RAG LEAGACY SYSTEM ARCHITECHTURE

1. **Document Loading**

- The system reads files from the /documents directory.
- Supported types: **PDF, DOCX, TXT**
- Libraries used:
    - PyPDFLoader : handles structured text extraction from PDFs
    - Docx2txtLoader : extracts text from Word files
    - TextLoader : reads plain text files
- **Why these:** These loaders are reliable, lightweight, and preserve page metadata.
- **Alternatives:** Unstructured.io loaders or PDFMiner, but they introduce heavy dependencies and slower processing.

2. **Text Splitting / Chunking**

- Library: RecursiveCharacterTextSplitter
- Strategy: split documents into chunks with overlap
- Chunk size used= 800, Chunk overlap= 200
- **Why this chunking size:** common chunk sizes in a typical RAG model is from 500 to 1000 and 800 chunk size provides enough context size for the system with overloading it. And overlap is 200 because typically it 20-25% of chunk size hence 200.
- **Why chunking is needed:** LLMs cannot process large documents directly; splitting keeps context focused while preventing loss of meaning at boundaries.
- **Alternatives:** Sentence/window chunkers or token-based chunkers.
- **Why recursive splitter:** It is designed for RAG workloads and preserves **semantic coherence + flexibility**.

3. **Embedding Generation**

- Model: **Amazon Titan — titan-embed-text-v1**

- Embeddings convert text chunks into numerical vectors for similarity search.

- **Why Titan embeddings:**
    - Native to AWS ecosystem and low latency + no external API
    - Strong semantic performance for enterprise English data
    - Fully managed + scalable
- **Alternatives:** OpenAI embeddings, Hugging Face embeddings, or sentence-transformers.
- **Why Titan was better:** No need to host a model ourselves, no token limits for long-term scale, secure for corporate use.

4. Vector Database / Indexing

- Chosen system: **FAISS**
- Operation: FAISS.from_documents(chunks, embedder) = builds high-performance index locally
- **Why we use a vector database:**
    - Allows **semantic retrieval** = find the most relevant chunks based on meaning, not keyword matching.

- **Why we switched from ChromaDB to FAISS**

  Originally the design used **ChromaDB**, and the ingestion pipeline worked. However on Windows with remote embeddings (AWS Bedrock) + persistent storage, Chroma had a **known issue** where retrieval would:

- crash silently (no traceback)
- terminate during similarity_search_by_vector

FAISS advantages:

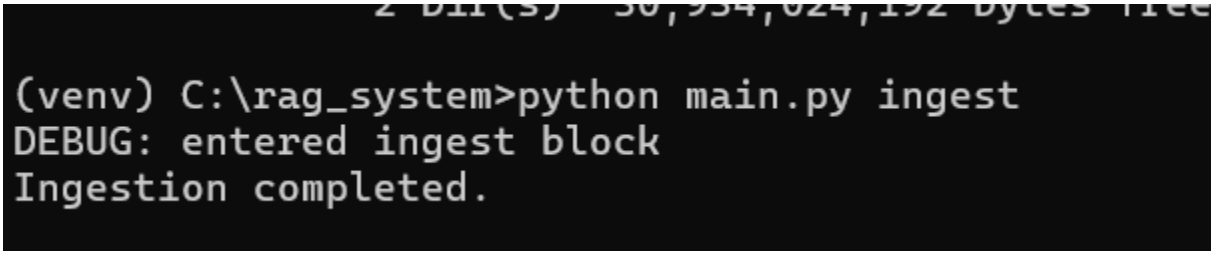
Reason	Benefit
Native C++ search engine	Very fast & stable
No multiprocessing issues on Windows	No silent crash
Integrates well with Titan embeddings	Clean pipeline
100% local	No external dependencies for retrieval

5. **Query (Retrieval Phase)**

- When a question is asked:
    1. The query is embedded using Titan (same embedding model as ingestion).
    2. FAISS performs nearest-neighbor search to return top-K relevant chunks.
    3. The retrieved text becomes **context** for the LLM.
- **Why retrieval improves accuracy:**
  The LLM is not answering from memory**; it** is answering based only on retrieved documents, reducing hallucination.

6. **Final Answer Generation**

- Model: **Amazon Titan — titan-text-lite**
- Prompt format:

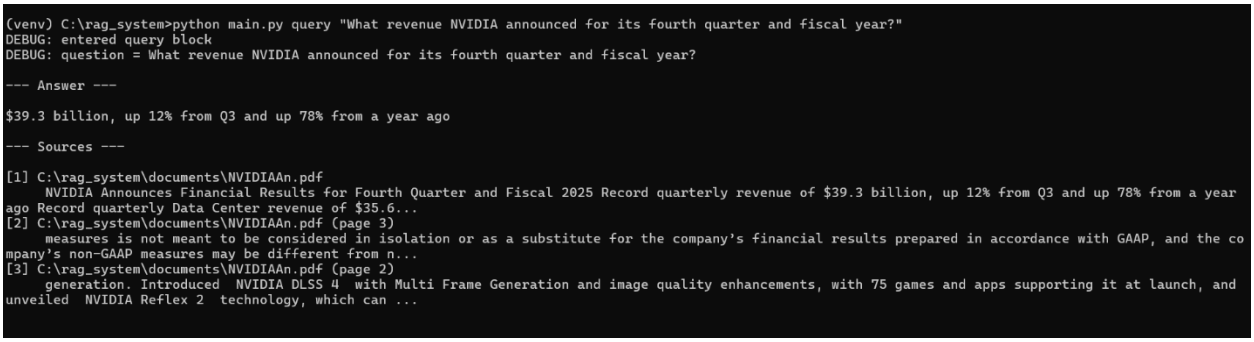


- **Why Titan LLM:**
    - Optimized for grounded, factual responses
    - Easy to control temperature for non-creative use cases
    - Integrated with Bedrock security policies
- The model returns:
    - Final answer text
    - We also extract metrics from headers (token counts + latency)

7. **Output Layer**

Printed to CLI with:

- Final Answer
- Source Documents (file names + snippets)
- Metrics (response time + token usage + latency)



Summary of Technology Choices

Layer	Your Choice	Why it fits
Embeddings	Titan	Enterprise-secured + high quality + no external API
LLM	Titan	Controlled, factual responses
Vector DB	FAISS	Fast + stable on Windows + scalable
Ingestion	LangChain loaders + splitter	Reliable text extraction + chunking
Interface	CLI	Fastest for MVP + demo control

Optional Alternatives

Component	Alternatives	Why not chosen
Titan Embeddings	OpenAI, Hugging Face	External billing + latency + VPN issues
Titan LLM	Llama, Claude, GPT	Overkill / licensing / confidentiality
Vector DB	Pinecone, Weaviate, Chroma	FAISS = fastest + simplest for demo
File Processing	Unstructured.io	Heavy dependencies, slower ingestion
UI	Streamlit	Not needed for first demo; CLI guaranteed stability