedText(text: string): Promise<number[]> {

    const out = await extractor(text, { pooling: "mean", normalize: true });

    return Array.from(out.data);
  }

  // Retrieve nearest documents for a text query
  // optional overrideMinScore filters out results with normalized score <
overrideMinScore
  async function retrieve(query: string, k?: number, overrideMinScore?:
number): Promise<RetrievedDocument[]> {
    const n = k || opts.k || 5;

    const effectiveMinScore = typeof overrideMinScore === "number" ?
overrideMinScore : (opts.minScore ?? 0);

    // Create embedding for the query
    const queryVector = await embedText(query);

    // Query ChromaDB collection for nearest neighbors
    const results = await collection.query({
      queryEmbeddings: [queryVector],
      nResults: n,
      include: ["metadatas", "documents", "distances"] as any,
    });

    // The ChromaDB response is columnar: arrays inside arrays (one query ->
index 0)
    const returnedIds = results.ids?.[0] || [];
    const returnedDocuments = results.documents?.[0] || [];
    const returnedMetadatas = results.metadatas?.[0] || [];
    const returnedDistances = results.distances?.[0] || [];



    // Convert distance -> similarity for cosine: similarity = 1 - distance
    // then normalize cosine from [-1,1] to [0,1] and apply the similarity
filter
    const retrievedResults: RetrievedDocument[] = [];

    returnedIds.forEach((docId: string, idx: number) => {
      const rawDistance = returnedDistances[idx];
```

```
      const cosineSimilarity = typeof rawDistance === "number" ? 1 -
rawDistance : 0;

      const normalizedScore = Math.max(0, Math.min(1, (cosineSimilarity + 1) /
2));

      if (normalizedScore >= effectiveMinScore) {
        retrievedResults.push({
          id: docId,
          score: normalizedScore,
          text: returnedDocuments[idx] || "",
          metadata: returnedMetadatas[idx] || {},
        });
      }
    });

    return retrievedResults;
  }

  return { retrieve };
}
```

retriever testing code in the repo

the reviver is where we can fine tune a lot about what we want to give as context to the LLM tings like needs to be x similar vector wise needs to be x metadata. basically handles all the context gathering.

augmented generation

this is where we add our personality and rules for the generation the LLM makes
as well as bundling in the context we get from the retriever

code example of augmented generation:

```
import { ChatOllama } from "@langchain/ollama";
import { HumanMessage, SystemMessage } from "@langchain/core/messages";

// Configuration
const MODEL_NAME = "llama3.1:8b";
const OLLAMA_BASE_URL = "http://localhost:11434"; // this should be able to
hook into an existing Ollama server instance
```

```
// System prompt instructing the LLM to answer based on provided context
const SYSTEM_PROMPT_STRICT = `You are a helpful assistant that answers
questions based on the provided context.

Your answers should be:
- Based ONLY on the information in the context provided
- Concise and directly addressing the question
- Honest about uncertainty - if the context doesn't contain enough
information, say so

If the context doesn't contain relevant information to answer the question,
respond with:
"I don't have enough information in the provided documents to answer that
question."

Do NOT make up information or use knowledge outside of the provided context.`;



// System prompt that allows fallback to general knowledge
const SYSTEM_PROMPT_FALLBACK = `You are a helpful assistant that answers
questions.

Your answers should be:
- Based primarily on the provided context when relevant
- Concise and directly addressing the question
- Clearly indicate when you're using general knowledge vs the provided context

If the context doesn't contain relevant information, you may use your general
knowledge to answer,
but clearly indicate that the answer is not from the provided documents.`;


interface GenerateOptions {
  stream?: boolean;
  allowFallback?: boolean;
  onToken?: (token: string) => void;
}

// Initialize the Ollama chat model (lazy singleton)
let llmInstance: ChatOllama | null = null;

function getLLM(): ChatOllama {
  if (!llmInstance) {
    llmInstance = new ChatOllama({
      model: MODEL_NAME,
```

```
      baseUrl: OLLAMA_BASE_URL,
      temperature: 0.3, // Lower temperature for more focused, factual
responses
    });
  }

  return llmInstance;
}
```

so now that we have our prompt set and our LLM model ready
we can start generating an answer for the user

code example of generation:

```
/**
 * Generate an answer using the LLM based on retrieved context
 * @param query - The user's question
 * @param context - The retrieved document content to use as context
 * @param options - Generation options (stream, allowFallback, onToken
callback)
 * @returns The generated answer
 */
export async function generateAnswer(
  query: string,
  context: string,
  options?: GenerateOptions
): Promise<string> {
  const llm = getLLM();
  const stream = options?.stream ?? false;
  const allowFallback = options?.allowFallback ?? false;
  const onToken = options?.onToken;

