RAG - test pdf

Tuesday, 10 February 2026 5:28 PM

1. Purpose of This Module

This module introduces **Retrieval-Augmented Generation (RAG)** from first principles.
It defines the problem RAG solves, explains why traditional LLM usage is insufficient for real-world applications, and establishes the base RAG architecture that all later modules build upon.
By the end of this module, the reader will:

- Understand the limitations of standalone LLMs
- Understand why RAG is required
- Be able to describe the minimal RAG pipeline
- Identify the base RAG type used throughout this documentation

2. Background: Limitations of Large Language Models

Large Language Models (LLMs) are probabilistic models trained to predict the next token based on prior context. While they demonstrate strong reasoning and language capabilities, they have **fundamental constraints**:

2.1 Knowledge Constraints

- Knowledge is fixed at training time
- No access to private or proprietary data
- No awareness of real-time or updated information

2.2 Reliability Constraints

- High risk of hallucination when information is missing
- Answers may sound confident but be factually incorrect
- No built-in verification mechanism

2.3 Scalability Constraints

- Cannot ingest large documents directly due to context window limits
- Prompt-based data injection does not scale

These limitations make standalone LLMs unsuitable for applications that require **accurate, up-to-date, and source-grounded responses**.

3. Problem Statement

Problem:
How can an LLM answer questions using information that is:

- Not part of its training data
- Private or organization-specific
- Too large to fit into a single prompt

Traditional approaches such as prompt stuffing or fine-tuning are insufficient due to cost, latency, and maintainability issues.

4. Definition of Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a system architecture that enhances an LLM by retrieving relevant external information at query time and supplying it as context for response generation.

Formal Definition

RAG is a pipeline that combines information retrieval with text generation, enabling language models to produce responses grounded in externally retrieved data.

5. Core Conceptual Insight

Instead of requiring an LLM to *store* all knowledge internally, RAG enables the model to:

1. Retrieve relevant information from an external knowledge source
2. Generate an answer using only the retrieved information and the user query

This mirrors how a human expert consults reference material before responding.

6. Minimal RAG Architecture

The minimal RAG system consists of the following logical stages:

User Query
↓
Information Retrieval
↓
Context Assembly
↓
Language Model Generation
↓
Final Response

Each stage is mandatory. Advanced RAG systems extend or optimize these stages but do not remove them.

7. Explanation of “Retrieval” and “Augmentation”

7.1 Retrieval

Retrieval is the process of identifying and selecting relevant information from an external knowledge base based on a user query.

7.2 Augmentation

Augmentation refers to injecting the retrieved information into the language model’s input context at generation time.
The language model itself remains unchanged and untrained.

8. What RAG Is Not

RAG should not be confused with:

- Model training or fine-tuning
- Persistent memory mechanisms
- Traditional database querying
- Search engines that return raw documents

RAG is a **context injection architecture**, not a learning mechanism.

9. Base RAG Type Used in This Documentation

This documentation focuses initially on:

Document-Based Naive RAG

Characteristics:

- Static document corpus (PDF, DOCX, TXT)
- One-pass retrieval per query
- No conversational memory
- No agents or tool invocation

This base RAG serves as the foundation for understanding all advanced RAG variants.

10. Summary

- LLMs alone cannot reliably answer questions about external or private data
- RAG addresses this limitation by introducing retrieval at inference time
- The minimal RAG pipeline consists of retrieval followed by generation
- All advanced RAG architectures are extensions of this base model

11. Next Module

Module 2: Document Ingestion and Knowledge Preparation

This module will cover:

- Document formats
- Text extraction
- Cleaning and normalization
- Chunking strategies
- Metadata design

1. Purpose of This Module

This module defines how **raw data** is transformed into **RAG-ready knowledge**.
It focuses on document ingestion, text extraction, normalization, chunking, and metadata design.
By the end of this module, the reader will:

- Understand what constitutes a document in RAG
- Know how documents are ingested and cleaned
- Understand chunking strategies and their trade-offs
- Be able to design metadata for reliable retrieval
- Prepare a corpus suitable for embedding and retrieval

2. What Is a “Document” in RAG?

In RAG, a **document** is any source of information that can be converted into text and used as retrievable knowledge.

