

CSE/DSC 234

Data-Centric AI and AI Engineering

Arun Kumar

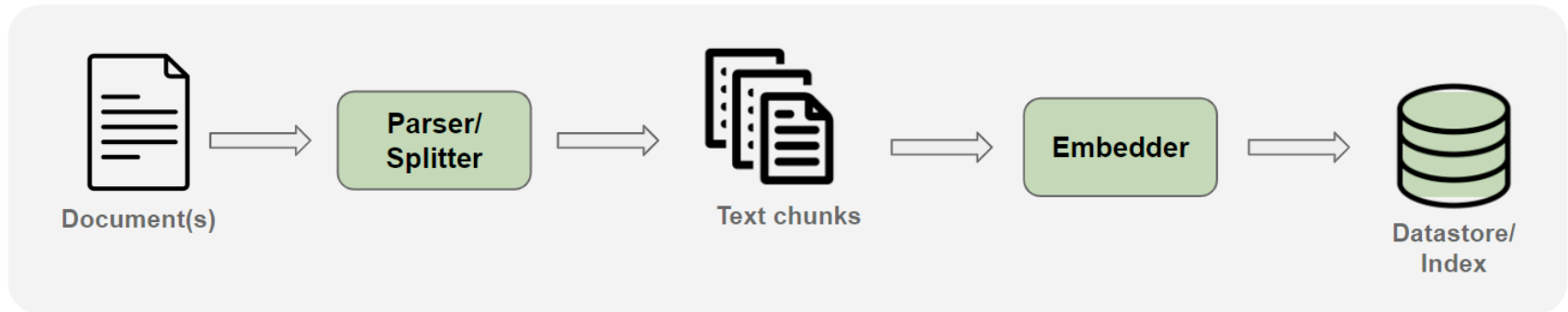
Topic 2: Retrieval and Context Engineering

Outline

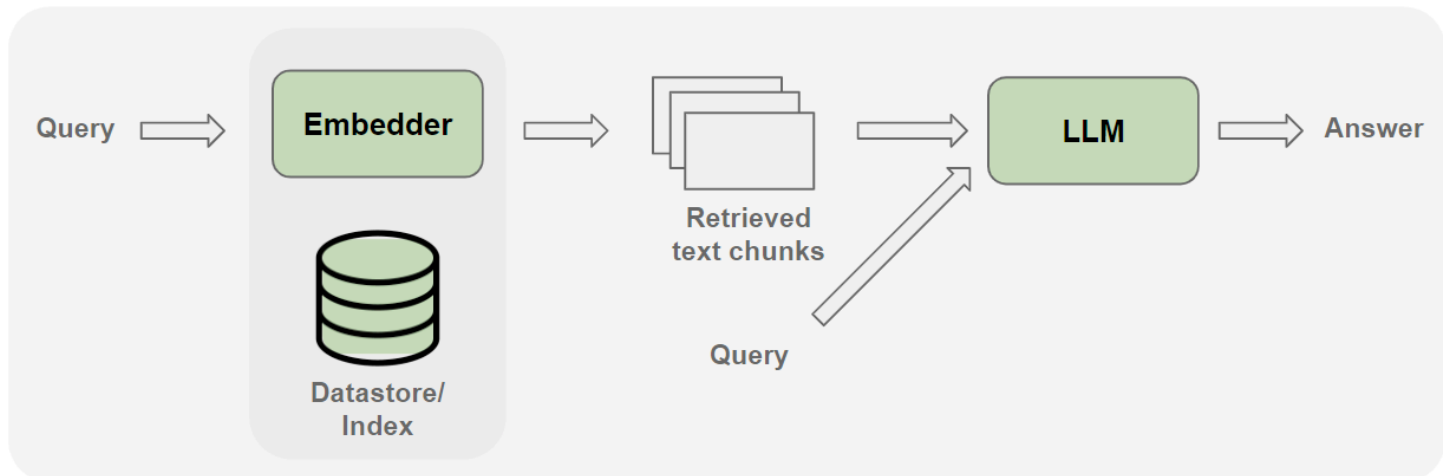
- ❖ Introduction
- ❖ Prompt Engineering
- ❖ Retrieval and Context Engineering
 - ❖ Retrieval vs. Long Context
-  ❖ VectorRAG
- ❖ GraphRAG
- ❖ Other Forms of RAG

