



AI
Bees

Generative AI

RAG & Vector Search

Revision Notes for Students

Embeddings • Vector Databases • RAG • Incremental RAG (iRAG)

This document builds your understanding from the ground up. Each section is a prerequisite for the next — start from Section 1 even if you already know some concepts:

Section	Topic	Why You Need It First
1	Embeddings	Core building block — how text becomes a number the computer can compare
2	Why Traditional Search Fails	Understand the problem that embeddings solve
3	Vector Databases	Where embeddings are stored and searched efficiently
4	Chunking	How large documents are broken down before embedding
5	RAG (Full Pipeline)	The complete system — bringing all concepts together
6	Incremental RAG (iRAG)	How to keep a RAG system up-to-date without rebuilding

1. Embeddings — Turning Words into Numbers

1.1 The Core Problem

Computers are fundamentally mathematical machines — they only understand numbers. But most of the world's information is stored as text: documents, emails, articles, chat messages. To make an AI system that can search, compare, or reason over text, we first need to convert that text into a mathematical form.

Simply converting letters to ASCII codes doesn't work — those numbers carry no meaning about what the word means. The number for 'cat' would be completely unrelated to the number for 'kitten', even though they mean almost the same thing. We need a richer representation.

✗ Problem: Computers can't compare meaning. ASCII('cat') has nothing to do with ASCII('kitten') — but the two words are semantically almost identical.

✓ Solution: Embeddings map words (and sentences) into a high-dimensional mathematical space where semantically similar words are placed close together geometrically.

1.2 What is an Embedding?

An embedding is a numeric representation of a word, phrase, sentence, or document as a dense vector — a list of floating-point numbers — in a high-dimensional space. The key property that makes embeddings powerful is semantic preservation: words or sentences with similar meaning are represented by vectors that are geometrically close to each other.

- **Dense vector**: every dimension has a meaningful value (unlike sparse vectors where most values are zero).
- **High-dimensional**: real models use 768 dimensions (BERT), 1536 dimensions (OpenAI ada-002), or more.
- **Semantic similarity preserved**: cat and kitten are close; cat and airplane are far apart.
- **Produced by an embedding model**: a neural network (BERT, SentenceTransformer, OpenAI Embeddings) that was trained to produce these meaningful representations.

```
# Example: what an embedding looks like in Python
# (768 numbers for BERT, only first few shown)

'cat' → [0.013, -0.278, 0.114, 0.331, -0.052, ..., 0.005]
      (768 numbers total)

'kitten' → [0.015, -0.265, 0.118, 0.327, -0.049, ..., 0.003]
          (very similar numbers → close in space)

'airplane' → [-0.412, 0.753, -0.289, -0.114, 0.831, ..., 0.241]
            (very different numbers → far apart in space)
```

1.3 How Embeddings Work — The Intuition

Think of embedding as answering a long list of questions about a word and packing all the answers into a single vector. Here is a simplified 3-dimensional example using the word 'cat':

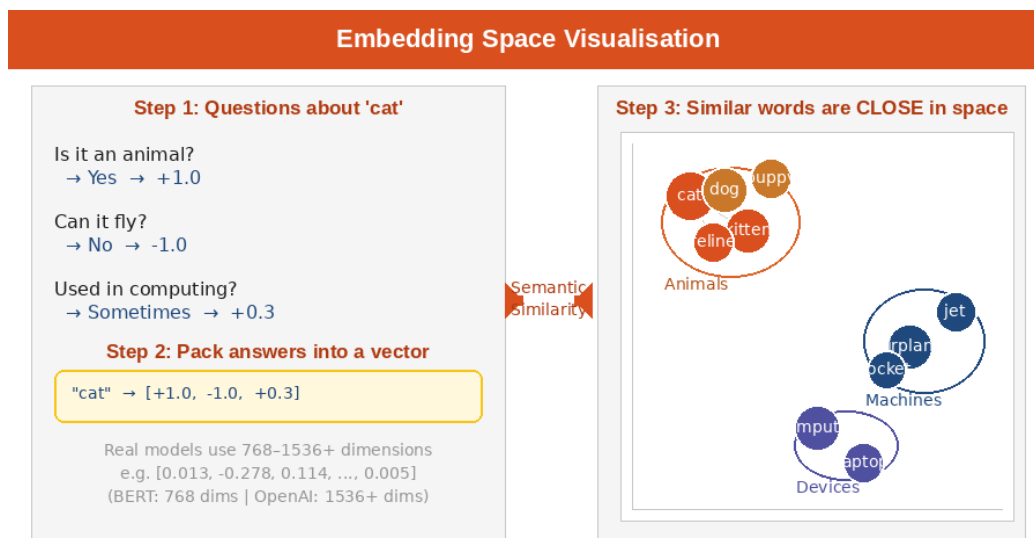



Figure 1: How words become vectors and how similar words cluster together in embedding space

In real models, instead of 3 questions, the model answers hundreds or thousands. And instead of humans designing the questions, the neural network learns the most useful dimensions automatically from training on billions of text examples.