2.1 Common Document Sources

- PDF files
- Word documents (DOCX)
- Text files (TXT, Markdown)
- HTML web pages
- Knowledge base articles
- Logs, FAQs, manuals

2.2 Logical vs Physical Documents

- **Physical document**: the original file (e.g., a PDF)
- **Logical document**: the extracted textual content used by the RAG system

- **Logical document:** the extracted textual content used by the RAG system

RAG systems operate on **logical documents**, not files.

3. Document Ingestion Pipeline

Document ingestion is the process of converting raw files into clean, structured text.

3.1 Standard Ingestion Stages

Raw File
↓
Text Extraction
↓
Cleaning & Normalization
↓
Segmentation (Chunking)
↓
Metadata Attachment

Each stage directly affects retrieval accuracy and answer quality.

4. Text Extraction

4.1 Purpose

To extract human-readable text from documents while preserving meaning.

4.2 Extraction Considerations

- PDFs may contain layout noise (headers, footers, page numbers)
- Tables and multi-column layouts may require special handling
- Scanned documents may require OCR

4.3 Extraction Output

The output should be:

- Plain text
- Ordered logically
- Free from repeated or irrelevant elements

Poor extraction quality leads to poor embeddings and retrieval failure.

5. Cleaning and Normalization

5.1 Why Cleaning Is Required

Raw extracted text often contains:

- Broken lines
- Redundant whitespace
- Page artifacts
- Encoding issues

5.2 Common Cleaning Steps

- Remove headers and footers
- Normalize whitespace
- Fix broken sentences
- Remove non-informative symbols
- Standardize punctuation

The goal is **semantic clarity**, not visual fidelity.

6. Chunking: Splitting Documents into Units of Meaning

6.1 Why Chunking Is Necessary

- LLMs have limited context windows
- Retrieval operates on small text units
- Smaller units improve retrieval precision

A RAG system does not retrieve full documents—it retrieves **chunks**.

7. Chunking Strategies

7.1 Fixed-Size Chunking

- Text split by character or token count
- Simple and fast
- May split sentences or ideas

Use case: baseline systems, large homogeneous text

7.2 Sentence-Based Chunking

- Splits on sentence boundaries
- Preserves semantic units
- Slightly more complex

Use case: explanatory or narrative documents

7.3 Semantic Chunking

- Splits based on topic or meaning
- Maintains conceptual coherence
- Computationally expensive

Use case: high-accuracy enterprise RAG

7.4 Overlapping Chunks

- Adjacent chunks share some content
- Prevents context loss at boundaries

Trade-off:

- Higher storage and embedding cost
- Better answer completeness

8. Chunk Size Trade-offs

Chunk Size	Advantages	Disadvantages
Small	High precision	May lose context
Medium	Balanced	Requires tuning
Large	Better context	Lower retrieval accuracy

There is **no universal optimal chunk size**.
Chunking must align with:

- Document type
- Query complexity
- Model context limits

9. Metadata Design

9. Metadata Design

9.1 What Is Metadata?
Metadata is structured information attached to each chunk.

9.2 Common Metadata Fields
- Source document name
- Page or section number
- Title or heading
- Creation date
- Category or tag

9.3 Why Metadata Matters
- Enables filtered retrieval
- Improves answer traceability
- Supports citations and auditing

A chunk without metadata is operationally weak.

10. Output of the Knowledge Preparation Stage
At the end of this module, the system should produce:
- Clean text chunks
- Each chunk representing a coherent unit of meaning
- Each chunk paired with metadata
- Ready for embedding and storage

This output becomes the **input to the embedding pipeline**.

11. Common Failure Modes
- Overly large chunks → irrelevant retrieval
- Overly small chunks → fragmented answers
- No metadata → poor filtering and traceability
- Dirty text → meaningless embeddings

Most RAG accuracy issues originate here.

12. Summary
- RAG operates on text, not files
- Document ingestion quality directly determines system performance
- Chunking is a critical design decision
- Metadata is essential for robust retrieval and explainability
- Proper knowledge preparation is foundational to all RAG systems

1. Purpose of This Module
This module explains how text is transformed into **numerical representations (embeddings)** that enable semantic search. Embeddings are the technical foundation that makes retrieval in RAG possible.
By the end of this module, the reader will:
- Understand what embeddings are and why they are required
- Know how embeddings encode semantic meaning

- Understand embedding dimensions and similarity metrics
- Be able to choose appropriate embedding models
- Produce embedding-ready outputs for vector storage

2. Why Embeddings Are Necessary

Traditional text search relies on **exact keyword matching**, which fails when:

- Different words have similar meanings
- Queries are phrased differently from documents
- Context matters more than literal terms

RAG requires **semantic retrieval**, not keyword search.