The Canonical VectorRAG

Indexing

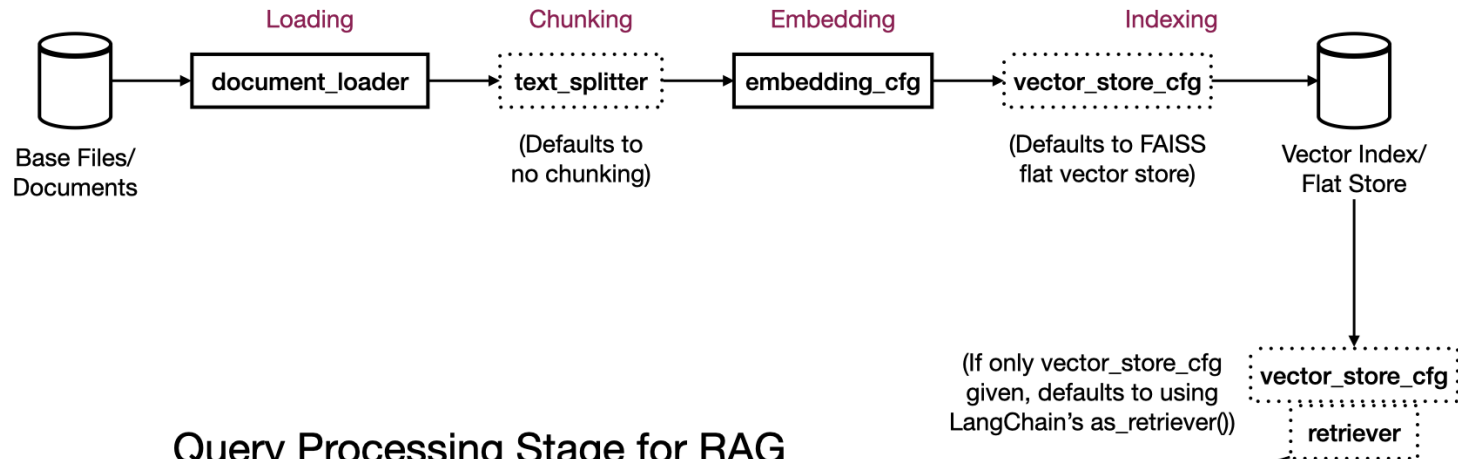


Generating

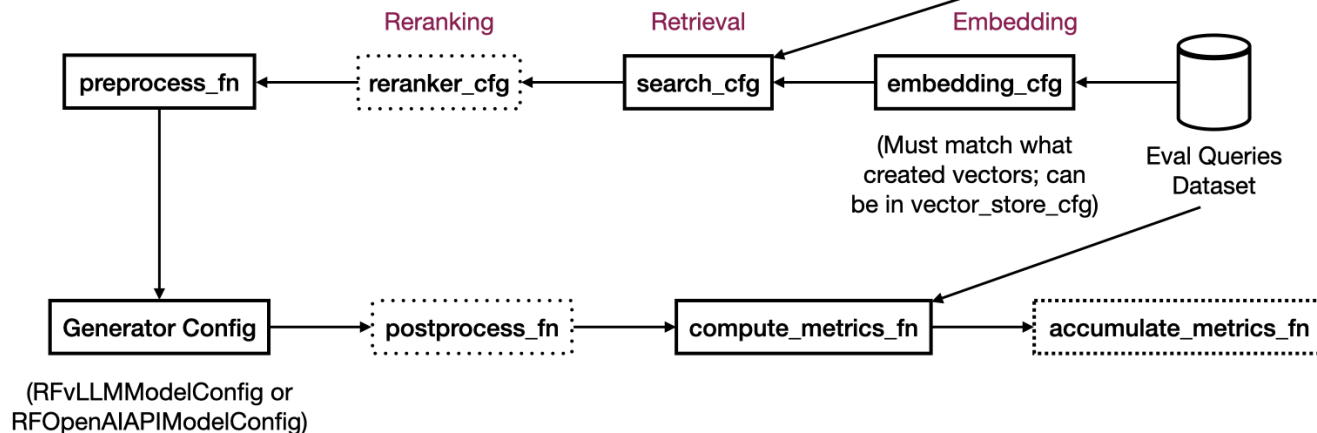


VectorRAG Pipeline in RapidFire AI

(Optional) Document Preprocessing Stage for RAG



Query Processing Stage for RAG



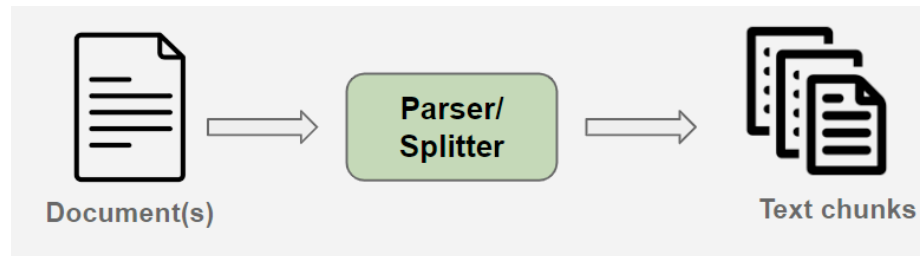
Parts of Canonical VectorRAG

1. **Parsing and Splitting/Chunking:** Turn raw data/docs into retrieval units
2. **Embedding:** Encode each chunk as a dense vector
3. **Indexing:** Store vectors for fast (approximate) nearest-neighbor (NN/ANN) search
4. **Retrieval:** Fetch top-K candidates from one or more sources; fuse into a single ranked list
5. **Reranking:** Rescore top-K to reorder chunks by relevance to query
6. **Context Stitching:** Assemble the final prompt—system instructions plus ordered, deduped, formatted, budgeted context
7. **Generation:** LLM produces a grounded answer with citations or abstains

More Advanced VectorRAG

Architecture	Description	Control	Flexibility	Latency	Example Use Case
2-Step RAG	Retrieval always happens before generation. Simple and predictable	✓ High	✗ Low	⚡ Fast	FAQs, documentation bots
Agentic RAG	An LLM-powered agent decides <i>when</i> and <i>how</i> to retrieve during reasoning	✗ Low	✓ High	🕒 Variable	Research assistants with access to multiple tools
Hybrid	Combines characteristics of both approaches with validation steps	⚖ Medium	⚖ Medium	🕒 Variable	Domain-specific Q&A with quality validation

1. Parsing and Splitting/Chunking



Q: How to divvy up the base data/docs for effective retrieval?

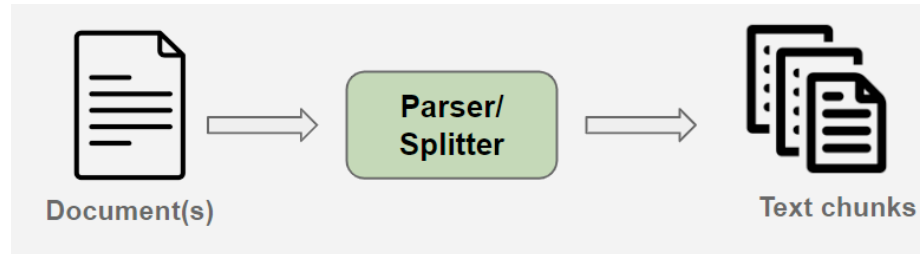
❖ Options for Parsing:

- ❖ Deterministic parsers to extract exact components
- ❖ ML/LLM-infused parsers to include OCR on multimodal docs

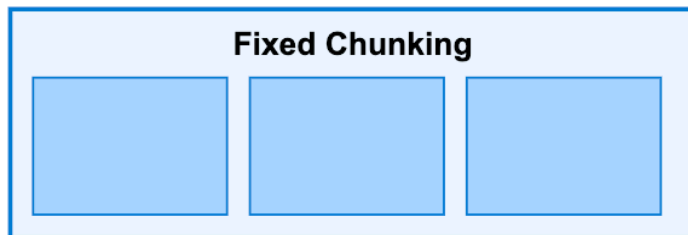
❖ Options for Splitting/Chunking:

- ❖ Fixed chunk size; optional overlap via sliding window
- ❖ Sentence/paragraph-aware
- ❖ Semantic: Metadata-based, topic-based, LLM-based, etc.
- ❖ Hierarchical, Hybrid, Multimodal, etc.

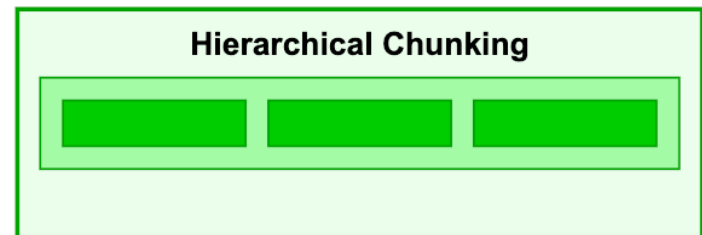
1. Parsing and Splitting/Chunking



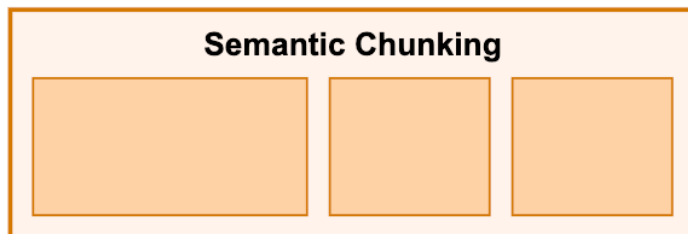
Q: How to divvy up the base data/docs for effective retrieval?



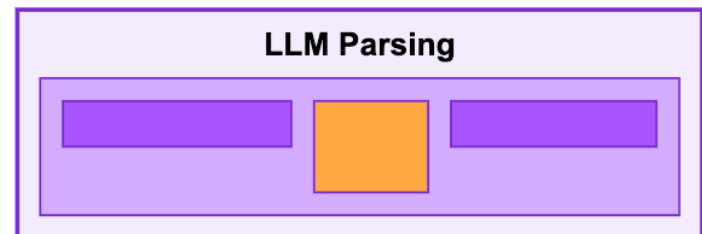
Fixed-size chunks with optional overlap



Parent and child chunks

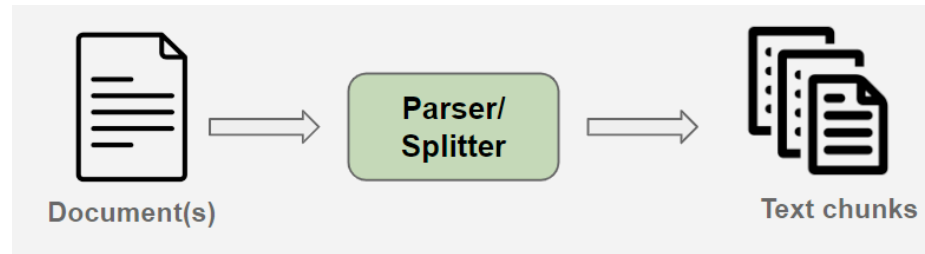


Chunks based on semantic similarity



Custom parsing with LLM instructions

1. Parsing and Splitting/Chunking



Q: How to divvy up the base data/docs for effective retrieval?

- ❖ **Basic Tradeoff** on chunking scheme + size:
 - ❖ Too small -> fragments answers, lower quality
 - ❖ Too large -> dilutes embeddings, wastes context, higher latency
- ❖ **Some best practices:**
 - ❖ Try to align with in-built structure in docs, e.g., subsections
 - ❖ Use a hierarchy of chunks if possible
- ❖ *No universal optimal chunking/size. Tune on your own eval set!*

1. Parsing and Splitting/Chunking

- ❖ RAG/agent engineering frameworks such as LangChain and LlamaIndex have many off-the-shelf parsers and chunkers/splitters

Class UnstructuredRTFLoader
Load RTF files using Unstructured.
Class UnstructuredPDFLoader
Load PDF files using Unstructured.
Class BasePDFLoader
Base Loader class for PDF files.
Class OnlinePDFLoader
Load online PDF.
Class PyPDFLoader
Load and parse a PDF file using 'pypdf' library.
Class PyPDFium2Loader
Load and parse a PDF file using the pypdfium2 library.

Class HTMLSectionSplitter
Splitting HTML files based on specified tag and font sizes.
Class HTMLSemanticPreservingSplitter
Split HTML content preserving semantic structure.
Class SentenceTransformersTokenTextSplitter
Splitting text to tokens using sentence model tokenizer.
Class KonlpyTextSplitter
Splitting text using Konlpy package.
Class PythonCodeTextSplitter
Attempts to split the text along Python syntax.
Class MarkdownTextSplitter
Attempts to split the text along Markdown-formatted headings.

