RAG & Retrievers

Comprehensive Knowledge Base

A Complete Guide to Retrieval-Augmented Generation and Retrieval Systems

1. Introduction to Retrieval-Augmented Generation (RAG)

1.1 What is RAG?

Retrieval-Augmented Generation (RAG) is a hybrid AI architecture that combines the strengths of information retrieval systems with the generative capabilities of large language models (LLMs). RAG enhances the quality and accuracy of AI-generated responses by grounding them in retrieved, relevant information from external knowledge sources.

The core principle of RAG is to augment the context window of an LLM with relevant documents or text chunks retrieved from a knowledge base, enabling the model to generate responses based on current, accurate, and domain-specific information rather than relying solely on its training data.

1.2 Why RAG is Important

- Mitigates Hallucinations: LLMs can generate plausible-sounding but incorrect information. RAG grounds responses in actual retrieved documents.
- Up-to-Date Information: Unlike static training data, RAG systems can access current information from updated knowledge bases.
- Domain-Specific Knowledge: RAG enables AI systems to leverage specialized knowledge bases without retraining the entire model.
- Cost-Effective: More economical than fine-tuning or training custom models for specific domains.
- Transparency: Retrieved sources can be cited, providing verifiable evidence for generated responses.
- Scalability: Knowledge bases can be updated and expanded without modifying the underlying LLM.

1.3 RAG Architecture Components

A typical RAG system consists of the following components:

1. Knowledge Base: Collection of documents, text chunks, or structured data
2. Embedding Model: Converts text into dense vector representations
3. Vector Database: Stores and indexes document embeddings for efficient retrieval
4. Retriever: Mechanism to find relevant documents based on query similarity
5. Language Model: Generates responses using retrieved context
6. Orchestration Layer: Manages the workflow between retrieval and generation

1.4 How RAG Works: The Pipeline

7. Query Processing: User query is converted into an embedding vector
8. Retrieval: Similar documents are retrieved from the vector database using the retriever
9. Context Assembly: Retrieved documents are ranked and formatted
10. Prompt Construction: Query and retrieved context are combined into a prompt
11. Generation: LLM generates response based on the augmented prompt

12. Response Delivery: Generated answer is returned to the user, optionally with citations

2. Types of RAG Systems

2.1 Naive RAG

Description: The simplest RAG implementation follows a straightforward retrieve-then-generate pattern.

Process:
- Index documents into chunks
- Embed chunks and store in vector database
- Retrieve top-k similar chunks for a query
- Pass chunks to LLM for generation

Use Cases: Simple Q&A systems, document search, basic chatbots

Limitations: Poor retrieval quality, context fragmentation, no query optimization

2.2 Advanced RAG

Description: Incorporates pre-retrieval and post-retrieval optimizations to improve accuracy and relevance.

Enhancements:
- Query transformation and expansion
- Hybrid retrieval (combining multiple retrieval methods)
- Re-ranking of retrieved documents
- Context compression and filtering
- Chunking strategies optimization

Use Cases: Enterprise search, technical documentation systems, customer support

2.3 Modular RAG

Description: Highly flexible architecture with specialized modules that can be combined and configured based on use case requirements.

Components:
- Routing: Directs queries to appropriate retrievers or knowledge sources
- Query Construction: Converts natural language to structured queries
- Multi-Hop Retrieval: Performs iterative retrieval for complex queries
- Memory Integration: Maintains conversation history and context
- Tool Augmentation: Integrates APIs, databases, and external tools

Use Cases: Complex research applications, multi-domain knowledge systems, agent-based applications

2.4 Agentic RAG

Description: Combines RAG with autonomous agents that can reason, plan, and execute multi-step workflows.

Capabilities:
- Self-directed information gathering

- Dynamic query reformulation
- Multi-source synthesis
- Adaptive retrieval strategies
- Action execution based on retrieved information

Use Cases: Research assistants, automated analysis systems, decision support systems

3. Comprehensive Guide to Retriever Types

Retrievers are the core components of RAG systems responsible for finding relevant information from knowledge bases. This chapter covers all major retriever types, their characteristics, and optimal use cases.

3.1 Dense Retrievers (Vector/Semantic Search)

3.1.1 Overview

Dense retrievers use neural embeddings to represent both queries and documents as dense vectors in a high-dimensional space. Retrieval is performed by computing similarity (typically cosine similarity or dot product) between query and document vectors.