 **Key Point:** The magic of embeddings: you never need to define the dimensions yourself. The embedding model learns what dimensions are most useful to distinguish meaning — through training on massive amounts of text.


1.4 Sentence and Paragraph Embeddings

Embeddings work not just for single words but for entire sentences, paragraphs, or even whole documents. A sentence embedding captures the overall meaning of the full sentence — not just individual words. This is critical for RAG, where we embed entire text chunks and then compare them to a user's full question.

```
# Sentences with similar meaning → similar embeddings


Sentence A: 'What is the capital of France?'
Sentence B: 'Which city is the capital of France?'
Sentence C: 'How do I bake a chocolate cake?'

cosine_similarity(A, B) → 0.97  (very close — same meaning)
cosine_similarity(A, C) → 0.12  (very far — different topics)
```

 **Key Point:** Cosine similarity is the standard way to compare two embeddings. It measures the angle between two vectors — 1.0 means identical direction (same meaning), 0 means unrelated, -1 means opposite.

1.5 Common Embedding Models

Model	Provider	Dimensions	Best For
text-embedding-ada-002	OpenAI	1536	General text, RAG pipelines, production use
text-embedding-3-small	OpenAI	1536	Fast, cost-efficient general embeddings
BERT / DistilBERT	Google / HuggingFace	768	Research, on-premise, fine-tuning
all-MiniLM-L6-v2	SentenceTransformers	384	Lightweight, fast, good quality — great for learning
embedding-001	Google Gemini	768	Google ecosystem, fast Gemini integration
multilingual-e5-large	Microsoft	1024	Multilingual text — 100+ languages

 **Remember:** For building your first RAG system, 'all-MiniLM-L6-v2' from SentenceTransformers is a great free starting point. For production, OpenAI's text-embedding-3-small gives excellent quality at low cost.

2. Why Traditional Search Methods Fail

2.1 The Problem with a 1,000-Page Document

Imagine you have a 1,000-page PDF document — a company policy manual, a legal contract, or a technical specification. If a user asks a question about it, the naive approach would be to send the entire document to an LLM and ask it to answer. This approach has several serious problems:

Problem	Details	Impact
Token Limits	Most LLMs have a strict context window (e.g. 128K tokens for GPT-4, ~200K for Claude). A 1,000-page PDF is far beyond any current limit.	Impossible — the entire document cannot fit
Focus Dilution	Even with a large context window, feeding the entire document dilutes the LLM's attention. It struggles to pinpoint the 2 relevant paragraphs in 1,000 pages.	Poor quality answers — LLM gets confused
Cost	API pricing is per token. Sending 1,000 pages costs enormous amounts of money — for every single user question.	Economically unviable at scale
Speed	Processing 1,000 pages takes significant time. Users expect answers in seconds, not minutes.	Terrible user experience

✅ **Solution:** RAG (Retrieval-Augmented Generation): instead of sending the whole document, retrieve only the 3-5 most relevant paragraphs for each specific question, then send only those to the LLM.

2.2 Why Keyword Search Isn't Enough

Before embeddings, the standard solution was keyword matching or TF-IDF-based search. These methods work well for exact word matches, but fail for semantic understanding — which is critical for a good Q&A system.