<https://reference.langchain.com/python/langchain-community/document-loaders>

<https://reference.langchain.com/python/langchain-text-splitters>

Parsing and Chunking for Project 1

- ❖ **For Project 1:** Peruse the docs of LangChain and any additional third party loaders and/or splitters you use
- ❖ **Example:** Unstructured.io for table structure extraction

	Year Ended December 31,	
	2021	2020
As Previously Reported		
Revenue:		
Domains:	\$ 1,809.9	\$ 1,515.1
Hosting and presence	1,283.4	1,200.6
Business applications	722.4	601.0
Total Revenue	\$ 3,815.7	\$ 3,316.7
As Revised		
Revenue:		
Applications and commerce	\$ 1,128.3	\$ 926.1
Core platform	2,687.4	2,390.6
Total Revenue	\$ 3,815.7	\$ 3,316.7

ChatGPT 4.0

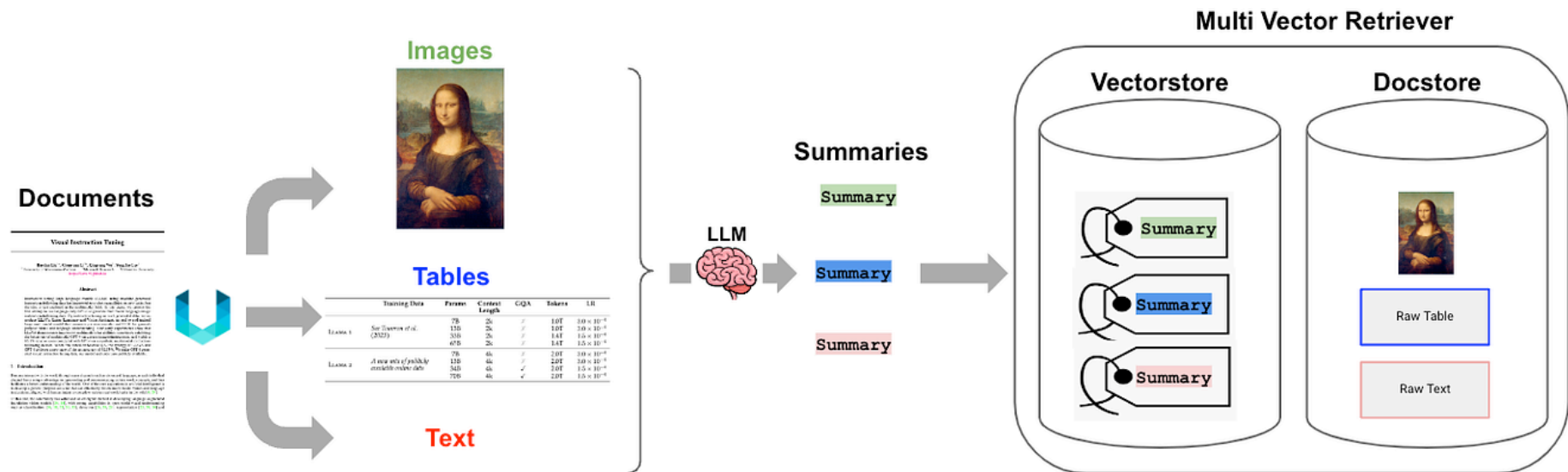
Table

```
{
  "Table": "
<table>
<tr>
<td colspan='2'> Year Ended December 31,</td>
<tr>
<td> 2021 </td>
<td> 2020 </td>
<tr>
<td> As Previously Reported </td>
<td> </td>
<tr>
<td> Revenue: </td>
<td> </td>
<tr>
<td> Domains: </td>
<td> $ 1,809.9 </td>
<td> $ 1,515.1 </td>
<tr>
<td> Hosting and presence </td>
<td> 1,283.4 </td>
<td> 1,200.6 </td>
<tr>
<td> Business applications </td>
<td> 722.4 </td>
<td> 601.0 </td>
<tr>
<td> Total revenue </td>
<td> $ 3,815.7 </td>
<td> $ 3,316.7 </td>
<tr>
<td> As Revised </td>
<td> </td>
<tr>
<td> Revenue: </td>
<td> </td>
<tr>
<td> Applications and commerce </td>
<td> $ 1,128.3 </td>
<td> $ 926.1 </td>
<tr>
<td> Core platform </td>
<td> 2,687.4 </td>
<td> 2,390.6 </td>
<tr>
<td> Total revenue </td>
<td> $ 3,815.7 </td>
<td> $ 3,316.7 </td>
</table>
"
```

Based on the table above, what is the revenue for 2021?

1. Complexity of Multimodal Data

- ❖ Multimodal data, e.g., PDFs or DOCs with text, tables, images, etc. present additional complexity for parsing + splitting/chunking
- ❖ Typical workflow decomposes them into text, tables, and images to handle each modality separately

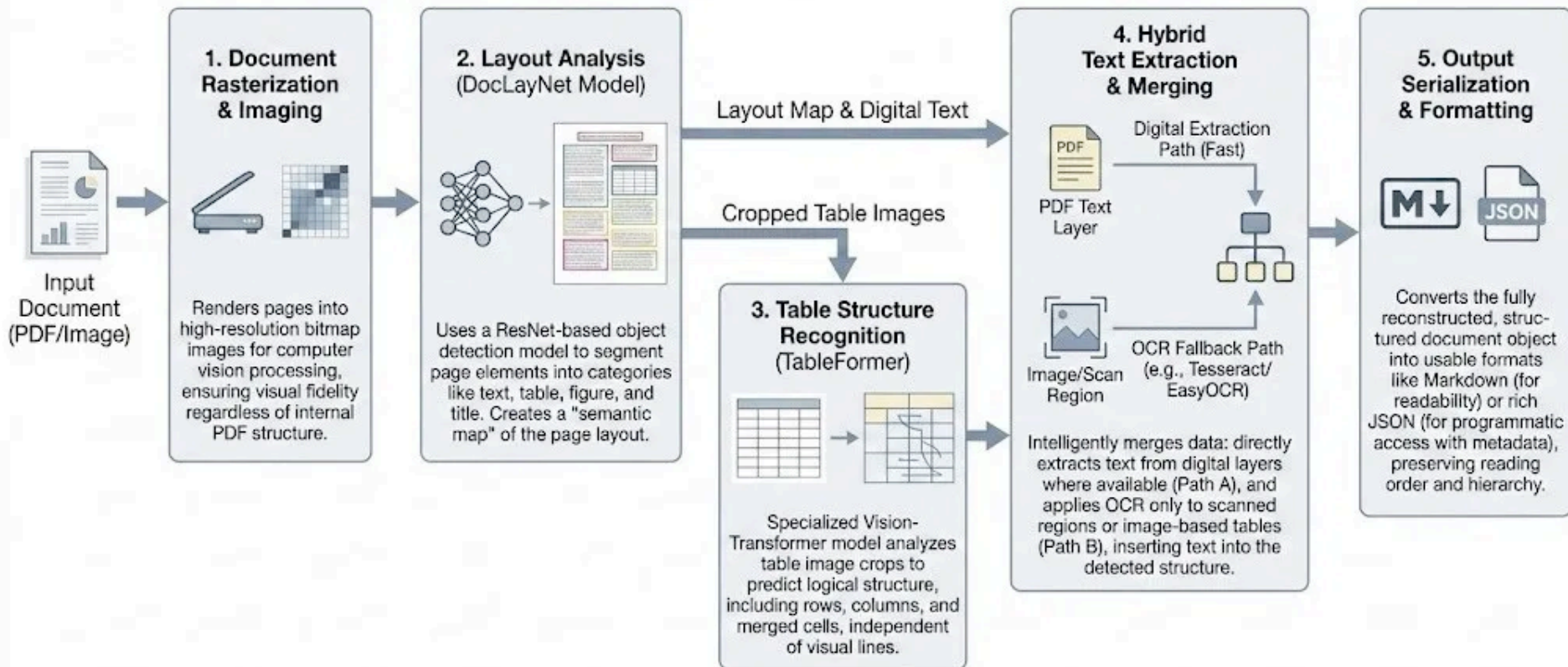


1. Complexity of Multimodal Data

- ❖ Multiple tradeoffs inherent in such complex extraction:
 - ❖ **Layout** itself might be information: Multi-column flow, headers/ footers, footnotes, sidebars. Naive top-to-bottom extraction scrambles reading order.
 - ❖ **Tables:** Cell structure carries meaning; flattening to plaintext loses rows/columns; chunking mid-table fragments the relationship
 - ❖ **Figures and Images:** need OCR (scanned), captions, and/or VLM descriptions to be retrievable
 - ❖ **Equations and Code:** LaTeX/MathML and indentation are semantically load-bearing; easily mangled by generic parsers
 - ❖ **Cross-references:** “see Figure 3”, “Table 2 above”, etc., break when the referent lands in a different chunk

1. Complexity of Multimodal Data

- ❖ PDF parsers such as unstructured.io have complex AI-infused workflows to extract the unimodal components