3.1.2 How It Works

- Documents are encoded into dense vectors using embedding models (e.g., BERT, Sentence Transformers)
- Vectors are stored in vector databases (Pinecone, Weaviate, FAISS, ChromaDB, Qdrant)
- Query is encoded into the same vector space
- Approximate Nearest Neighbor (ANN) algorithms find most similar documents

3.1.3 Key Characteristics

Aspect	Details
Strengths	Captures semantic meaning, handles synonyms, works across languages, robust to paraphrasing
Weaknesses	Computationally expensive, requires quality embeddings, black box nature, struggles with rare entities
Latency	Medium (10-100ms depending on index size and ANN algorithm)

3.1.4 Use Cases

- Semantic search where meaning matters more than exact matches
- Question answering systems
- Cross-lingual information retrieval
- Document similarity and recommendation
- General-purpose RAG applications

3.1.5 Popular Models

- OpenAI text-embedding-3-small/large
- all-MiniLM-L6-v2 (Sentence Transformers)
- BGE-M3, BGE-large
- E5-large, E5-mistral
- Cohere embed-v3

3.2 Sparse Retrievers (Lexical/Keyword Search)

3.2.1 Overview

Sparse retrievers use traditional information retrieval methods based on term frequency and statistical matching. They represent documents and queries as sparse vectors where most dimensions are zero.

3.2.2 Main Algorithms

BM25 (Best Matching 25):

- Most popular sparse retrieval algorithm
- Probabilistic ranking function based on term frequency (TF) and inverse document frequency (IDF)
- Includes document length normalization

TF-IDF (Term Frequency-Inverse Document Frequency):

- Classic information retrieval method
- Weights terms by frequency in document and rarity across corpus

3.2.3 Key Characteristics

Aspect	Details
Strengths	Fast, interpretable, excellent for exact matches, no training required, handles rare terms well
Weaknesses	Cannot understand semantics, vocabulary mismatch problem, poor with synonyms, language-dependent
Latency	Very fast (1-10ms with proper indexing)

3.2.4 Use Cases

- Code search where exact identifier matching is crucial
- Legal and compliance documents requiring precise term matching
- Technical documentation with specific terminology
- Product catalogs with model numbers and SKUs
- Scientific literature with precise terminology
- First-stage retrieval in hybrid systems

3.3 Hybrid Retrievers

3.3.1 Overview

Hybrid retrievers combine sparse and dense retrieval methods to leverage the strengths of both approaches. They typically use score fusion techniques to merge results from multiple retrieval systems.

3.3.2 Fusion Techniques

Reciprocal Rank Fusion (RRF):

- Combines rankings from different retrievers
- Formula: RRF(d) = Σ 1/(k + rank(d)) where k is a constant (typically 60)
- Robust and doesn't require score normalization

Linear Combination:

- Weighted sum of normalized scores
- Score = α × sparse_score + (1-α) × dense_score
- Requires careful weight tuning

Learned Fusion:

- Machine learning models trained to combine scores
- Can capture complex interaction patterns

3.3.3 Key Characteristics

Aspect	Details
Strengths	Best of both worlds, improved recall and precision, robust across query types, handles diverse content
Weaknesses	Higher complexity, increased latency, requires parameter tuning, more infrastructure
Latency	Medium (sum of both retrieval methods, typically 20-150ms)

3.3.4 Use Cases

- Enterprise search across diverse document types
- E-commerce product search (combining exact and semantic matches)
- Medical and healthcare information systems
- Multi-domain knowledge bases
- Production RAG systems requiring high accuracy

3.4 Multi-Vector Retrievers

3.4.1 Overview

Multi-vector retrievers represent documents using multiple embedding vectors rather than a single vector. This allows capturing different aspects, sections, or representations of the same document.

3.4.2 Approaches

ColBERT (Contextualized Late Interaction over BERT):

- Creates embeddings for each token in the document
- Uses MaxSim operation for scoring (maximum similarity between query and document tokens)
- Balances effectiveness and efficiency

Proposition-based Retrieval:

- Breaks documents into atomic propositions
- Each proposition is embedded separately
- Improves granularity and precision

3.4.3 Use Cases

- Long documents requiring fine-grained retrieval
- Complex queries needing partial matching
- Multi-aspect documents (e.g., research papers with methods, results, conclusions)
- High-precision retrieval requirements

3.5 Re-ranker/Cross-Encoder Retrievers

3.5.1 Overview

Re-rankers are not standalone retrievers but are used as a second-stage refinement step. They take the initial retrieved results and re-order them using more sophisticated scoring methods.