  // Choose system prompt based on fallback setting
  const systemPrompt = allowFallback ? SYSTEM_PROMPT_FALLBACK :
SYSTEM_PROMPT_STRICT;

  // Build the prompt with context and question
  const userPrompt = `Context:
${context}

Question: ${query}

Answer:`;

  const messages = [
    new SystemMessage(systemPrompt),
```

```
    new HumanMessage(userPrompt),
  ];



  if (stream && onToken) {
    // Streaming mode
    const streamResponse = await llm.stream(messages);
    let fullAnswer = "";

    for await (const chunk of streamResponse) {
      const token = typeof chunk.content === "string" ? chunk.content : "";
      fullAnswer += token;
      onToken(token);
    }

    return fullAnswer;
  } else {
    // Non-streaming mode
    console.log("Generating answer with Ollama...");
    const response = await llm.invoke(messages);

    // Extract the text content from the response
    const answer = typeof response.content === "string"
      ? response.content
      : JSON.stringify(response.content);

    return answer;
  }
}
```

this is an example with multiple flag options
like if you want it streamed or if you want it

data flow:

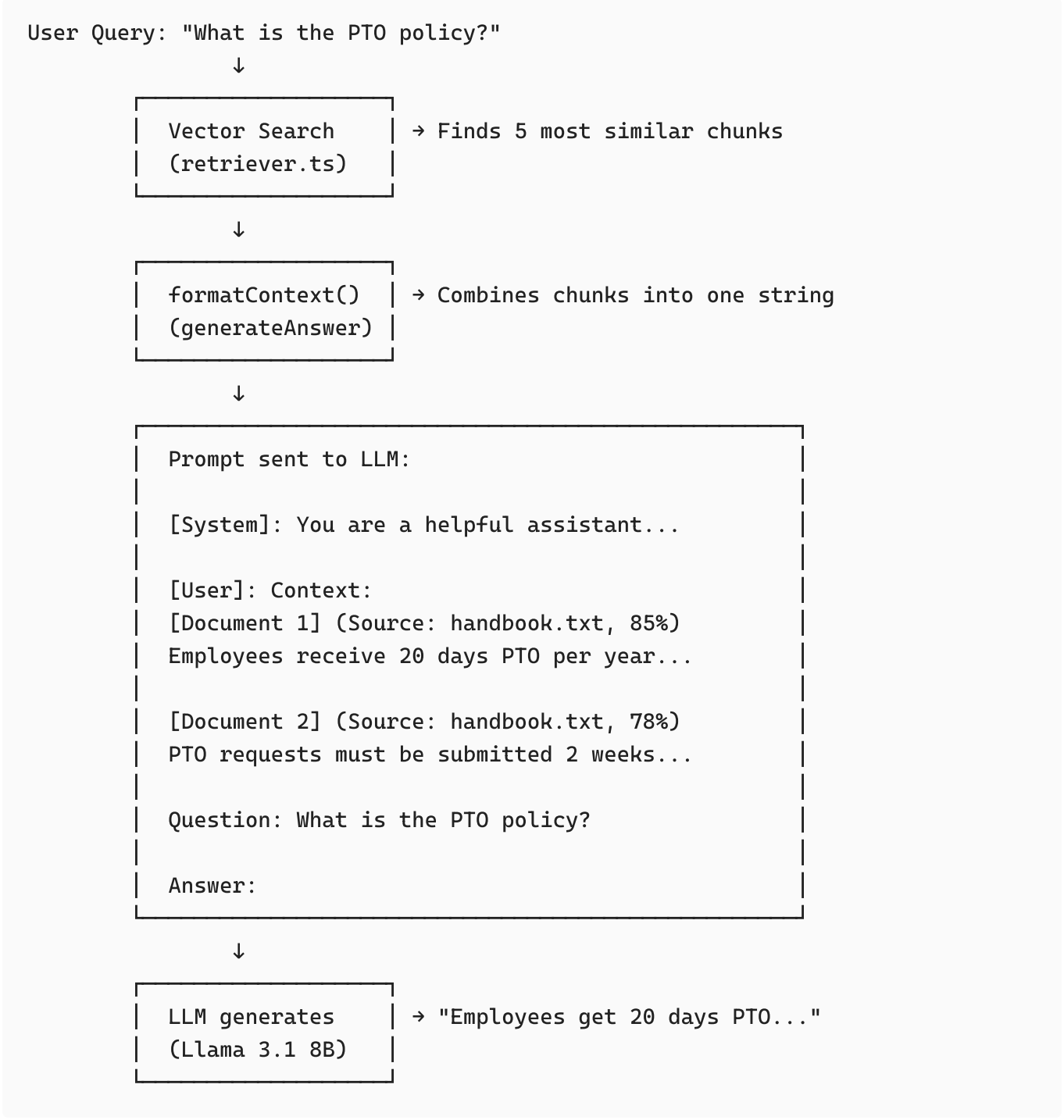
```
User Query
    ↓
ragChain.ts calls retriever
    ↓
Retrieved docs → formatContext() → context string
    ↓
generateAnswer(query, context, options)
    ↓
ChatOllama generates answer
```

