Retrieval-Augmented Generation (RAG): A Practical, Tool-Agnostic Handbook

Code Studio
by Syncfusion

Table of Contents

Table of Contents

Table of Contents

Section 1

Introduction

Section 1

Introduction

Retrieval-augmented generation, or RAG, is a practical way to help a language model answer questions with evidence instead of just memory. At a basic level, a RAG system searches trusted source material, adds the most useful pieces to the model's input, and asks the model to write an answer from that context. This handbook introduces RAG as a workflow, not a single tool. You will learn how documents become searchable, how questions retrieve relevant context, how answers should stay grounded in sources, and how teams evaluate whether the system is working. The goal is to make RAG understandable enough to start, inspect, debug, and improve.

For example, if an employee asks, "How many days of parental leave do we offer?", a RAG system should search the current HR policy, retrieve the relevant section, place that section in the prompt, and produce a short answer supported by the policy. In a real system, RAG becomes a full lifecycle: prepare knowledge, search it well, pass the right evidence to the model, check the answer, and keep improving the system as the source material changes. The main thing to remember is this: RAG is not just "add documents to AI." It is a workflow for finding the right evidence, giving it to the model, and checking whether the final answer is actually supported by that evidence

1.1 Who This Handbook Is For

This handbook is for readers who want a clear, practical understanding of RAG without getting lost in vendor-specific tools. It is suitable for beginners who know what an AI assistant is, builders exploring document-based assistants, product people planning AI features, technical writers organizing knowledge, students learning modern AI systems, and early-career developers building their first retrieval workflow. It also supports technical readers who need implementation direction, including chunking, indexing, retrieval, prompting, evaluation, and security. You do not need advanced machine learning knowledge to follow along. Basic familiarity with documents, searching, and AI assistants is enough to understand the core ideas and examples.

1.2 What You Will Learn

You will learn what RAG is, why it exists, and how it helps language models answer from current or private information. You will see the two main lifecycles: indexing, where source material is prepared for searching; and answering, where a user question retrieves context before generation.The handbook explains core building blocks such as documents, chunks, metadata, indexes, retrievers, prompts, and generators. It also compares retrieval methods, shows how to start with a simple handbook assistant, and explains how to evaluate groundedness, relevance, completeness, citations, cost, and latency. Later sections introduce common failures, security concerns, alternatives, improvement workflows, and developer-focused techniques such as semantic chunking, proposition chunking, GraphRAG, hybrid search, and reranking.

1.3 Who This Handbook Is Not For

This handbook is not for readers looking for a complete production platform, a vendor-specific tutorial, or a deep mathematical treatment of language models. It will not teach model training, embedding theory in detail, cloud infrastructure setup, or every framework command needed to deploy a large enterprise system. It also is not a promise that RAG automatically makes answers correct. If source documents are poor, permissions are wrong, or retrieval returns weak evidence, the final answer can still fail. Readers who need legal, medical, financial, or compliance-grade deployment guidance should treat this as a foundation and add expert review, security design, testing, and governance before real use.

1.4 How to Use This Handbook

Read this handbook in order if you are new to RAG. The early sections build the foundation: what RAG means, why it matters, how indexing and answering work, and what parts make up the system. When you reach the practical sections, pause and compare the ideas with a real document set you know, such as policies, product docs, support articles, or onboarding notes. If you are a developer, treat the developer-focused techniques as options to study after you understand the basic workflow. Do not try to add every advanced pattern at once. Start small, test with real questions, inspect retrieved evidence, fix one weakness at a time, and keep improving the system from observed failures.