3.5.2 How It Works

- Cross-encoder models process query and document together
- Computes relevance score through joint encoding
- More expensive but more accurate than bi-encoder approaches
- Applied to top-k results from first-stage retrieval (typically k=20-100)

3.5.3 Popular Re-ranker Models

- Cohere Rerank
- bge-reranker-large
- ms-marco-MiniLM
- MonoT5

3.5.4 Use Cases

- Improving precision of initial retrieval results
- High-stakes applications requiring maximum accuracy
- Legal and compliance systems
- Customer support with quality requirements

3.6 Contextual Retrieval

3.6.1 Overview

Contextual retrieval augments document chunks with surrounding context before embedding, improving retrieval accuracy by reducing ambiguity and providing more complete information.

3.6.2 How It Works

- LLM generates context for each chunk (document summary, section headers, metadata)
- Context is prepended to chunk before embedding
- Original chunk (without context) is stored for generation
- Retrieval uses context-enhanced embeddings

3.6.3 Use Cases

- Long documents where chunks lose meaning in isolation
- Technical documentation with nested structure
- Multi-chapter books and reports
- Reducing retrieval failures from context loss

3.7 Parent Document Retriever

3.7.1 Overview

Retrieves on smaller chunks but returns larger parent documents to the LLM, balancing precise retrieval with comprehensive context.

3.7.2 How It Works

- Documents are split into both small chunks (for retrieval) and larger parent chunks (for context)
- Small chunks are embedded and indexed
- When small chunk matches, its parent document is retrieved
- Maintains mapping between child and parent chunks

3.7.3 Use Cases

- When precise retrieval but broad context is needed
- Hierarchical documents (sections within chapters)
- Avoiding context fragmentation
- Improving LLM reasoning with fuller context

3.8 Self-Query Retriever

3.8.1 Overview

Uses an LLM to decompose user queries into semantic search components and metadata filters, enabling structured retrieval from vector stores.

3.8.2 How It Works

- LLM parses natural language query
- Extracts semantic search query and metadata filters
- Example: 'Papers about transformers from 2020' → query='transformers', filter={year: 2020}
- Applies both semantic search and filters to vector database

3.8.3 Use Cases

- Structured data with rich metadata (date, author, category, tags)
- Research paper databases
- Product catalogs with attributes
- Content management systems with taxonomies

3.9 Ensemble Retriever

3.9.1 Overview

Combines results from multiple different retrievers (not just sparse + dense) using various aggregation strategies.

3.9.2 Possible Combinations

- Multiple embedding models (different perspectives)
- Different chunking strategies
- Specialized domain retrievers
- Graph-based and vector-based retrievers

3.9.3 Use Cases

- Maximum recall requirements
- Multi-domain systems with specialized retrievers
- Research and discovery applications

3.10 Multi-Query Retriever

3.10.1 Overview

Uses an LLM to generate multiple variations of the original query, retrieves documents for each variation, and combines results.

3.10.2 How It Works

- LLM generates 3-5 query variations with different phrasings
- Each variation is used for retrieval independently
- Results are combined and deduplicated
- Improves recall by covering different query formulations

3.10.3 Use Cases

- Ambiguous queries that could be interpreted multiple ways
- Improving recall for complex questions
- Research applications needing comprehensive coverage

3.11 Time-Weighted Retriever

3.11.1 Overview

Incorporates temporal relevance by boosting recent documents in retrieval ranking.

3.11.2 How It Works

- Documents tagged with timestamps
- Recency score calculated using decay function
- Final score = semantic_similarity × time_decay
- Configurable decay rate based on domain

3.11.3 Use Cases

- News and current events
- Social media and trending content
- Time-sensitive documentation (software updates, policies)
- Conversation history and chat applications

3.12 Graph RAG / Knowledge Graph Retriever

3.12.1 Overview

Uses knowledge graphs to represent relationships between entities, enabling retrieval based on graph structure in addition to semantic similarity.