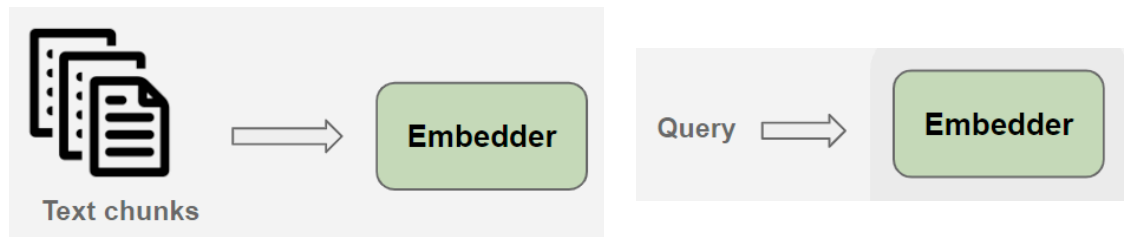


1. Complexity of Multimodal Data

❖ Some best practices:

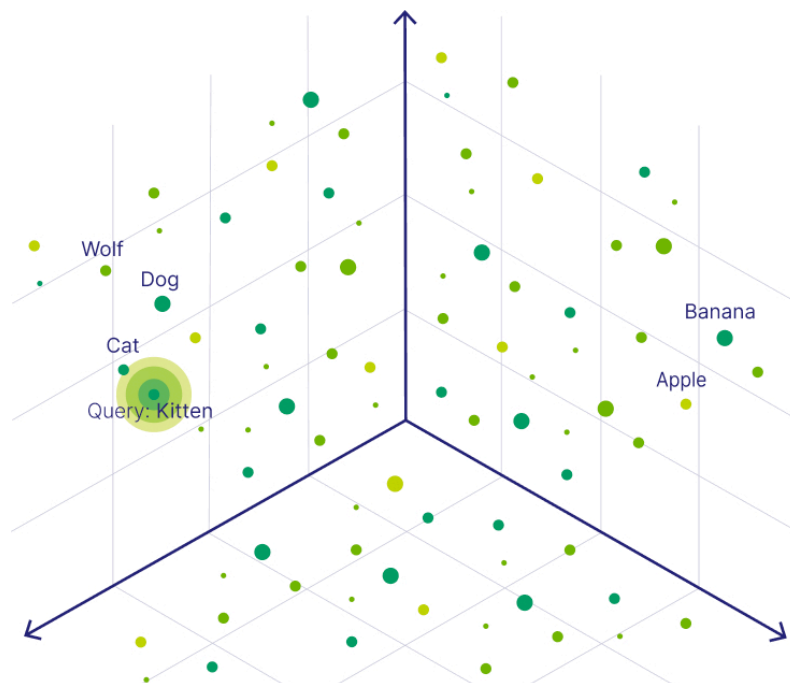
- ❖ Rolling out your own could be a rabbit hole!
- ❖ Off-the-shelf tooling: Unstructured.io, LlamaParse, Docling, Azure Document Intelligence, Marker; varying fidelity vs. cost vs. latency
- ❖ Typical pattern in practice: Parse to a semi-structured intermediate (markdown/JSON + image refs + table objects), then chunk semantically by section, not by token count alone

2. Embedding



Q: How to map chunks and queries to useful latent vector space?

- ❖ “Useful” means geometric proximity ~ semantic proximity



2. Embedding: Design Decisions

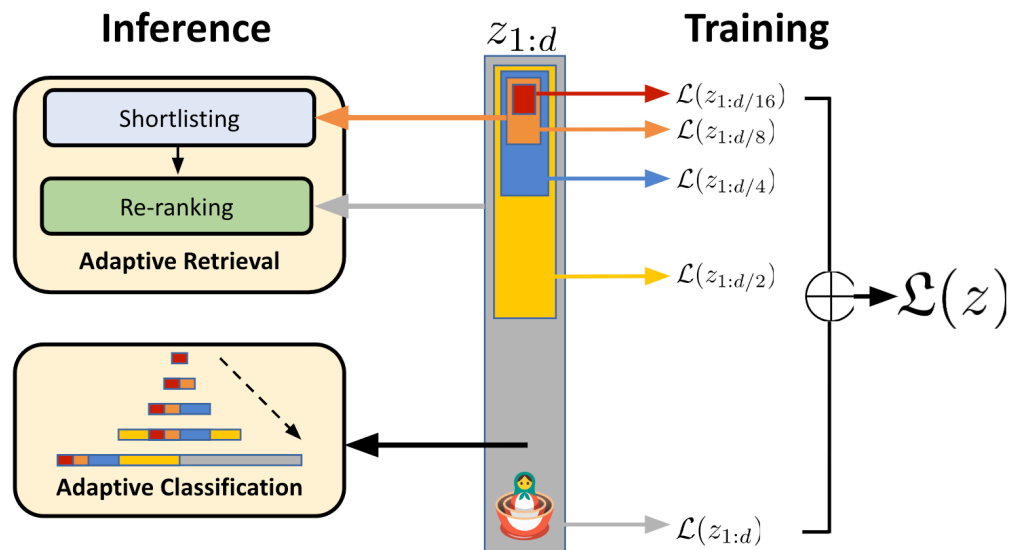
- ❖ **Model class:** *Closed* (e.g., OpenAI, Cohere, Voyage) vs. *open* (e.g., BGE, E5, Jina, Nomic)
- ❖ **Dimensionality:** Typically 384 to 3072; *Matryoshka* models allow nested truncation
- ❖ **Domain:** General-purpose vs. domain-tuned (legal, medical, code, multilingual)
- ❖ **What to embed:** chunk only vs. chunk + metadata vs. parent-aware
- ❖ **Hosting:** API vs. self-hosted GPU vs. CPU edge

2. Embedding: Tradeoffs

- ❖ **Knowledge Level:** General models offer broad coverage vs. domain-tuned models are narrower but more precise and likely more familiar with domain-specific lingo
- ❖ **DevOps Burden:** Closed APIs (zero infra. vs. rate limits and lock-in) vs. self-hosted (control vs. complexity)
- ❖ **Versioning:** Embedding decision is a long-ish commitment; model swap -> full re-index required!
- ❖ **Quality vs. Dimensionality:** Storage, memory, latency, and thus cost all scale linearly
- ❖ *No universal optimal embedding scheme/size. Tune on your own eval set!*

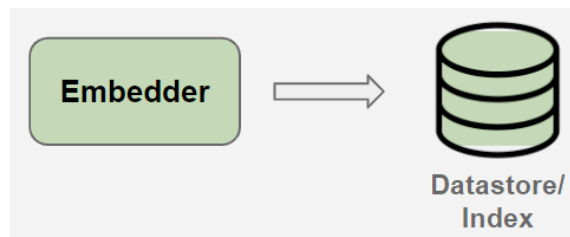
2. Embedding: Matryoshka

- ❖ Model trained with multiple usable *nested* embeddings, i.e., the first 64, 128, etc. dimensions are each *independently* meaningful (like the Russian dolls)

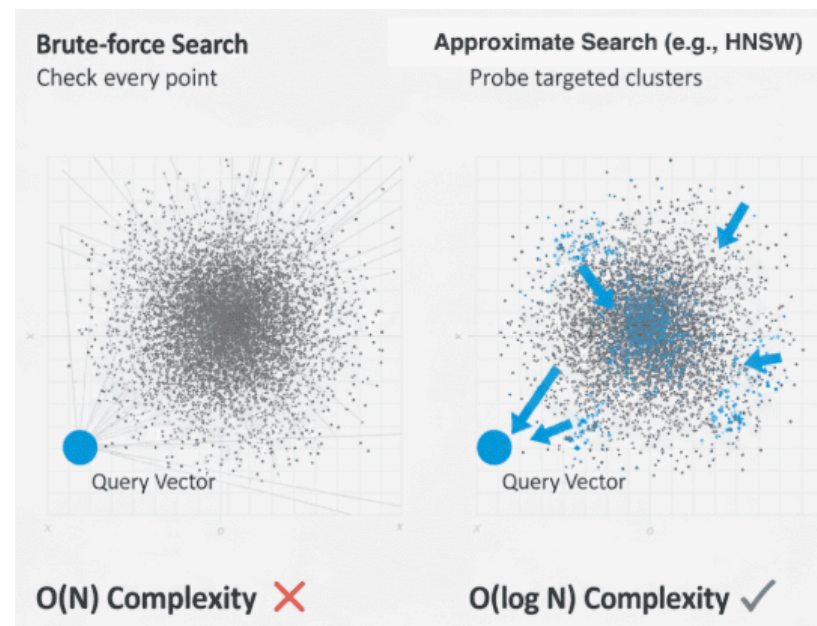
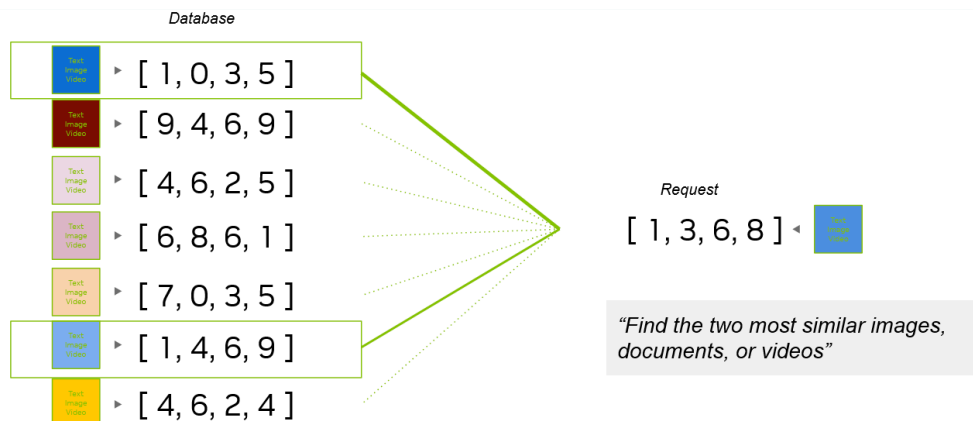