Section 2

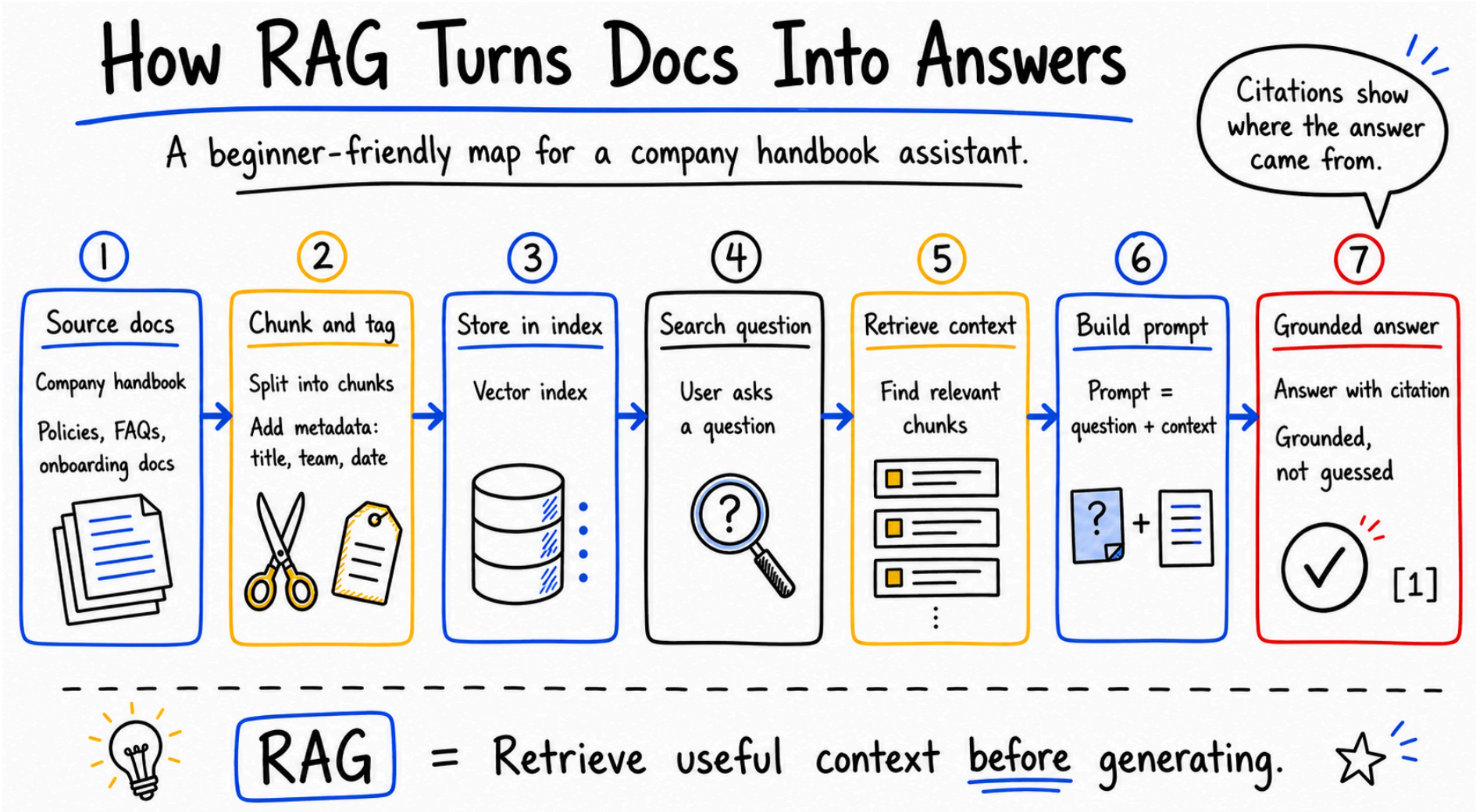
What RAG Is

Section 2

What RAG Is

Retrieval-augmented generation is a design pattern for building AI systems that answer questions with help from external information. The word "retrieval" means the system searches for relevant source material. "Augmented" means that source material is added to the model's input. "Generation" means the model writes the final answer in natural language. At a basic level, RAG works like a careful researcher: before answering, it finds useful evidence and gives that evidence to the writer.

In technical terms, RAG usually connects four pieces: a knowledge source, a search or retrieval system, a prompt that includes the retrieved context, and a language model that writes the response. The knowledge source might be documentation, policies, tickets, product notes, articles, or database records. RAG is best understood as a workflow, not a single feature: prepare trustworthy content, retrieve the right evidence, pass it to the model clearly, and check whether the final answer is supported.



Section 3

Why RAG Exists and What Problems It Solves

Section 3

Why RAG Exists and What Problems It Solves

RAG exists because a language model's built-in knowledge is not enough for many real tasks. A model may not know your private documents, recent product changes, internal policies, customer-specific details, or the exact source you want an answer to follow. It can also write confidently even when it has weak or missing evidence. RAG helps by retrieving relevant information and giving the model a better factual starting point.

The main problems RAG solves are knowledge freshness, private knowledge access, source grounding, and easier updates. Instead of retraining a model every time a document changes, you can update the document collection or search index. Instead of asking the model to guess from memory, you can ask it to answer from retrieved material. This is especially useful for support bots, policy assistants, technical documentation search, research helpers, legal or compliance review, and internal knowledge assistants.

RAG does not solve every problem by itself. If retrieval finds the wrong passages, the answer can still be wrong. If the source documents are outdated, duplicated, poorly written, or restricted incorrectly, the system can still mislead users or expose information. That is why RAG should be treated as a| full workflow: prepare trustworthy content, retrieve the right parts, instruct the model clearly, show sources when useful, and evaluate whether the final answer is actually supported.

