COMPREHENSIVE CURRICULUM

Multimodal RAG

Retrieval-Augmented Generation Across Text, Images, Audio & Video

12 Modules	150+ Hours	Mid-Advanced	Industry-Ready

Designed for AI/ML Engineers, Data Scientists & NLP Practitioners

Topics: RAG Architecture | Vector Databases | Multimodal Embeddings | LoRA Finetuning | RAGOps | Evaluation

Table of Contents

Course Overview

Multimodal Retrieval-Augmented Generation (Multimodal RAG) is one of the fastest-growing competencies demanded by top AI employers. This curriculum is engineered for practitioners who already understand fundamentals of machine learning and computer vision, and are ready to build production-grade, knowledge-grounded AI systems that operate across text, images, audio, and video.

Who This Course Is For

Target Audience

- ML Engineers building production retrieval-augmented systems
- Data Scientists extending NLP pipelines to multimodal data
- NLP researchers moving from text-only to vision-language models
- Software Engineers integrating LLMs into enterprise applications
- AI candidates targeting roles with 'RAG', 'LLM', or 'multimodal' in the job description

Prerequisites

Prerequisite	Proficiency Level	Key Concepts Required
Python Programming	Proficient	OOP, async, decorators, type hints
Machine Learning Fundamentals	Intermediate	Supervised learning, loss functions, optimization
Deep Learning	Intermediate	Neural networks, backprop, PyTorch/TensorFlow
Computer Vision	Intermediate	CNNs, ViT, object detection, image embeddings
NLP Basics	Intermediate	Tokenization, transformers, attention mechanisms
Software Engineering	Intermediate	APIs, Docker, Git, basic cloud usage
Linear Algebra & Statistics	Working Knowledge	Vectors, dot products, probability distributions

Course Structure at a Glance

Module	Title	Duration	Level
1	Foundations of RAG	1 week / 10 hrs	Intermediate

2	Data Ingestion & Knowledge Sources	1 week / 12 hrs	Intermediate
3	Text Processing & Chunking	1 week / 10 hrs	Intermediate
4	Embeddings & Vectorization	1.5 weeks / 14 hrs	Intermediate-Advanced
5	Vector Databases & Indexing	1 week / 12 hrs	Intermediate-Advanced
6	Retrieval Strategies	1.5 weeks / 14 hrs	Advanced
7	Multimodal RAG: Core Integration	2 weeks / 20 hrs	Advanced
8	Generation, Prompting & Evaluation	1.5 weeks / 14 hrs	Advanced
9	Advanced RAG Techniques	2 weeks / 18 hrs	Advanced
10	Finetuning & LoRA for Multimodal RAG	2 weeks / 20 hrs	Advanced
11	Production, Ops & Security	1.5 weeks / 12 hrs	Advanced
12	Capstone Projects & Career Prep	2 weeks / 16 hrs	Integrated

Total estimated duration: 16–20 weeks (self-paced) or 12 weeks (intensive). Each module includes video/reading content, hands-on labs, and an assessment component.

MODULE 1

Foundations of RAG

1 Week / ~10 Hours Intermediate

Topics covered:

- What is Retrieval-Augmented Generation and why it matters in production AI
- Retriever-Generator architecture and its components
- Closed-book vs. Open-book LLMs: parametric vs. non-parametric memory
- Knowledge-grounded generation and hallucination mitigation
- Open-domain QA framing: from question to grounded answer
- Context-augmented prompting and external knowledge integration
- RAG vs. fine-tuning: when to use which approach
- Overview of the RAG ecosystem: LangChain, LlamaIndex, Haystack

Module 1: Foundations of RAG

1.1 The Case for Retrieval-Augmented Generation

Large language models encode knowledge in their weights at training time — but the world changes. Retrieval-Augmented Generation (RAG) addresses this fundamental limitation by coupling a parametric model (the LLM) with a non-parametric memory store (the knowledge base). At inference time, relevant documents are retrieved and injected into the context, enabling the model to produce knowledge-grounded responses without expensive retraining.

Core Concepts in this Section

- Retrieval-Augmented Generation (RAG): the paradigm of retrieval + generation
- Retriever-Generator architecture: the two-stage pipeline
- Parametric memory: knowledge baked into model weights during pretraining
- Non-parametric memory: external documents, databases, vector stores
- Knowledge injection: how retrieved content enters the generation process
- Grounded responses: answers tied explicitly to source documents
- Hallucination mitigation: why grounding reduces confabulation
- Open-domain QA: the task setting that originally motivated RAG
- Closed-book vs. Open-book LLMs: generation with and without retrieval
- Context-augmented prompting: the mechanics of injecting retrieved text
- External knowledge integration: APIs, databases, knowledge graphs as sources

1.2 RAG vs. Alternatives

Approach	Pros	Cons	Best For
Closed-book LLM	Simple, fast, no external deps	Stale, hallucinates, no citations	General conversation

Fine-tuning only	Deeply adapted to domain	Expensive, stale after training cutoff	Style/format adaptation
RAG (standard)	Fresh knowledge, citable, cheaper to update	Retrieval latency, chunking errors	Knowledge-intensive QA
RAG + Fine-tune	Best of both worlds	Most complex to build/maintain	High-stakes production systems

1.3 RAG System Architecture: End-to-End

A complete RAG system involves five logical stages: (1) Ingestion — documents are loaded, chunked, and embedded; (2) Indexing — embeddings are stored in a vector database; (3) Retrieval — at query time, the user query is embedded and similar chunks are fetched; (4) Prompt Construction — retrieved chunks are assembled into a context-augmented prompt; (5) Generation — the LLM produces a grounded response.

1.4 Hands-On Lab 1: Your First RAG Pipeline

Lab 1.1 — Build a Basic RAG System

Goal: Build a minimal but functional RAG system in 100 lines of Python.
Stack: LangChain + OpenAI Embeddings + FAISS + GPT-4o
Steps:

1. Load a set of Wikipedia articles using LangChain's WikipediaLoader
2. Chunk documents with RecursiveCharacterTextSplitter (chunk_size=500)
3. Embed chunks with text-embedding-ada-002 and index in FAISS
4. Retrieve top-5 chunks for a user query and assemble a prompt
5. Generate and display a grounded, cited response

Deliverable: A CLI chatbot that cites its sources.

1.5 Key Papers & Resources

- Lewis et al. (2020) — Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks (original RAG paper)
- Guu et al. (2020) — REALM: Retrieval-Enhanced Language Model Pre-Training
- Tool: LangChain documentation — https://docs.langchain.com
- Tool: LlamaIndex documentation — https://docs.llamaindex.ai
- Tutorial: LangChain RAG quickstart guide

1.6 Module 1 Assessment

Quiz 1

Q1: What is the key difference between parametric and non-parametric memory in LLMs?

Q2: Describe the five stages of a RAG pipeline.
Q3: When would you choose fine-tuning over RAG? Give two concrete scenarios.
Q4: What is hallucination, and how does retrieval grounding reduce it?
Q5: What is context-augmented prompting and how does it differ from standard prompting?

MODULE 2

Data Ingestion & Knowledge Sources

1 Week / ~12 Hours Intermediate

Topics covered:

- Understanding structured, semi-structured, and unstructured data types
- Document loaders for PDFs, HTML, Markdown, CSV, JSON, code, and more
- Web scraping and crawling strategies for knowledge base construction
- ETL pipelines for large-scale document ingestion
- Incremental indexing and change data capture (CDC)
- Deduplication, canonicalization, and metadata extraction
- Multimodal ingestion: loading images, audio files, video frames
- Handling tables, emails, chat transcripts, and wikis

Module 2: Data Ingestion & Knowledge Sources

2.1 Data Types for RAG Knowledge Bases

RAG systems are only as good as the knowledge they can access. Building a comprehensive, well-maintained knowledge base requires ingesting a wide variety of data types. Each type presents unique parsing, cleaning, and chunking challenges.

Data Type	Category	Common Formats	Key Challenges
Prose text	Unstructured	TXT, DOCX, PDF	Layout reconstruction, font issues
Web pages	Semi-structured	HTML, XML	Boilerplate removal, rendering JS
Structured records	Structured	CSV, JSON, SQL	Relational context, type coercion
Scientific papers	Semi-structured	PDF, LaTeX	Figures, equations, citations
Source code	Structured-ish	.py, .js, .ts	Semantic segmentation by function
Tables	Structured	HTML table, CSV	Header inference, multi-row spans
Wikis / Knowledge Bases	Semi-structured	Wiki markup, Markdown	Hyperlinks, templates
Emails / Chat	Unstructured	EML, JSON export	Threading, sender metadata
Logs	Structured	Syslog, JSONL	Parsing timestamps, error codes
Images	Unstructured	PNG, JPEG, TIFF	OCR, captioning for indexing
Audio	Unstructured	MP3, WAV, M4A	Transcription (Whisper), diarization
Video	Multimodal	MP4, MOV	Frame sampling, audio+visual alignment

2.2 Document Loaders

Document loaders are responsible for reading raw content and converting it into a uniform representation (typically a list of Document objects with page_content and metadata). LangChain and LlamaIndex both provide rich loader ecosystems.