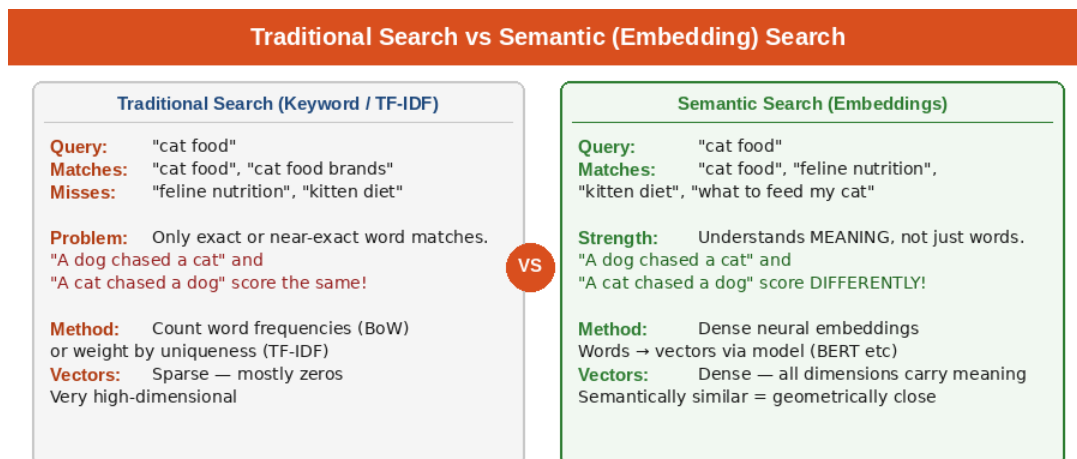


Figure 2: Keyword / TF-IDF search vs Semantic (Embedding) search — the core problem and solution

Bag-of-Words (BoW) and TF-IDF

These methods represent documents as vectors where each dimension corresponds to a word in the vocabulary, and the value represents word frequency or TF-IDF score. They look at how often words appear but have no understanding of word meaning or context.

Example: **"A dog chased a cat"** and **"A cat chased a dog"** would produce identical BoW vectors — yet they describe opposite events!

- **Sparse vectors**: most dimensions are zero (only words present in a document have non-zero values). A vocabulary of 50,000 words = 50,000 dimensions, mostly zeros.
- **No semantic understanding**: 'feline' and 'cat' are completely unrelated because they share no characters.
- **Computationally expensive at scale**: comparing sparse 50,000-dimensional vectors across millions of documents is slow.


N-grams

N-grams are contiguous sequences of N words. A bi-gram (N=2) of 'The quick brown fox' produces: 'The quick', 'quick brown', 'brown fox'. While N-grams capture some local word order (better than pure BoW), they still completely fail to understand sentence-level meaning.

Key limitation: *N-grams can capture that 'quick brown' often appears together, but they cannot understand that 'The quick brown fox' and 'The fast auburn fox' mean the same thing.*

Keyword Matching

The simplest form — find documents containing the exact words in the query. Fast and predictable, but semantically blind. A user asking 'How do I configure the settings?' will not find a document that says 'You can adjust the preferences in the options panel' — even though they mean the same thing.

 **Remember:** All traditional methods share the same fundamental flaw: they match words (symbols), not meanings (semantics). Embeddings solve this by encoding meaning directly into the numerical representation.

3. Vector Databases

3.1 The Storage Problem

Once we have embeddings for thousands or millions of text chunks, we need somewhere to store them and — crucially — search through them extremely fast. A regular database (PostgreSQL, MySQL) stores rows of structured data and searches with exact matches or range queries. It has no built-in concept of 'similarity' or 'nearest neighbour'.

Storing 1 million 768-dimensional vectors and finding the top-10 most similar to a query vector by brute force (computing cosine similarity with every single stored vector) would be too slow for real-time applications. This is exactly the problem that vector databases solve.

3.2 What is a Vector Database?

A vector database is a specialised database designed to store, manage, and efficiently search high-dimensional vectors — such as those generated by embedding models. Instead of exact match queries (WHERE name = 'John'), it answers nearest neighbour queries ("find the 5 vectors most similar to this query vector").

- **ANN (Approximate Nearest Neighbour)**: vector databases use ANN algorithms (like HNSW, IVF) that find results that are near-optimal but much faster than exhaustive search. In practice, the approximation is imperceptible.
- **Storage + metadata**: alongside each vector, the database also stores the original text chunk and any metadata (source document, page number, date) needed to construct the answer.
- **Scalability**: designed to handle millions to billions of vectors with millisecond query times.