- ❖ **Adaptive cost vs. recall per query**, not per corpus, e.g., store full 3072-dim, use only 256 for retrieval, full dim for reranking
- ❖ 2-6x faster ANN search with minimal recall loss; smooth quality-cost curve instead of step function
- ❖ Rising popularity in practice: OpenAI text-embedding-3-*, Nomic Embed, Snowflake Arctic, etc.

3. Indexing

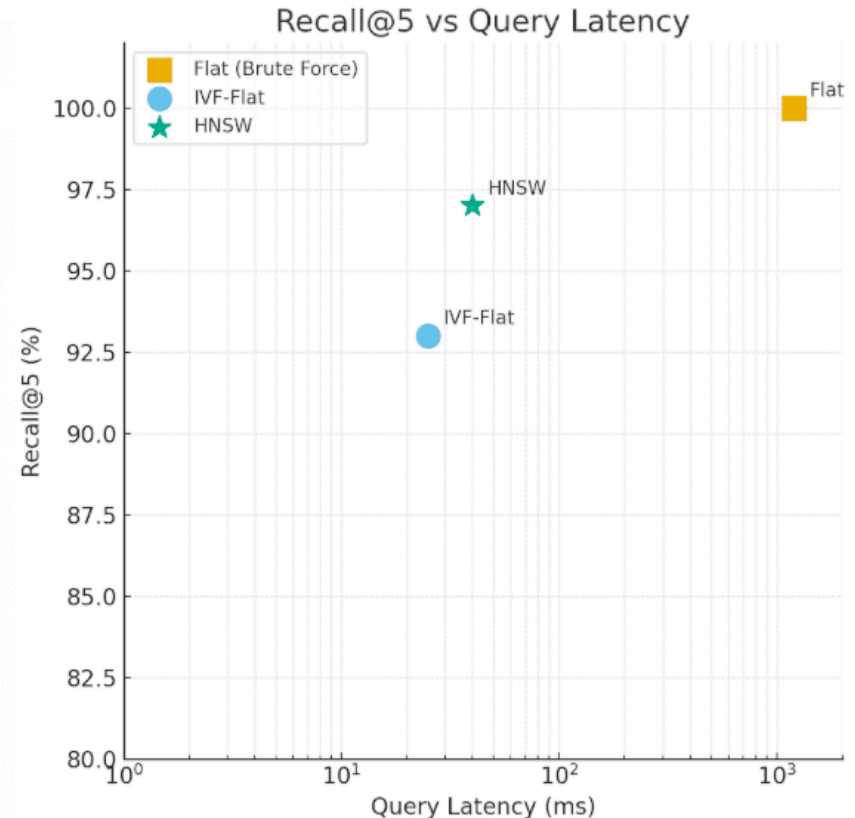
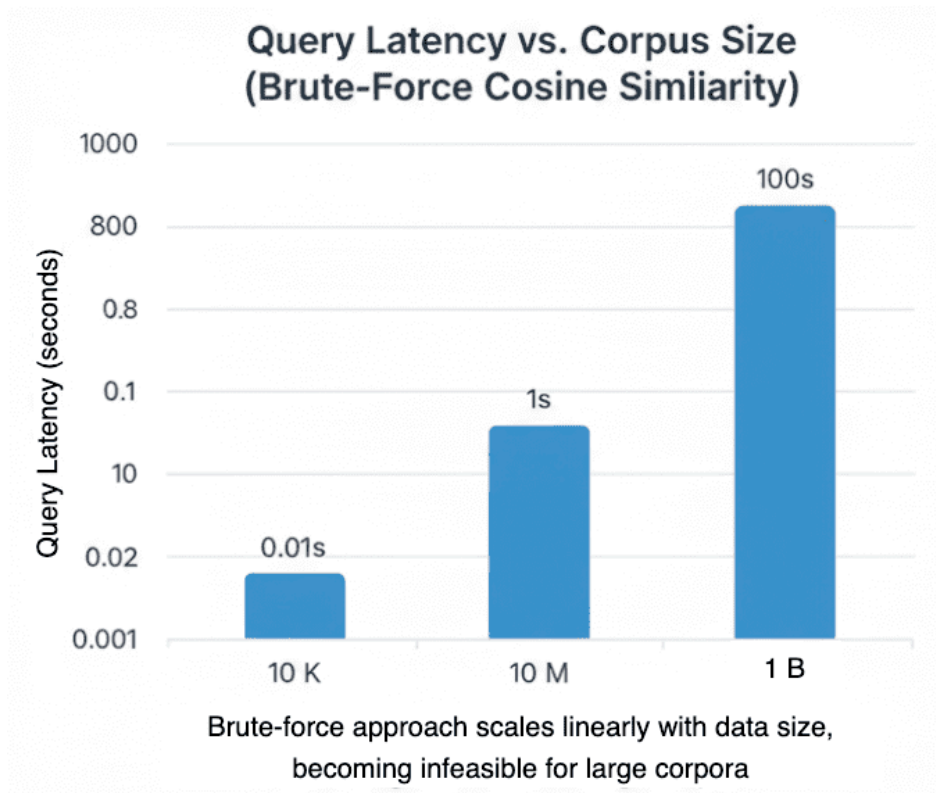


Q: How to store embeddings to support fast retrieval with (A)NN?



3. Indexing: When ANN?

- ❖ Small scale: Brute force exact matrix multiply on GPU suffices
- ❖ Large scale: ANN becomes only feasible option in production

