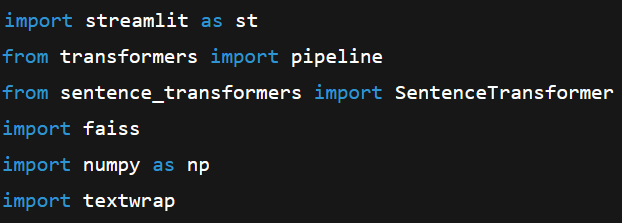
# **Detailed Project details 📘**

# **A — High-level flow**

1. **Load two ML models: a sentence embedder and a generative QA model.**
2. **Split the user’s text into overlapping chunks.**
3. **Convert chunks to embeddings and build a FAISS index for fast similarity search.**
4. **For a user question, compute its embedding, search the index for top-k similar chunks.**
5. **If similarity is good, feed those chunks + the question into the generative model (Flan-T5) and return an answer.**
6. **The whole thing is wrapped in a Streamlit UI so users can paste text and ask questions.**

**B — Imports (what and why)**

****

**streamlit as st — Streamlit is the web UI framework used to create the interactive app. st is the usual alias.**

**pipeline from transformers — a high-level API that loads text generation / seq2seq models (here used for Flan-T5).**

**SentenceTransformer — from the sentence-transformers library; used to create sentence embeddings (semantic vectors).**

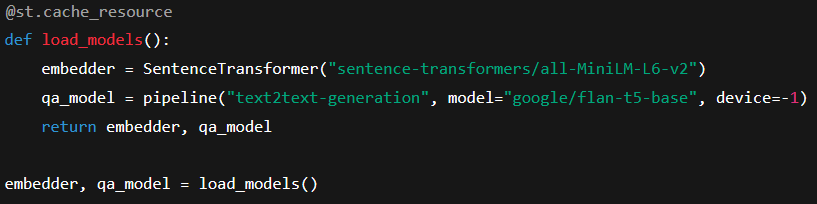
**faiss — Facebook AI Similarity Search: a fast, C++/CUDA library (with Python bindings) for similarity search over vectors.**

**numpy as np — numerical arrays operations and conversions (FAISS and models expect numpy arrays).**

**textwrap — standard lib to nicely wrap long answer strings for display.**

# **C — line-by-line explanation**

### **1) Loading models efficiently (caching)**



**@st.cache\_resource:**

* **Streamlit decorator that caches expensive-to-create resources (models, DB connections, etc.) across app reruns.**
* **Without caching, Streamlit would reload models every time a widget interaction triggers a rerun, which is slow and wasteful.**
* **Use for objects that are relatively static and expensive to construct.**

**def load\_models(): — a function to encapsulate model creation.**

**SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2"):**

* **Loads a pre-trained embedding model (All-MiniLM-L6-v2). This model maps sentences to fixed-size vectors.**
* **It downloads model weights the first time (or loads from cache) — another reason to cache it.**

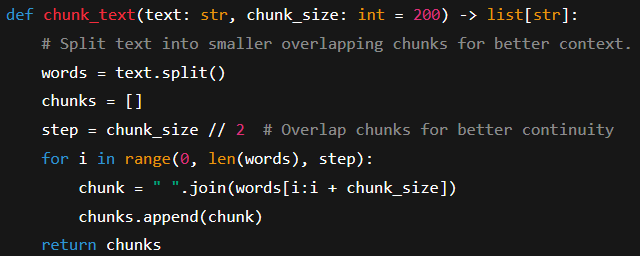
**pipeline("text2text-generation", model="google/flan-t5-base", device=-1):**

* **Loads a Hugging Face pipeline for sequence-to-sequence generation (Flan-T5).**
* **text2text-generation is used for models that map text → text (question answering, summarization, instruction following).**
* **device=-1 means CPU. For GPU use device=0 (or appropriate CUDA device id).**