Embeddings solve this by converting text into vectors that capture meaning rather than surface form.

3. Definition of an Embedding

An **embedding** is a fixed-length numerical vector that represents the semantic meaning of a piece of text.

Formal Definition

> An embedding is a dense vector representation of text in a high-dimensional space where semantically similar texts are located closer together.

4. Intuition Behind Embeddings

Conceptually:

- Each dimension in an embedding represents a latent semantic feature
- Similar meanings → similar vectors
- Dissimilar meanings → distant vectors

For example:

- “attendance policy” and “leave rules” will have embeddings close to each other
- “attendance policy” and “network security” will be far apart

This spatial property enables semantic retrieval.

5. Embedding Granularity in RAG

Embeddings are generated for:

- Individual **chunks**, not full documents
- Queries submitted by users

Key Principle

> Retrieval works by comparing **query embeddings** with **chunk embeddings**.

This comparison determines which chunks are relevant.

6. Embedding Vector Dimensions

6.1 What Is Dimensionality?

Dimensionality refers to the number of numerical values in an embedding vector.

Examples:

- 384 dimensions
- 768 dimensions
- 1024+ dimensions

6.2 Trade-offs

Dimensionality	Advantage	Disadvantage
Lower	Faster, cheaper	Less expressive
Higher	Better semantic nuance	More memory and compute

Higher dimensionality generally improves retrieval quality but increases cost.

7. Embedding Models

7.1 Characteristics of a Good Embedding Model

- Semantic consistency
- Domain robustness
- Stable vector space
- Efficient inference

7.2 Categories of Embedding Models

- General-purpose text embeddings
- Domain-specific embeddings
- Multilingual embeddings

The embedding model choice must align with:

- Document domain
- Language requirements
- Performance constraints

8. Similarity Metrics

Once embeddings are generated, similarity must be measured.

8.1 Common Similarity Measures

Cosine Similarity

- Measures angular similarity
- Most commonly used in RAG
- Scale-invariant

Dot Product

- Measures vector alignment and magnitude
- Faster but sensitive to vector norms

Euclidean Distance (L2)

- Measures absolute distance
- Less commonly used for text embeddings

8.2 Selection Guidance

Cosine similarity is the default choice for most RAG systems due to its stability and interpretability.

9. Embedding Normalization

Normalization ensures:

- Consistent vector magnitude
- More reliable similarity computation

Some embedding models output normalized vectors by default. Others require explicit normalization.

Failure to normalize can degrade retrieval quality.

10. Embedding Consistency Requirement

A critical constraint in RAG systems:

The same embedding model must be used for both documents and queries.

Mixing embedding models results in incompatible vector spaces and invalid similarity comparisons.

11. Common Failure Modes

- Embedding entire documents instead of chunks
- Using different models for queries and documents
- Ignoring normalization requirements
- Choosing embeddings not suited for the domain
- Overly large or noisy chunks reducing embedding quality

Most retrieval issues trace back to incorrect embedding decisions.

12. Output of This Module

At the end of this module, the system should have:

- A set of text chunks
- One embedding vector per chunk
- A consistent embedding model selected
- Embedding vectors ready for indexing in a vector database

This output feeds directly into the retrieval layer.

13. Summary

- Embeddings convert text into semantic vectors
- Retrieval depends on vector similarity, not keywords
- Dimensionality and model choice affect quality and cost
- Consistency between query and document embeddings is mandatory
- Embeddings form the backbone of all RAG systems

1. Purpose of This Module

This module explains how embedded knowledge is **stored, indexed, and retrieved** efficiently using vector databases. It formalizes the retrieval step of the RAG pipeline and introduces the mechanisms that determine *which* information reaches the language model.

By the end of this module, the reader will:

- Understand what a vector database is and why it is required
- Know how similarity search works at scale
- Understand indexing strategies and retrieval parameters
- Be able to design a reliable retrieval layer
- Produce ranked, relevant chunks for downstream generation

2. Why a Vector Database Is Required

After Module 3, the system has:

- Thousands to millions of embedding vectors
- Each vector representing a text chunk

A traditional relational or document database cannot efficiently answer the question:

“Which vectors are *closest in meaning* to this query vector?”

Which vectors are *closest in meaning* to this query vector?