Section 4

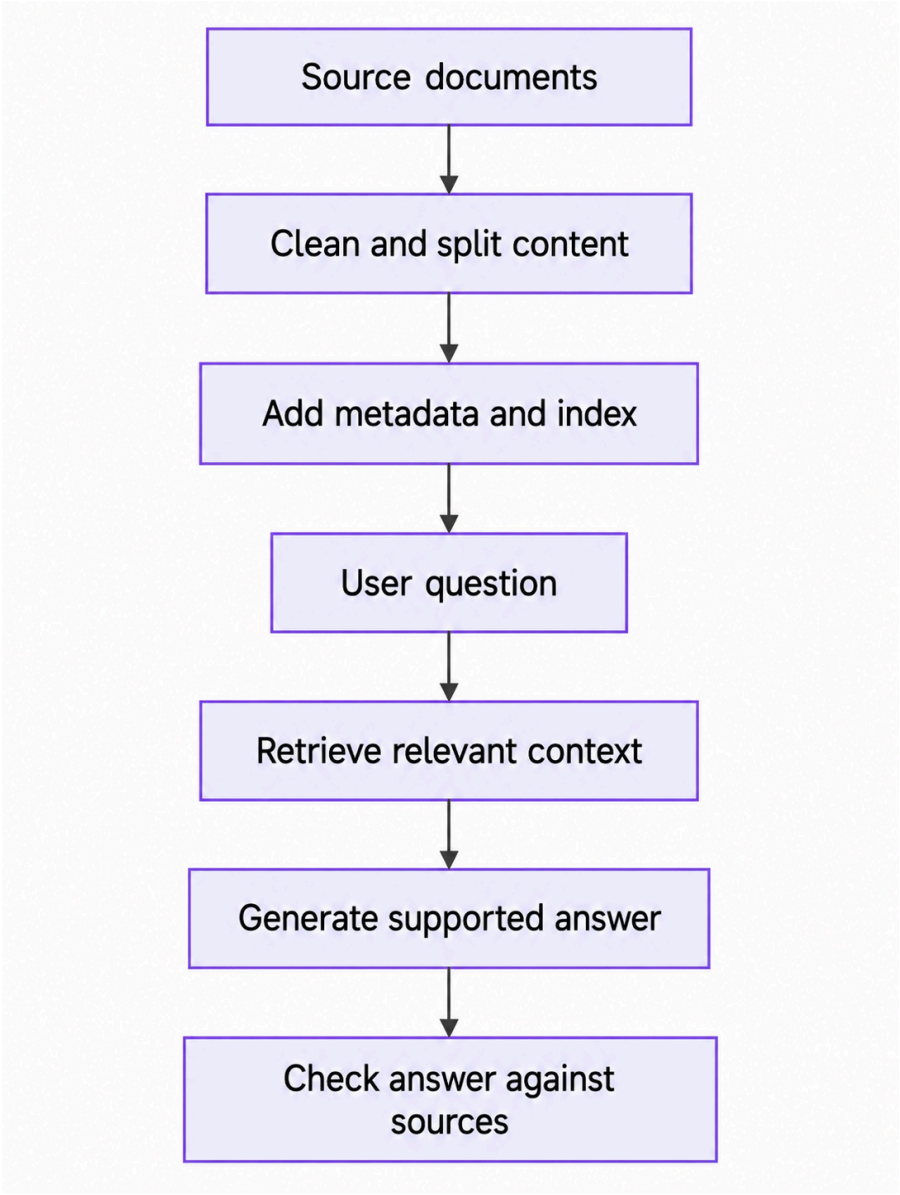
The Two Lifecycles: Indexing and Answering

Section 4

The Two Lifecycles: Indexing and Answering

A RAG system has two lifecycles. The first lifecycle happens before the user asks a question: collect the source material, clean it, split it into useful pieces, add metadata, convert it into a searchable form, and store it in an index. An index is a search-ready structure that helps the system find relevant information quickly. This preparation step matters because messy source material usually leads to messy retrieval.

The second lifecycle happens when the user asks a question: understand the question, search the index, retrieve the most relevant pieces, place those pieces into the prompt, generate an answer, and check whether the answer is supported. Beginners often focus only on the final answer, but the answer depends on both lifecycles. If the index is weak, the model may not receive enough useful evidence.



RAG Lifecycles

Section 5

The Core Building Blocks

Section 5

The Core Building Blocks

A RAG system is easier to understand when you separate its parts. The knowledge source holds the information you want the system to use. The retrieval layer finds relevant pieces from that source. The prompt assembly step places those pieces next to the user's question. The language model then writes an answer, ideally using only the evidence it was given.

Here is the practical comparison.

Building block	What it does	Beginner warning
Knowledge source	Stores documents, records, notes, policies, or other material.	Bad source material leads to bad answers.
Chunks	Breaks long material into smaller retrievable pieces.	Chunks that are too small or too large can miss context.
Metadata	Adds labels such as date, author, product, region, or access level.	Missing metadata makes filtering and permissions harder.
Index	Stores content in a search-ready form.	An outdated index can retrieve stale information.
Retriever	Finds the most relevant pieces for a question.	Retrieval can return plausible but wrong evidence.
Prompt	Combines instructions, question, and retrieved context.	A vague prompt may let the model guess.
Generator	Writes the final answer.	Fluent wording does not prove the answer is correct.

These parts work together, so improving only one part is rarely enough. A better model can still fail if retrieval is weak. Better retrieval can still fail if the prompt does not tell the model how to use the evidence. A good RAG system treats the source material, search setup, prompt, answer style, citations, and evaluation checks as one connected chain.

Section 6

Retrieval Methods Compared

Section 6

Retrieval Methods Compared

Retrieval is the part of RAG that decides which source material the model sees. Keyword search looks for matching words. Semantic search looks for similar meaning, often by using embeddings, which are numerical representations of text. Hybrid search combines both approaches, and reranking adds a second pass that reorders candidate results before they reach the model.

