RAG 101 - Workshop

Who are we?

Aishwarya Srinivasan

- Head of AI Developer Relations- Growth @ Fireworks AI
- Masters in Data Science- Columbia University
- Ex-Microsoft, Ex-Google, Ex-IBM - Data Scientist
- Startup Advisor & Angel Investor

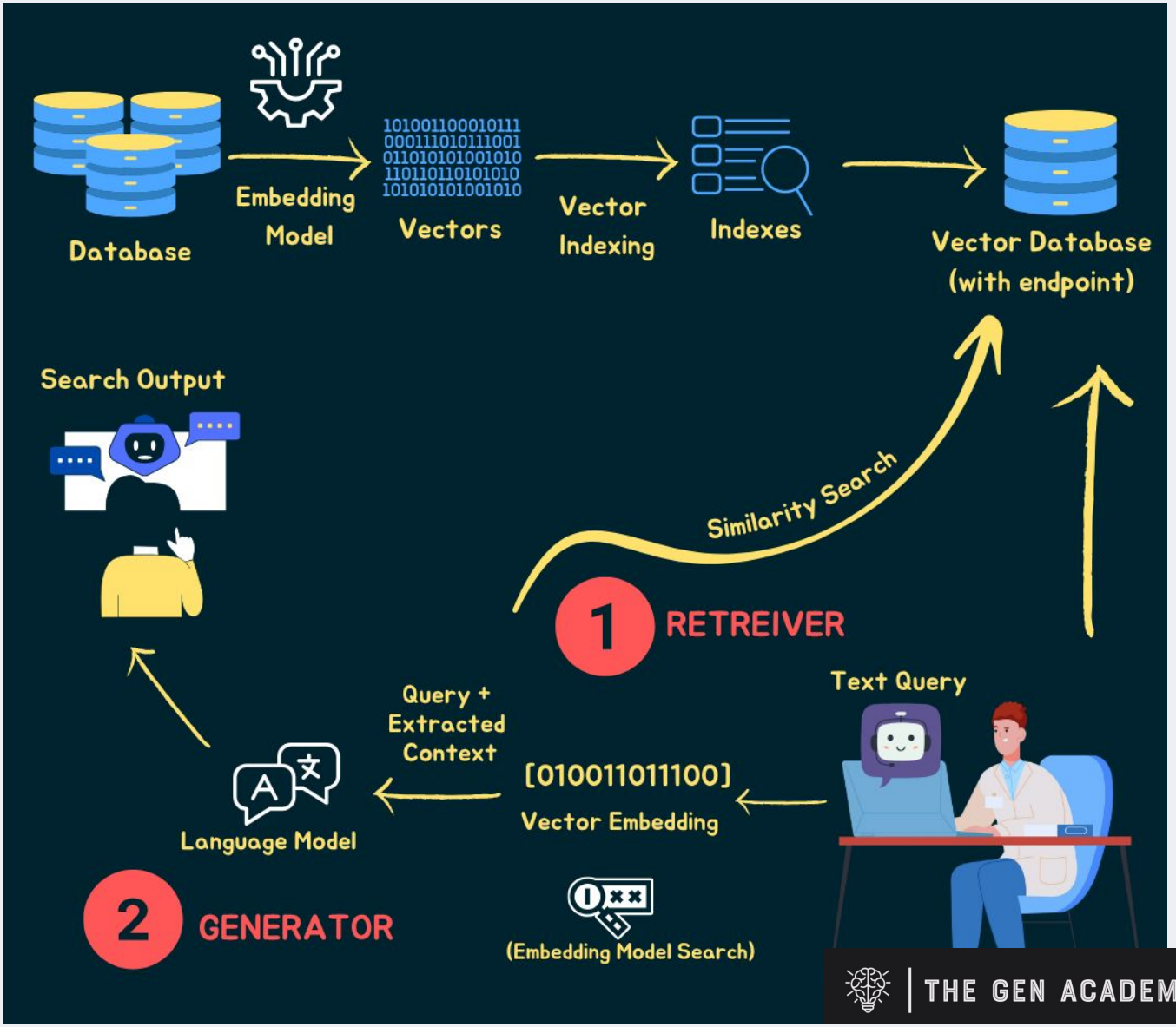
Arvind Narayanamurthy

- Founder, Principal AI Consultant @ Eikos
- Master's degree from Carnegie Mellon University
- Ex-Adobe, Ex-Microsoft, Ex-IBM - Data Scientist
- Expert in AI-driven workflow automation