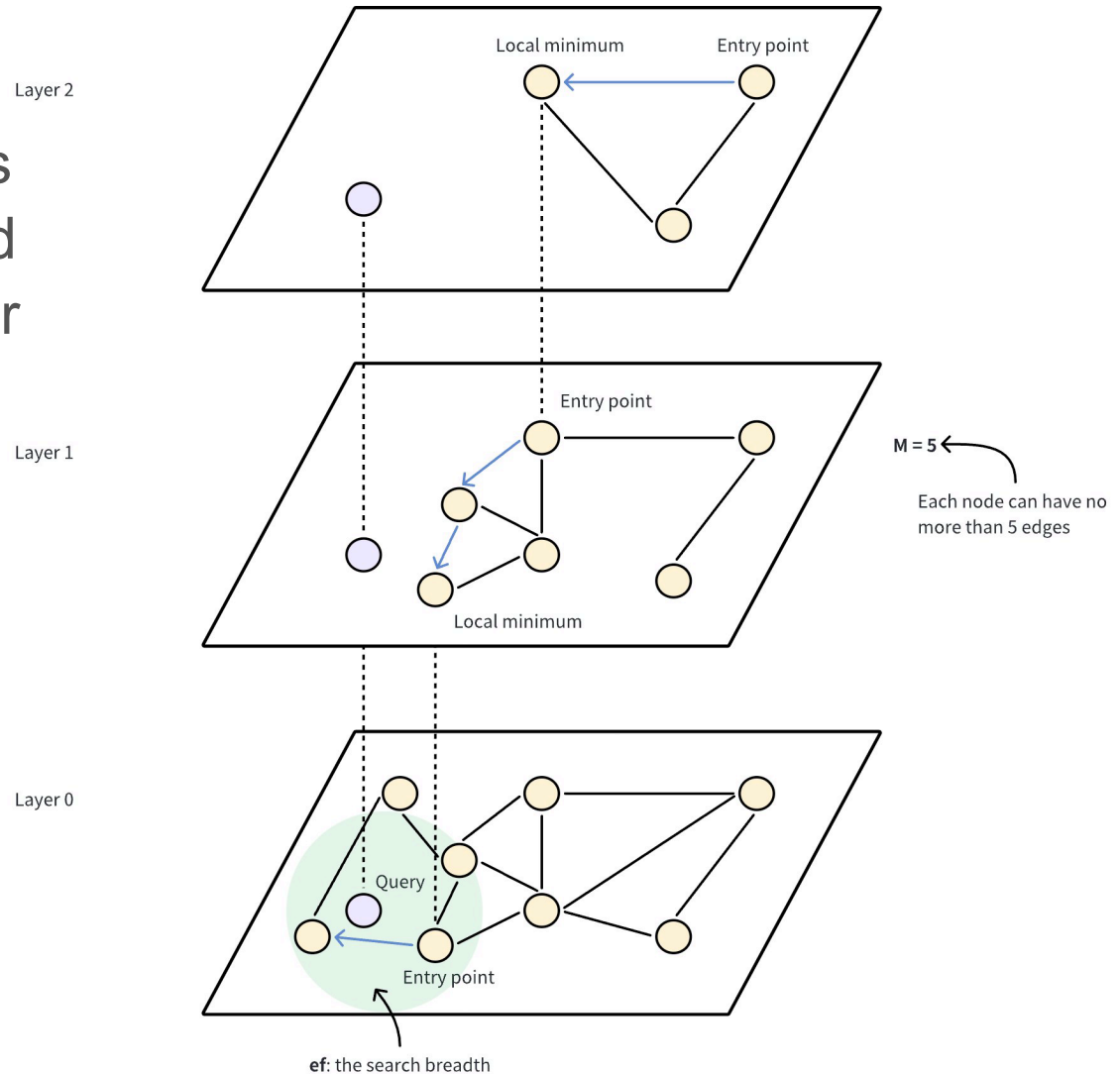


3. Indexing: Design Decisions

- ❖ **ANN Index Structure:** HNSW (graph-oriented), IVF (clustering-based), product quantization (compression)
- ❖ **Hybrid Indexes:** Vector + Text Inverted Index (BM25) for dense+sparse fusion
- ❖ **Metadata Filters:** Date, source, owner, etc. might have filter predications: pre-search vs. post-search vs. in-line during search
- ❖ **Update Model:** Batch re-indexing vs. incremental upsert
- ❖ **Common Stores Today:** FAISS, pgvector, Chroma, Pinecone, Weaviate, Qdrant; DBMS vendors too have their own

3. Indexing: HNSW

- ❖ Hierarchical Navigable Small Worlds (HNSW) is a popular graph-oriented vector index structure for fast retrieval
- ❖ Nodes represent data points (vectors)
- ❖ Layers consolidate proximity info.



3. Indexing: Tradeoffs and Best Practices

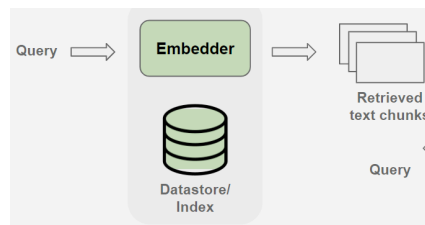
❖ Tradeoffs:

- ❖ Recall vs. latency vs. DRAM vs. update cost—pick 3!
 - ❖ HNSW: fast and accurate, but DRAM-heavy, slow update
 - ❖ IVF + PQ: more compact in memory, more tunable, but lower recall ceiling
- ❖ DevOps Burden: Vendor-hosted vs. self-hosted

❖ Some best practices:

- ❖ HNSW popular for read-heavy, infrequent-update workloads; IVF + PQ for when DRAM tight and/or corpus too huge
- ❖ Always co-locate metadata for filtering; vector-only indexes hit access-control walls fast!
- ❖ *No universal optimal indexing scheme. Tune for your workload!*

4. Retrieval

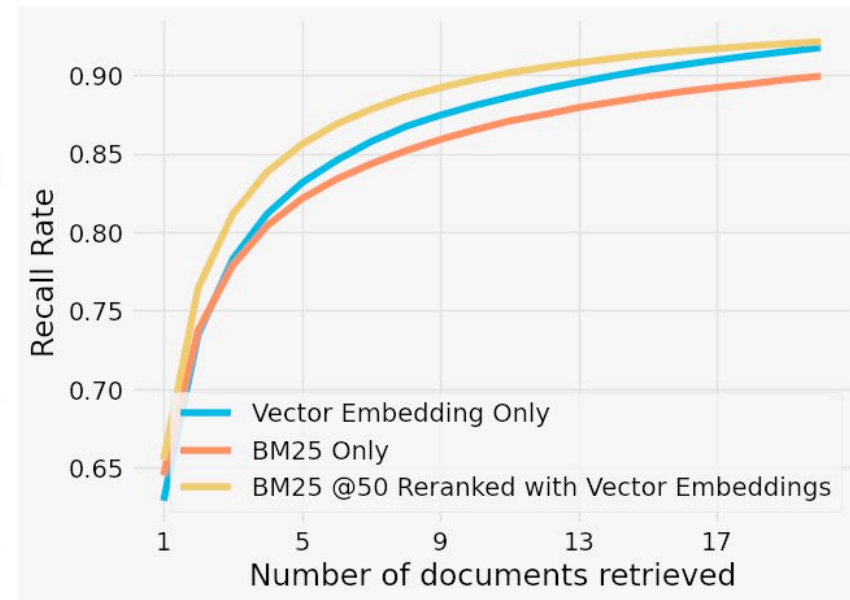
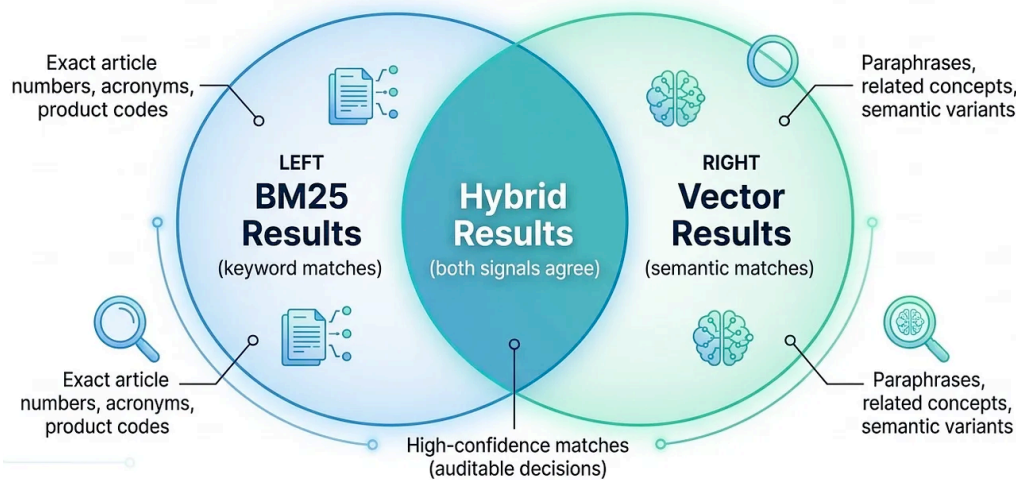


Q: How to retrieve top-K matches quickly and accurately?

- ❖ Standalone vector search is limited; often exact keyword/phrase (e.g., IDs, names, error codes, etc.) match needed
- ❖ Good ol' IR-style keyword search (e.g., BM25) helps there; AKA “sparse” retrieval in RAG world
- ❖ Typically, “hybrid” search = dense (ANN on vectors) + sparse (BM25) + rerank (e.g., with RRF) or weighted scores