- PyPDFLoader / PDFMinerLoader — layout-aware PDF extraction
- UnstructuredLoader — universal loader for 20+ file types via the Unstructured library
- WebBaseLoader / BeautifulSoup — HTML ingestion with noise removal
- GitLoader — code repository ingestion with file-path metadata
- CSVLoader / JSONLoader — structured data ingestion with column mapping
- WikipediaLoader / ArxivLoader — curated knowledge source loaders
- DirectoryLoader — batch loading entire folder hierarchies
- ImageCaptionLoader — uses vision models to generate text descriptions of images
- YoutubeLoader — transcribes YouTube videos using Whisper or YouTube captions

2.3 Web Scraping & Crawling

- Scrapy — production-grade crawling with pipelines and middleware
- Playwright/Selenium — JavaScript-rendered page scraping
- sitemap.xml parsing — efficient discovery of pages to crawl
- robots.txt compliance — ethical and legal scraping practices
- Rate limiting and politeness policies
- Incremental crawling — only re-fetch changed pages (ETags, Last-Modified headers)

2.4 ETL Pipelines for RAG

Production knowledge bases are not static. You need ETL (Extract, Transform, Load) pipelines that run continuously, handle failures gracefully, and support incremental updates.

- Change Data Capture (CDC): detect new/modified/deleted source documents
- Incremental indexing: only re-embed and re-index changed content
- Deduplication: exact-match (hash-based) and near-duplicate (MinHash/SimHash) detection
- Canonicalization: normalizing URLs, document IDs, entity names
- Metadata extraction: author, date, source URL, version, access level
- Airflow / Prefect / Dagster for pipeline orchestration

2.5 Multimodal Ingestion Pipeline

Multimodal Ingestion: Key Steps

Images: Load -> OCR (Tesseract/Google Vision) -> Caption (BLIP-2/LLaVA) -> Embed (CLIP) -> Store

Audio: Load -> Transcribe (Whisper) -> Diarize (optional) -> Embed text transcript -> Store audio embedding separately

Video: Load -> Sample frames (1fps or scene-change) -> Caption frames -> Transcribe audio -> Align timestamps -> Create multimodal chunks

Tables: Load -> Parse to structured representation -> Convert to markdown/natural language -> Embed

2.6 Hands-On Lab 2: Building a Multimodal Ingestion Pipeline

Lab 2.1 — Multimodal Document Ingestion

Goal: Build an ingestion pipeline that handles PDFs (with embedded images), web pages, and a YouTube video.

Stack: LlamaIndex + Unstructured + Whisper + BLIP-2 + PostgreSQL for metadata

Tasks:

6. Ingest a 50-page technical PDF with figures — extract text and caption images separately
7. Crawl 20 web pages, remove boilerplate, and deduplicate overlapping content
8. Transcribe a YouTube video and align transcript segments to timestamps
9. Store all artifacts with rich metadata in a PostgreSQL metadata store

Deliverable: A reusable IngestionPipeline class with pluggable loaders.

MODULE 3

Text Processing & Chunking

1 Week / ~10 Hours Intermediate

Topics covered:

- Document chunking strategies: fixed-size, semantic, recursive, hierarchical
- Chunk size, chunk overlap, and their impact on retrieval quality
- Token-aware chunking for LLM context windows
- Sentence-level and paragraph-level boundary preservation
- Sliding window chunking and parent-child hierarchical chunking
- Semantic chunking with embedding-based boundary detection
- Special chunking: code, tables, structured data
- Context window optimization and max token limit management

Module 3: Text Processing & Chunking

3.1 Why Chunking Matters

Vector databases store and retrieve fixed-size vector embeddings. Since LLMs have finite context windows and embedding models perform best on semantically coherent inputs, raw documents must be split into chunks before embedding. Poor chunking is one of the most common causes of poor retrieval quality.

3.2 Chunking Strategies: Taxonomy

Strategy	Method	Best For	Tradeoffs
Fixed-size	Split every N characters	Simple, fast baseline	Cuts mid-sentence; poor coherence
Recursive	Split by \n\n, \n, . , ' ' in order	General-purpose text	Better but not semantic
Sentence-level	Split on sentence boundaries (spaCy/NLTK)	Factual QA	May produce very short chunks
Paragraph-level	Split on double newline	Long-form prose	Variable chunk sizes
Token-aware	Count tokens via tiktoken; split at token boundary	LLM context management	Requires tokenizer per model
Semantic	Embed sentences; split where similarity drops	Best coherence	Slow; requires embeddings upfront
Hierarchical / Parent-Child	Store small chunks; index by parent summary	Multi-granularity retrieval	More complex index management

Sliding window	Overlap chunks (e.g. 20%) to preserve boundary context	Dense retrieval	Increases chunk count

3.3 Key Chunking Parameters

- chunk_size: Target chunk size in characters or tokens (common: 256–1024 tokens)
- chunk_overlap: Number of overlapping tokens/chars between consecutive chunks (common: 10–20%)
- Boundary preservation: Never cut mid-sentence, mid-code-block, or mid-table
- Context window optimization: Ensure chunk_size + prompt overhead fits in the LLM's context window
- Max token limits: Account for prompt template tokens + retrieved chunks + generated answer

3.4 Semantic Chunking Deep Dive

Semantic chunking uses embedding similarity to detect topic boundaries. The algorithm embeds consecutive sentences, computes pairwise cosine similarities, and identifies breakpoints where similarity falls below a threshold. This produces semantically coherent chunks at the cost of pre-embedding every sentence.

- Implementation: LlamaIndex SemanticSplitterNodeParser, LangChain SemanticChunker
- Threshold tuning: percentile-based breakpoints vs. fixed threshold
- Embedding model choice for chunking vs. retrieval: can differ (smaller is fine for chunking)

3.5 Hierarchical Chunking & Parent-Child

Hierarchical chunking stores both a coarse and a fine representation: small chunks (e.g., 128 tokens) are embedded and retrieved for precision; their full parent chunk (e.g., 512 tokens) is passed to the LLM for richer context. This improves both retrieval precision and generation quality.

3.6 Special Cases: Code, Tables, Structured Data

- Code: Split on function/class boundaries using AST parsing (ast module in Python), not character count
- Tables: Convert to markdown or serialized natural language before chunking (avoids splitting header from rows)
- JSON/CSV: Chunk per record or per logical row group, preserving schema headers
- Markdown: Respect heading structure — chunk within ## sections

3.7 Hands-On Lab 3: Chunking Strategy Comparison

Lab 3.1 — Chunk Quality Evaluation

Goal: Empirically compare 4 chunking strategies on a test retrieval benchmark.
Dataset: A 100-page technical document (e.g., a machine learning textbook chapter PDF).
Evaluation: For 20 test questions, measure Recall@5 for each strategy.
Strategies to compare: Fixed-size (512 chars), Recursive, Sentence-level, Semantic
Bonus: Implement parent-child chunking and compare against flat strategies.
Deliverable: A benchmark report with Recall@5, MRR, and qualitative analysis.

MODULE 4

Embeddings & Vectorization

1.5 Weeks / ~14 Hours Intermediate–Advanced

Topics covered:

- Text embeddings: dense, sparse, hybrid; sentence vs. token-level
- Multilingual and domain-specific embedding models
- Vector concepts: cosine similarity, dot product, Euclidean distance, normalization
- High-dimensional spaces: curse of dimensionality, dimensionality reduction (PCA, UMAP)
- Image embeddings: CLIP, ViT, convolutional feature extractors
- Audio embeddings: Wav2Vec2, EnCodec, audio spectrograms
- Cross-modal embeddings: joint image-text embedding spaces
- Embedding drift detection and monitoring
- Choosing the right embedding model for your domain

Module 4: Embeddings & Vectorization

4.1 Text Embeddings: Dense, Sparse, Hybrid

Embeddings convert raw content into numerical vectors in a high-dimensional space, where semantic similarity corresponds to geometric proximity. Choosing the right embedding type is one of the most impactful decisions in a RAG system.