Course Agenda

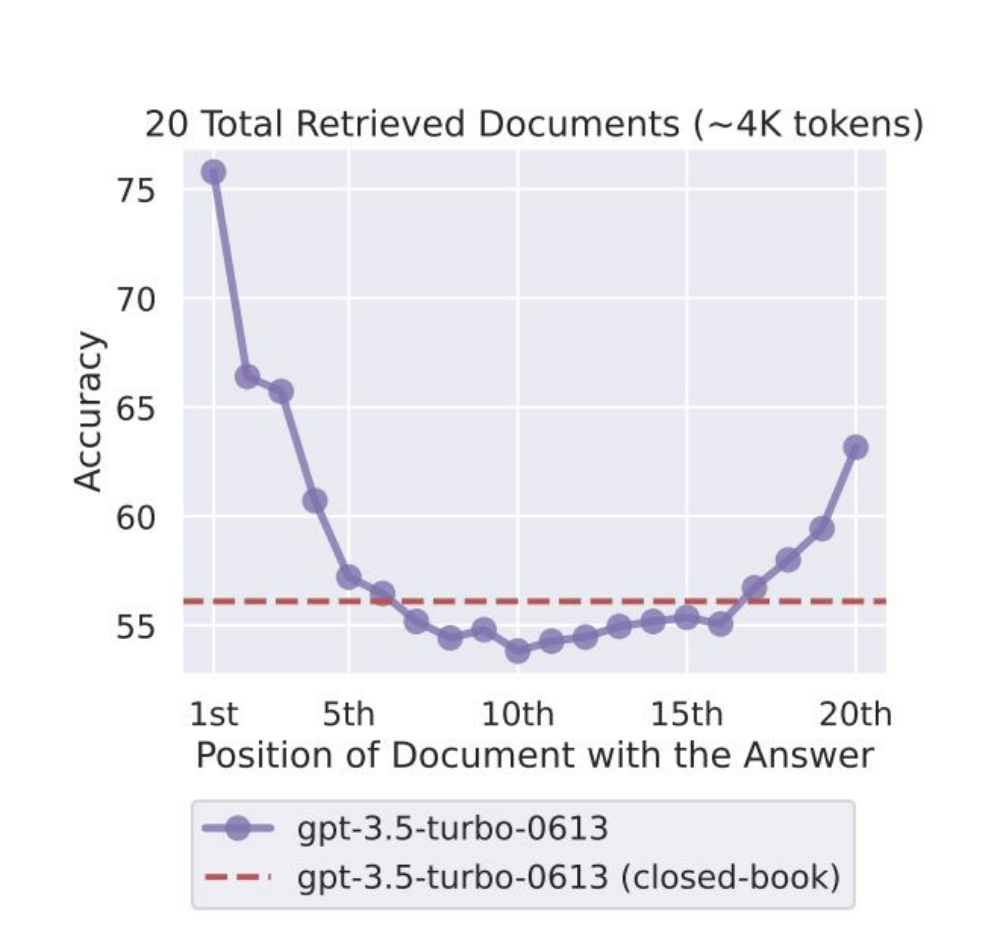
1. What is RAG & Why It Matters
2. Core Components of a RAG System
3. Chunking & Indexing Strategies
4. Retrieval Methods
5. Reranking for Precision
6. Building the Full RAG Pipeline
7. Metadata Filtering & Scaling Tips
8. Hands-on Demo
9. RAG FAQs & Q&A

What is RAG?


Database
Embedding Model
101001100010111
000111010111001
011010101001010
110110110101010
101010101001010
Vectors
Vector Indexing
Indexes
Vector Database (with endpoint)
Search Output
Similarity Search
1 RETREIVER
Text Query
Query + Extracted Context
[0100110111100]
Vector Embedding
Language Model
2 GENERATOR
(Embedding Model Search)
THE GEN ACADEMY

Why RAG?

1. Knowledge cut-off in LLMs
2. Domain Specific scenarios
3. Long Context ≠ Better Retreival



Core Components of RAG

0. Ingestion

Loading authoritative data sources into a vector store for efficient access and retrieval

1. Retrieval

Finding the most relevant chunks of information that match a user's query

2. Augmentation

Injecting the retrieved knowledge into the model's prompt to provide context

3. Generation

Producing a response that incorporates both the model's parameters and the retrieved information

These components work together to create a system that leverages both the reasoning capabilities of language models and the factual accuracy of curated knowledge sources.

Indexing: Preparing Data for Efficient Retrieval

Indexing is the foundational process of transforming raw data into searchable units that can be efficiently retrieved when needed.

This critical step involves:

- Converting documents into searchable **chunks**
- Enriching chunks with **metadata** for filtering
- Creating **vector embeddings** to capture semantic meaning
- **Organizing** data for fast search and retrieval

Effective indexing is essential when scaling RAG systems to handle massive knowledge bases like Wikipedia or enterprise document collections.

Overview of Chunking in RAG

What is Chunking?

Chunking is the process of dividing large documents into smaller, manageable pieces of text that can be independently retrieved and processed.

Optimal Chunk Sizes

- Typically 100-300 words per chunk
- 500-1000 tokens for embedding models
- Varies based on model context window

Effective chunking balances granularity with contextual completeness to ensure relevant information is retrieved.

Why Chunking Matters

Improved Retrieval Precision

Smaller chunks allow more precise matching to specific queries, reducing noise and irrelevant information in responses.

Computational Efficiency

Processing smaller chunks **reduces memory requirements** and allows for parallel processing, improving system performance.

Context Window Optimization

Smaller chunks ensure more **relevant information** can fit within the **limited context window** of language models.

Easier Maintenance

When source documents change, **only affected chunks need to be updated** rather than reprocessing entire documents.

The quality of chunking directly impacts both the accuracy and efficiency of RAG systems, making it a critical design decision.

Chunking Strategies: Key Methods

Fixed-size Chunking

Splits text into chunks of predetermined token count, regardless of content boundaries.

Pros: Simple, fast implementation

Cons: May break semantic units

Semantic Chunking

Divides content based on topic or semantic similarity, creating contextually coherent units.

Pros: Maintains topical relevance

Cons: Computationally expensive

Sentence-based Chunking

Creates chunks based on natural sentence boundaries, maintaining linguistic coherence.