**return embedder, qa\_model — return both objects so the rest of app can use them.**

**embedder, qa\_model = load\_models() — call the cached loader so models are available.**

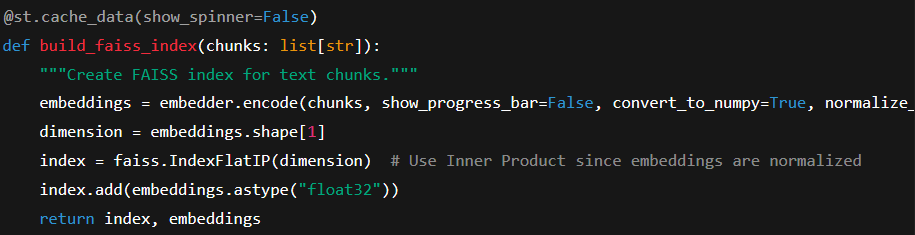
**2) Chunking the text**



* **def chunk\_text(text: str, chunk\_size: int = 200) -> list[str]:**
  + **Function signature with type hints: text is string, chunk\_size default 200 (words), returns list[str].**
  + **Type hints are optional but good for readability and static analysis.**
* **words = text.split() — simple whitespace tokenization; results in a list of words.**
* **chunks = [] — initialize list to store chunks.**
* **step = chunk\_size // 2 — sets overlap to 50% (sliding window). Overlap preserves context that might otherwise fall between chunks.**
* **for i in range(0, len(words), step): — iterate with stride step.**
* **chunk = " ".join(words[i:i + chunk\_size]) — create a chunk with up to chunk\_size words starting at i.**
* **chunks.append(chunk) — add chunk to list.**
* **return chunks — return list of overlapping text chunks.**

**Why overlap? Suppose an idea spans two consecutive chunks; overlap increases chance that at least one chunk contains enough context to answer a query.**

**3) Building FAISS index (cached)**



**@st.cache\_data(show\_spinner=False):**

* **Another Streamlit cache decorator targeted at function outputs (data). It caches the returned index and embeddings keyed by the function arguments.**
* **show\_spinner=False disables the small Streamlit spinner while the cached function runs (optional UX choice).**
* **Caching here avoids recomputing embeddings and index on every rerun if the chunks argument hasn't changed.**

**embeddings = embedder.encode(chunks, show\_progress\_bar=False, convert\_to\_numpy=True, normalize\_embeddings=True):**

* **embedder.encode() converts each chunk into a numerical vector.**
* **show\_progress\_bar=False prevents a CLI progress bar.**
* **convert\_to\_numpy=True returns a NumPy array (shape: [num\_chunks, dim]).**
* **normalize\_embeddings=True scales each vector to unit length (L2 norm = 1). This makes dot product = cosine similarity.**

**dimension = embeddings.shape[1] — number of embedding dimensions (e.g., 384 for many MiniLM models).**

**index = faiss.IndexFlatIP(dimension):**

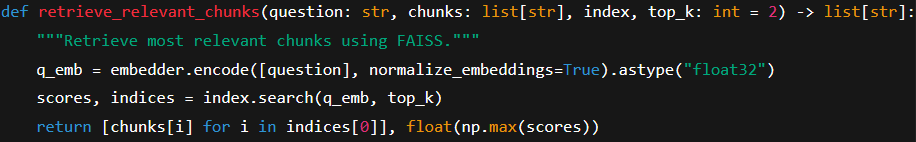
* **IndexFlatIP builds a *flat* (exact search) index using Inner Product (IP). When vectors are normalized, inner product equals cosine similarity, so this gives nearest-by-cosine.**
* **Flat index = exact brute-force search. Fast C++ code but O(N) per query; OK for small-medium N.**

**index.add(embeddings.astype("float32")):**

* **FAISS expects np.float32. astype("float32") ensures correct dtype and memory layout.**
* **Adds all embeddings to the index so subsequent index.search() can find nearest neighbors.**

**return index, embeddings — the function returns both index and original embeddings (you might keep embeddings if you want to examine them).**

**4) Retrieve relevant chunks**



**def retrieve\_relevant\_chunks(...) — function to find top-k chunks matching the question.**

**q\_emb = embedder.encode([question], normalize\_embeddings=True).astype("float32"):**

* **Encode the question into a vector; normalize\_embeddings=True so it’s on same scale (unit length).**
* **Wrap question in a list so encode returns shape [1, dim].**

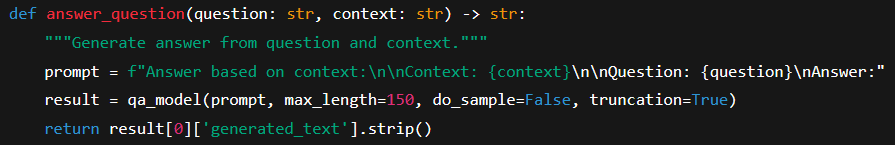
**scores, indices = index.search(q\_emb, top\_k):**

* **index.search(query\_vectors, k) returns (scores, indices) arrays:**
  + **scores shape (num\_queries, k) — similarity scores (inner product) for each returned neighbor.**
  + **indices shape (num\_queries, k) — indices into the original chunks array.**
* **For IndexFlatIP, higher scores = more similar (not distances).**