- Dense embeddings: Fixed-length real-valued vectors (e.g., 768-dim). Capture semantic meaning. Models: text-embedding-ada-002, sentence-transformers (all-MiniLM-L6-v2, all-mpnet-base-v2), E5, GTE, BGE
- Sparse embeddings: High-dimensional vectors where most values are zero (like BM25 or SPLADE). Strong for exact keyword matching and rare terms
- Hybrid embeddings: Combine dense and sparse scores at retrieval time. Superior to either alone for most real-world tasks
- Sentence embeddings (SBERT): Pooled representation of entire sentence/passage
- Token embeddings: Per-token vectors (useful for late-interaction models like ColBERT)
- Multilingual embeddings: LaBSE, multilingual-e5 — support 100+ languages in shared space
- Domain-specific embeddings: BioASQ (biomedical), LegalBERT (legal), FinBERT (finance) — fine-tuned on domain corpora

4.2 Vector Similarity Metrics

Metric	Formula	When to Use	Notes
Cosine Similarity	dot(a,b) / (|a| * |b|)	Most common for text/image embeddings	Invariant to magnitude; range [-1,1]
Dot Product	sum(a_i * b_i)	When embeddings are L2-normalized	Fast; equals cosine for unit vectors

Euclidean Distance	sqrt(sum((a_i-b_i)^2))	Metric learning, low-dim spaces	Sensitive to magnitude
Manhattan Distance	sum(|a_i-b_i|)	Sparse or tabular features	Rarely used in modern RAG

4.3 Image Embeddings for RAG

Images cannot be embedded with text models. Specialized vision encoders produce fixed-length image representations that can be stored in vector databases alongside text embeddings.

- CLIP (Contrastive Language-Image Pretraining): Produces aligned image and text embeddings in a shared space. Enables text-to-image and image-to-text retrieval without a human-labeled dataset
- ViT (Vision Transformer): Patch-based image encoding; produces strong general-purpose image features
- Convolutional features: ResNet-50/DINO features — useful when CLIP is overkill
- BLIP-2: Can generate textual descriptions of images and produce aligned embeddings
- SigLIP: Improved CLIP training with sigmoid loss — better zero-shot performance
- Image embedding dimensions: Typically 512–1024 dim (CLIP: 512 or 768 depending on variant)

4.4 Audio Embeddings

- Wav2Vec2 / HuBERT: Self-supervised speech representations — useful for speaker identification, emotion, phonetics
- Whisper embeddings: Speech encoder features from OpenAI Whisper — useful for ASR-aligned retrieval
- EnCodec / SoundStream: Neural audio codecs producing compact, high-fidelity audio representations
- Log-Mel spectrograms + CNN: Classical approach — still competitive for audio classification tasks
- CLAP (Contrastive Language-Audio Pretraining): Audio-text aligned embeddings, analogous to CLIP

4.5 Cross-Modal Embeddings

Cross-modal embedding models produce representations in a shared semantic space where similar concepts from different modalities (e.g., an image of a dog and the word 'dog') are geometrically close. This is the foundation of cross-modal retrieval.

- CLIP: The canonical cross-modal model for image-text. Query with text, retrieve images (or vice versa)
- ImageBind (Meta): Extends alignment to 6 modalities: image, text, audio, video, depth, IMU
- ALIGN: Google's CLIP analog trained on noisy internet data at scale
- CoCa: Combined contrastive + generative pretraining for aligned multimodal representations

4.6 Embedding Drift & Monitoring

- Embedding drift: The semantic meaning of terms shifts over time (new models, new vocabulary)
- Detection: Compare embedding distributions across time using cosine similarity histograms or MMD tests
- Mitigation: Scheduled re-embedding pipelines, versioned embedding stores
- Dimensionality reduction for visualization: PCA, t-SNE, UMAP — useful for debugging embedding quality

4.7 Hands-On Lab 4: Embedding Space Exploration

Lab 4.1 — Multimodal Embedding Comparison

Part A: Embed 1,000 text passages with 3 models (ada-002, all-MiniLM-L6-v2, BGE-large) and compare MTEB benchmark scores

Part B: Embed 500 image-caption pairs with CLIP and visualize the joint embedding space using UMAP

Part C: Demonstrate cross-modal retrieval — query with text, retrieve images; query with images, retrieve text

Part D: Simulate embedding drift by swapping embedding models mid-index and observe retrieval quality degradation

Stack: Hugging Face Transformers, OpenAI SDK, FAISS, UMAP-learn, matplotlib

MODULE 5

Vector Databases & Indexing

1 Week / ~12 Hours Intermediate–Advanced

Topics covered:

- Vector database landscape: Pinecone, Weaviate, Qdrant, Chroma, Milvus, pgvector
- ANN algorithms: HNSW, IVF, PQ, ScaNN, Annoy — how they work and when to choose
- Index operations: build, refresh, re-index, upserts, deletes
- Metadata filtering: combining vector search with structured filters
- Index sharding and replication for scale
- Multimodal vector stores: storing mixed modality embeddings
- Hybrid indexes: combining dense + sparse in one store
- Exact vs. Approximate Nearest Neighbor search tradeoffs

Module 5: Vector Databases & Indexing

5.1 Why Vector Databases?

Standard relational databases cannot efficiently search in high-dimensional spaces. Vector databases are purpose-built for Approximate Nearest Neighbor (ANN) search, enabling retrieval of the top-K most semantically similar vectors in milliseconds, even across millions of embeddings.

5.2 Vector Database Landscape

Database	Type	Highlights	Best For
Pinecone	Managed cloud	Serverless, fully managed, namespace support	Production with low ops burden
Weaviate	Open source + cloud	Multi-tenancy, BM25 hybrid built-in, GraphQL	Hybrid search, graph traversal
Qdrant	Open source + cloud	Rich filtering, payload indexing, Rust-based	Performance + filtering
Chroma	Open source, local	Minimal setup, good for prototyping	Development / small scale
Milvus	Open source	Billion-scale, cloud-native, multi-index	Large-scale production
pgvector	Postgres extension	SQL + vector in one DB, familiar tooling	Existing Postgres users
FAISS	Library (Meta)	Industry benchmark, GPU support, not a DB	Offline indexing / research
Vespa	Open source	Hybrid search + ranking, ML inference	Advanced ranking pipelines

5.3 ANN Algorithms Deep Dive

HNSW (Hierarchical Navigable Small World)

HNSW builds a multi-layered graph where each node connects to its nearest neighbors. Search begins at the top (sparse) layer and descends to finer layers, enabling O(log n) retrieval. It is the most widely used ANN algorithm in production vector databases.

- Parameters: M (number of neighbors per node), ef_construction (search depth during build), ef (search depth at query time)
- Tradeoff: High recall but memory-intensive — each vector requires ~50-100 bytes overhead

IVF (Inverted File Index)

IVF clusters the vector space into nlist centroids using k-means. At query time, only nprobe nearest clusters are searched, dramatically reducing the search space.

- Parameters: nlist (number of clusters), nprobe (number of clusters to search)
- Tradeoff: Lower memory than HNSW; recall depends on nprobe setting

PQ (Product Quantization)

PQ compresses high-dimensional vectors into compact codes by splitting the vector into subvectors and quantizing each independently. Enables billion-scale indexes at the cost of some recall.

- Often combined with IVF: IVF-PQ is the standard large-scale configuration
- ScaNN (Scalable Approximate Nearest Neighbors): Google's optimized ANN library, outperforms FAISS on many benchmarks

5.4 Index Operations

- Index build: Batch embedding and indexing of the full corpus (offline step)
- Index refresh: Periodic rebuild or incremental update to incorporate new documents
- Re-indexing: Full rebuild required when changing embedding model or index parameters
- Upserts: Insert or update a vector by ID (supported in Pinecone, Qdrant, Weaviate)
- Deletes: Remove vectors by ID or metadata filter
- Metadata filtering: Filter by structured fields (date, source, category) during vector search — crucial for access control and freshness

5.5 Multimodal & Hybrid Indexes

- Separate indexes per modality: maintain independent image, text, and audio indexes; merge results at retrieval time
- Unified embedding space: Use cross-modal models (CLIP, ImageBind) to index all modalities in a single shared space
- Hybrid indexes: Combined dense + sparse vectors (Weaviate's hybrid search, Qdrant's hybrid mode)
- Index sharding: Distribute shards across nodes for horizontal scaling

- Replication: Copy shards across multiple nodes for high availability

5.6 Hands-On Lab 5: Vector Database Benchmarking

Lab 5.1 — Build & Benchmark Three Vector Stores

Index 100K Wikipedia article chunks in: FAISS (local), Qdrant (Docker), Weaviate (Docker)
Benchmark: Recall@10, Query latency (p50, p95, p99), Index build time, Memory usage
Experiment: Compare HNSW vs IVF-PQ configurations in FAISS
Multimodal extension: Index COCO image embeddings (CLIP ViT-B/32) alongside text in Weaviate
Deliverable: A benchmark report with plots and a recommendation for a given use case.