Vector databases are purpose-built to solve this problem.

3. Definition of a Vector Database

A **vector database** is a data store optimized for:

- High-dimensional vector storage
- Approximate or exact nearest-neighbor search
- Low-latency similarity queries
- Metadata-aware filtering

Core Responsibilities

- Store vectors + metadata
- Build search indexes
- Execute similarity queries efficiently

4. Data Model in a Vector Database

Each stored record typically consists of:

- **ID**: Unique identifier
- **Vector**: Embedding representation
- **Payload / Metadata**: Source, page, tags, timestamps

Conceptually:

```
{
 id: "chunk_123",
 vector: [v1, v2, v3, ... vn],
 metadata: {
  source: "policy.pdf",
  page: 5,
  section: "Medical Leave"
 }
}
```

Metadata is not optional in production systems.

5. Similarity Search Fundamentals

5.1 Query Flow

```
User Query
  ↓
Query Embedding
  ↓
Similarity Search
  ↓
Top-K Relevant Chunks
```

The goal is to retrieve the **K most semantically similar chunks**.

5.2 Nearest Neighbor Search

Similarity search identifies vectors closest to the query vector using a chosen distance metric (cosine, dot product, or L2).

Because exact comparison against all vectors is expensive at scale, most systems use **Approximate Nearest Neighbor (ANN)** techniques.

6. Indexing Strategies

Indexes accelerate similarity search by organizing vectors.

6.1 Flat (Brute Force) Index

- Compares query against all vectors
- Exact results
- Poor scalability

Use case: small datasets, prototyping

6.2 Inverted File Index (IVF)

- Groups vectors into clusters
- Searches within relevant clusters only
- Faster with minimal accuracy loss

Use case: medium to large datasets

6.3 Graph-Based Indexes (e.g., HNSW)

- Builds a navigable graph of vectors
- Fast and high recall
- Higher memory usage

Use case: production-grade RAG systems

7. Retrieval Parameters

7.1 Top-K Retrieval

Determines how many chunks are returned.

- Low K → high precision, low recall
- High K → higher recall, more noise

Typical values: **3–10**

7.2 Score Thresholding

Filters out chunks below a similarity score.

Benefits:

- Prevents irrelevant context
- Improves answer grounding

7.3 Metadata Filtering

Restricts retrieval to subsets of data.

Examples:

- Source = “policy”
- Date > 2023
- Department = “HR”

Metadata filtering improves relevance and compliance.

8. Retrieval Output

The retriever returns:

- A ranked list of chunks
- Each with similarity scores
- [illegible]

- Each with metadata

This output is **not yet an answer**.
It is **raw context** for the language model.

9. Retrieval Quality Considerations

Good retrieval must balance:

- Relevance
- Diversity
- Context completeness

Poor retrieval leads to:

- Hallucinations
- Partial answers
- Overconfidence on wrong data

In RAG systems:

> **Retrieval quality often matters more than model choice.**

10. Common Failure Modes

- Retrieving too many irrelevant chunks
- Ignoring metadata filtering
- Using incompatible similarity metrics
- Over-reliance on Top-K without re-ranking
- Treating retrieval as a one-time design decision

Retrieval requires tuning and iteration.

11. Role of Retrieval in RAG Types

At this stage, the system still implements:

Naive Document RAG

Advanced RAG types (multi-hop, conversational, agentic) extend this retrieval layer but do not replace it.

12. Output of This Module

At the end of this module, the system produces:

- Ranked, relevant chunks
- Similarity scores
- Metadata-enriched context

This output feeds directly into the **generation layer**.

13. Summary

- Vector databases enable efficient semantic search
- Retrieval is based on nearest-neighbor similarity
- Indexing strategies trade accuracy for speed
- Top-K, thresholds, and filters strongly affect performance
- Retrieval quality is critical to RAG success

1. Purpose of This Module

This module explains how retrieved information is transformed into a **controlled input context** for a language model and how prompts are designed to ensure responses are **grounded, accurate, and verifiable**.

By the end of this module, the reader will:

- Understand how retrieved chunks are assembled into context
- Know how context size and ordering affect outputs
- Be able to design RAG-specific prompt templates
- Apply grounding techniques to reduce hallucinations
- Manage context window constraints effectively

2. Position of This Module in the RAG Pipeline

At this stage, the system has already:

- Ingested and chunked documents
- Embedded chunks
- Retrieved relevant chunks via similarity search

This module governs what happens **between retrieval and generation**.