```
    ↓
Return to user (streamed or buffered)
```

the repo has a ragChain.ts that is meant for an application and testRag.ts that you can use in the CLI to do things

understanding fine tuning and context

the flow:



all LLMs have different context window. llama3.1:8b has 128k tokens ~ 500k Approx Characters

what if we have to little context?

- the LLM does not have enough information to answer
- with strict mode in this repo it will not be able to answer
- with fallback it will use general knowledge and the info might be wrong from the docs provided

what could be some causes for too little context (fine tuning):

- --k is too small
- --minScore is too strict and it filters out useful documents
- chunks are too small and context is lost
- user query does not match any index

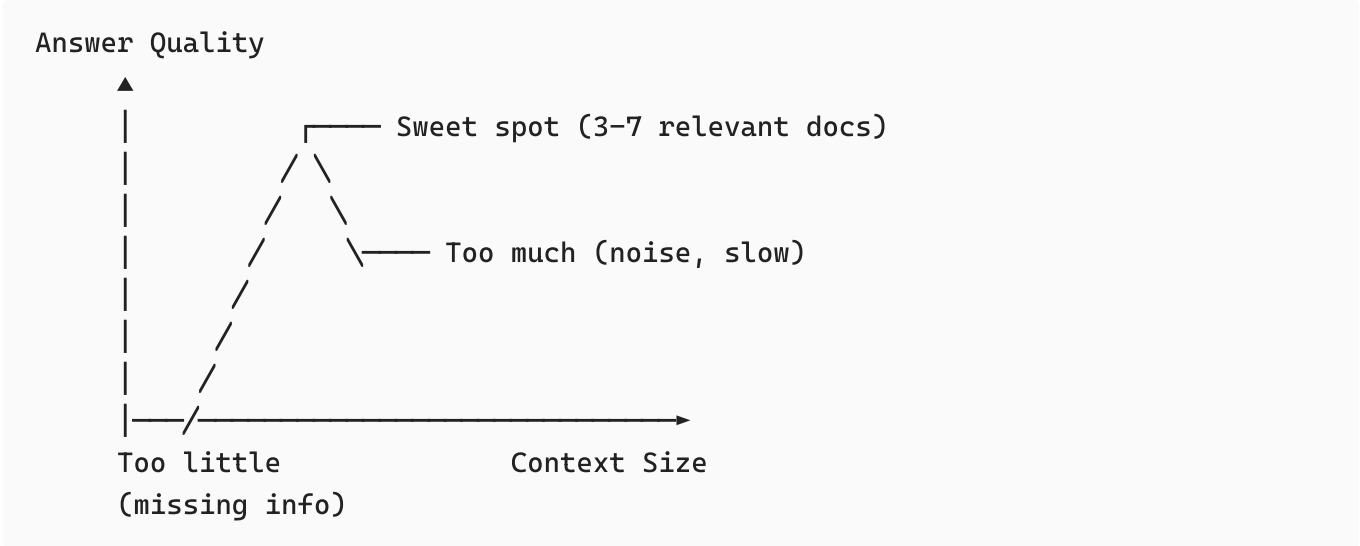
what happens when there is too much context:

- performance degrades, more tokens = slower generation
- if using a paid api increased costs (ollama since its local this is not an issue but might lead to higher server traffic)
- irrelevant noise for the LLM to sort through

what could be the cause of too much context (fine tuning):

- --k too many documents are retrieved
- --minScore too low 0.1 (grabs random stuff if it just seems to be a document)
- chunks are too large so the chunk has to much different info

this is an example 3-7 is not always the golden spot



things that need work

- Still have not found a good solution for when unique id's when a chuck contains the same text in the same file (this can be fixed with bigger chunks most of the time)
- Still need to add other loaders and find out how that works with the pipeline

Sanity has it own RAG system

this is still in beta and is a paid feature, but once you understand how this little repo works it might be easier to understand the underlying mechanics of this sanity feature

- Semantic search for relevant recommendations
- https://www.sanity.io/docs/compute-and-ai/embeddings-index-api-overview