MODULE 6

Retrieval Strategies

1.5 Weeks / ~14 Hours Advanced

Topics covered:

- Basic vector similarity search: Top-K, threshold-based, metadata-filtered
- Hybrid search: combining dense + sparse (BM25/TF-IDF) with score fusion
- Advanced query strategies: query expansion, rewriting, decomposition, step-back prompting
- Multi-query retrieval and sub-query generation
- Hierarchical retrieval and graph-based retrieval
- Time-aware and freshness-aware retrieval
- Contextual retrieval (Anthropic's approach)
- Cross-modal retrieval: text-to-image, image-to-text, audio-to-text
- Re-ranking: cross-encoders, bi-encoders, RRF, MMR, Cohere Rerank

Module 6: Retrieval Strategies

6.1 Basic Retrieval

- Vector similarity search: Embed the query, search the index for the top-K most similar vectors
- Top-K retrieval: Return the K highest-scoring chunks (typical K: 3–20)
- Threshold-based retrieval: Only return results above a minimum similarity score (avoids injecting irrelevant chunks)
- Metadata-filtered retrieval: Combine semantic search with SQL-like filters (e.g., date > 2024, source = 'pubmed')

6.2 Hybrid Search

Hybrid search combines dense (semantic) and sparse (keyword) retrieval signals, which are complementary: dense search excels on paraphrase matching; sparse search excels on rare entity names and exact terminology.

- BM25: Probabilistic keyword scoring based on term frequency and inverse document frequency
- TF-IDF: Classical keyword vector approach
- SPLADE: Neural sparse retrieval that learns term expansion — bridges sparse and dense
- Score fusion: Reciprocal Rank Fusion (RRF) is the most robust fusion method — combines ranked lists without normalizing scores
- RRF formula: RRF_score(d) = sum(1 / (k + rank_i(d))) where k=60 is the standard constant
- Alpha weighting: Linear combination alpha * dense_score + (1-alpha) * sparse_score — requires calibration

6.3 Advanced Query Strategies

Strategy	Description	When to Use
Query Expansion	Add synonyms/related terms to the query before retrieval	Recall-sensitive, sparse retrieval
Query Rewriting	Use LLM to rephrase ambiguous queries for better retrieval	Conversational RAG, unclear queries
Query Decomposition	Break complex multi-part queries into sub-queries, retrieve for each	Multi-hop, complex QA
Step-back Prompting	Generate a more abstract 'step-back' question to retrieve background context first	Physics, law, complex reasoning
HyDE (Hypothetical Document Embeddings)	Generate a hypothetical answer, embed it, use for retrieval	Asymmetric retrieval tasks
Multi-query Retrieval	Generate N paraphrases of the query, retrieve for each, union results	Robustness to query phrasing
Sub-query Generation	Decompose into atomic sub-questions (e.g. LlamaIndex SubQuestionQueryEngine)	Complex document corpora

6.4 Cross-Modal Retrieval

Cross-modal retrieval is a defining capability of Multimodal RAG systems. The query and retrieved items may be in different modalities.

- Text-to-image: Embed user text query with CLIP text encoder; retrieve nearest image embeddings
- Image-to-text: Embed query image with CLIP image encoder; retrieve nearest text chunk embeddings
- Audio-to-text: Transcribe audio query with Whisper; then text-to-text or text-to-image retrieval
- Video retrieval: Sample key frames, embed with CLIP; embed transcript; fuse scores; retrieve matching video segments
- Use case example: 'Find me the slide from the conference video where they show the architecture diagram' — image+text query against a video index

6.5 Re-Ranking & Post-Processing

Initial retrieval (using bi-encoders for speed) is followed by a more expensive but more accurate re-ranking step.

- Cross-encoders: Process query and passage jointly (attention over both) — much more accurate than bi-encoders but 10-100x slower. Models: ms-marco-MiniLM-L-6-v2, monoT5
- Bi-encoders: Encode query and passage independently — fast, used for first-stage retrieval
- Late interaction models: ColBERT — token-level similarity between query and passage, O(n) retrieval
- Cohere Rerank: Hosted re-ranking API, easy to integrate into any RAG pipeline

- Maximal Marginal Relevance (MMR): Re-ranks to balance relevance and diversity — avoids returning redundant chunks
- Diversity re-ranking: Ensures retrieved chunks cover different aspects of the topic
- Context pruning: Remove low-relevance or redundant sentences from retrieved chunks before injection
- Passage compression: Use an LLM to compress retrieved passages to their essential content (reduces token count)

6.6 Specialized Retrieval

- Time-aware retrieval: Weight or filter results by recency (e.g., prefer documents from last 30 days)
- Freshness-aware retrieval: Track document update timestamps; trigger re-embedding on change
- Session-aware retrieval: Use conversation history to bias retrieval toward recently discussed entities
- Graph-based retrieval: Traverse a knowledge graph from entity mentions in the query to retrieve structured facts
- RAPTOR: Hierarchical tree-based retrieval — clusters and summarizes documents at multiple granularity levels

6.7 Hands-On Lab 6: Retrieval Strategy Shootout

Lab 6.1 — Retrieval Benchmark on BEIR

Evaluate 5 retrieval configurations on the BEIR benchmark (or a subset: NFCorpus, TREC-COVID, SciFact):

Config 1: Dense only (BGE-large)

Config 2: Sparse only (BM25 via Elasticsearch)

Config 3: Hybrid (dense + BM25 with RRF)

Config 4: Hybrid + Cross-encoder re-ranking (ms-marco-MiniLM-L-6-v2)

Config 5: Hybrid + MMR diversity re-ranking

Metrics: NDCG@10, Recall@100, Mean Reciprocal Rank (MRR)

Deliverable: A comparison table and a written recommendation for each use case.

MODULE 7

Multimodal RAG: Core Integration

2 Weeks / ~20 Hours Advanced

Topics covered:

- Architecture of multimodal RAG systems: text + image + audio + video pipelines
- Vision-language models: GPT-4V, LLaVA, Flamingo, InstructBLIP, Qwen-VL
- Multimodal prompt construction: interleaving text and images
- Image-text QA: retrieving images and using them in generation
- Document understanding: PDFs with figures, slides, infographics
- Video QA: retrieval over video frames and transcripts
- Audio-grounded QA: retrieving and citing audio segments
- Multimodal memory: storing and retrieving past visual contexts
- Modality alignment challenges and visual hallucination mitigation
- Cross-modal alignment metrics and evaluation

Module 7: Multimodal RAG — Core Integration

7.1 Multimodal RAG Architecture

A multimodal RAG system extends the standard text-only pipeline to handle heterogeneous modalities at every stage. The architecture must handle modality-specific ingestion, modality-aware embeddings, a unified or federated index, cross-modal retrieval, multimodal prompt construction, and vision-language generation.

Multimodal RAG: Full Pipeline

1. INGESTION: Text chunks + Image embeddings (CLIP) + Audio transcripts (Whisper) + Video frame embeddings
2. INDEXING: Unified cross-modal vector store (CLIP embedding space) OR federated modal-specific indexes
3. RETRIEVAL: Cross-modal ANN search + optional BM25 on text transcripts + metadata filtering
4. RE-RANKING: Cross-encoder re-ranking (text) + image relevance scoring + modality fusion
5. PROMPT CONSTRUCTION: Interleave retrieved text + inline images (base64 or URL) + citations
6. GENERATION: Vision-Language Model (GPT-4V, LLaVA, Qwen-VL) produces grounded multimodal answer

7.2 Vision-Language Models (VLMs) for RAG Generation

Model	Organization	Context	Strengths	Access
GPT-4V / GPT-4o	OpenAI	128K tokens	Best overall, strong instruction following	API

LLaVA-1.6 / LLaVA-Next	Haotian Liu et al.	4K-8K	Open source, strong visual reasoning	HuggingFace
InstructBLIP	Salesforce	4K	Strong zero-shot VQA	HuggingFace
Qwen-VL	Alibaba	8K+	Excellent on Chinese + multilingual	HuggingFace
Flamingo / IDEFICS	DeepMind/HuggingFace	2K	Few-shot multimodal learning	HuggingFace
Gemini 1.5 Pro	Google	1M tokens	Long context, native multimodal	API
Claude 3.5 Sonnet	Anthropic	200K tokens	Vision + long context + grounding	API
MiniCPM-V	Tsinghua	8K+	Efficient, on-device capable	HuggingFace

7.3 Multimodal Prompt Construction

Constructing effective prompts for VLMs requires careful ordering of modalities. Most VLMs accept interleaved image-text sequences. The retrieved images should be placed near the question they are meant to answer, with clear source attribution.

- Image injection formats: Inline base64 (for API calls), URL references, or pre-uploaded image IDs
- Modality ordering: System context -> Retrieved text chunks -> Retrieved images (captioned) -> User question
- Caption augmentation: Always attach a generated or original caption to each retrieved image for text-model fallback
- Context window budgeting: Images consume significant token budget; balance image count vs. text context
- Source attribution: Include image source metadata (document name, page number, timestamp for video)
- Prompt compression: If context is too long, compress text chunks before injection; images cannot be compressed

7.4 Document Understanding with Multimodal RAG

Document understanding is one of the most common enterprise applications of Multimodal RAG. Complex PDF documents contain figures, tables, equations, and charts that pure-text extractors cannot adequately capture.