3.3 How Vector Search Works


When you query a vector database, it does not scan every stored vector. Instead, it uses index structures (like HNSW — Hierarchical Navigable Small World graphs) to quickly navigate to the neighbourhood of the query vector in high-dimensional space and return the closest matches.

```
# Conceptual flow of a vector DB query:

1. User asks: 'What is chunking in RAG?'
2. Embed the question: [-2.1, 4.3, 0.8, ..., 1.2] (768 dims)
3. Query vector DB: find_top_k(query_vector, k=5)
4. Vector DB returns: 5 text chunks most similar to the question
5. Feed chunks + question to LLM → grounded answer
```

3.4 Vector Database Tools

Tool	Type	Best For	Key Feature
FAISS	Library (Meta/Facebook)	Research, local development, offline use	Extremely fast, no server needed — runs in-memory
ChromaDB	Open-source (local/cloud)	Learning, prototyping, small projects	Simple Python API, easy to start, stores metadata
Pinecone	Managed cloud service	Production, enterprise, managed infrastructure	Fully managed, auto-scaling, no maintenance
Qdrant	Open-source (local/cloud)	Production, on-premise, advanced filtering	Fast, supports rich payload filtering alongside vectors
Weaviate	Open-source (local/cloud)	Knowledge graphs, multi-modal (text + images)	GraphQL interface, automatic vectorisation
Milvus	Open-source (local/cloud)	Large-scale enterprise, billions of vectors	Distributed architecture for massive scale

 **Key Point:** For learning and building your first RAG system, start with ChromaDB — it runs locally with zero configuration and has an intuitive Python API. For production, evaluate Pinecone (easiest to manage) or Qdrant (most feature-rich open-source option).

3.5 Vector DB vs Traditional DB — Side by Side

Aspect	Traditional Database (SQL)	Vector Database
Query type	Exact match, range, joins	Nearest neighbour (semantic similarity)
Data stored	Structured rows and columns	Vectors (float arrays) + metadata
Search method	B-tree index, full-table scan	ANN index (HNSW, IVF, etc.)
Use case	Transactions, user accounts, orders	Semantic search, recommendation, RAG
Speed at scale	Fast for exact queries	Fast for similarity queries at millions of vectors
Examples	PostgreSQL, MySQL, SQLite	FAISS, ChromaDB, Pinecone, Qdrant, Milvus

4. Chunking — Preparing Documents for RAG

4.1 What is Chunking?

Chunking is the process of splitting large documents into smaller, manageable pieces (chunks) before embedding them. Each chunk is embedded independently and stored as a separate entry in the vector database. When a user asks a question, only the most relevant chunks are retrieved — not the whole document.

Chunking is one of the most important and often underestimated steps in building a RAG system. The size and method of chunking directly affects the quality of retrieval, which directly affects the quality of the LLM's answers.

💡 Key Point: Think of chunking like creating index cards for a textbook. Each card covers one topic, is small enough to be quickly scanned, and contains enough context to be understood standalone.

4.2 Why Chunk Size Matters

Chunk Size	Trade-off	Typical Use Case
Too small (< 100 tokens)	High precision but missing context. A sentence in isolation may not make sense.	Not recommended — context loss is too high
Small (100-300 tokens)	Good for very specific fact retrieval. Minimal noise but may miss surrounding context.	FAQ systems, specific fact lookup
Medium (300-500 tokens)	Best balance of precision and context. One complete idea per chunk.	Most RAG systems — recommended default
Large (500-1000 tokens)	More context preserved but retrieval less precise. LLM gets more noise.	Narrative documents, long explanations
Too large (> 1000 tokens)	Retrieval becomes imprecise. Too much irrelevant text passed to LLM.	Not recommended — defeats the purpose of RAG

4.3 Chunk Overlap — Preserving Context at Boundaries

A natural chunk boundary (e.g. every 400 tokens) can split a sentence or a concept across two chunks, losing context. Chunk overlap solves this by including a repeated portion of text between consecutive chunks — typically 50-100 tokens. This ensures that content at the boundary of a chunk is not lost.

Example with `chunk_size=400`, `overlap=50`:

```
Chunk 1: [tokens 1 → 400]
Chunk 2: [tokens 350 → 750] ← starts 50 tokens before Chunk 1 ends
Chunk 3: [tokens 700 → 1100] ← starts 50 tokens before Chunk 2 ends
```

The 50-token overlap means context around each boundary is captured in both adjacent chunks → better retrieval.

4.4 Chunking Strategies

Strategy	Method	Best For
Fixed-size chunking	Split every N tokens regardless of content	Simple, fast — good starting point
Sentence-based chunking	Split at sentence boundaries (full stop, newline)	Preserves grammatical units — better quality
Paragraph-based chunking	Split at paragraph breaks	Narrative text, articles, documentation
Semantic chunking	Split when topic changes (using embeddings to detect)	Best quality — more complex to implement
Recursive chunking	Split by section, then paragraph, then sentence if needed	Structured documents like PDFs, reports
Document-specific	Custom rules (e.g. split HTML by <div>, code by function)	HTML, Markdown, source code

5. RAG — Retrieval-Augmented Generation

5.1 What is RAG?

Retrieval-Augmented Generation (RAG) is an AI architecture that enhances an LLM's ability to answer questions by first retrieving relevant information from an external knowledge base — and then using that retrieved information as context in the prompt to generate the final answer.