Retrieved Chunks
↓
Context Construction
↓
Prompt Assembly
↓
LLM Generation

3. What Is "Context" in RAG?

In RAG, **context** refers to the structured text supplied to the LLM alongside the user query.

Context typically includes:

- Retrieved chunks (primary knowledge)
- Instructions or system rules
- The user's question

The LLM's response quality is highly sensitive to how this context is constructed.

4. Context Assembly Strategy

4.1 Chunk Selection

Only the most relevant chunks should be included.

Considerations:

- Similarity score
- Redundancy between chunks
- Complementary information

Including irrelevant chunks increases hallucination risk.

4.2 Chunk Ordering

Chunks should be ordered logically, typically by:

1. Similarity score (highest first)
2. Source hierarchy (same document grouped)
3. Chronological or structural order (if relevant)

3. Chronological or structural order (if relevant)

Poor ordering can confuse the model even if the right information is present.

4.3 Chunk Formatting

Chunks should be clearly separated and labeled.
Example structure (conceptual):
[Source: policy.pdf | Page: 5]
<chunk text>
[Source: policy.pdf | Page: 6]
<chunk text>
Explicit boundaries improve grounding and citation.

5. Context Window Constraints

5.1 What Is a Context Window?

The context window is the maximum number of tokens the LLM can process in a single request.
The total input includes:

- System prompt
- Instructions
- Retrieved context
- User query

Exceeding this limit causes truncation or failure.

5.2 Context Budgeting

A practical RAG system allocates tokens intentionally.
Typical priorities:

1. Retrieved knowledge
2. User query
3. Instructions

Context budgeting often requires:

- Limiting Top-K
- Reducing chunk size
- Removing redundant text

6. Prompt Design for RAG

6.1 Why RAG Prompts Are Special

Standard prompts encourage creativity.
RAG prompts must encourage **faithfulness to context**.
The prompt must:

- Restrict the model to retrieved information
- Explicitly forbid unsupported assumptions

6.2 Core Components of a RAG Prompt

A well-structured RAG prompt typically contains:

1. **System Instruction**
   - Defines the model’s role and constraints
2. **Context Block**
   - Injected retrieved chunks
3. **User Query**

    - The question to answer

6.3 Grounding Instructions

Grounding instructions explicitly limit the model’s behavior.
Common grounding rules:

- “Answer only using the provided context”
- “If the answer is not present, state that it is unavailable”
- “Do not use external knowledge”

These instructions significantly reduce hallucinations.

7. Handling “No Answer Found” Scenarios

A robust RAG system must handle cases where retrieval fails.
Design options:

- Return a fallback response
- Ask the user to rephrase
- Indicate insufficient information

Silently hallucinating an answer is unacceptable in production systems.

8. Citations and Traceability

Context formatting and metadata enable:

- Source attribution
- Auditing
- Trust and explainability

Even if citations are not displayed to users, they are critical internally.

9. Common Failure Modes

- Overloading the context window
- Mixing instructions with knowledge
- Weak grounding language
- Including low-relevance chunks
- Allowing the model to answer beyond context

Most hallucination issues originate here, not in retrieval.

10. Output of This Module

At the end of this module, the system can:

- Assemble retrieved chunks into a coherent context
- Construct RAG-specific prompts
- Control LLM behavior through instructions
- Generate answers grounded strictly in retrieved data

This completes the **core RAG pipeline**.

11. Summary

- Context construction directly impacts answer quality
- Prompt design in RAG prioritizes faithfulness over creativity
- Grounding instructions are mandatory for reliability
- Context window limits require deliberate budgeting
- This layer enforces correctness in RAG systems

1. Purpose of This Module

This module integrates all prior components into a **single, coherent, end-to-end RAG system**. It defines the complete architecture, data flow, and operational lifecycle for a **document-based (naive) RAG** implementation.

By the end of this module, the reader will:

- Understand the full RAG pipeline from ingestion to response
- Be able to design a complete system architecture
- Identify component responsibilities and interfaces
- Understand runtime and offline workflows
- Establish a reference RAG system suitable for extension

2. Scope of the System

The system built in this module supports:

- Static document ingestion (PDF, DOCX, TXT)
- Semantic search over document content
- Question answering grounded strictly in documents
- Single-turn (non-conversational) queries

This system serves as the **baseline RAG architecture** upon which all advanced RAG types are built.