- Layout-aware extraction: pdfplumber, PyMuPDF, Adobe PDF Extract API
- Figure extraction: Identify and crop figures from PDFs (PyMuPDF + heuristics)
- Figure captioning: Run BLIP-2 or GPT-4V on extracted figures to generate searchable captions
- Indexing strategy: Index figure captions as text + figure embeddings (CLIP) in parallel indexes
- Table extraction: Camelot / Tabula for structured tables; convert to markdown for embedding

- Equation handling: LaTeX rendering -> image -> caption pipeline for mathematical content

7.5 Video RAG

Video RAG enables question-answering over video content by combining visual frame retrieval with audio transcript retrieval.

- Frame sampling: Extract frames at fixed intervals (1fps) or at scene boundaries (PySceneDetect)
- Frame embedding: Embed each frame with CLIP — store with timestamp metadata
- Audio transcription: Transcribe with Whisper (or Google Speech-to-Text) — align to timestamps
- Chunk alignment: Create multimodal chunks that pair frame embedding + corresponding transcript segment
- Retrieval: Search frames by visual query (image CLIP), transcripts by text (BM25/dense); fuse results
- Generation: Pass retrieved frame images + transcript context to GPT-4V or Gemini for answer
- Citation: Reference specific video timestamps in the generated answer

7.6 Visual Hallucination in Multimodal RAG

VLMs are prone to 'visual hallucination' — generating statements about image content that is not actually present. This is the multimodal analog of text hallucination.

- Hallucination types: Object hallucination (claiming non-existent objects), attribute hallucination (wrong color/size/position), relational hallucination (wrong relationships between objects)
- Mitigation: POPE benchmark for evaluation; grounding-focused VLM fine-tuning; explicit instruction to only describe what is visible
- Detection: Cross-checking generation against retrieved image metadata; using object detection to verify claimed objects
- Faithfulness scoring: Adapt NLI-based faithfulness metrics to vision-language settings

7.7 Hands-On Lab 7: End-to-End Multimodal RAG System

Lab 7.1 — Image+Text QA over Technical Documents (MAIN PROJECT)

Goal: Build a production-quality multimodal RAG system for technical documentation QA.

Data: AWS Architecture documentation PDFs (100+ pages with diagrams), supplemented with related blog post HTML.

Pipeline:

10. Extract text and figures from PDFs (PyMuPDF). Caption figures with GPT-4V.
11. Embed text chunks with BGE-large; embed figures with CLIP ViT-L/14.
12. Index both modalities in Qdrant with metadata (source, page, type: text|image).
13. Implement hybrid cross-modal retrieval: text query retrieves text + relevant images.

14. Construct multimodal prompt with retrieved context and images; generate with GPT-4o.
15. Build a Gradio UI showing citations, retrieved images, and the generated answer.

Evaluation: Answer correctness, faithfulness (RAGAS), image relevance (manual annotation), visual hallucination rate.

Deliverable: A publicly accessible Gradio app + evaluation report.

MODULE 8

Generation, Prompting & Evaluation

1.5 Weeks / ~14 Hours Advanced

Topics covered:

- Grounded generation: faithfulness, extractive vs. abstractive answers
- Chain-of-Thought (CoT) and Retrieval-aware CoT
- Prompt templates, context stuffing, context window budgeting
- Source attribution and inline citations
- Prompt compression techniques (LLMLingua, Selective Context)
- RAG evaluation frameworks: RAGAS, TruLens, DeepEval
- Retrieval metrics: Recall@K, Precision@K, MRR, NDCG
- Generation metrics: faithfulness, answer relevance, hallucination rate, citation accuracy
- Multimodal evaluation: image relevance, cross-modal alignment scoring
- End-to-end system evaluation: latency, cost, throughput

Module 8: Generation, Prompting & Evaluation

8.1 The Generation Phase

- Grounded generation: The LLM generates answers using retrieved context as its primary knowledge source
- Faithfulness: Generated answer is factually consistent with retrieved context (not model's parametric knowledge)
- Answer synthesis: Combining information from multiple retrieved passages into a coherent response
- Extractive vs. Abstractive: Extractive copies spans verbatim; Abstractive paraphrases — both can be appropriate
- Multi-document synthesis: Identifying consensus, contradictions, and complementary information across chunks
- Self-consistency: Sample multiple answers and aggregate — improves reliability for complex questions
- Retrieval-aware CoT: Structure prompts to first reason about which parts of context are relevant, then answer

8.2 Prompt Construction Deep Dive

Prompt construction is the critical interface between retrieval and generation. A well-designed prompt template dramatically improves faithfulness and reduces hallucinations.

- System prompt grounding: Instruct the model to only use provided context: 'Answer based solely on the context below. If the answer is not in the context, say so.'
- Context stuffing: Concatenate retrieved chunks into the prompt — maintain document order or relevance order
- Instruction vs. context separation: Use clear delimiters (XML tags, triple backticks) to separate system instructions, retrieved context, and the user question
- Source attribution: Prepend each chunk with its source identifier [DOC_1], [DOC_2]

- Inline citations: Instruct the model to cite sources like [1], [2] next to claims
- Context window budgeting: Reserve tokens for: system prompt (~200t) + retrieved context (variable) + generated answer (200-500t)
- Prompt chaining: Use a chain of prompts — first retrieve and compress, then generate, then cite/verify
- Prompt token optimization: Remove stopwords and boilerplate from retrieved chunks to fit more content

8.3 Prompt Compression

- LLMLingua: Token-level prompt compression using a small language model to identify and remove unimportant tokens
- Selective Context: Sentence-level filtering by self-information score
- RECOMP: Retrieve, Compress, Prepend — train a compressor model specifically for RAG
- Typical compression ratios: 2x-5x compression with <5% retrieval quality loss

8.4 RAG Evaluation Framework

Evaluating a RAG system requires assessing both the retrieval stage and the generation stage independently, as well as the end-to-end pipeline.

Metric	Stage	Description	Tool
Recall@K	Retrieval	Fraction of relevant docs in top-K retrieved	BEIR, custom
Precision@K	Retrieval	Fraction of top-K that are relevant	BEIR, custom
MRR (Mean Reciprocal Rank)	Retrieval	Average of 1/rank of first relevant result	BEIR
NDCG@K	Retrieval	Normalized graded relevance in top-K	BEIR
Hit Rate	Retrieval	Whether any relevant doc appears in top-K	LlamaIndex eval
Faithfulness	Generation	Answer consistent with retrieved context (NLI-based)	RAGAS, TruLens
Answer Relevance	Generation	Answer addresses the question asked	RAGAS
Contextual Precision	Generation	Relevant chunks ranked higher	RAGAS
Contextual Recall	Generation	All needed context was retrieved	RAGAS
Hallucination Rate	Generation	Rate of factually incorrect claims	DeepEval, FActScore
Citation Accuracy	Generation	Cited sources actually support the claims	Custom NLI

Answer Completeness	Generation	All aspects of question addressed	G-Eval
End-to-End Latency	System	Time from query to response	Custom tracing
Token Efficiency	System	Answer quality per token consumed	Custom

8.5 Multimodal Evaluation

- Image relevance: Does the retrieved image actually depict the content described? (Human or VLM judge)
- Cross-modal alignment: Cosine similarity between text query embedding and retrieved image embedding (CLIP score)
- Visual faithfulness: Does the generated answer accurately describe the retrieved image content?
- Visual hallucination rate: Rate of generated claims about image content not visually present
- POPE benchmark: Probing Object Hallucination Evaluation — standard benchmark for VLM hallucination
- NoCaps / VQAv2: Standard image QA benchmarks for evaluating VLM generation quality

8.6 Hands-On Lab 8: RAG Evaluation with RAGAS

Lab 8.1 — Systematic RAG Evaluation

Use RAGAS to evaluate the system built in Lab 7.1 across all core metrics
Create a test set of 50 QA pairs with ground truth answers and relevant chunks
Compute: Faithfulness, Answer Relevance, Contextual Precision, Contextual Recall
Implement custom citation accuracy metric using an NLI model (DeBERTa-v3)
Identify the top 10 failure cases and categorize by failure mode
Deliverable: An evaluation dashboard (Weights & Biases or Streamlit) showing all metrics

MODULE 9

Advanced RAG Techniques

2 Weeks / ~18 Hours Advanced

Topics covered:

- Naive, Advanced, Modular, and Adaptive RAG architectures
- Self-RAG: retrieval as a conditional action based on model confidence
- Agentic RAG: LLM agents that plan, retrieve, and reason iteratively
- Iterative and Multi-hop RAG for complex multi-step questions
- Conversational RAG with memory and session awareness
- Graph-RAG and Knowledge Graph-augmented generation
- RAG-Fusion and HyDE (Hypothetical Document Embeddings)
- RAPTOR: tree-based hierarchical retrieval
- Corrective RAG (CRAG): retrieval quality self-correction
- Tool-augmented RAG: integrating APIs, calculators, code execution

Module 9: Advanced RAG Techniques

9.1 RAG Architecture Taxonomy

Architecture	Key Idea	When to Use
Naive RAG	Retrieve once, generate once. No query modification.	Simple QA, prototyping
Advanced RAG	Query rewriting + re-ranking + iterative refinement	Production QA systems
Modular RAG	Plug-and-play modules: any retriever, ranker, generator	Experimentation, flexibility
Adaptive RAG	Route queries to different strategies based on complexity	Mixed-difficulty corpora
Self-RAG	Model decides when and what to retrieve using special tokens	Efficiency-critical systems
Agentic RAG	LLM plans and executes multi-step retrieval as an agent	Complex research tasks
Iterative RAG	Retrieve -> generate partial answer -> retrieve again -> refine	Multi-hop questions
Conversational RAG	Maintains dialogue history; retrieves session-aware context	Chatbots, assistants
Multi-hop RAG	Chains multiple retrievals to traverse knowledge graph	Wikipedia multi-hop QA
Graph-RAG	Retrieves from knowledge graph nodes + edges + text chunks	Entity-rich corpora

9.2 Self-RAG

Self-RAG (Asai et al., 2023) trains a model to decide whether retrieval is needed, retrieve when appropriate, critically evaluate retrieved passages (relevance, support, utility), and generate accordingly. This is achieved by fine-tuning with special 'reflection tokens'.

- Reflection tokens: [Retrieve], [IsREL], [IsSUP], [IsUSE] — emitted mid-generation
- Selective retrieval: Model only retrieves when it determines retrieval is necessary — reduces latency
- Segment-level critique: Each retrieved passage is evaluated for relevance and support
- Training: Standard next-token training on corpus augmented with reflection tokens

9.3 Agentic RAG

Agentic RAG wraps a RAG system in an LLM agent that can plan multi-step retrieval strategies, call external tools, and iterate on its answer. The agent uses RAG as one of several tools at its disposal.

- ReAct (Reason + Act): Agent reasons about what to retrieve, issues retrieval actions, observes results, reasons again
- Plan-and-Execute: First plan all retrieval steps, then execute in batch — more efficient than sequential
- Tool-augmented RAG: Agent can call: vector search, SQL query, web search, calculator, code interpreter, APIs
- Frameworks: LangChain Agents, LlamaIndex Agentic Pipeline, AutoGen, CrewAI
- Memory integration: Agent maintains short-term (working memory) and long-term (episodic memory) across sessions

9.4 Graph-RAG & Knowledge Graph Augmentation

Graph-RAG combines traditional vector retrieval with structured knowledge graph traversal. Entity mentions in the query are linked to knowledge graph nodes, and graph traversal retrieves structured facts, related entities, and relationships — content that would be fragmented across many chunks in a flat document store.

- Microsoft GraphRAG: Hierarchical summarization of document community clusters — excellent for global thematic queries
- Neo4j + LangChain: Store entities and relationships as graph nodes; query with Cypher + vector search
- Entity linking: Connect query terms to knowledge graph entities (BLINK, spaCy entity linker)
- KGRAG: Retrieve triples (subject, predicate, object) from KG and inject as structured context

9.5 HyDE, RAG-Fusion, RAPTOR, CRAG

HyDE (Hypothetical Document Embeddings)

Ask the LLM to generate a hypothetical answer to the query. Embed that hypothetical answer and use it for retrieval. The hypothesis often uses more domain-specific vocabulary than the raw query, improving retrieval of technical content.

RAG-Fusion

Generate multiple paraphrase queries, retrieve for each independently, then fuse the ranked lists using Reciprocal Rank Fusion. Significantly improves recall vs. single-query retrieval.

RAPTOR

Recursively clusters document chunks and generates summaries at each level of a tree hierarchy. Enables retrieval at multiple levels of granularity — bottom-level for specific facts, top-level for thematic overview queries.

Corrective RAG (CRAG)

Evaluates the quality of retrieved documents using a lightweight grader. If all retrieved documents are deemed irrelevant, CRAG triggers a web search fallback or query reformulation before generating.

9.6 Multi-hop & Conversational RAG

- Multi-hop RAG: A multi-step question (e.g., 'Who is the CEO of the company that acquired DeepMind?') requires chaining multiple retrieval steps, using intermediate answers as queries
- Conversational RAG: Reformulate questions in context of dialogue history (query rewriting using history). Use session-aware memory to bias retrieval
- Chat history compression: Summarize earlier dialogue to prevent context overflow
- Entity memory: Maintain a structured record of entities mentioned in conversation for retrieval biasing

9.7 Hands-On Lab 9: Agentic Multimodal RAG

Lab 9.1 — Research Agent with Multimodal RAG

Build an agentic system that answers complex research questions using multiple sources:

Tool 1: Vector search over a technical document corpus (text + images)

Tool 2: Web search (SerpAPI / Tavily) for current information

Tool 3: Python code interpreter (E2B / LangChain sandbox) for computation

Tool 4: SQL query over a structured database

Agent: GPT-4o with ReAct prompting, 5-step max reasoning chain

Implement multi-hop retrieval: each agent step can issue new retrieval queries

Deliverable: An agent that can answer questions requiring 3+ retrieval steps with multimodal evidence

MODULE 10

Finetuning & LoRA for Multimodal RAG

2 Weeks / ~20 Hours Advanced

Topics covered:

- When and why to fine-tune vs. prompt engineer for RAG
- Full model fine-tuning: data preparation, training, evaluation
- Parameter-Efficient Fine-Tuning (PEFT): LoRA, QLoRA, Prefix Tuning, Adapter Layers
- LoRA mathematics: low-rank decomposition of weight updates
- QLoRA: 4-bit quantization + LoRA for memory-efficient fine-tuning
- Fine-tuning the retriever: DPR training, ANCE, in-batch negatives
- Fine-tuning the reader/generator for faithfulness and citation
- Multimodal fine-tuning: LLaVA training, visual instruction tuning
- LoRA for vision-language models: fine-tuning VLMs for domain-specific RAG
- Data synthesis: generating training data for RAG fine-tuning

Module 10: Finetuning & LoRA for Multimodal RAG

10.1 When to Fine-Tune

Fine-tuning is powerful but expensive. The decision should be based on evidence that prompt engineering and RAG alone are insufficient for the target task.

Scenario	Recommended Approach
Need current knowledge not in training data	RAG only (no fine-tuning)
Need specific output format or style	Fine-tuning (or few-shot prompting)
Poor retrieval quality on domain jargon	Fine-tune the embedding model
High hallucination rate on domain content	Fine-tune the generator for faithfulness
High cost due to long prompts	Fine-tune for implicit knowledge + shorter prompts
Need optimal combo: knowledge + style	RAG + fine-tuning together
Limited compute budget	QLoRA (4-bit) fine-tuning

10.2 LoRA: Low-Rank Adaptation — Theory

LoRA (Hu et al., 2021) is a parameter-efficient fine-tuning method that freezes the original model weights and injects trainable rank-decomposition matrices into each layer of the Transformer. Instead of updating a weight matrix W (d x k parameters), LoRA trains two small matrices A (d x r) and B (r x k) where r << d.

- Adapted weight: W' = W + BA where B is initialized to zero (so initial LoRA output = zero)
- Rank r: Typical values 4, 8, 16, 64. Higher r = more capacity but more parameters
- Alpha (scaling): LoRA scales BA by alpha/r. Common: alpha = 2*r

- Target modules: Typically applied to Q, V projection matrices; sometimes K, O, FFN layers
- Parameter reduction: For a 7B model, LoRA with r=8 trains ~4M params vs 7B for full fine-tuning
- Merge at inference: LoRA weights can be merged into the base model (W + BA) with zero inference overhead

10.3 QLoRA: Quantized LoRA

QLoRA (Dettmers et al., 2023) combines 4-bit quantization of the frozen base model with LoRA adapters in 16-bit precision. This enables fine-tuning of 65B+ parameter models on a single GPU.

- 4-bit NormalFloat (NF4): A new data type optimized for normally distributed weights
- Double quantization: Quantize the quantization constants to save additional memory
- Paged optimizers: NVIDIA unified memory to handle memory spikes during backprop
- Practical result: 65B LLaMA fine-tuned on a single 48GB A100 GPU
- Library: Hugging Face PEFT + bitsandbytes + transformers

10.4 Fine-Tuning the Retriever

A domain-specific embedding model dramatically improves retrieval quality in specialized domains. The retriever can be fine-tuned using contrastive learning with positive and negative passage pairs.