3.12.2 How It Works

- Documents processed to extract entities and relationships
- Knowledge graph constructed (Neo4j, Amazon Neptune, etc.)
- Retrieval combines vector similarity and graph traversal
- Can answer multi-hop reasoning questions

3.12.3 Key Advantages

- Explicit relationship modeling
- Multi-hop reasoning capabilities
- Better handling of complex queries requiring relationship understanding
- Explainable retrieval paths

3.12.4 Use Cases

- Enterprise knowledge management with complex relationships
- Scientific research with citation networks
- Fraud detection and network analysis
- Recommendation systems

3.13 SQL Database Retriever

3.13.1 Overview

Converts natural language queries into SQL statements to retrieve structured data from relational databases.

3.13.2 How It Works

- LLM translates natural language to SQL
- Schema information provided to LLM for context
- SQL query executed against database
- Results formatted and returned

3.13.3 Use Cases

- Business intelligence and analytics
- Customer data queries
- Inventory and logistics systems
- Financial reporting and analysis

3.14 Long Context Retriever (No Traditional Retrieval)

3.14.1 Overview

With modern LLMs supporting large context windows (100K-1M+ tokens), an alternative approach is to include entire documents in the context without traditional retrieval.

3.14.2 When to Use

- Small to medium knowledge bases (fits within context window)
- High accuracy requirements where all context is needed
- Dynamic or frequently changing content
- Simplifying architecture by eliminating retrieval infrastructure

3.14.3 Considerations

- Cost: Increases proportionally with context size
- Latency: Longer processing time for large contexts
- Lost in the middle: LLMs may struggle to use all information in very long contexts

4. Retriever Selection Decision Framework

Choosing the right retriever depends on multiple factors. Use this decision framework to guide your selection.

4.1 Decision Matrix

Requirement	Priority	Recommended Retriever
Semantic understanding	High	Dense Retriever
Exact term matching	High	Sparse Retriever (BM25)
Maximum accuracy	Critical	Hybrid + Re-ranker
Low latency	Critical	Sparse Retriever (BM25)
Structured data	High	SQL/Self-Query Retriever
Complex relationships	High	Graph RAG
Time-sensitive content	High	Time-Weighted Retriever
Small knowledge base	N/A	Long Context (No Retrieval)

5. Implementation Best Practices

5.1 Evaluation Metrics

Retrieval Metrics:

- Recall@K: Percentage of relevant documents in top K results
- Precision@K: Percentage of retrieved documents that are relevant
- MRR (Mean Reciprocal Rank): Average of reciprocal ranks of first relevant result
- NDCG (Normalized Discounted Cumulative Gain): Measures ranking quality

End-to-End Metrics:

- Answer accuracy (exact match, F1 score)
- Faithfulness (generated answer grounded in retrieved context)
- Relevance (answer addresses the question)
- Latency and throughput

5.2 Common Pitfalls

- Poor chunking strategies (too large or too small chunks)
- Not considering hybrid approaches when appropriate
- Ignoring metadata and structured information
- Insufficient evaluation and testing
- Over-engineering for simple use cases
- Not monitoring and iterating on retrieval quality

5.3 Optimization Strategies

- Start simple (dense retriever) and add complexity as needed
- Experiment with different chunk sizes (256, 512, 1024 tokens)
- Use overlapping chunks to preserve context at boundaries
- Implement re-ranking for high-value use cases
- Leverage metadata filtering to narrow search space
- Monitor failure cases and iterate on retrieval strategy
- Cache frequently accessed documents
- Batch queries when possible for efficiency

6. Conclusion and Summary

RAG represents a powerful paradigm for building AI systems that combine the reasoning capabilities of large language models with access to external knowledge. The choice of retriever is crucial to RAG system performance and should be guided by:

- Nature of your data (structured vs. unstructured, relationships, temporality)
- Query characteristics (semantic vs. exact match needs)
- Performance requirements (latency, accuracy, cost)
- Scale and complexity of the knowledge base

This knowledge base provides a comprehensive foundation for understanding and selecting retrievers for your RAG applications. Remember that the field is rapidly evolving, and new techniques emerge regularly. Always evaluate multiple approaches with your specific data and use cases to find the optimal solution.

Quick Reference Summary

- Starting a new project? Begin with Dense Retriever (simple, effective)
- Need production-grade accuracy? Use Hybrid Retriever + Re-ranker
- Have structured data? Consider Self-Query or SQL Retriever
- Complex relationships matter? Explore Graph RAG
- Very small knowledge base? Skip retrieval and use Long Context
- Always measure and iterate based on real performance data