Pros: Preserves grammatical structure

Cons: Variable chunk sizes

Semantic Double-Pass Merging

First divides text into small chunks, then merges related chunks based on semantic similarity.

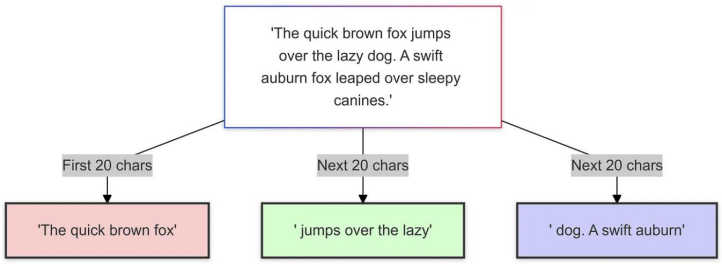
Pros: Optimizes for both granularity and context

Cons: Complex implementation

....and more

Fixed-Length Chunking

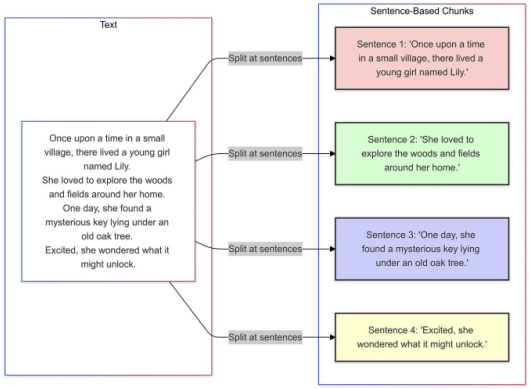
How it works: Divides text into chunks of a predefined length, typically based on tokens or characters.



Fixed-Length Chunking — Visualization

Sentence-Based Chunking

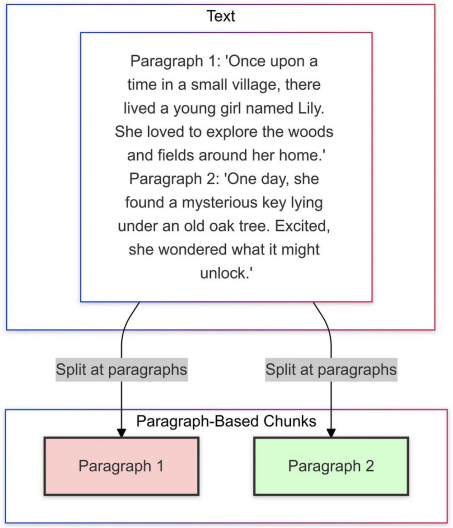
How it works: Splits text at sentence boundaries, ensuring each chunk is a complete thought.



Sentence-Based Chunking — Visualization

Paragraph-Based Chunking

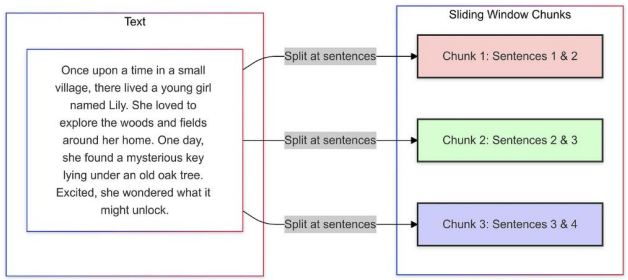
How it works: Splits documents into paragraphs, which often encapsulate a complete idea or topic.



Paragraph-Based Chunking — Visualization

Sliding Window Chunking

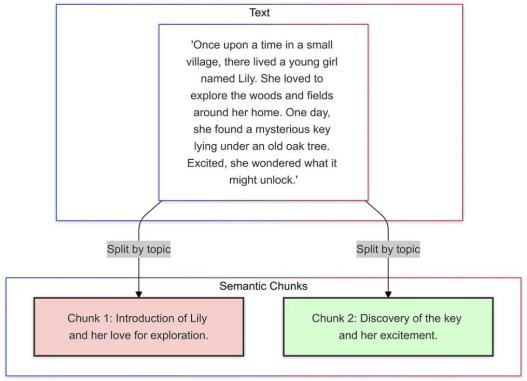
How it works: Creates overlapping chunks by sliding a window over the text, ensuring adjacent chunks share content.



Sliding Window Chunking — Visualization

Semantic Chunking

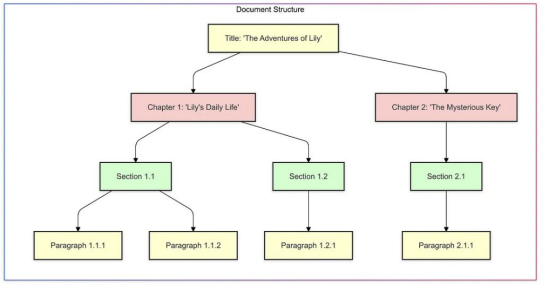
How it works: Utilizes embeddings or machine learning models to split text based on semantic meaning, ensuring each chunk is cohesive in topic or idea.



Semantic Chunking — Visualization

Recursive Chunking

How it works: Breaks down text progressively into smaller chunks using hierarchical delimiters like headings, subheadings, paragraphs, and sentences.