RAG solves the two biggest limitations of standard LLMs: they have a knowledge cutoff date (they don't know about recent events), and they have no access to your private data (company documents, personal files, proprietary databases). RAG gives LLMs access to any knowledge base you build — with no fine-tuning required.

LLM Limitation	How RAG Solves It
Knowledge cutoff	RAG retrieves from a vector DB that can be updated with new documents at any time
No private data access	You index your own documents into the vector DB — the LLM gets access to them via retrieval
Context window limit	Only the 3-5 most relevant chunks (not the full document) are sent to the LLM
Hallucination risk	LLM is instructed to answer ONLY from the provided context — grounding reduces hallucination
Cost of large context	Sending 500 tokens of retrieved context is far cheaper than sending 100,000 tokens of full documents

5.2 RAG Architecture — The Two Phases

RAG has two distinct phases. The Indexing Phase happens once (or when documents change). The Query Phase happens for every single user question. It is critical to understand this distinction:

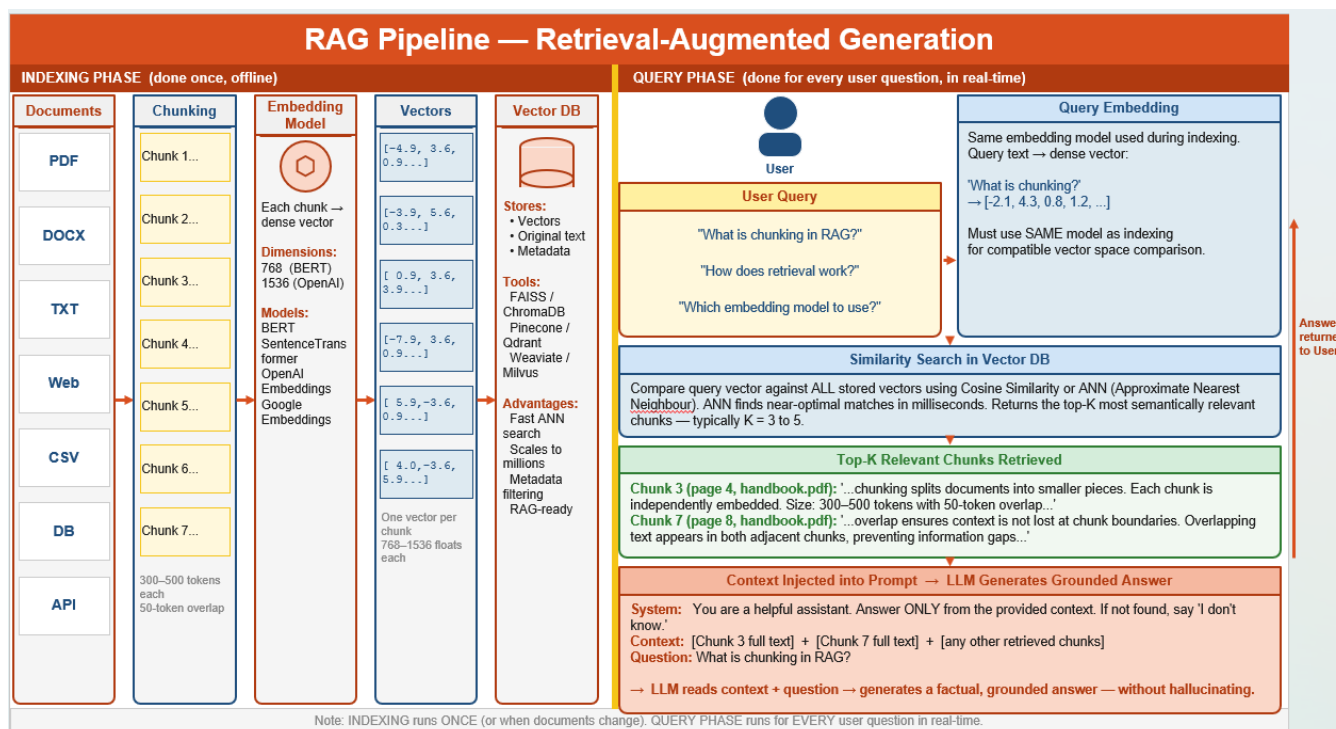


Figure 3: Complete RAG Pipeline — Indexing Phase (left, offline) and Query Phase (right, per-question)

5.3 Phase 1 — Indexing (Done Once, Offline)

The indexing phase processes your documents and builds the searchable vector database. This is typically done as a batch job before the application goes live, and re-run whenever documents change. Because it only runs once, it can take minutes or even hours for large document sets — that is acceptable.