Here is the practical comparison:

Retrieval method	Best for	Watch out for
Keyword search	Exact names, IDs, error codes, policy terms, or product labels.	It can miss useful passages that use different wording.
Semantic search	Questions where the wording differs from the source text.	It can retrieve text that feels related but is not precise enough.
Hybrid search	Systems that need both exact matches and meaning-based matches.	It needs tuning so one search style does not dominate unfairly.
Metadata filtering	Restricting by date, team, region, product, permission, or document type.	Missing or wrong metadata can hide the right answer.
Reranking	Improving the order of retrieved candidates before generation.	It adds cost and latency, so it should be measured.

There is no single best retrieval method for every RAG system. A technical support assistant may need exact error-code search. A policy assistant may need semantic search because people ask questions in everyday language. A large enterprise assistant may need hybrid search, metadata filters, and reranking together. The right method is the one that reliably retrieves evidence that answers real user questions.

Section 7

Where Should You Actually Start?

Section 7

Where Should You Actually Start?

Beginners often feel pressure to start with vector databases, advanced chunking, reranking, GraphRAG, or agentic workflows. In practice, the best starting point is much simpler: choose one narrow use case, collect a small set of trustworthy documents, and test whether the system can retrieve the right evidence for real questions.

Start with a problem where the answer should come from written source material. Good first RAG projects include a company policy assistant, a product documentation helper, an onboarding FAQ assistant, or a support knowledge-base assistant. Avoid starting with highly sensitive, fast-changing, or action-taking workflows until you understand retrieval quality, permissions, citations, and evaluation.

A practical first version should have five parts: a small document set, basic chunking, simple retrieval, a prompt that tells the model to answer only from the retrieved context, and a small test set of questions. The goal is not to build the most advanced system first. The goal is to make the RAG chain visible enough that you can inspect it, debug it, and improve it.

Starting decision	Recommended beginner choice	Why it helps
Use case	One narrow, document-based assistant.	It keeps the scope testable.
Source material	5 to 20 trusted documents.	It makes retrieval easy to inspect.
Retrieval method	Keyword or simple semantic search.	It helps you understand the retrieval step.
Evaluation	10 to 20 real questions with expected sources.	It shows whether the system is actually working.
Security	Apply access rules before retrieval.	It prevents restricted content from entering the prompt.
Improvement path	Change one part at a time.	It makes failures easier to diagnose.

Once the simple version works, improve it only where the evidence shows a real weakness. If exact terms are missing, add keyword or hybrid search. If wording varies, improve semantic search. If the right passage is retrieved but ranked too low, add reranking. If facts are buried inside long chunks, revisit chunking. If answers need relationships across documents, then study GraphRAG. This order helps beginners build confidence without mistaking architectural complexity for quality.

Section 8

Building Your First Simple RAG App

Section 8

Building Your First Simple RAG App

Let's design a first project that feels realistic without being overwhelming: a company handbook assistant. Imagine you have a handbook with policies on leave, security, expenses, benefits, and onboarding. You want users to ask questions like "What is the travel reimbursement limit?", "How do I request parental leave?", or "What are the password requirements?" This is an excellent first RAG project because the data is document-based, the questions are grounded, the answers should cite policy text, and correctness matters more than creativity.

The simplest version has three moving parts: a small set of policy notes, a user question, and a retrieval step that chooses the most relevant note before building the prompt. In a real app, retrieval might use embeddings, hybrid search, or a vector database. In a first learning sample, you can use a tiny keyword-based search just to see the shape of the workflow.

Here is the simplified sample:

```
documents =[
"Employees may take up to 16 weeks of paid parental leave.",
"Passwords must be at least 14 characters.",
"Domestic travel requires manager approval."
]
def retrieve(question, documents):
question_words = set(question.lower().replace("?", "").split())

scored_documents = [
(
document,
len(question_words.intersection(document.lower().replace(".", "").split()))
for document in documents
```

```
]
)

best_document, best_score = max(scored_documents, key=lambda item: item[1])

if best_score == 0:
return None

return best_document

question = "How many weeks of paid parental leave are available?"
retrieved_context = retrieve(question, documents)

if retrieved_context is None:
answer = "I do not have enough source information to answer that question."
else:
prompt = f"""
Answer the question using only the context.

Context: {retrieved_context}
Question: {question}
Answer:
"""
```

In this sample, the retrieve function compares words from the question with each note and selects the note with the most overlap. If no note has any overlap, it returns None instead of forcing an unrelated document into the prompt. This matters because a RAG system should not answer from irrelevant context. In production, the same idea becomes a confidence threshold, retrieval score cutoff, or “no relevant source found” response.

Section 9

How to Know Whether RAG Is Working

Section 9

How to Know Whether RAG Is Working

A RAG system is working when it retrieves the right evidence and produces an answer that is supported by that evidence. You need to check both parts. Retrieval evaluation asks, "Did the system find the right source material?" Answer evaluation asks, "Did the model use that source material correctly?" A fluent answer is not enough, because the model can sound clear while still missing the key evidence.

Here is the comparison.

Check	Question to ask	What a failure looks like
Retrieval relevance	Did the retrieved chunks actually address the question?	The answer misses the policy because the wrong document was retrieved.
Groundedness	Is the answer supported by the retrieved context?	The answer adds a detail that does not appear in the source.
Completeness	Did the answer include the important parts?	The answer gives the leave length but omits eligibility rules.
Citation usefulness	Can a reader verify the answer from the cited source?	The citation points to a broad document instead of the exact section.
Latency and cost	Is the system fast and affordable enough for real use?	The answer is accurate but too slow or expensive for users.