Recursive Chunking — Visualization

https://masteringllm.medium.com/11-chunking-strategies-for-rag-simplified-visualized-df0dbec8e373

Chunking Strategies: Comparison

1 Fixed-length Chunking

Speed: Very Fast

Accuracy: Low to Medium

Best for: Large-scale initial processing

2 Sentence-based Chunking

Speed: Fast

Accuracy: Medium

Best for: General-purpose applications

3 Semantic Chunking

Speed: Slow

Accuracy: Medium to High

Best for: Precise technical content

4 Semantic Double-Pass

Speed: Medium

Accuracy: Very High

Best for: Critical applications requiring precision

Retrieval Strategies in RAG

Dense Vector Search

Uses embedding similarity to find semantically related content, even when vocabulary differs

Strength: Semantic understanding

Sparse Search

Leverages keyword matching through BM25 or TF-IDF algorithms to find exact terminology

Strength: Precise term matching

Metadata Filtering

Narrows results using structured attributes like date, source, or topic

Strength: Contextual relevance

Hybrid Retrieval

Combines dense and sparse approaches to leverage strengths of both methods

Strength: Best overall performance

Each retrieval strategy offers different tradeoffs between accuracy, speed, and resource requirements. The best approach often depends on specific use case requirements.

Embedding Models for Retrieval

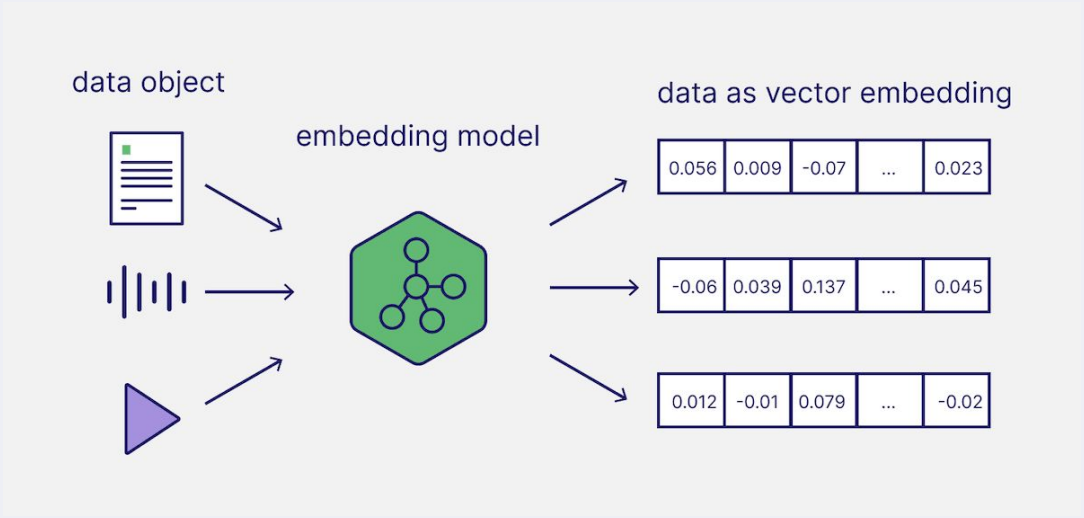
What are Embedding Models?

These specialized neural networks transform text into high-dimensional vectors that capture semantic meaning, enabling similarity comparisons.

Popular Embedding Models

BERT & RoBERTa: Contextual embeddings with strong performance
Sentence Transformers: Optimized for sentence-level similarity
OpenAI Embeddings: High-quality semantic matching

E5 & GTR: State-of-the-art retrieval performance



The quality of embedding models directly impacts retrieval performance, with more advanced models capturing nuanced semantic relationships.

Embedding Models for Retrieval

Understanding Vision Embeddings

- Vision embeddings are c**ompact numerical representations** that encode the semantic content of images into high-dimensional vector
- Unlike raw pixel data that only contains color and intensity information, vision embeddings **capture abstract visual concepts** like object shapes, spatial relationships, and contextual meaning
- When an image embedding is generated, it can be stored **alongside text embeddings** in the same vector space, allowing for **cross-modal retrieval** where users can search for relevant images using text queries or find related text using image queries

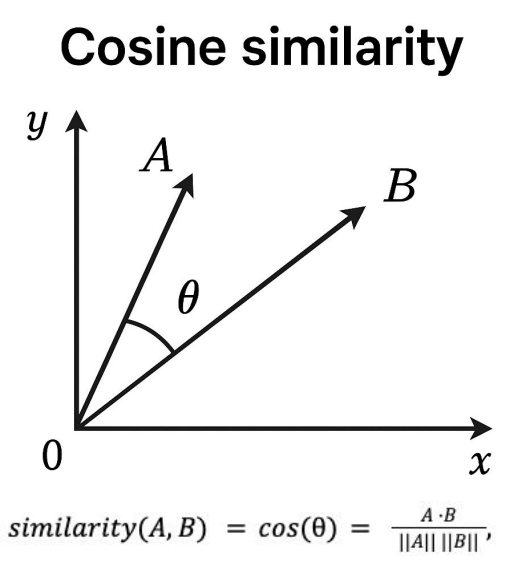
Vision Embedding Models

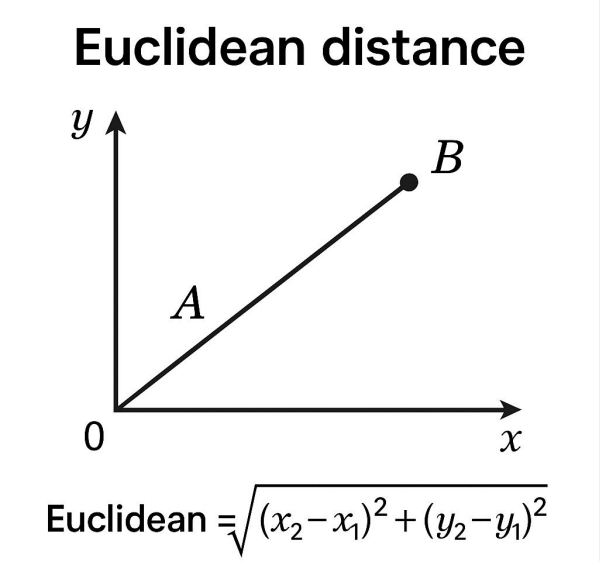
- **CLIP:** Contrastive Language-Image Pre-training