Step 1: Load Documents

Ingest documents from any source — PDFs, Word files, web pages, databases, CSV files, code repositories. LangChain provides ready-made document loaders for all common formats (PDFLoader, WebBaseLoader, CSVLoader, etc.).

Step 2: Chunk Documents

Split each document into chunks of the appropriate size (typically 300–500 tokens) with overlap (50–100 tokens). Each chunk becomes one independently searchable unit in the vector database.

Step 3: Embed Each Chunk

Pass every chunk through an embedding model to convert it into a dense vector. This step is computationally intensive for large document sets but only happens during indexing — not at query time.

Step 4: Store in Vector Database

Store each (vector, original_text, metadata) triplet in the vector database. The metadata — source file name, page number, date, section title — is important for providing citations in the final answer.

```
# Indexing phase — pseudocode (conceptual)

documents = load_documents('company_handbook.pdf')
chunks    = chunk_documents(documents, chunk_size=400, overlap=50)
vectors    = embedding_model.embed(chunks)    # one vector per chunk
vector_db.store(vectors, chunks, metadata)    # save for querying

# This runs ONCE. After this, the vector DB is ready.
```

5.4 Phase 2 — Query (Every User Question)

The query phase is the real-time part that runs for every user question. It is fast (milliseconds) because the heavy work was already done during indexing. The flow:

Step 1: Embed the User's Question

Convert the user's question into a vector using the same embedding model used during indexing. This is critical — you must use the same model for indexing and querying. Using a different model produces incompatible vector spaces.

Step 2: Similarity Search

Query the vector database with the question's vector to find the top-K most similar chunks (typically K=3 to 5). The database returns the original text of those chunks along with their metadata.

Step 3: Build Augmented Prompt

Construct a prompt that includes: (1) a system instruction telling the LLM to answer only from the provided context, (2) the retrieved chunks as context, and (3) the user's original question.

```
# Augmented prompt structure:

System: You are a helpful assistant. Answer ONLY using the
       context provided below. If the answer is not in the
       context, say 'I don't know based on the documents.'

Context:
  [Chunk 1]: ...retrieved text paragraph 1...
  [Chunk 2]: ...retrieved text paragraph 2...
  [Chunk 3]: ...retrieved text paragraph 3...

Question: What is the company's parental leave policy?

Answer:
```

Step 4: LLM Generates Grounded Answer

The LLM reads the context and generates an answer based only on that information. This grounding dramatically reduces hallucination because the model is constrained to the retrieved evidence rather than relying on potentially outdated or incorrect training knowledge.

💡 **Key Point:** The quality of a RAG system is roughly: Retrieval quality × LLM quality. A great LLM cannot compensate for bad retrieval — if the wrong chunks are retrieved, the LLM will give a wrong or irrelevant answer. Spend 70% of your optimisation effort on the retrieval step.

5.5 RAG Use Cases

Use Case	Description	Example
Document Q&A	Answer questions about specific documents	"What does Section 4.2 of the contract say about penalties?"
Knowledge Base Chatbot	Company internal assistant answering from policy docs	HR bot, IT helpdesk, onboarding assistant
Customer Support	Answer support questions from product documentation	"How do I reset my password in the mobile app?"
Research Assistant	Answer from a library of research papers or reports	Summarise findings from 50 uploaded PDFs
Code Documentation	Q&A over a codebase or API documentation	"What does the get_user() function return?"
Legal / Compliance	Answer regulatory questions from legal documents	"Does GDPR require consent for this data processing?"
Medical Information	Answer from verified medical literature or protocols	Clinical decision support from approved guidelines

6. Incremental RAG (iRAG)

6.1 The Problem with Traditional RAG

Traditional RAG works brilliantly for static document collections — index once, query forever. But the real world has dynamic data: news articles published every hour, product documentation updated weekly, company policies revised monthly, customer support tickets arriving constantly.