**return [chunks[i] for i in indices[0]], float(np.max(scores)):**

* **Returns the list of top chunks corresponding to indices for the single query.**
* **Also returns the maximum similarity score across the top\_k results as a float — useful to decide whether the question is related enough.**

**5) Ask the generative model for an answer**



* **prompt = f"Answer based on context:...:**
  + **Prompt engineering: the function builds a single text string that instructs the model to answer based on the provided context.**
  + **Keep prompts explicit and directive for better outputs.**
* **result = qa\_model(prompt, max\_length=150, do\_sample=False, truncation=True):**
  + **Calls the Hugging Face pipeline to generate text.**
  + **max\_length=150 — maximum tokens in generated answer. (Tokens, not characters.)**
  + **do\_sample=False — deterministic decoding (greedy / beam by default) rather than sampling; helps repeatability.**
  + **truncation=True — if the prompt is longer than model max input length, it will be truncated to the model’s tokenizer limit.**
* **return result[0]['generated\_text'].strip():**
  + **The pipeline returns a list of outputs; each output is a dict with 'generated\_text'.**
  + **.strip() removes leading/trailing whitespace.**

**Important: truncation=True can cause loss of context if context is too long. Flan-T5 has a limited input size (e.g., ~512–1024 tokens depending on model); so the retrieved context must be short enough. That's why retrieval reduces the context to top-k chunks.**

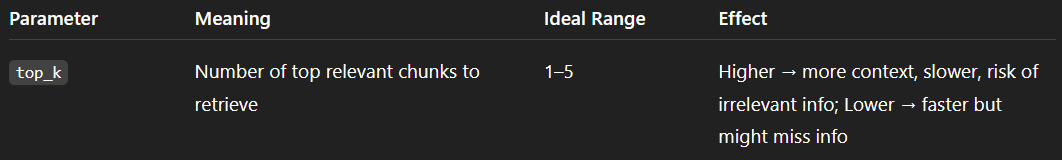
**6) Streamlit UI: layout and interaction**

* **st.title(...), st.write(...) — basic page header and explanatory text.**
* **text\_input = st.text\_area(...) — multi-line text area widget; returns string content.**
* **question = st.text\_input(...) — single-line input for the question.**
* **if st.button("🔍 Get Answer"): — when the user clicks this button, the app runs the body once. Streamlit reruns the script on every interaction; the button returns True only on click.**
* **The if not text\_input.strip() / elif not question.strip() checks ensure required input exists; .strip() removes whitespace.**
* **with st.spinner("..."): shows a spinner during processing. Streamlit shows UI while backend operations execute.**
* **chunks = chunk\_text(text\_input) — chunk the input text.**
* **index, \_ = build\_faiss\_index(chunks) — create (or reuse cached) FAISS index and embeddings for chunks.**
* **relevant\_chunks, max\_sim = retrieve\_relevant\_chunks(question, chunks, index) — find top-k chunks and get top similarity.**
* **RELEVANCE\_THRESHOLD = 0.35 — chosen cut-off below which answer is declared “unrelated”. (Why 0.35? Empirical; values between 0.2–0.5 are commonly used depending on model and embedding space.)**
* **st.subheader("Answer:") — UI section header.**
* **if max\_sim < RELEVANCE\_THRESHOLD: st.info("...") — shows a hint that question isn’t closely related to the text.**
* **Else: context = " ".join(relevant\_chunks) — concatenate retrieved chunks into a single context string for the generative model.  
   answer = answer\_question(question, context) — ask Flan-T5 to answer.  
   st.success(textwrap.fill(answer, width=80)) — display answer and wrap text to width 80.**

**UX notes: Because Streamlit reruns the whole file on each interaction, caching models and cached index building is essential to keep the app responsive.**

### **Cosine similarity vs inner product (dot product)**

* **Cosine similarity between vectors u and v: cos(u,v) = (u·v) / (||u||\*||v||).**
* **If vectors are normalized to unit length (||u|| = ||v|| = 1), then cos(u,v) = u·v (dot product / inner product).**
* **This code uses normalize\_embeddings=True so inner product equals cosine similarity. IndexFlatIP (Inner Product) returns higher values for more similar vectors.**
* **Score ranges: if normalized, similarity is within [-1, 1], with 1 = identical direction, 0 = orthogonal, -1 = opposite.**