3. High-Level System Architecture

3.1 Architectural Overview

```
Offline Pipeline (Indexing)
--------------------------
Documents
  ↓
Text Extraction & Cleaning
  ↓
Chunking + Metadata
  ↓
Embedding Generation
  ↓
Vector Database (Index)

Online Pipeline (Query-Time)
----------------------------
User Query
  ↓
Query Embedding
  ↓
Vector Retrieval
  ↓
Context Assembly
  ↓
```

↓
LLM Generation
↓
Final Answer

The system is divided into **offline** and **online** pipelines to optimize performance and cost.

4. Offline Pipeline: Knowledge Indexing

4.1 Document Ingestion

- Documents are uploaded or collected
- Each document is assigned a unique identifier
- Source-level metadata is recorded

4.2 Text Processing

- Text is extracted
- Noise is removed
- Content is normalized

4.3 Chunking

- Text is split into coherent chunks
- Overlap may be applied
- Each chunk inherits document metadata

4.4 Embedding Generation

- Each chunk is converted into a vector
- A single embedding model is used consistently
- Vectors are optionally normalized

4.5 Vector Storage

- Embeddings are indexed in a vector database
- Metadata is stored alongside vectors
- Indexing strategy is selected (e.g., HNSW)

This pipeline runs **once per document update**, not per query.

5. Online Pipeline: Query-Time Execution

5.1 Query Intake

- User submits a natural language question
- Input is validated and normalized

5.2 Query Embedding

- The same embedding model used offline is applied
- The query is converted into a vector

5.3 Retrieval

- Similarity search is performed
- Top-K relevant chunks are retrieved
- Metadata filters may be applied
- Results are ranked by relevance

5.4 Context Construction

- Retrieved chunks are ordered
- Redundant or low-quality chunks are removed
- Chunks are formatted with metadata

5.5 Prompt Assembly

- System instructions are added
- Context is injected
- User query is appended

5.6 Generation

- The LLM generates an answer
- The response is constrained to provided context

6. Control Flow Responsibilities

Component	**Responsibility**
Ingestion Service	Document intake and preprocessing
Embedding Service	Vector generation
Vector Database	Similarity search and filtering
Retriever	Query execution and ranking
Prompt Builder	Context and instruction assembly
LLM	Controlled response generation

Clear separation of responsibilities improves maintainability and scalability.

7. Error Handling and Edge Cases

7.1 No Relevant Chunks Found

- Return a “no answer available” response
- Avoid fallback hallucination

7.2 Partial Context Matches

- Answer with explicit uncertainty
- Limit response scope

7.3 Context Window Overflow

- Reduce Top-K
- Truncate lowest-ranked chunks
- Compress context if necessary

8. Observability and Debugging

A production-ready RAG system should log:

- Retrieved chunk IDs
- Similarity scores
- Prompt size
- Model latency
- Response confidence indicators

These logs are essential for evaluation and tuning.

9. Reference Characteristics of the Built System

The system constructed in this module is:

- Deterministic
- Explainable
- [illegible]

- Grounded
- Extensible

However, it is:

- Single-turn
- Non-agentic
- Limited to static knowledge

These limitations are addressed in the next module.

10. Summary

- A complete RAG system consists of offline indexing and online retrieval-generation
- Separation of pipelines improves efficiency
- Each module contributes a critical layer
- This baseline system is the foundation for all advanced RAG types
- Correctness depends on orchestration, not just individual components

1. Purpose of This Module

This module expands the baseline document-based RAG system into a **design space**. It formalizes the **types of RAG architectures**, explains how RAG systems are **evaluated**, and outlines the **engineering and business considerations** required to deploy RAG in production environments.

By the end of this module, the reader will:

- Understand and classify major RAG types
- Know when and why to choose a specific RAG architecture
- Understand how to evaluate RAG systems objectively
- Identify latency, cost, and scalability trade-offs
- Apply security, governance, and production best practices

2. Classification of RAG Types

All RAG systems share the same core pipeline.

RAG “types” represent **system-level variations** in retrieval, context management, and control flow.

3. Baseline: Naive (Standard) RAG

Description

A single-pass retrieval followed by generation.

Characteristics

- One query → one retrieval
- No memory or iteration
- Static knowledge base

Strengths

- Simple
- Predictable
- Easy to debug

Limitations

- Weak for complex questions
- No conversational continuity

This is the **foundation** for all other RAG types.