With traditional RAG, every time a single document is added, modified, or deleted, you must rebuild the entire index from scratch: re-chunk all documents, re-embed all chunks, and rebuild the entire vector database. For a knowledge base with millions of documents, this is extremely slow, expensive, and completely impractical.

✗ Problem: Traditional RAG: add 1 new document to a 1,000,000-document collection → must re-embed all 1,000,000 documents and rebuild the entire vector database. Rebuilding might take hours and cost significant money in API calls.

6.2 What is Incremental RAG (iRAG)?

Incremental RAG (iRAG) is an advanced approach to RAG that addresses the challenge of managing dynamic, frequently changing data. Instead of rebuilding the entire knowledge base for every update, iRAG intelligently detects what has changed and processes only the new or modified information.

The core difference is the method of updating. Traditional RAG requires a full system rebuild for every change, while iRAG allows targeted, efficient additions and modifications to the existing system — leaving unchanged content untouched.

✓ Solution: iRAG: add 1 new document → detect only the new chunks → embed only those chunks → insert only those vectors into the vector DB. The rest of the 1,000,000 documents are untouched.

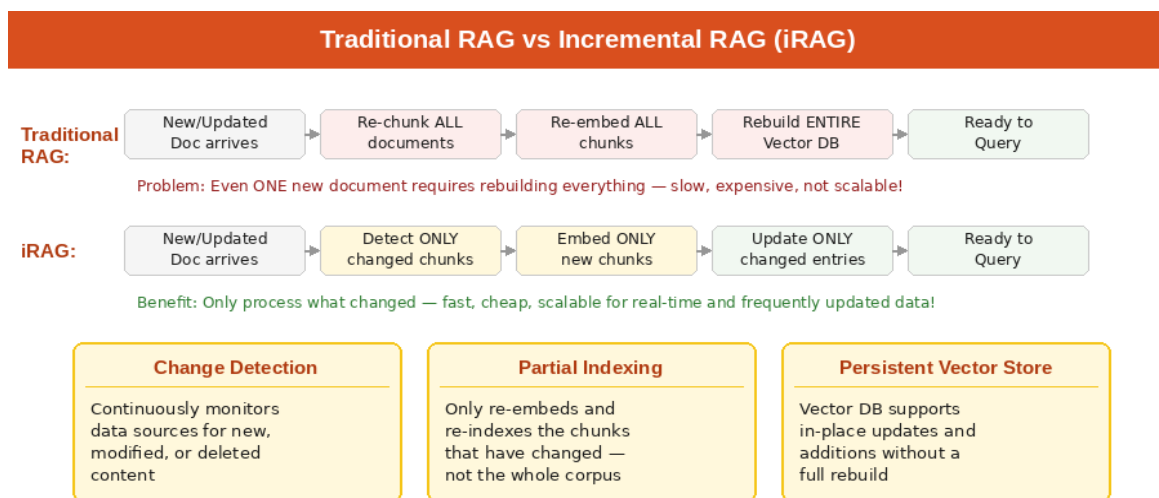


Figure 4: Side-by-side comparison of Traditional RAG rebuild vs iRAG incremental update with three core principles

6.3 iRAG Core Principles

Principle 1 — Change Detection

The iRAG system continuously monitors data sources (file systems, databases, web feeds, APIs) to identify new, modified, or deleted content. It uses techniques like file checksums, modification timestamps, database change data capture (CDC), or document version IDs to determine exactly what has changed since the last indexing run.

- **New document:** detected → chunks extracted → embedded → inserted into vector DB.
- **Modified document:** detected → old chunks removed from vector DB → new chunks embedded → inserted.
- **Deleted document:** detected → all associated vectors removed from vector DB.


Principle 2 — Partial Indexing

Instead of re-processing the entire document corpus, iRAG re-embeds only the specific chunks that have changed. If a 500-page manual has one section updated, iRAG re-embeds only the 3 chunks from that section — not all 1,200 chunks from the full manual.

This requires maintaining a mapping from each chunk back to its source document and location, so the system knows exactly which chunks to delete and replace when a document changes.

Principle 3 — Persistent Vector Store

Traditional RAG could use an in-memory vector store that is rebuilt from scratch each run. iRAG requires a persistent vector database that supports efficient in-place insertions, updates, and deletions without requiring a full database rebuild. Tools like ChromaDB, Qdrant, Pinecone, and Weaviate all support these operations natively.

 **Remember:** Not all vector databases support efficient incremental updates. FAISS (in-memory) requires a full rebuild to add vectors. For iRAG, choose ChromaDB, Qdrant, Pinecone, or Weaviate — they support add/delete operations on individual vectors.