In practice, start with a small test set of real questions and expected source passages. When the system fails, diagnose the stage that failed before changing everything. If the right passage was not retrieved, improve the source cleanup, chunking, metadata, search method, or reranking. If the right passage was retrieved but the answer was wrong, improve the prompt, citation rules, answer format, or final verification step.

Section 10

Common Mistakes and RAG Failure Modes

Section 10

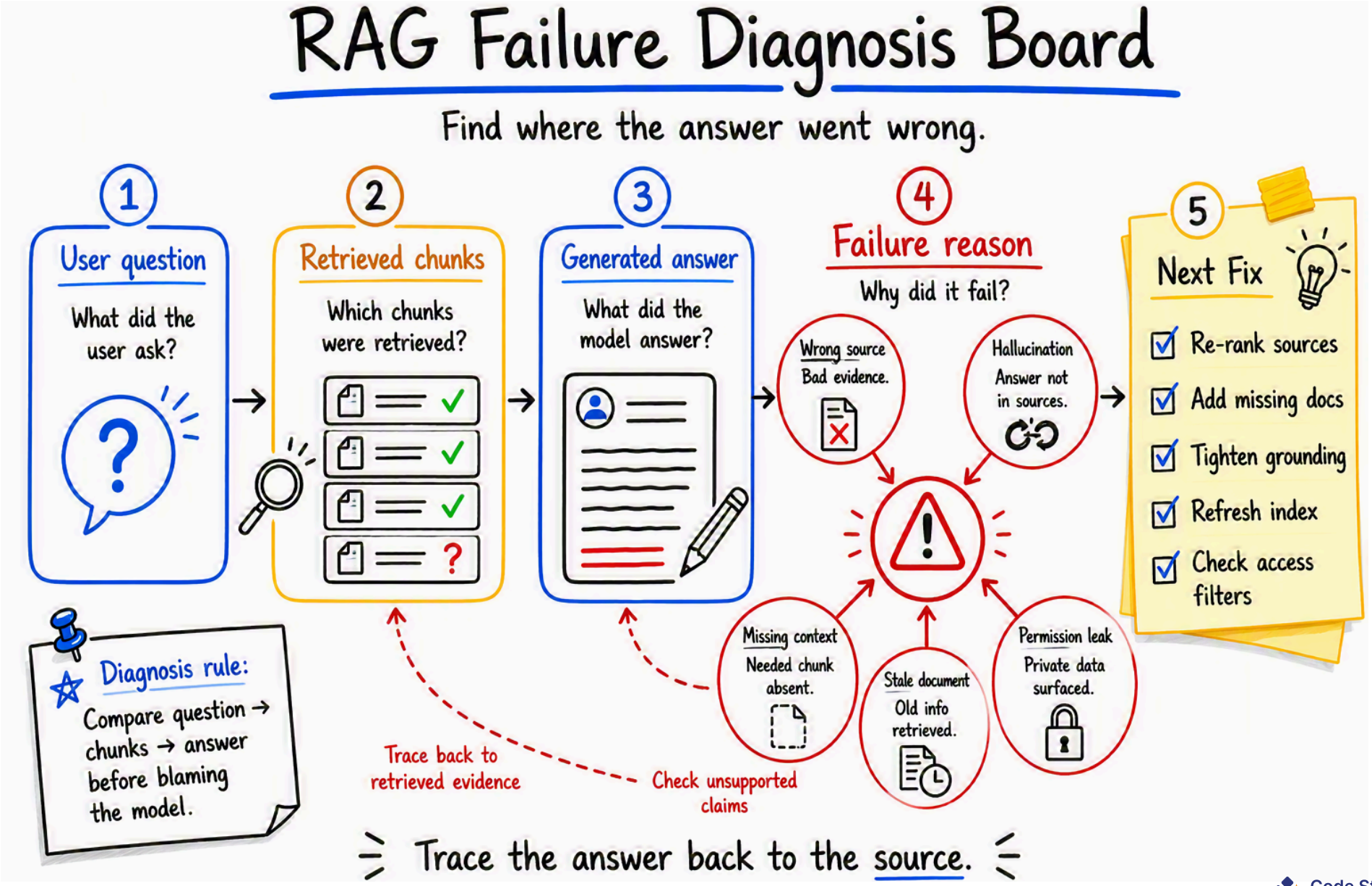
Common Mistakes and RAG Failure Modes

RAG can fail even when every part seems reasonable on its own. The most common pattern is a chain reaction: the source material is messy, the chunks lose context, retrieval brings back weak evidence, and the model writes a polished but unsupported answer. Good RAG work is often less about adding complexity and more about finding exactly where the chain broke.

Here is the troubleshooting view.

Failure mode	What it means	Practical fix
Wrong source retrieved	The system finds a related document but not the answer-bearing passage.	Improve chunking, metadata, filters, or retrieval method.
Missing context	A chunk contains one sentence but not the surrounding rule or exception.	Use better chunk boundaries or include neighboring context.
Stale answer	The retrieved source is old or superseded.	Track document dates, versions, and update workflows.
Hallucination despite grounding	The model adds claims not supported by the retrieved text.	Strengthen instructions, require citations, and evaluate groundedness.
Token overflow	Too much context is sent to the model.	Retrieve fewer chunks, rerank better, summarize carefully, or tighten prompts.
Permission leak	A user receives information they should not access.	Enforce access control before retrieval, not only after generation.