- DPR (Dense Passage Retrieval): Trains a bi-encoder (question encoder + passage encoder) using in-batch negatives
- ANCE (Approximate Nearest-Neighbor Negative Contrastive Learning): Mines hard negatives from the index during training — superior to in-batch negatives
- Training data: (question, positive passage, hard negative passage) triples
- Data sources: MS MARCO, Natural Questions, domain-specific QA pairs
- Synthetic data: Use an LLM to generate questions from document chunks (GPT-4 or Llama-3) — synthetic QA pairs for retriever training
- Frameworks: sentence-transformers library, FlagEmbedding (BGE training), tevatron

10.5 Fine-Tuning the Generator for RAG

Fine-tuning the generator on domain-specific RAG data teaches the model to correctly cite context, remain faithful to retrieved passages, and format answers appropriately.

- Training format: (context, question, faithful_answer) triples
- Faithfulness reward: Can use RLHF or RLAIF with faithfulness as the reward signal
- Negative training: Include examples where the model would hallucinate; train to abstain or cite context
- Citation training: Fine-tune to produce inline citations [1][2] matching retrieved chunk IDs
- Self-RAG fine-tuning: Train model to emit reflection tokens during generation

10.6 Multimodal Fine-Tuning: Vision-Language Models

Fine-tuning VLMs for domain-specific multimodal RAG enables better visual grounding, domain vocabulary understanding, and instruction-following for specialized tasks.

LLaVA Training Pipeline

- Stage 1 — Feature alignment: Freeze LLM + vision encoder; train only the projection layer on image-caption pairs
- Stage 2 — Instruction tuning: Train all components (or LoRA adapters) on visual instruction data
- Data: LLaVA-Instruct-150K or domain-specific image-instruction pairs

LoRA for VLMs

- Apply LoRA to both the LLM backbone and optionally the vision encoder
- Target modules in VLM: LLM Q/V projections + multimodal projection layer
- Typical setup: r=16, alpha=32, target_modules=['q_proj','v_proj','mm_projector']
- Memory: 7B VLM + QLoRA + gradient checkpointing fits in 24GB VRAM

10.7 Synthetic Training Data Generation

High-quality training data is often the bottleneck. LLMs can generate synthetic QA pairs from existing documents, dramatically reducing annotation costs.

- Question generation: Prompt GPT-4 to generate diverse questions from each document chunk
- Answer generation: Generate ground-truth answers with chain-of-thought reasoning
- Hard negative mining: Use the vector index to retrieve plausible-but-wrong passages as negatives
- Data quality filtering: Filter generated QA pairs by answer faithfulness score
- Frameworks: Ragas synthetic data generation, LlamaIndex dataset generation, Distilabel

10.8 Hands-On Lab 10: LoRA Fine-Tuning for Domain RAG

Lab 10.1 — QLoRA Fine-Tune for Medical Multimodal RAG

Goal: Fine-tune a VLM (LLaVA-1.6-7B or Qwen-VL) using QLoRA for radiology report generation + image QA.

Dataset: PubMedQA + MIMIC-CXR radiology images (publicly available subsets)

Step 1: Generate synthetic (image, question, answer) triples from radiology reports using GPT-4

Step 2: Fine-tune LLaVA-1.6 with QLoRA (r=16, alpha=32) using HuggingFace PEFT + TRL SFTTrainer

Step 3: Integrate the fine-tuned VLM as the generator in a multimodal RAG pipeline

Step 4: Evaluate: faithfulness (radiology-specific), visual hallucination rate, answer accuracy vs. base model

Compute: Runs on 1x A100 40GB or Google Colab A100 (within free tier with small batch sizes)

Deliverable: Fine-tuned LoRA adapter weights + evaluation report comparing base vs. fine-tuned RAG performance

MODULE 11

Production, Ops, Security & Failure Modes

1.5 Weeks / ~12 Hours Advanced

Topics covered:

- RAGOps: observability, tracing, logging, feedback loops for RAG pipelines
- Performance optimization: caching, async retrieval, batching, load balancing
- Memory & conversation handling: short-term, long-term, episodic memory
- Security: prompt injection, retrieval poisoning, data leakage, PII masking
- Access control: document-level permissions, row-level security
- Common failure modes: context dilution, hallucination, embedding mismatch, stale data
- Multimodal-specific failures: modality misalignment, visual hallucinations
- Model versioning, embedding versioning, rollbacks, canary deployments
- Cost optimization: token efficiency, embedding model selection, caching strategies
- Human-in-the-loop and continuous evaluation in production

Module 11: Production, Ops, Security & Failure Modes

11.1 RAGOps: Observability & Monitoring

A RAG system in production must be observable. Without visibility into retrieval quality, generation faithfulness, and end-to-end latency, silent failures will go undetected.

- Distributed tracing: Instrument every stage — ingestion, embedding, retrieval, re-ranking, generation — with trace IDs (OpenTelemetry, Arize Phoenix, LangSmith)
- Logging: Log queries, retrieved chunk IDs, scores, generated answers, and user feedback
- Retrieval quality dashboards: Track Recall@K, latency distributions, and score distributions over time
- Faithfulness monitoring: Run automated faithfulness scoring on production responses (RAGAS in evaluation mode)
- Feedback loops: Thumbs-up/down, correctness flags → retrain embedding or generator
- Data freshness: Monitor time since last index update; alert on stale data
- Continuous evaluation: Run evaluation test suite on every deployment
- Human-in-the-loop: Route low-confidence responses to human reviewers before delivery

11.2 Performance & Scalability

- Embedding cache: Cache embeddings of common queries — avoid re-embedding repeated queries
- Query cache: Cache (query_hash -> retrieved_chunks) for identical queries
- Asynchronous retrieval: Use asyncio to parallelize retrieval from multiple indexes
- Batch retrieval: Group multiple user queries into a single batch embedding call
- Warm indexes: Keep vector indexes loaded in memory — cold start adds 1-5 seconds
- Horizontal scaling: Shard vector indexes across multiple nodes with a load balancer

- Cost optimization: Profile token usage per query; compress prompts; use smaller embedding models where recall is sufficient
- Latency budgets: Set p99 latency SLOs; profile each pipeline stage to identify bottlenecks

11.3 Memory & Conversation Handling

- Short-term memory: Current session's conversation history — injected directly into prompt
- Long-term memory: Persistent storage of past sessions — retrieved based on relevance to current query
- Episodic memory: Specific past interactions indexed by time/context
- Semantic memory: Distilled facts from past conversations stored as structured records
- Memory summarization: Use LLM to compress older conversation history — prevents context overflow
- Context carry-over: Extract entities and topics from current conversation and use to bias retrieval
- Multimodal memory: Store past visual inputs (images viewed) and their embeddings for future retrieval

11.4 Security & Governance

Security Threat Model for RAG Systems

Prompt Injection: Malicious content in retrieved documents instructs the LLM to deviate from its instructions. Mitigation: Input sanitization, system prompt hardening, separate instruction and context roles.

Retrieval Poisoning: An attacker inserts malicious documents into the knowledge base to manipulate retrieval. Mitigation: Document provenance verification, rate limiting on indexing APIs, content moderation on ingested docs.

Data Leakage: Sensitive documents retrieved and exposed to unauthorized users. Mitigation: Row-level security in vector DB, document-level ACL enforcement at retrieval time.

PII Exposure: Retrieved context contains PII (names, SSNs, email addresses). Mitigation: PII detection (Presidio) and redaction before indexing; or at retrieval time.

Model Inversion: Adversarial queries designed to extract training data from the LLM. Mitigation: Rate limiting, output filtering, differential privacy.