6.4 Traditional RAG vs iRAG — Full Comparison

Aspect	Traditional RAG	Incremental RAG (iRAG)
Update method	Re-index everything from scratch	Only process changed chunks
Update time	Hours to days for large collections	Seconds to minutes for typical changes
Update cost	Re-embed millions of chunks — very expensive	Only new/changed chunks — minimal cost
Data freshness	Only as fresh as the last full rebuild	Near real-time — updated continuously
Best for	Static document collections	Dynamic, frequently updated content
Vector DB requirement	Can use in-memory (FAISS OK)	Requires persistent DB (ChromaDB, Qdrant)
Implementation complexity	Simple	Moderate — needs change detection logic
Example data sources	Product manuals, legal archives, books	News feeds, support tickets, live wikis, APIs

6.5 When to Choose iRAG

Scenario	Use Traditional RAG	Use iRAG
Document change frequency	Documents are stable (months between updates)	Documents change daily, hourly, or in real-time
Collection size	Small to medium (< 100K documents)	Large or growing (100K+ documents)
Latency tolerance for updates	Downtime for rebuilding is acceptable	Knowledge base must be up-to-date continuously
Budget	Rebuild cost is affordable	Rebuild cost is prohibitive — only pay for changes
Example use case	Legal archive, historical documents	News site, customer support, live product docs


6.6 iRAG Practical Example

Consider an AI customer support bot for a software company. Their product documentation is updated with every software release (weekly). Here is how each approach handles a new release:

```
# Documentation: 5,000 pages | 12,500 chunks | Weekly releases

# Traditional RAG — every release:
Re-chunk 5,000 pages      → 60 minutes
Re-embed 12,500 chunks    → ~$3.50 per rebuild (OpenAI)
Rebuild entire vector DB  → 15 minutes downtime
Total per release: 75+ minutes, $3.50, service downtime

# iRAG — same release (50 pages changed out of 5,000):
Detect 125 changed chunks → 2 minutes
Re-embed only 125 chunks  → ~$0.035 (100x cheaper!)
Update 125 entries in DB  → 30 seconds, no downtime
Total per release: ~3 minutes, $0.035, zero downtime
```

 **Key Point:** At scale, iRAG can reduce update costs by 100x and update time by 95%. For any production RAG system with regularly changing content, iRAG is not an optimisation — it is a necessity.

7. The Complete Picture — How Everything Connects

7.1 The Full Knowledge Flow

Each concept in this document is a layer in the same knowledge stack. Understanding how they connect gives you the full picture of how modern AI knowledge systems work:

Layer	Concept	Role in RAG	Without It...
Foundation	Embeddings	Convert text to searchable numbers	No way to measure semantic similarity
Why Needed	Traditional Search Fail	Proves why embeddings are necessary	Would use keyword search — miss semantic matches
Storage	Vector Database	Store and search millions of embeddings fast	No efficient way to retrieve relevant chunks
Preparation	Chunking	Break documents into embeddable pieces	Can't embed 500-page documents as a whole
Architecture	RAG Pipeline	Combine retrieval + generation for Q&A	LLM uses only training knowledge — no private data
Scalability	iRAG	Keep the system up-to-date efficiently	Full rebuild required for every document change

7.2 Quick Revision — Key Definitions

Term	One-Line Definition
Embedding	A list of numbers (dense vector) that represents the meaning of a text
Cosine Similarity	A measure (0 to 1) of how similar two embedding vectors are in direction
Chunking	Splitting a long document into smaller, independently searchable text pieces
Chunk Overlap	Repeated text between adjacent chunks to preserve context at boundaries
Vector Database	Specialised database for storing and fast-searching high-dimensional vectors
ANN	Approximate Nearest Neighbour — fast algorithm to find similar vectors without checking all
RAG	Retrieval-Augmented Generation — retrieve relevant chunks, then generate a grounded answer
Context Window	Maximum number of tokens an LLM can process in a single request
Grounding	Anchoring LLM output to retrieved evidence to reduce hallucination
iRAG	Incremental RAG — update only changed document chunks instead of full rebuilds
Change Detection	iRAG mechanism that identifies new, modified, or deleted content automatically
Partial Indexing	Re-embedding only changed chunks — not the entire document corpus
Hallucination	When an LLM confidently generates incorrect information not supported by evidence
Knowledge Cutoff	The date after which an LLM has no training data — RAG solves this