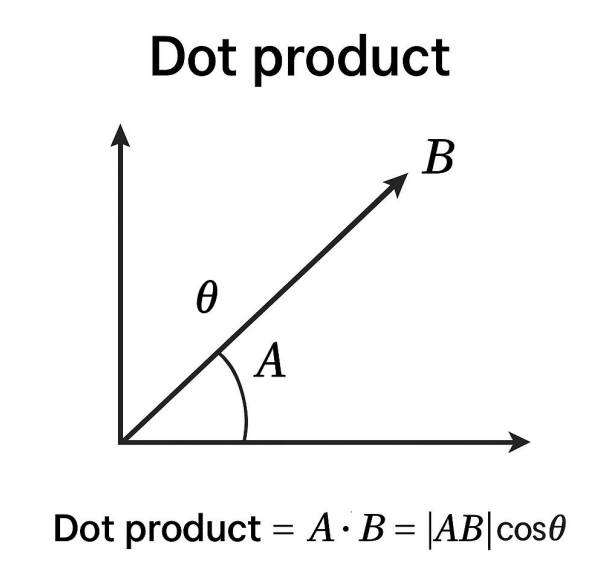
  ViT: Vision Transformer

  BLIP: Bootstrapping Language-Image Pre-training >> BLIP-2
- **ALIGN:** A Large-scale ImaGe and Noisy-text embedding
- **Amazon Titan Multimodal Embeddings**

Similarity Measures in Retrieval







Cosine Similarity

Measures the cosine of the angle between vectors, focusing on direction rather than magnitude. Values range from -1 (opposite) to 1 (identical).
Best for: General semantic similarity

Euclidean Distance

Calculates the straight-line distance between vectors in multi-dimensional space. Lower values indicate greater similarity.
Best for: When magnitude matters

Dot Product

Multiplies corresponding elements and sums them. For normalized vectors, equivalent to cosine similarity.
Best for: Optimized computation

The choice of similarity measure affects both retrieval accuracy and computational performance of RAG systems.

Sparse Retrieval: TF-IDF

Term Frequency–Inverse Document Frequency

TF-IDF is a statistical measure that evaluates how important a word is to a document in a collection by combining:

Term Frequency (TF): How often a term appears in a document
Inverse Document Frequency (IDF): How rare the term is across all documents

Words that appear frequently in a specific document but rarely in others receive higher TF-IDF scores, making them good discriminators.

Sparse Retrieval: BM25

What is BM25?

BM25 (Best Matching 25) is a probabilistic ranking function used to estimate the relevance of documents to a given search query based on term frequency and document length. It's an improved version of TF-IDF that incorporates document length normalization and diminishing returns for term frequency.

Key Advantages

- Handles term saturation (diminishing returns for repeated terms)
- Accounts for document length bias
- Proven effectiveness in information retrieval
- Still widely used in search engines

Formula Components

BM25 score is calculated using:

- Term frequency in document
- Inverse document frequency
- Document length normalization
- Tunable parameters k1 and b

Dense Retrieval: Vector Search

How Dense Retrieval Works

Dense retrieval uses neural networks to map text to dense vector representations, capturing semantic meaning rather than just keywords:

1. Documents are embedded into vector space during indexing
2. Queries are embedded using the same model
3. Nearest neighbor search finds similar vectors
4. Results are ranked by vector similarity

Vector Database Solutions

Pinecone: Managed vector DB service

FAISS: Facebook's efficient similarity search

Milvus: Open-source vector database

Weaviate: Vector search with GraphQL

Hybrid Search: Dense + Sparse Fusion

Score Fusion

Combines relevance scores from both dense and sparse retrievers using weighted averaging or more sophisticated methods

Rank Fusion

Merges ranked lists from different retrievers using algorithms like Reciprocal Rank Fusion (RRF)

Ensemble Methods

Uses machine learning to predict the most effective retrieval method for each query type

Dynamic Weighting

Adjusts the influence of each retrieval method based on query characteristics

Hybrid search leverages the strengths of both approaches- keyword precision from sparse methods and semantic understanding from dense methods- to achieve superior retrieval performance.

Metadata Filtering

Enhancing Retrieval with Structured Attributes

Metadata filtering allows RAG systems to narrow retrieval results based on document attributes rather than just content, ensuring more contextually appropriate responses.

Common Metadata Filters

Temporal: Creation date, last updated, expiration

Source: Author, department, authority level

Content Type: FAQ, policy, technical doc

Domain: Product line, geographic region

Status: Draft, published, archived

Implementation Approaches

Metadata filtering can be applied:

- Pre-retrieval to limit search space
- During retrieval as weighted conditions
- Post-retrieval to refine candidates

This capability is especially valuable in enterprise settings where information governance and regulatory compliance are concerns.