4. Retrieval: Hybrid Search

BM25 vs. Vector only vs. Both

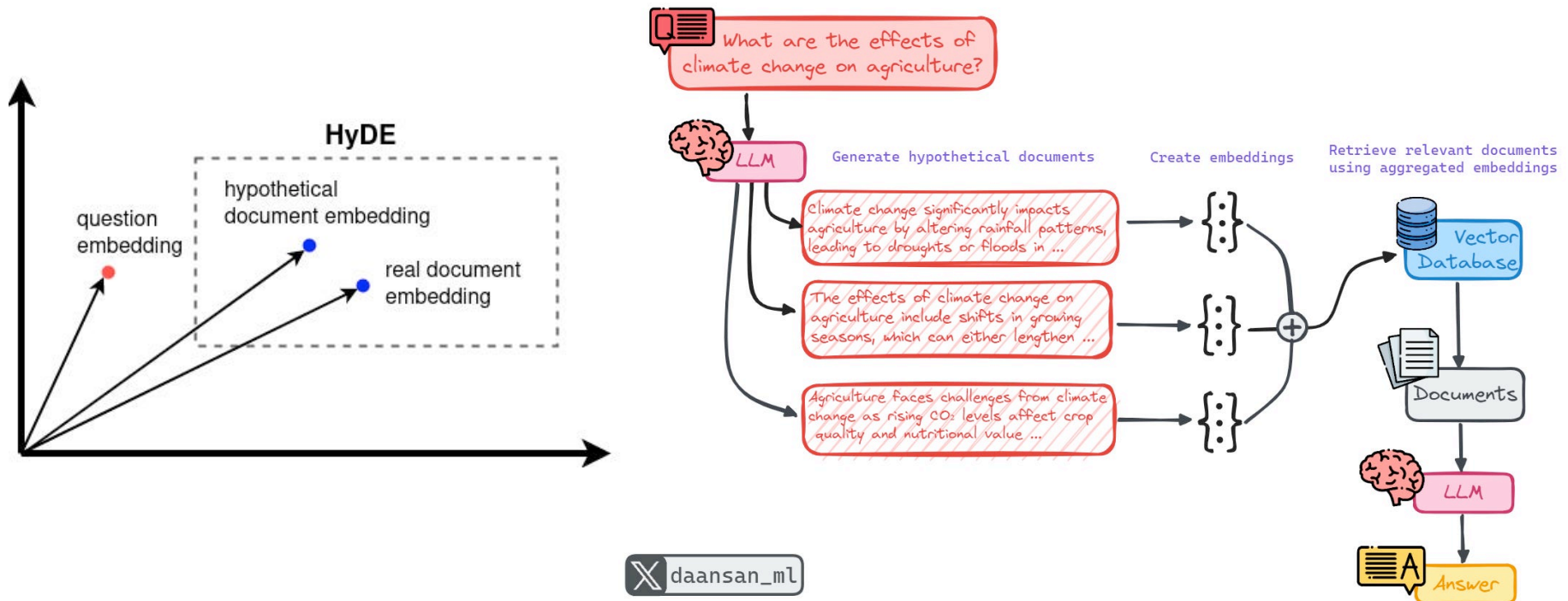


4. Retrieval: More Design Decisions

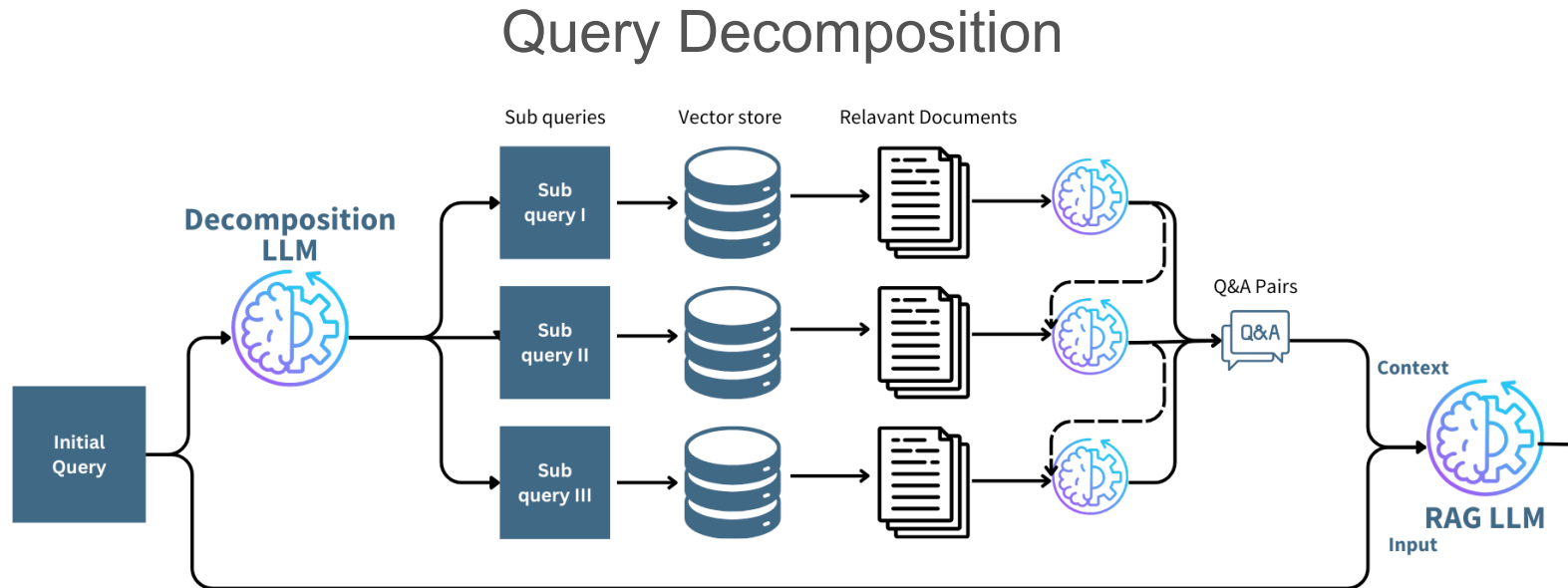
- ❖ **K** in top-K: How many to retrieve per index?
- ❖ **Predicates:** How to constitute them on metadata?
- ❖ **Query Rewriting Upstream:** Before query sent to search:
 - ❖ HyDE (Hypothetical Document Embeddings): Ask LLM to “guess” an answer and embed that too for actual search!
 - ❖ Multi-query: Ask LLM to create variations of query
 - ❖ Decomposition / Expansion: Classical IR style techniques
 - ❖ Can also mix and match among the above
- ❖ **Source-Separated Indexes:** Maintain separate indexes for different corpora / modalities / teams

4. Retrieval: More Design Decisions

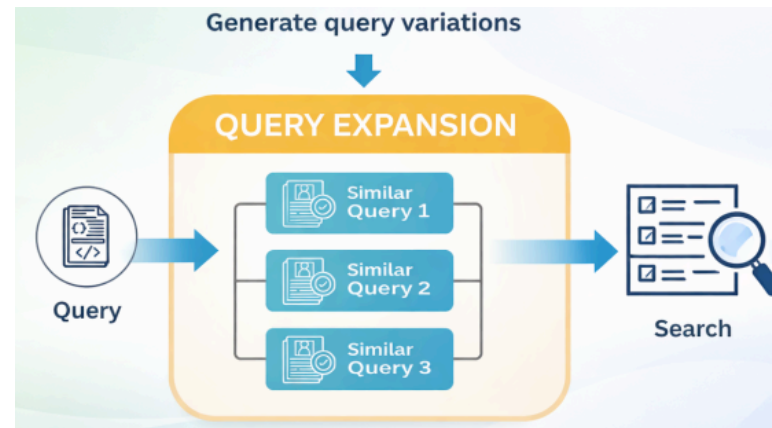
Why HyDE usually helps



4. Retrieval: More Design Decisions



Multi-Query Expansion



4. Retrieval: Tradeoffs

- ❖ **Sparse vs. Dense:** Sparse can miss paraphrasing, while dense can miss exact terms
- ❖ **K:** Larger value helps recall, but adds more latency and downstream costs (for reranking, context length, etc.)
- ❖ **Query Rewriting:** Can help recall, but adds more latency, possibly more costs, and another non-trivial knob to debug!
- ❖ **Multi-Source Fusion:** Helps coverage but complicates evals and debugging
- ❖ *No universal optimal retrieval scheme or K! Tune on eval set*

4. Retrieval: Some Best Practices

- ❖ Typically $K \sim 10$ to 50 per index; narrow further with reranking
- ❖ Do query rewriting only if eval shows recall is bottleneck
- ❖ Carefully debug queries that have zero retrieval results!
- ❖ Default to hybrid search with **Reciprocal Rank Fusion (RRF)**; almost always beats unimodal retrieval

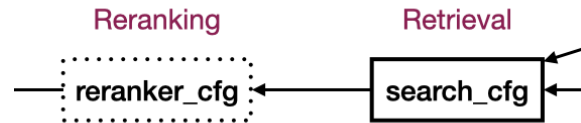
$$\text{RRF} = \sum \frac{1}{C + \text{Rank}(\text{document})}$$

$C \sim 60$ in practice

Typically not much impact
from tuning up or down

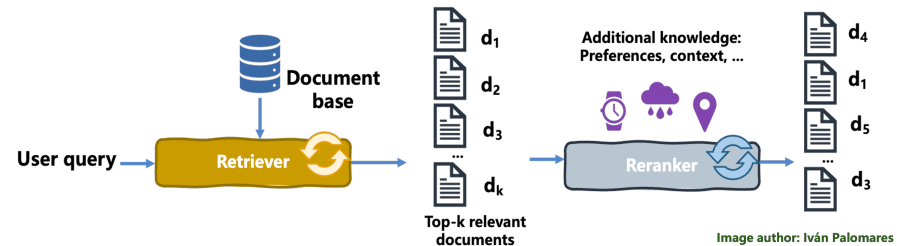
BM25 (sparse)	Vector (dense)	Fused
A	B	$B = 1/61 + 1/62 = 0.0325$
B	C	$C = 1/62 + 1/62 = 0.0323$
C	D	$A = 1/61 = 0.0164$
		$D = 1/63 = 0.0159$

5. Reranking



Q: When does reordering the retrieved and fused results help?

