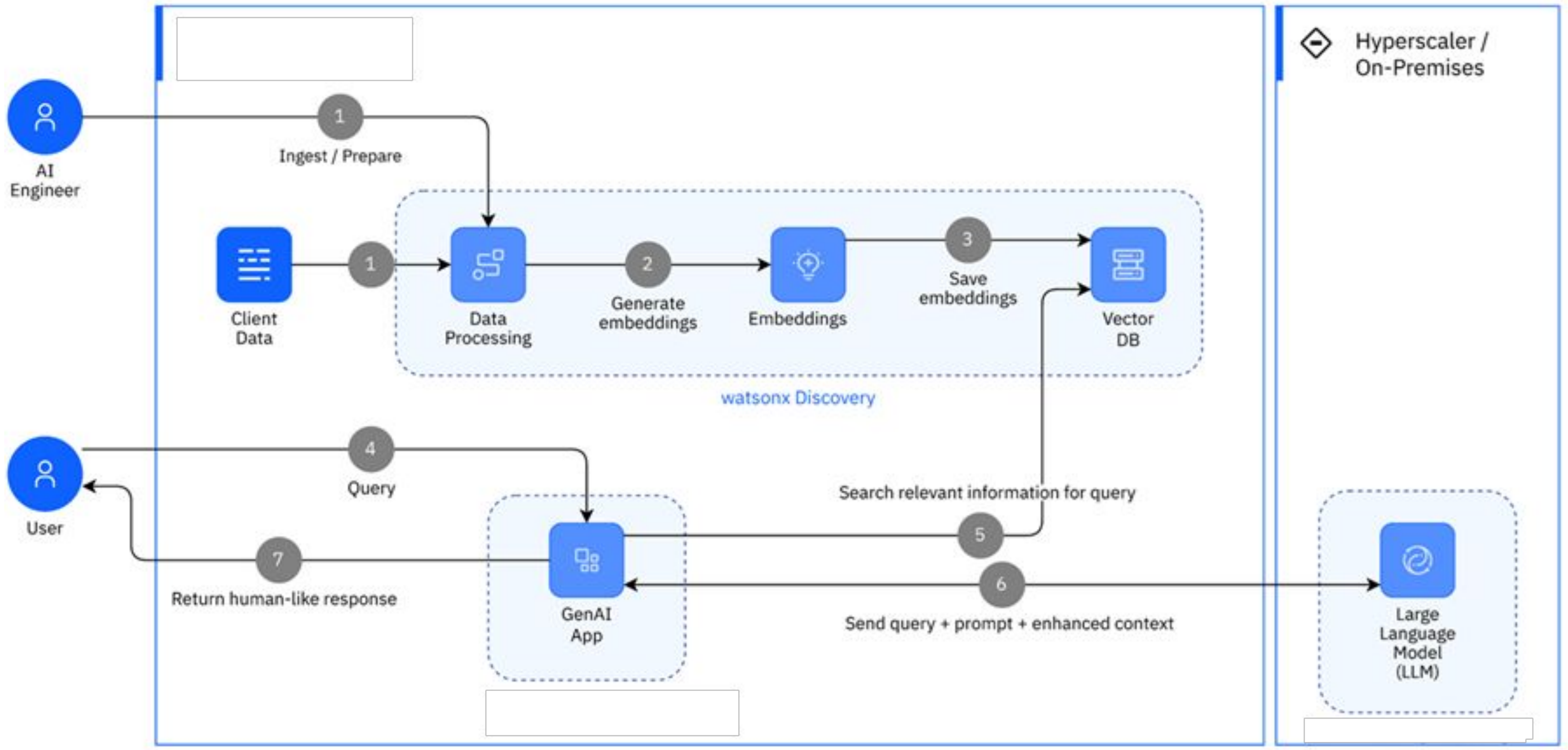