Reranking in RAG Pipelines

What is Reranking?

Reranking is a second-pass process that takes the top-k results from initial retrieval and reorders them using a more sophisticated (and typically more computationally expensive) model.

Reranking Models

Cross-Encoders: Process query and document pairs together for precise relevance scoring
Bi-Encoders: Generate separate embeddings but with deeper semantic understanding
LLM-based Rerankers: Use language models to judge relevance

Performance Impact

Reranking typically improves retrieval precision by:

- 10-30% increase in retrieval accuracy
- Better handling of nuanced queries
- Improved response quality in RAG systems

This comes at the cost of increased latency and computational resources.

RAG Pipeline End-to-End

1. Query Processing

User submits a question which is analyzed, preprocessed, and embedded into a vector representation

2. Initial Retrieval

System retrieves top-k relevant chunks from the knowledge base using vector similarity and/or keyword matching

3. Reranking

Retrieved chunks are reordered using more sophisticated models to prioritize the most relevant information

4. Context Augmentation

Top results are formatted and injected into the prompt template to provide the LLM with relevant context

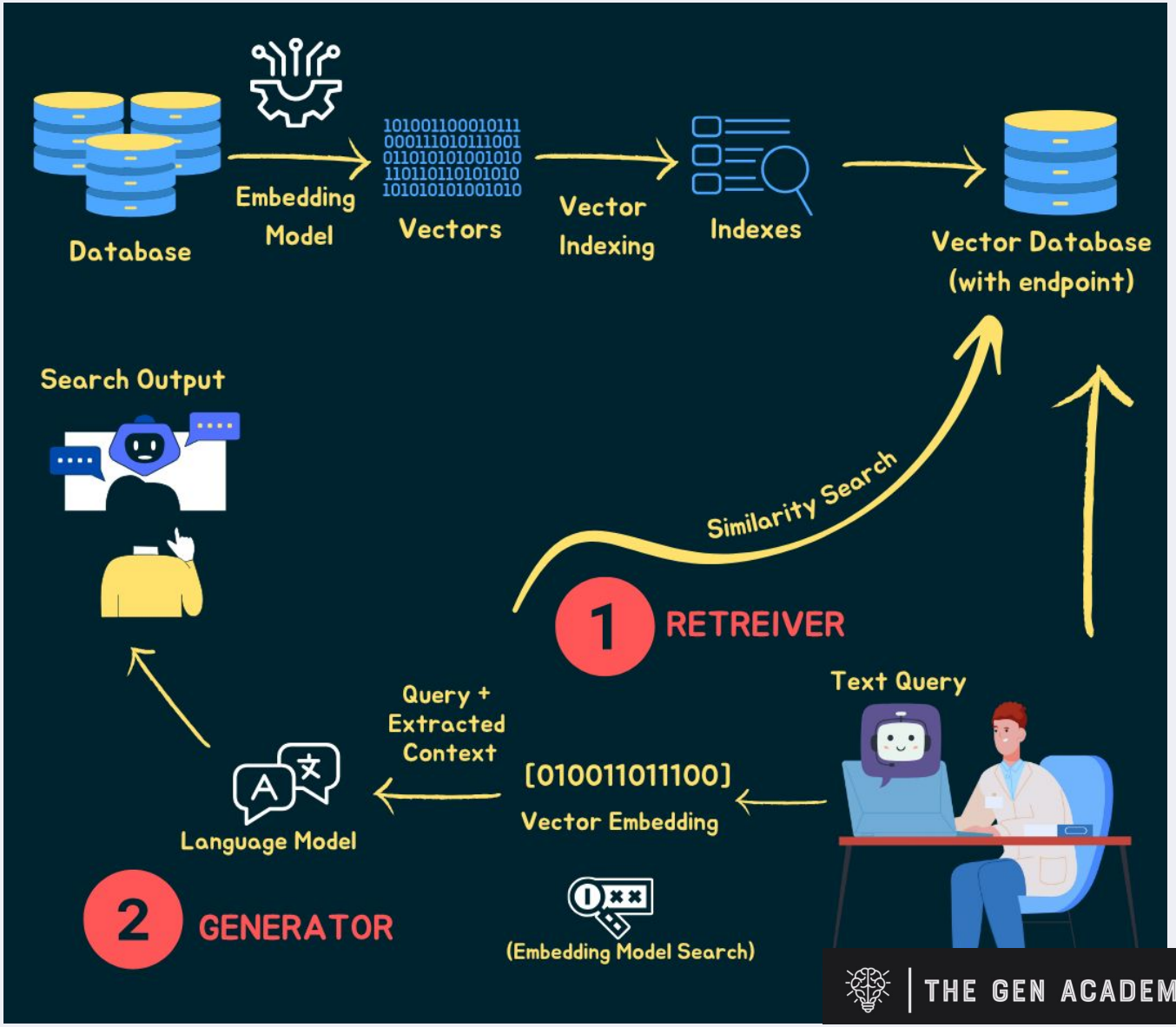
5. Response Generation

LLM generates a comprehensive answer based on the query and the retrieved context

6. Post-processing

System adds citations, confidence scores, or additional metadata to enhance explainability and trustworthiness

How RAG works?



Questions?

(Before we get into the demo)

Demo Time!

