

All 15 RAG Components Overview

Here's a quick overview of all the components we'll cover:

1 Retrieval

2 Embeddings

3 Vector Database

4 Retriever

5 Chunking

6 Context Window

7 Grounding

8 Re-Ranking

9 Hybrid Search

10 Metadata Filtering

11 Similarity Search

12 Prompt Injection

13 Hallucination

14 Agentic RAG

15 Latency

#1: Retrieval

The process of finding and extracting relevant information from a knowledge base before generating a response. Think of it as your AI research phase—it searches through your documents to gather context before answering.

Instead of relying only on pre-trained knowledge, retrieval allows the AI to access up-to-date, domain-specific information from your documents, databases, or knowledge sources.

PYTHON EXAMPLE

```
# Basic retrieval flow
query = "What's our refund policy?"
relevant_docs = vector_db.search(query, top_k=5)
answer = llm.generate(query, context=relevant_docs)
```

#2: Embeddings

Numerical representations of text that capture semantic meaning.
Embeddings convert words, sentences, or documents into dense vectors (arrays of numbers) that preserve relationships and context.

"King" - "Man" + "Woman" ≈ "Queen" – embeddings capture these semantic relationships mathematically.

PYTHON EXAMPLE

```
from openai import OpenAI
client = OpenAI()

embedding = client.embeddings.create(
    model="text-embedding-3-small",
    input="Retrieval-Augmented Generation"
)
vector = embedding.data[0].embedding # [0.123, -0.456,
... ]
```

#3: Vector Databases

Specialized databases optimized for storing and querying high-dimensional vectors (embeddings). Unlike traditional databases that search by exact matches, vector databases find semantically similar content using distance metrics.

They enable lightning-fast similarity searches across millions of documents in milliseconds, making them essential for RAG systems at scale.

PYTHON EXAMPLE

```
from chromadb import Client

client = Client()
collection = client.create_collection("docs")

# Add documents
collection.add(
    documents=["RAG improves accuracy", "LLMs can
hallucinate"],
    ids=["doc1", "doc2"]
)

# Search
results = collection.query(query_texts=["reduce AI
errors"], n_results=2)
```

#4: Retriever

A component that orchestrates the retrieval process. The retriever takes a user query, converts it to an embedding, searches the vector database, and returns the most relevant document chunks.

Think of it as a smart librarian that understands your question, where to look, and brings back exactly what you need from a vast collection of documents.

PYTHON EXAMPLE

#5: Chunking

The process of splitting large documents into smaller, manageable segments called "chunks." Effective chunking preserves semantic meaning while fitting within model context limits.

Good chunking is critical—it directly impacts retrieval quality. Well-chunked documents improve precision, maintain context, and enable more accurate answers.

PYTHON EXAMPLE

```
def chunk_text(text, chunk_size=500, overlap=50):
    chunks = []
    start = 0

    while start < len(text):
        end = start + chunk_size
        chunks.append(text[start:end])
        start = end - overlap

    return chunks

# Usage
chunks = chunk_text(document, chunk_size=500, overlap=50)
```

#6: Context Window

The maximum number of tokens (words/subwords) that an LLM can process in a single request. It's the model's "working memory" that determines how much context you can include.

Context windows range from 4K tokens (older models) to 200K+ tokens (modern models). You must fit your retrieved chunks within this limit, making chunk size and selection crucial.

PYTHON EXAMPLE

```
def fit_context(chunks, max_tokens=4000):
    context = []
    token_count = 0

    for chunk in chunks:
        chunk_tokens = count_tokens(chunk)
        if token_count + chunk_tokens < max_tokens:
            context.append(chunk)
            token_count += chunk_tokens

    return context
```

#7: Grounding

The practice of ensuring AI responses are based on retrieved, verifiable sources rather than hallucinated information. Grounding keeps AI "honest" by tethering answers to real data.

Effective grounding requires the AI to cite specific sources, reference retrieved documents, and only make claims supported by the provided context. This reduces hallucinations and increases trustworthiness.

PYTHON EXAMPLE

```
prompt = f"""
```

```
Answer the question using ONLY the provided context.  
Cite the source for each claim.
```

```
Context: {retrieved_docs}
```

```
Question: {user_question}
```

```
Answer with citations:
```

```
"""
```

```
response = llm.generate(prompt)
```

#8: Re-Ranking

A two-stage retrieval strategy that improves result quality. First, a fast initial retrieval finds many candidates (e.g., top 100). Then, a more expensive but precise cross-encoder model re-ranks them to find the best matches.