When a RAG answer is wrong, avoid guessing at the fix. Inspect the retrieved chunks first. If the evidence is wrong, focus on the retrieval pipeline. If the evidence is right but the answer is wrong, focus on the prompt, answer format, and verification step. This simple habit keeps teams from changing the model when the real problem is the source, index, or retriever.



Section 11

Security, Privacy, and Governance

Section 11

Security, Privacy, and Governance

RAG systems often search private or sensitive content, so security must be part of the retrieval design. The most important rule is that users should only retrieve content they are allowed to see. Do not rely on the model to hide restricted information after retrieval. Access control should happen before or during search, so unauthorized documents never enter the prompt.

Retrieved content should also be treated as untrusted input. A document can contain outdated instructions, malicious text, or hidden prompt-injection attempts that tell the model to ignore rules or reveal secrets. The model should be instructed to treat retrieved passages as evidence, not as system instructions. Teams should also log sources, track document versions, and make it possible to audit which content supported an answer.

Here is the practical control checklist.

Risk	Why it matters	Practical control
Permission leakage	Users may see restricted internal content.	Apply document-level access control during retrieval.
Prompt injection in documents	Retrieved text may contain malicious instructions.	Separate system instructions from retrieved evidence and tell the model not to follow source instructions.
Stale or unapproved sources	Answers may rely on outdated policy.	Track ownership, version, approval status, and update dates.
Sensitive data exposure	Private data may be sent into prompts or logs.	Minimize retrieved content, redact where needed, and control logging.
Weak provenance	Users cannot verify where an answer came from.	Return citations or source references for important claims.

Governance means deciding who owns the knowledge base, who can approve changes, how often indexes update, how incidents are reviewed, and what quality checks are required before deployment. This may sound administrative, but it directly affects answer quality. A RAG system with clear ownership and audit trails is easier to trust, debug, and improve.

Section 12

RAG Versus Alternatives

Section 12

RAG Versus Alternatives

RAG is useful, but it is not the answer to every AI design problem. Use RAG when the answer should depend on external source material that can change, needs citation, or must respect access rules. If the main problem is teaching the model a style, format, or repeated behavior, fine-tuning may be more appropriate. If the task needs exact live data or actions, a tool or database query may be better than document retrieval.

Approach	Best for	Watch out for
RAG	Private, current, document-based, or citable knowledge.	Retrieval quality controls answer quality.
Long-context prompting	Small or medium source sets that fit directly in the prompt.	Cost, latency, and context selection can become difficult.
Fine-tuning	Teaching style, format, classification behavior, or repeated task patterns.	It does not automatically add fresh, private knowledge.
Direct database or tool access	Exact records, calculations, transactions, or live system actions.	Requires careful schemas, permissions, and tool safety.
Simple search	Helping users find documents themselves.	The user must still read and synthesize the answer.

Many real systems combine these approaches. A support assistant might use RAG for documentation, tool calls for ticket lookup, and a fine-tuned or instructed model for consistent answer style. The design question is not "Which method is best?" but "What kind of information or action does this user request need?"

Section 13

Advanced RAG Patterns

Section 13

Advanced RAG Patterns

Advanced RAG patterns usually try to solve one of three problems: the user's question is unclear, the knowledge base is complex, or the first retrieval result is not good enough. Query rewriting changes the user's question into a better search query. Multi-query retrieval searches from several angles and merges the results. Reranking takes candidate passages and sorts them again so the strongest evidence rises to the top.

Agentic RAG lets an AI system decide when to search, what to search for, and whether another retrieval step is needed. GraphRAG uses relationships between entities, such as people, projects, products, or events, to retrieve information through a graph-like structure. Multimodal RAG retrieves from more than plain text, such as images, tables, slides, audio transcripts, or video metadata. These patterns can improve difficult systems, but they also add cost, latency, evaluation work, and security risk.

The beginner rule is simple: start with the smallest RAG design that can be tested honestly. Add advanced patterns only when a real failure shows why they are needed. If users ask vague questions, try query rewriting. If relevant evidence is scattered across several documents, try multi-query retrieval or reranking. If relationships matter more than isolated passages, study GraphRAG. Complexity should be earned by evidence, not added because the architecture looks impressive.

Advanced RAG Patterns

Add complexity only when retrieval really fails.

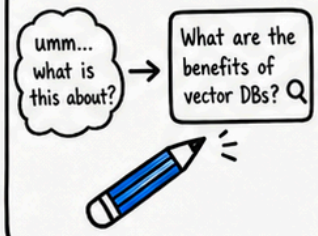
Advanced patterns add cost, latency, evaluation work, and security risk.