4. Conversational RAG

Description

Extends naive RAG by incorporating conversation history.

Key Additions

- Chat history summarization
- Context carry-over across turns

Use Cases

- Customer support
- Knowledge assistants
- Tutoring systems

Design Considerations

- History compression
- Avoiding context window overflow
- Preventing topic drift

5. Multi-Query and Multi-Hop RAG

Description

Decomposes a single user query into multiple retrieval steps.

Characteristics

- Query rewriting or expansion
- Multiple retrieval calls
- Aggregated context

Use Cases

- Analytical questions
- Research assistants
- Policy or legal reasoning

Trade-offs

- Higher latency
- Improved factual coverage

6. Re-Ranking and Hybrid RAG

Description

Introduces a second-stage ranking model to improve retrieval quality.

Variants

- Dense + sparse (hybrid) retrieval
- Cross-encoder re-ranking

Use Cases

- Large corpora
- High-precision enterprise search

Cost Consideration

Cost Consideration
Re-ranking significantly improves accuracy but increases compute cost.

7. Agentic RAG
Description
Uses an LLM as a controller to decide:
- What to retrieve
- When to retrieve
- Whether to iterate

Characteristics
- Dynamic control flow
- Tool invocation
- Conditional retrieval

Use Cases
- Complex workflows
- Autonomous research
- Multi-step decision systems

Risks
- Non-deterministic behavior
- Higher operational complexity

8. Graph RAG
Description
Retrieval is guided by structured relationships rather than pure similarity.

Key Components
- Knowledge graph
- Entity linking
- Relationship traversal

Use Cases
- Enterprise knowledge systems
- Compliance and auditing
- Highly structured domains

Strength
Improves reasoning over interconnected facts.

9. Evaluation of RAG Systems
9.1 Why Evaluation Is Hard
- Answers are generative
- Correctness is context-dependent
- Retrieval and generation must be evaluated jointly

9.2 Retrieval Evaluation Metrics
- Recall@K
- Precision@K
- Mean Reciprocal Rank (MRR)

- Mean Reciprocal Rank (MRR)

These metrics measure **whether the right information was retrieved**.

9.3 Generation Evaluation Metrics

- Faithfulness to context
- Answer completeness
- Hallucination rate

Often evaluated using:

- Human review
- LLM-based evaluators
- Ground-truth comparisons

9.4 End-to-End Evaluation

A production RAG system must be evaluated on:

- Answer correctness
- Source alignment
- User satisfaction

10. Performance and Cost Optimization

10.1 Latency Optimization

- Precompute embeddings
- Use efficient ANN indexes
- Cache frequent queries

10.2 Cost Control

- Limit Top-K
- Reduce embedding calls
- Optimize context length

Trade-offs between quality and cost must be explicit.

11. Security and Data Governance

11.1 Data Isolation

- Tenant-level vector separation
- Metadata-based access control

11.2 Prompt Injection Risks

- Sanitize retrieved content
- Enforce strict system instructions

11.3 Compliance

- Audit logs
- Source traceability
- Data retention policies

Security failures in RAG systems often originate from retrieval, not generation.

12. Production Deployment Considerations

12.1 System Architecture

- Stateless query services
- Scalable vector storage

- Observability pipelines

12.2 Monitoring

Track:

- Retrieval accuracy
- Hallucination incidents
- Latency distributions
- Cost per query

13. Decision Framework: Choosing the Right RAG Type

Requirement	Recommended RAG Type
Simple Q&A	Naive RAG
Chat-based systems	Conversational RAG
Complex reasoning	Multi-Hop RAG
Large corpora	Hybrid + Re-ranking
Autonomous workflows	Agentic RAG
Structured knowledge	Graph RAG

There is no universally “best” RAG type—only **fit-for-purpose architectures**.

14. Summary

- RAG types are architectural patterns, not separate technologies
- All advanced RAG systems extend the same core pipeline
- Evaluation must consider retrieval and generation together
- Production RAG requires performance, security, and governance planning
- Mastery of RAG is the ability to **design**, not just implement

15. Completion of the RAG Curriculum

You have now progressed from:

- First principles
- To document ingestion
- To embeddings and retrieval
- To controlled generation
- To a complete RAG system
- To advanced architectures and production readiness

This documentation provides a **complete conceptual and architectural foundation** for building, extending, and operating Retrieval-Augmented Generation systems.