Two-stage process: Fast retrieval (broad search) → Precise ranking (fine-grained scoring)

PYTHON EXAMPLE

```
from sentence_transformers import CrossEncoder

reranker = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

# Get initial results
candidates = retriever.search(query, top_k=100)

# Re-rank with cross-encoder
scores = reranker.predict([(query, doc) for doc in candidates])
top_docs = [candidates[i] for i in scores.argsort()[-5:]]
```

#9: Hybrid Search

A search strategy that combines keyword-based search (BM25) semantic/vector search. Hybrid search leverages both exact term matching and meaning-based similarity to improve retrieval accuracy.

By blending keyword and semantic scores, you get the precision of exact matches for specific terms and the flexibility of semantic understanding for conceptual queries—best of both worlds.

PYTHON EXAMPLE

```
def hybrid_search(query, alpha=0.5):
    # Keyword search (BM25)
    keyword_results = bm25_search(query)

    # Semantic search
    semantic_results = vector_search(query)

    # Combine scores
    final_scores = (
        alpha * keyword_results.scores +
        (1 - alpha) * semantic_results.scores
    )
    return final_scores.top_k(5)
```

#10: Metadata Filtering

Filtering search results using document metadata (dates, authors, document types, departments, etc.) to narrow down results before semantic search. This reduces noise and improves precision.

By applying filters like "year: 2024" or "department: sales" before searching, you focus the AI on the most relevant subset of documents, making retrieval faster and more accurate.

PYTHON EXAMPLE

```
results = collection.query(  
    query_texts=["quarterly revenue"],  
    n_results=10,  
    where={  
        "year": {"$eq": 2024},  
        "department": {"$eq": "sales"},  
        "type": {"$in": ["report", "presentation"]}  
    }  
)
```

#11: Similarity Search

The core search mechanism in RAG that finds documents with embeddings most similar to your query's embedding. It measures semantic closeness, not just keyword matches.

Typically uses cosine similarity (measures angle between vectors) dot product to compare query embeddings with document embeddings. Higher similarity scores indicate more relevant content.

PYTHON EXAMPLE

```
import numpy as np

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm1 = np.linalg.norm(vec1)
    norm2 = np.linalg.norm(vec2)
    return dot_product / (norm1 * norm2)

# Find similar documents
query_vec = embed(query)
similarities = [cosine_similarity(query_vec, doc_vec)
                for doc_vec in document_vectors]
```

#12: Prompt Injection

A security vulnerability where malicious users inject instructions or queries to manipulate the AI's behavior. Attackers try to override system prompts or extract sensitive information.

Common attacks include phrases like "ignore previous instruction," "reveal your system prompt." RAG systems must sanitize inputs and use prompt engineering to prevent these attacks.

PYTHON EXAMPLE

```
def sanitize_input(user_input):
    # Remove potential injection attempts
    dangerous_patterns = [
        "ignore previous instructions",
        "disregard system prompt",
        "reveal your instructions"
    ]

    for pattern in dangerous_patterns:
        if pattern.lower() in user_input.lower():
            return "Invalid input detected"

    return user_input
```

#13: Hallucination

When AI generates information that sounds plausible but isn't supported by the source documents. LLMs can confidently produce false facts, fake citations, or invented details.

RAG significantly reduces hallucinations by grounding responses retrieved documents. However, proper grounding, source verification, and citation requirements are essential to minimize this risk.

PYTHON EXAMPLE

```
def check_hallucination(response, source_docs):
    # Use LLM to verify claims against sources
    verification_prompt = f"""
        Does this response contain claims not supported by
        sources?

    Response: {response}
    Sources: {source_docs}

    Answer: Yes/No
    """
    is_hallucinated = llm.verify(verification_prompt)
    return is_hallucinated
```

#14: Agentic RAG

An advanced RAG architecture where the AI agent actively plans, reasons, and decides its own retrieval strategy. Instead of a single search, the agent can perform multiple searches, analyze results, iterate.

The agent autonomously determines what information to retrieve, when to search again, which tools to use, and how to synthesize multiple sources. This enables complex, multi-step reasoning tasks.

PYTHON EXAMPLE

```
class AgenticRAG:
    def answer(self, query):
        plan = self.llm.create_plan(query)

        for step in plan:
            if step.action == "search":
                results =
self.retriever.search(step.query)
                self.context.add(results)
            elif step.action == "reason":
                analysis = self.llm.analyze(self.context)
                self.context.add(analysis)
```

#15: Latency

Critical for user experience. Optimize by caching embeddings, using faster models, reducing search scope, and parallelizing operations. Typical RAG systems target sub-second to few-second latency.

PYTHON EXAMPLE

```
import time

def measure_rag_latency(query):
    t0 = time.time()

    # Step 1: Embed query
    embedding = embed(query)
    t1 = time.time()

    # Step 2: Search
    results = vector_db.search(embedding)
    t2 = time.time()

    # Step 3: Generate
    response = llm.generate(query, results)
    t3 = time.time()

    print(f"Embed: {t1-t0:.3f}s, Search: {t2-t1:.3f}s,
          Generate: {t3-t2:.3f}s")
```