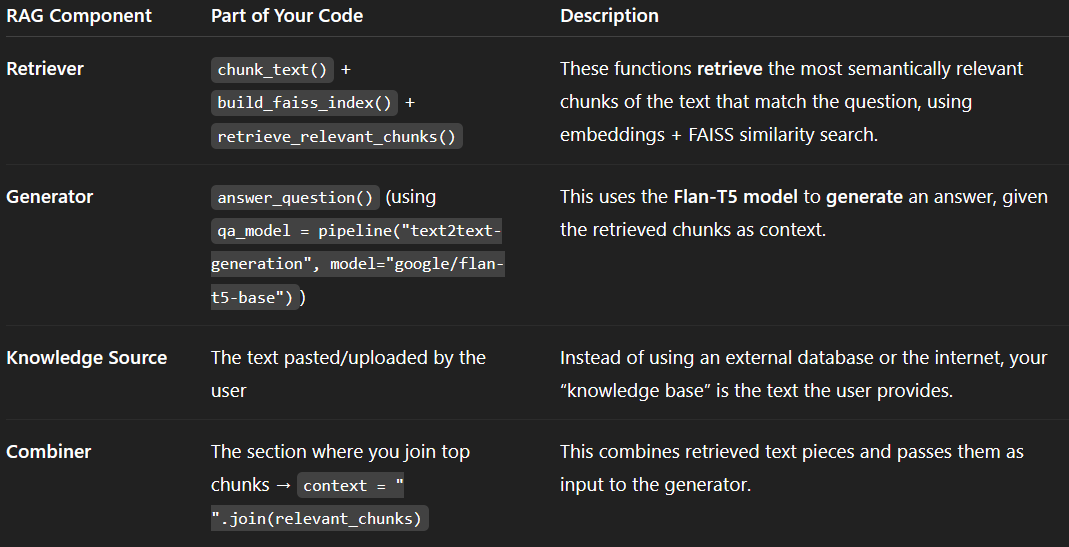


## **🔹 First — What is RAG?**

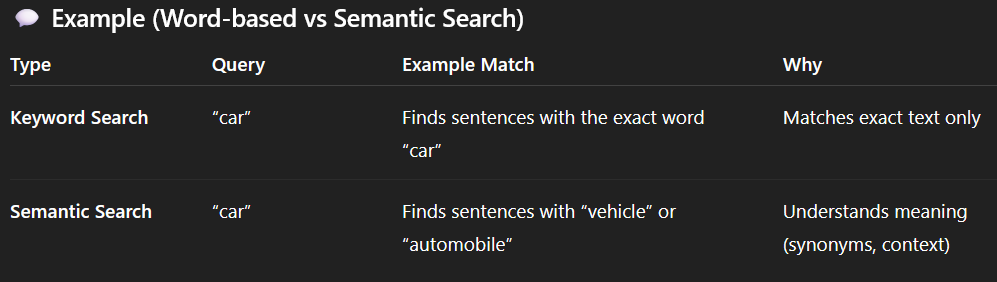
**RAG = Retrieval-Augmented Generation.**

**It’s an AI architecture that combines two stages:**

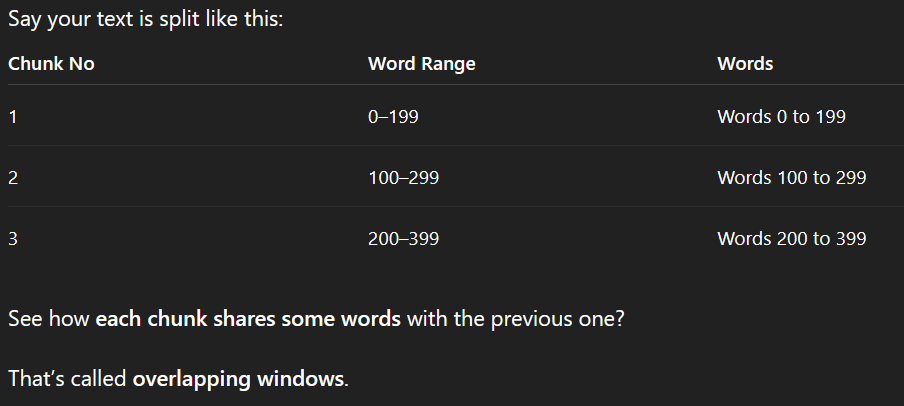
1. **Retrieval stage → find the most relevant information (from documents, text chunks, databases, etc.) related to a question.**
2. **Generation stage → use a language model to generate an answer based on that retrieved information. ( Used in current ChatGPT , Gemini …..)**



**Important small features :**



**Overlapping windows:**



## **Why We Do This**

**It’s mainly for context continuity.**

**Without overlap:**

* **If one sentence ends at the end of chunk 1 and continues in chunk 2, the model might miss part of the meaning.**

**With overlap:**

* **The shared region ensures no loss of context between chunks.**
* **FAISS can still find that chunk even if your question relates to a sentence spanning two chunks.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_⭐⭐⭐⭐⭐\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_