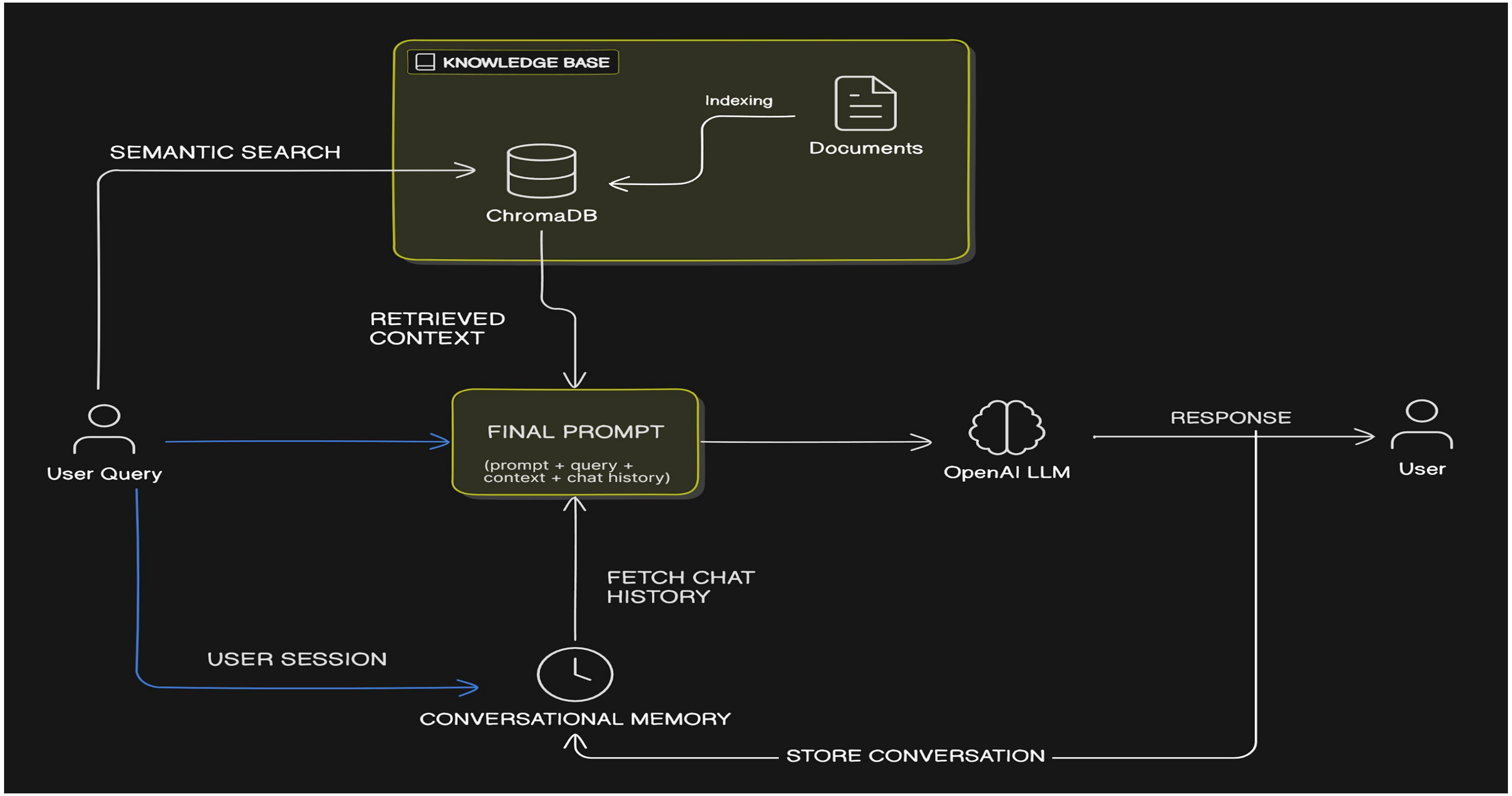
RAG – Retrieval Augmented Generation