11.5 Common Failure Modes

Failure Mode	Description	Diagnosis	Mitigation
Context dilution	Too many retrieved chunks; key info buried	Answer quality drops with high K	Reduce K; use re-ranking; MMR
Irrelevant retrieval	Retrieved chunks don't address the query	Low Recall@K; high hallucination	Better chunking; re-rank; hybrid search

Over-retrieval	Retrieving too much, diluting the prompt	Long latency; low faithfulness	Tune K downward; passage compression
Under-retrieval	Missing key facts from retrieval	Answer incomplete or hallucinated	Query expansion; RAG-Fusion; broader K
Hallucinations	Model generates unsupported facts	Low faithfulness score	Grounding instructions; CRAG; fine-tune
Stale data	Knowledge base not updated	Outdated answers; user complaints	CDC pipeline; freshness-aware retrieval
Embedding mismatch	Query/doc embedding from different models	Retrieval quality collapse	Use same model for query and index
Prompt overflow	Context exceeds model's context window	API error or truncation	Compression; reduce K; summarize chunks
Modality misalignment	Wrong modality retrieved for query	Image retrieved for text question	Metadata filtering by modality
Visual hallucination	VLM claims non-existent image content	Low visual faithfulness	POPE eval; grounding fine-tuning

11.6 Model & Embedding Versioning

- Embedding versioning: Tag every chunk with the embedding model name and version used
- Forced re-indexing: When changing embedding models, all chunks must be re-embedded (versions are incompatible)
- Model versioning: Track generator model version per response for reproducibility
- Canary deployments: Route 5% of traffic to new model version; monitor metrics before full rollout
- Rollbacks: Maintain the previous embedding version in a secondary index for rapid rollback

MODULE 12

Capstone Projects & Career Preparation

2 Weeks / ~16 Hours Integrated

Topics covered:

- Capstone Project 1: Enterprise Document Intelligence System (text + image)
- Capstone Project 2: Video QA System with Multimodal RAG
- Capstone Project 3: Domain-Specific RAG with LoRA Fine-Tuning
- Portfolio presentation: structuring your RAG projects for job applications
- Job description analysis: mapping curriculum skills to AI role requirements
- System design interview prep: designing RAG systems under constraints
- Common interview questions on RAG, embeddings, and VLMs
- Open-source contributions and research directions

Module 12: Capstone Projects & Career Preparation

12.1 Capstone Projects

Capstone A: Enterprise Document Intelligence

Capstone A — Full-Stack Multimodal RAG

Build a production-quality multimodal document QA system for a chosen domain (legal, medical, financial, or technical).

Requirements: Handle PDFs with figures, tables, and equations. Support text and image queries. Implement document-level access control. Serve via a REST API with <2s p95 latency. Include observability (LangSmith or Arize Phoenix tracing).

Evaluation: RAGAS full suite + visual faithfulness + latency benchmarks

Deliverable: GitHub repo, live demo, 2-page system design document

Capstone B: Video Question-Answering with Multimodal RAG

Capstone B — Video QA System

Build a system that answers questions about a corpus of YouTube/conference talk videos.

Requirements: Transcribe and timestamp all videos (Whisper). Extract and embed key frames (CLIP). Build a unified text+visual index. Answer queries with timestamped citations (link to exact moment in video).

Deliverable: Gradio demo accepting natural language questions and returning answers with clickable video timestamp citations

Capstone C: Domain RAG with LoRA Fine-Tuning

Capstone C — Fine-Tuned Multimodal RAG

Pick a specialized domain (e.g., biomedical, legal, financial). Fine-tune both the embedding model and a lightweight VLM using LoRA/QLoRA. Integrate into a RAG pipeline and demonstrate measurable improvement over the base model baseline.
Deliverable: Fine-tuned weights on HuggingFace Hub + evaluation report + blog post or technical writeup

12.2 Career Preparation

Mapping Skills to Job Descriptions

Job Role Keyword	Required Skills from This Curriculum
RAG Engineer	Modules 1-9: full RAG stack, chunking, retrieval, evaluation
Multimodal AI Engineer	Modules 4, 7: CLIP, VLMs, cross-modal retrieval, image embedding
LLM Engineer	Modules 1, 8, 9: prompt engineering, generation, advanced RAG techniques
ML Engineer (NLP)	Modules 1-6, 8: full text RAG + embedding fine-tuning
AI Research Scientist	Modules 9, 10: Self-RAG, Graph-RAG, fine-tuning, PEFT/LoRA
MLOps / AI Platform	Module 11: RAGOps, observability, versioning, security, scalability

System Design Interview Framework for RAG

16. Clarify requirements: domain, query types, data volume, latency SLA, update frequency
17. Choose data modalities and ingestion pipeline: loaders, chunking, metadata
18. Select embedding strategy: text model + optional image/audio model; explain tradeoffs
19. Choose vector database: justify based on scale, filtering needs, managed vs. self-hosted
20. Design retrieval strategy: hybrid search, re-ranking, query expansion
21. Design prompt construction: context window budget, attribution, compression
22. Choose generation model: API vs. self-hosted, fine-tuning if needed
23. Design for production: caching, async, observability, access control, versioning
24. Define evaluation: retrieval metrics + generation metrics + E2E latency + cost

Common Interview Questions

- 'What is the difference between HNSW and IVF-PQ, and when would you use each?'
- 'How would you handle a domain where retrieval quality is poor due to jargon?'
- 'Explain how you would build a RAG system that handles both text and image queries'
- 'How would you detect and mitigate hallucinations in a production RAG system?'
- 'What is LoRA and how does it differ from full fine-tuning? When would you choose QLoRA?'
- 'How would you implement access control in a multi-tenant RAG system?'
- 'Walk me through how you would debug a RAG pipeline where answers are consistently wrong'
- 'How does CLIP work, and why is it useful for multimodal retrieval?'

Appendix: Resources & Research Papers

A. Foundational Papers

Paper	Authors	Year	Contribution
RAG for Knowledge-Intensive NLP Tasks	Lewis et al.	2020	Original RAG paper — the foundation
REALM: Retrieval-Enhanced LM Pretraining	Guu et al.	2020	End-to-end differentiable retrieval
Dense Passage Retrieval (DPR)	Karpukhin et al.	2020	Bi-encoder for passage retrieval
CLIP	Radford et al. (OpenAI)	2021	Cross-modal image-text alignment
LoRA	Hu et al.	2021	Low-rank adaptation for efficient fine-tuning
Self-RAG	Asai et al.	2023	Retrieval as a conditional model action
QLoRA	Dettmers et al.	2023	4-bit quantization + LoRA
LLaVA	Liu et al.	2023	Visual instruction tuning for VLMs
GraphRAG	Edge et al. (Microsoft)	2024	Graph-based community retrieval
RAPTOR	Sarthi et al.	2024	Tree-based hierarchical retrieval
ImageBind	Girdhar et al. (Meta)	2023	6-modality joint embedding space
CRAG	Yan et al.	2024	Corrective retrieval augmented generation

B. Essential Tools & Libraries

Tool	Purpose	URL
LangChain	RAG pipeline orchestration, agents, chains	langchain.com
LlamaIndex	Data framework for LLM applications, advanced RAG	docs.llamaindex.ai
Hugging Face Transformers	Models, fine-tuning, embedding models	huggingface.co
Hugging Face PEFT	LoRA, QLoRA, Prefix Tuning implementation	github.com/huggingface/peft
sentence-transformers	Text embedding models and fine-tuning	sbert.net
FAISS	Fast ANN search library (CPU + GPU)	github.com/facebookresearch/faiss

Qdrant	High-performance vector database	qdrant.tech
Weaviate	Hybrid search vector database	weaviate.io
RAGAS	RAG evaluation framework	docs.ragas.io
DeepEval	LLM evaluation framework	docs.confident-ai.com
LangSmith	LLM observability and tracing	smith.langchain.com
Arize Phoenix	RAG observability and evals	phoenix.arize.com
Unstructured.io	Universal document parsing	unstructured.io
bitsandbytes	4-bit/8-bit quantization for PyTorch	github.com/TimDettmers/bitsandbytes
OpenCLIP	Open-source CLIP implementation	github.com/mlfoundations/open_clip
Whisper	State-of-the-art speech transcription	github.com/openai/whisper

C. Benchmark Datasets

Dataset	Task	Modality	Use in Curriculum
MS MARCO	Passage retrieval	Text	Module 6: Retrieval benchmarking
BEIR	Heterogeneous retrieval benchmark (18 datasets)	Text	Module 6: Retrieval strategy comparison
Natural Questions (NQ)	Open-domain QA	Text	Module 1, 8: End-to-end RAG eval
TriviaQA	Trivia question answering	Text	Module 1: Basic RAG
HotpotQA	Multi-hop question answering	Text	Module 9: Multi-hop RAG
VQAv2	Visual question answering	Image+Text	Module 7, 8: Multimodal eval
NoCaps	Novel object captioning	Image+Text	Module 7: VLM evaluation
COCO	Image captioning + detection	Image	Module 4, 5: Image embedding
PubMedQA	Biomedical QA	Text	Module 10: Domain fine-tuning
MIMIC-CXR	Radiology report generation	Image+Text	Module 10: Medical multimodal RAG
POPE	VLM hallucination evaluation	Image+Text	Module 7, 8: Visual hallucination eval

MMMU	Massive Multidisciplinary Multimodal Understanding	Image+Text	Module 7, 8: Advanced VLM eval

D. Recommended Learning Path by Role

Role Target	Priority Modules	Capstone
LLM Application Engineer	1, 2, 3, 6, 8, 11	A
Multimodal AI Engineer	1, 4, 5, 6, 7, 8, 9	B
ML Research Scientist	1, 4, 6, 9, 10 + all papers	C
MLOps / Platform Engineer	1, 5, 11 + DevOps modules	A
Full-Stack AI Engineer	All modules	A + B

Multimodal RAG: Comprehensive Curriculum

A complete guide from RAG fundamentals to production multimodal AI systems