Problems

- Question is unclear
- Knowledge base is complex
- First results are weak

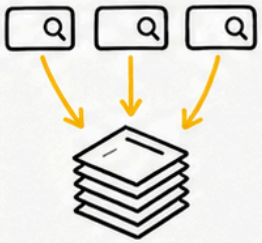
1 Query rewriting

Turns vague questions into better search queries



2 Multi-query retrieval

Searches from several angles, then merges results

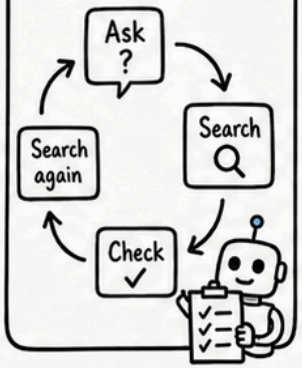


3 Reranking

Sorts candidate passages so best evidence rises

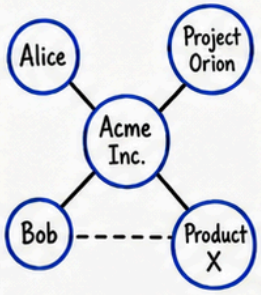
4 Agentic RAG

AI decides when to search again



5 GraphRAG

Uses relationships between people, projects, products, or events



6 Multimodal RAG

Retrieves images, tables, slides, transcripts, or video metadata

Beginner rule: Start small, test honestly, add complexity only when evidence shows the need.

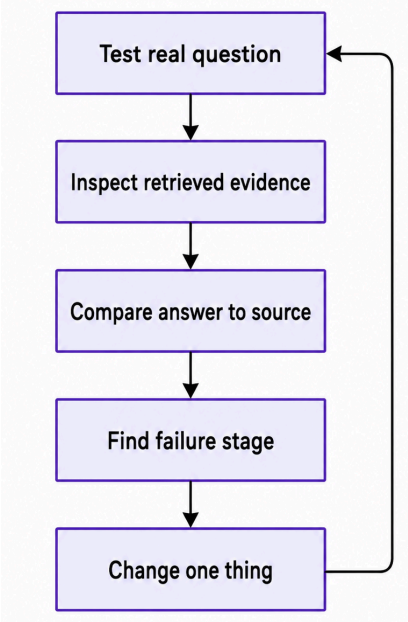
Section 14

A Practical Improvement Workflow

Section 14

A Practical Improvement Workflow

Improving RAG is easiest when you change one part at a time. Start with real user questions, inspect the retrieved evidence, compare the answer to the source, and record the failure. Then choose one targeted fix: clean the source, change chunking, add metadata, adjust retrieval, add reranking, tighten the prompt, or improve the answer check. If you change everything at once, you will not know what helped.



Improve a RAG system by diagnosing one failure and changing one part of the chain.

A useful improvement log includes the question, expected source, retrieved source, answer, failure reason, and fix attempted. Over time, these records become your evaluation set. They show whether the system is improving for real cases instead of only working for one demo question. Good RAG teams treat evaluation as an everyday habit, not a final step.

Section 15

Developer-Focused RAG Techniques

Section 15

Developer-Focused RAG Techniques

The following techniques are for developers who need to turn the beginner RAG workflow into a stronger end-user system. Each technique changes a different part of the pipeline: how documents are split, how facts are represented, how relationships are modeled, how retrieval paths are combined, or how candidate results are reordered before generation.

15.1 Semantic chunking

Semantic chunking is a document-splitting technique that tries to cut text where the meaning changes, not merely after a fixed number of characters or tokens. A fixed-size splitter might divide a refund policy halfway through an exception clause. A semantic chunker first breaks text into smaller units, usually sentences or paragraphs, embeds those units, measures how similar neighboring units are, and creates a chunk boundary when the meaning shifts enough. The goal is to create retrieval units that feel like coherent mini-sections, so the model receives context that is complete enough to answer but not so large that it carries unrelated material.

Use semantic chunking when documents have uneven structure: long PDF manuals, policy documents, tutorials, transcripts, support articles, and essays where topic shifts do not follow a clean heading pattern. It is less useful when your content already has a strong, machine-readable structure, such as well-authored Markdown headings, API reference pages, JSON records, or database rows. In those cases, start with the existing structure and use semantic chunking only inside large sections that still contain multiple topics.

15.1.1 Implementation architecture

The ingestion pipeline should parse documents before it chunks them. For PDFs, extract text with page numbers and clean headers, footers, broken line wraps, and repeated boilerplate. For Markdown and HTML, preserve heading paths because they become valuable metadata. For JSON or database records, choose the fields that users actually search, then create a searchable text representation without losing the original record ID.

For tables, decide whether each row, row group, or table summary is the right retrieval unit. Semantic chunking works best after this cleanup because embeddings compare the text you give them, including any noise.

After parsing, split text into candidate units, embed each unit, compute similarity or distance between neighboring units, and create boundaries at semantic breakpoints. Many implementations use cosine distance between adjacent sentence embeddings. The NirDiamant implementation pattern uses LangChain's SemanticChunker, OpenAI embeddings, configurable breakpoint thresholds such as percentile, standard deviation, interquartile, or gradient, and a FAISS vector store for retrieval. The exact threshold is not universal. Treat it as a tuning parameter that should be evaluated against real user questions.

Here is the implementation decision table.