By

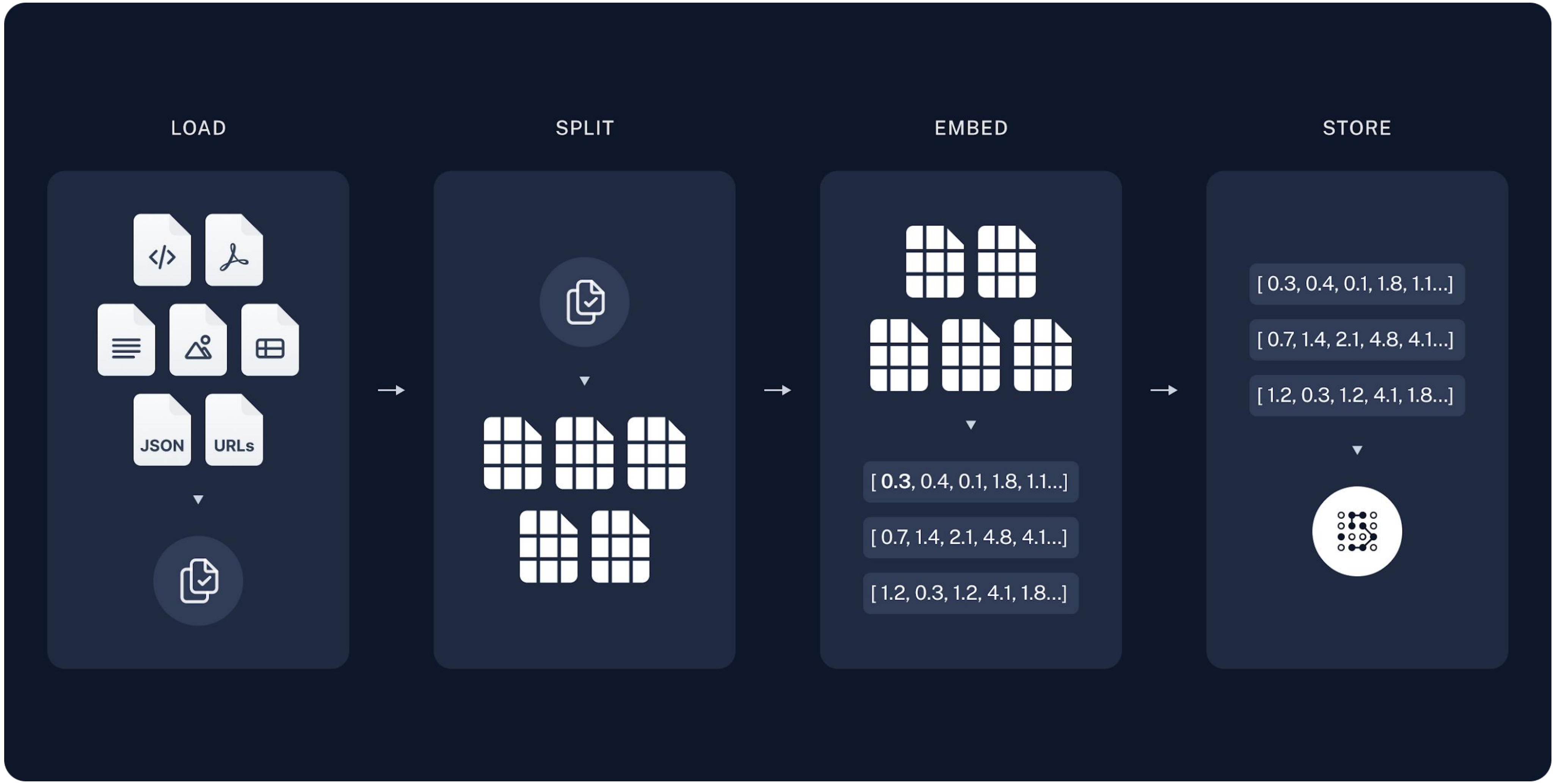
RAG – Retrieval Augmented Generation


Hyperscaler / On-Premises
AI Engineer
1
Ingest / Prepare
Client Data
1
Data Processing
2
Generate embeddings
Embeddings
3
Save embeddings
Vector DB
watsonx Discovery
User
4
Query
Search relevant information for query
5
7
Return human-like response
GenAI App
6
Send query + prompt + enhanced context
Large Language Model (LLM)

RAG – Retrieval Augmented Generation



RAG – Retrieval Augmented Generation



RAG – Retrieval Augmented Generation

Retrieval augmented generation (RAG) is an architectural pattern that enables foundation models to produce factually correct outputs for specialized or proprietary topics that were not part of the model's training data.

The **RAG (Retrieval-Augmented Generation) pattern** consists of two main stages:

- **Build Time (Data Embedding)**
  - Client data (manuals, documentation, tickets) is preprocessed through transformations (format conversion, restructuring) and enrichment (abbreviation expansion, metadata addition).
  - An **embedding model** converts the processed text into vector representations (chunks).
  - These embeddings are stored in a **vector database** (e.g., FAISS, Milvus, Chroma).

- **Runtime (User Query & Retrieval)**
  - A user query is converted into an embedding and matched against stored vectors using semantic search to retrieve the most relevant data.
  - The retrieved information is then used to enhance an AI model’s response.

- **User Query:** End-users enter a query in a GenAI-enabled application.
- **Search & Retrieval:** The application searches the vector database for the top **K** relevant passages.
- **LLM Processing:** Retrieved passages and a curated prompt are sent to the LLM.
- **Response Generation:** The LLM generates a human-like response based on the query and context.
- **Output:** The response is presented to the user.