- ❖ Reranking applies query-aware ML (cross-encoders, CoBERT, LLMs, etc.) to rescore retrieved (and fused) results



- ❖ Better-ranked top-K -> fewer chunks needed -> less lost-in-the-middle exposure downstream
- ❖ Reranking is how high recall can become high precision too; without it, you either lose accuracy (small K, miss answers) or eat lost-in-the-middle (big K, drown them)

5. Reranking: Design Decisions

- ❖ **Cross-Encoder Rerankers:** Joint (query, chunk) scoring; the standard precision-quality choice, e.g., Cohere Rerank, BGE, MS-MARCO MiniLM
- ❖ **Late-Interaction:** per-token scoring; often more accurate than cross-encoder but slower (e.g. ColBERT)
- ❖ **SLM-as-Reranker:** SLM scores (query, chunk); flexible reasoning, slower and costlier
- ❖ On top of any reranker, can do **Score Fusion:** blend reranker score with metadata signals (recency, authority, popularity, etc.)

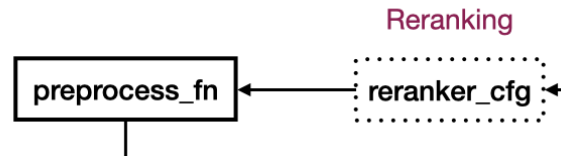
5. Reranking: Tradeoffs

- ❖ Often results in significant boost to precision@K (even 30%) but with slightly more latency and cost
- ❖ Cross-encoders cannot be pre-indexed; every query is a fresh forward pass
- ❖ Reranker model domain match matters as much as embedding model domain match
- ❖ LLM rerankers are flexible but expensive at high QPS

5. Reranking: Some Best Practices

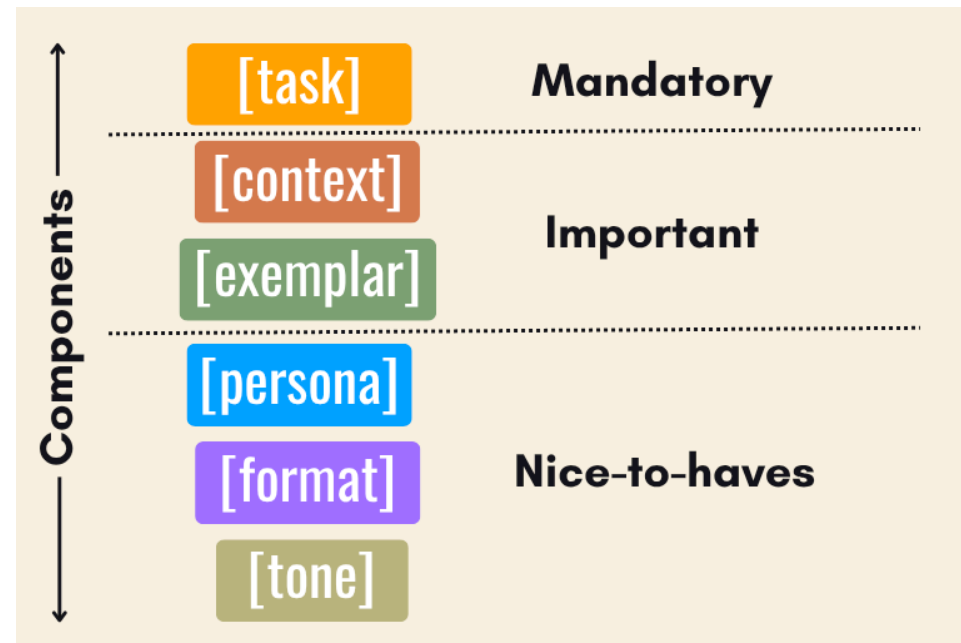
- ❖ Rerank when retrieval recall is high but precision is low—a common RAG failure pattern
- ❖ Default to a self-hosted Cross-Encoder (Cohere, BGE) before reaching for ColBERT or SLM rerankers
- ❖ Make it top-5 to top-10 *after* (fusion and) reranking; more chunks ≠ better answers (Lost-in-the-Middle)
- ❖ *No universal best option!* Eval and tune reranker quality separately from retriever quality

6. Context Stitching



Q: How to organize and compose the final prompt?

- ❖ Recollect best practices from prompt engineering
- ❖ **Ordering:** Put best chunks at start or end
- ❖ **Format:** MD, JSON, XML
- ❖ **Cleanup:** Dedupe overlaps, summarize chunks
- ❖ **Budget:** Account for token rate limits



6. Context Stitching: Tradeoffs and Best Practices

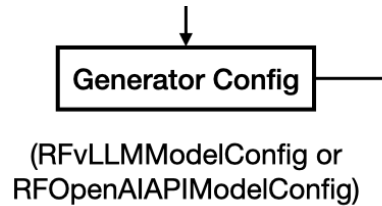
❖ Tradeoffs:

- ❖ Longer context can help recall but can cause context rot and raise latency and costs
- ❖ Compression: Reverse tradeoffs as above
- ❖ Heavy formatting helps citation but can cost token
- ❖ Aggressive abstention instructions help reduce hallucination

❖ Some best practices:

- ❖ Number the chunks and instruct the model to cite by number
- ❖ Put user's query at the end; recency bias can help
- ❖ Cap context at the model's effective recall length, not its advertised window
- ❖ *No universal best option! Eval it, version it, regression test it*

7. Generation: Design Decisions



Produce a faithful grounded answer under constraints—or abstain

- ❖ **Model:** Size, family (reasoning vs. instruct), open vs. closed
- ❖ **Decoding:** Temperature, top-p; determinism vs. diversity
- ❖ **Constrained Decoding:** JSON schema or grammar for parseable output
- ❖ **Tool calling:** Re-retrieve, calculator, code, etc. vs. free text
- ❖ **Reasoning Budget:** Think-before-answer for harder synthesis
- ❖ **Self-checks:** Self-consistency, draft-then-verify

7. Generation: Tradeoffs and Best Practices

❖ Tradeoffs:

- ❖ Bigger model, reasoning, self-checks: higher faithfulness, but also higher latency and cost
- ❖ Temperature: low => more consistent but brittle; high => more creative but hallucinatory
- ❖ Constrained decoding: parseable, but occasional quality hit

❖ Some best practices:

- ❖ Temperature: 0.0-0.3 typical for factual RAG
- ❖ Constrained decoding when output is parsed downstream
- ❖ Reasoning models when faithfulness > latency (e.g., legal, medical, audit)
- ❖ Add a cheap second-pass verifier for high-stakes answers
- ❖ *No universal optimal choice! Tune it on your eval set*

Parts of Canonical VectorRAG

1. **Parsing and Splitting/Chunking:** Turn raw data/docs into retrieval units
2. **Embedding:** Encode each chunk as a dense vector
3. **Indexing:** Store vectors for fast (approximate) nearest-neighbor (NN/ANN) search
4. **Retrieval:** Fetch top-K candidates from one or more sources; fuse into a single ranked list
5. **Reranking:** Rescore top-K to reorder chunks by relevance to query
6. **Context Stitching:** Assemble the final prompt—system instructions plus ordered, deduped, formatted, budgeted context
7. **Generation:** LLM produces a grounded answer with citations or abstains

Review Questions

1. Briefly distinguish prompt eng. vs. retrieval eng. vs. context eng.
2. Will ever-growing context windows obviate retrieval? Why or why not?
3. What is the lost-in-the-middle problem? How does context rot relate?
4. Name the 7 parts of a canonical VectorRAG pipeline. Why that order?
5. What is the core tradeoff in chunk scheme and size selection?
6. Why is swapping an embedding model a costly decision? What are Matryoshka embeddings and how do they help resolve that issue?
7. What is hybrid search? Why does it usually beat pure vector search?
8. How do the retriever and reranker components of a canonical RAG pipeline divide responsibility for recall vs. precision?
9. Name key levers in context stitching for configuring output quality.
10. You are an AI eng. at an enterprise AI team. Your AI colleague suggests leveraging the full 10M-token context window of new LLMs to skip RAG altogether for an internal Q&A chatbot. Do you agree? Why or why not?