Practical Example: Building a RAG System

Step 1: Data Preparation

1

Collect and clean documents from knowledge sources (e.g., product manuals, support documentation)

Tools: BeautifulSoup, Unstructured.io, PyPDF

Step 2: Chunking & Metadata Extraction

2

Split documents into semantic chunks and extract metadata like source, date, and category

Tools: LangChain, LlamaIndex, custom NLP pipelines

Step 3: Embedding Generation

3

Create vector embeddings for each chunk using a suitable embedding model

Tools: Sentence-Transformers, OpenAI Ada, HuggingFace models

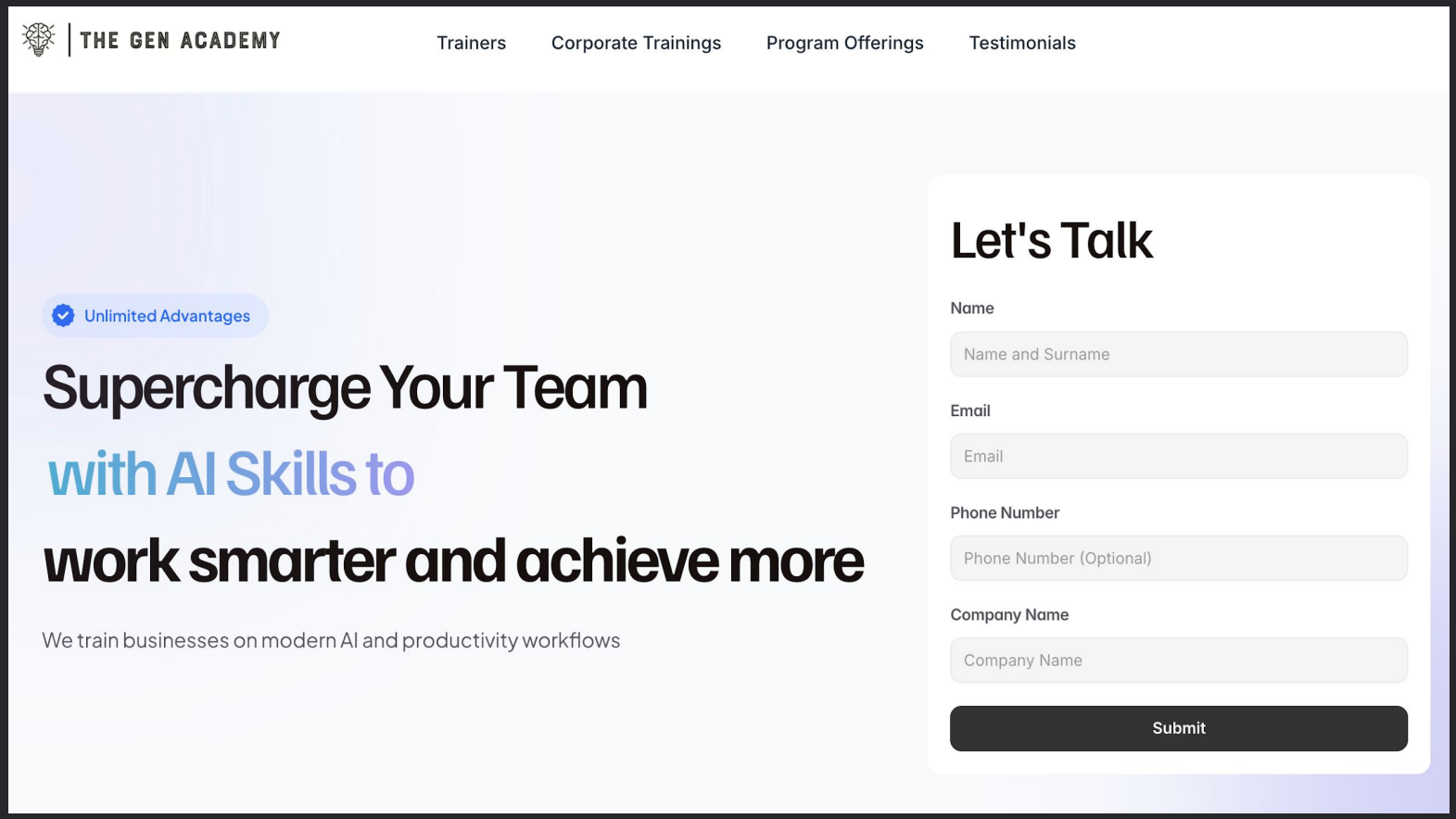
Step 4: Vector Database Setup

4

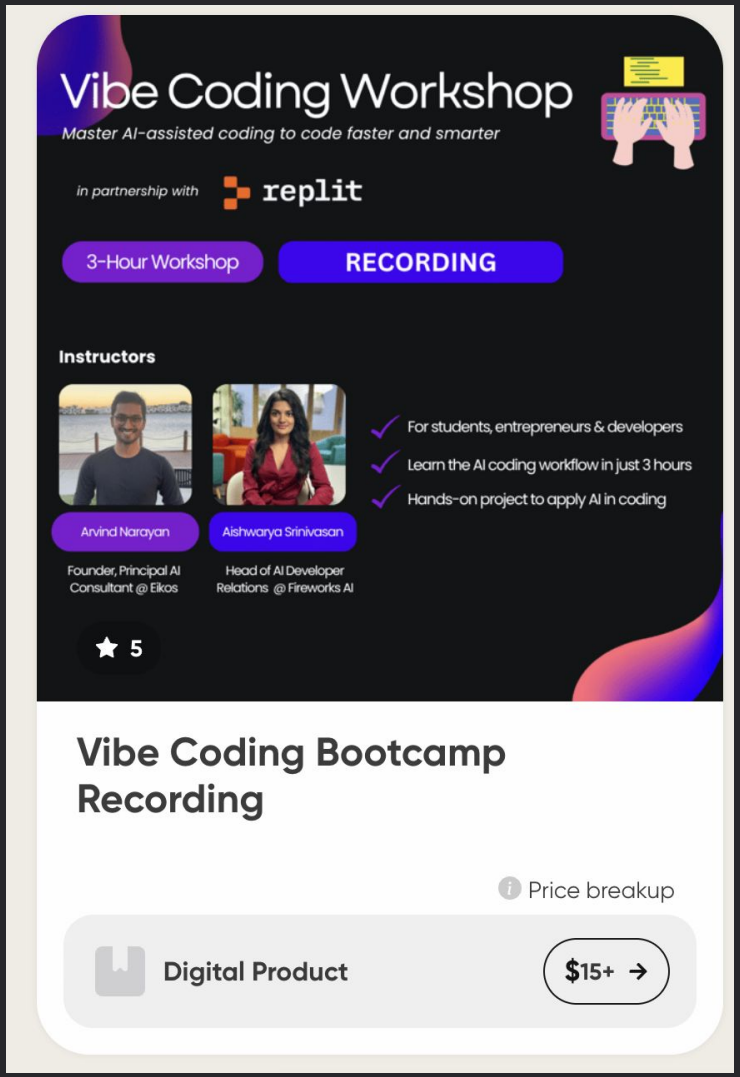
Resources

Recommended Reading:
https://8738733.fs1.hubspotusercontent-na1.net/hubfs/8738733/eBooks/Weaviate%20Agentic%20Architectures-ebook.pdf
https://weaviate.io/blog/introduction-to-rag
The 5 Levels Of Text Splitting For Retrieval
https://arxiv.org/pdf/2307.03172
https://developers.google.com/machine-learning/clustering/dnn-clustering/supervised-similarity

We are working on an initiative to help teams upskill on AI tools and concepts

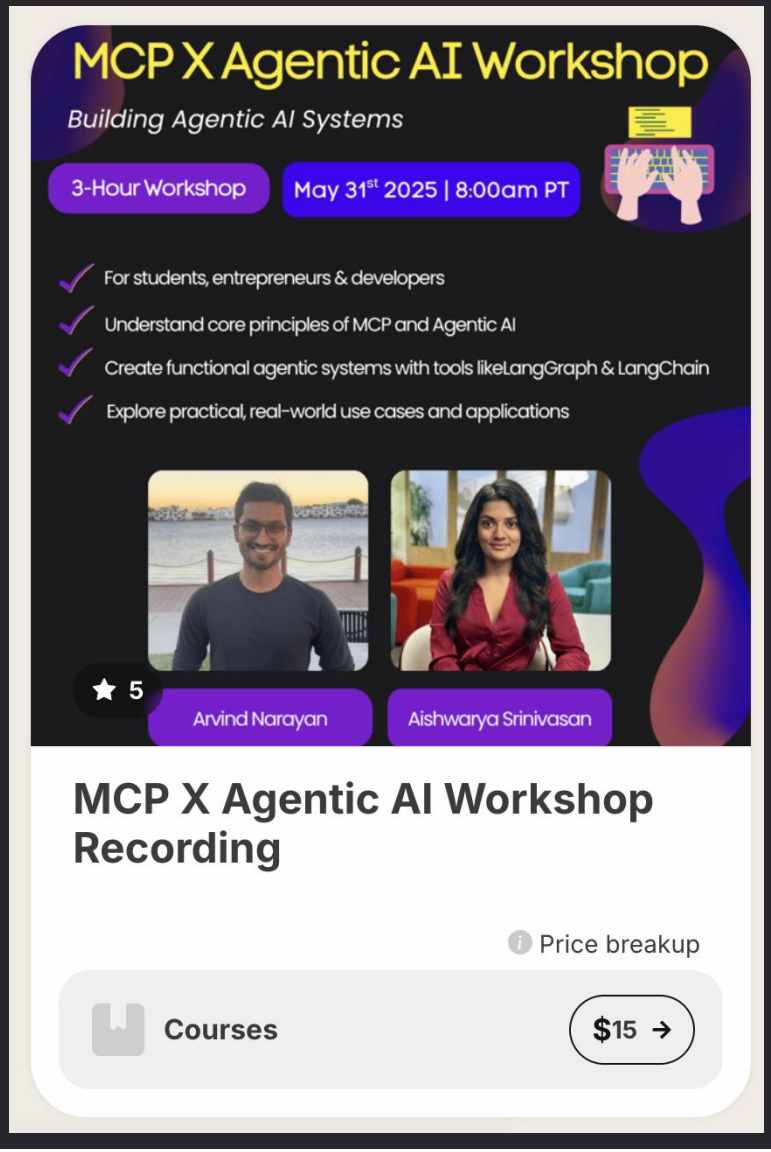


Check out more details at thegenacademy.com



Our last Vibe Coding Workshop Recording is now available

<< Click on it for link



Our last MCP X Agentic AI Workshop Recording is now available

<< Click on it for link

We would love your Feedback!

https://forms.gle/7EBrwoXM6awgE5DNA