Chunking Strategies in RAG

Fixed-Size Chunking

Aspect	Details
How it works	Splits text into chunks based on a fixed number of tokens/characters. Often includes overlap (e.g., 50 tokens).
Example	Text → split every 300 tokens with 50-token overlap. Chunk 1: tokens 1–300 Chunk 2: tokens 250–550 Chunk 3: tokens 500–800
Pros	Simple, fast, predictable.
Cons	Cuts meaning halfway; may split tables/code.
Best For	PDFs, long manuals, code docs, structured text.

Chunking Strategies in RAG

Semantic Chunking (Adaptive)

Aspect	Details
How it works	**Uses embeddings to detect where the text changes topic.** ▯ **Embedding similarity drop** → if adjacent sentences have low semantic similarity, create a new chunk. ▯ **Percentile threshold** (LangChain) → only split at strong semantic breaks. ▯ **Clustering-based chunking** → group semantically similar sentences the way LDA or K-Means would.
Example	Sentence similarity: Sim(S1,S2)=0.95 → same chunk Sim(S2,S3)=0.93 → same chunk Sim(S3,S4)=0.41 → **new chunk**
Pros	Meaning-preserving, very relevant chunks.
Cons	Slow, costly, may produce giant chunks if all sentences are similar.
Best For	Blogs, Q&A, knowledge bases, conversational docs.

Chunking Strategies in RAG

Structure-Aware Chunking

Aspect	Details
How it works	Use the document structure to split meaningfully.
Example	**a) Heading-based chunking -** Split at: H1, H2, H3 Sections / chapters **b) Markdown / HTML-aware chunking -** Split at: <p>, <ul>, <h2>, <code>, etc. **c) PDF positional chunking -** Split by: Layout Left/right columns Paragraphs Table detection **d) Code-aware chunking -** Split by: Classes Functions Docstrings
Pros	Very high contextual accuracy,Maintains document structure,Works best for technical docs
Cons	Requires document layout parsing
Best For	Best for: **PDFs, code, manuals, books, legal docs**

Chunking Strategies in RAG

Query-Aware Chunking (Dynamic RAG) Chunking

Aspect	Details
How it works	Chunking happens based on **user query intent**. **Techniques:** Re-rank chunk boundaries based on query similarity Dynamic window expansion around relevant sections
Example	For code, fetch the function + relevant calls For legal text, fetch the clause + definitions

Sentence-Level Embedding + Grouping

Aspect	Details
How it works	Each sentence has an embedding, then you merge **topically similar adjacent sentences**.
Example	These methods include: Agglomerative clustering DBSCAN on sentence embeddings Topic modeling (LDA-based)

Chunking Strategies in RAG

Graph-Based Chunking

Aspect	Details
How it works	Build a **knowledge graph** instead of linear chunks. Extract entities + relations Create graph nodes (chunks) Link them semantically Better retrieval than simple chunks.
Example	Nodes: {CPU}, {GPU}, {Tensor Core} Edges: “GPU → contains → Tensor Core” Query: “What is Tensor Core?” Chunks: traverse GPU → Tensor Core.

Recommended Best Practice

Best 2-Strategy Combo:

- Semantic chunking + fixed-size hard limit, This avoids over-large chunks when semantic similarity is high.

Best for production:

- Structure-aware + semantic within sections

Best for chat-like data:

- Overlapping sliding windows

Best for enterprise knowledge bases:

- GraphRAG (graph-based chunking)

Document Extraction Tools

Feature / Tool	PyMuPDF	Docling (NVIDIA)	Azure Document Intelligence
Type	Local parser	AI-powered layout + OCR	Cloud OCR + Document AI
Extraction	Raw text + coords	Structured MD/HTML/JSON	Structured JSON
Table extraction	Manual	✔ Very strong	✔ Very strong
OCR	❌ No	✔ Yes	✔ Yes
Layout detection	❌ Basic	✔ Strong AI	✔ Strong AI
Hierarchy reconstruction	❌ No	✔ Yes	✔ Yes
Ideal for	Clean PDFs	RAG / Scientific PDFs	Forms, receipts, enterprise docs
Speed	⚡ Fast	Medium	Medium
Cost	Free	Free	Paid (usage-based)
Customizable	High	Very high	Medium

Thank you