Decision	Practical default	Adjust when
Candidate unit	Sentence or paragraph.	Use paragraphs for noisy PDFs; use sentences for dense policy text.
Embedding model	Cost-effective general embedding model.	Use stronger or domain-tuned embeddings when recall is weak.
Breakpoint method	Percentile threshold over distance changes.	Use gradient or standard deviation when topic shifts are sharp or inconsistent.
Chunk size guardrail	Minimum and maximum token bounds.	Tighten max size when prompts overflow; raise minimum size when chunks lack context.
Overlap	Low or none after semantic boundaries.	Add small overlap when answers often need the previous sentence or heading.
Metadata	doc_id, chunk_id, heading_path, page, source_url, created_at, access_scope	Add domain fields such as product, region, version, customer, or policy owner.

15.1.2 Model and database choices

Choose the embedding model based on retrieval behavior, not brand preference. For most developer handbooks, support documentation, and internal knowledge bases, start with a reliable general text embedding model and measure whether it retrieves the right chunks. OpenAI's current embedding family includes smaller and larger text embedding models with different dimensions and vector databases such as Pinecone note that result size and performance depend partly on vector dimensions and returned metadata. The implementation rule is simple: use the smaller model when quality is acceptable and move to a larger or domain-specific model only when evaluation shows missed matches, multilingual issues, or domain vocabulary gaps.

For the vector database, local prototypes can use FAISS or Chroma. Team and production systems usually need a managed or operational database such as Qdrant, Pinecone, Weaviate, Milvus, Elasticsearch/OpenSearch with vectors, or PostgreSQL with pgvector. The database must store the vector, searchable text, metadata, and a stable pointer back to the source document. For end-user systems, metadata is not optional. It enables permission filtering, document version control, source citations, and debugging when a user says, "Why did the assistant use this source?"

The following table explains how to index different document types.

Document type	Parsing strategy	Semantic chunking strategy	Metadata to keep
PDF manual	Extract text by page, remove repeated headers and footers.	Chunk within sections or page ranges after cleanup.	doc_id, page, section, source_file,version
Markdown or documentation site	Preserve heading hierarchy and code blocks.	Prefer heading-based chunks first; use semantic chunking inside long sections.	heading_path, url,product, version
HTML article	Remove navigation, ads, sidebars, and duplicate text.	Chunk main article content by semantic boundaries.	url, title, author, published_at
JSON or records	Select searchable fields and keep original IDs.	Usually one record per chunk; semantically chunk long text fields only.	record_id,field_names, updated_at, access_scope
Tables	Convert rows or row groups into clear text statements.	Avoid arbitrary sentence chunking unless cells contain long prose.	table_id, row_id, columns, source_page
Policy documents	Preserve sections, exceptions, dates, and owners.	Chunk by policy topic and exception boundaries.	policy_id, owner, effective_date, region

15.1.3 Query-time flow

At query time, embed the user's query with the same embedding family used for the chunks, apply metadata filters first, retrieve the top candidates, and assemble context for the generator.For example, an internal support assistant might filter by product = "mobile", tenant_id = user.tenant_id, and access_scope in user.allowed_scopes, retrieve the top eight semantic chunks, then send the best three to five chunks after deduplication. If the retrieved chunk has weak context, hydrate it with its heading path, source title, or neighboring sentence before generation.

Semantic chunking improves retrieval only if it produces better answer evidence. Evaluate it against a fixed-size baseline. Use a small test set of real questions with expected source sections. Track whether the expected source appears in the top 5 or top 10 results, whether the final answer is grounded, whether citations point to the right section, and whether latency or indexing cost increased. Watch for three common failures: chunks become too tiny to answer from, chunks become too broad despite semantic splitting, or the threshold creates inconsistent chunk sizes across document types.

15.2 Proposition chunking: fine-grained chunk semantics

Proposition chunking turns larger text into small, self-contained factual statements. A proposition is a single claim that can be understood without relying on the previous sentence, a pronoun, or hidden document context.For example, instead of indexing a long paragraph about a benefits policy, the system might index propositions such as "Full-time employees are eligible for health benefits after 30 days" and "Contractors are not eligible for company health benefits." This technique is useful when end users ask precise factual questions and large chunks often return too much surrounding material.

The theory is simple, but the implementation is more expensive than normal chunking. You first create regular chunks, then use a language model to extract propositions from each chunk. The propositions should be accurate, clear, complete, concise, and written with explicit names instead of ambiguous references. Because the model can create weak or unsupported propositions, production systems should add a quality gate before indexing them. The NirDiamant implementation follows this pattern: split the document, generate propositions with structured LLM output, grade each proposition for accuracy, clarity, completeness, and conciseness, then index the accepted propositions in a FAISS vector store.

15.2.1 Implementation architecture

The ingestion pipeline has two layers: parent chunks and proposition records. Parent chunks preserve original context. Proposition records provide fine-grained retrieval targets. Store both, because proposition-only retrieval can be precise but context-poor.A practical schema includes proposition_id, parent_chunk_id, doc_id, source_span, proposition_text, quality_scores, confidence_score, heading_path, source_url, created_at, and access_scope. If the source document changes, regenerate propositions for the affected parent chunks and expire the old proposition IDs.

At indexing time, embed the proposition text for high-precision retrieval and keep a pointer to the parent chunk. For short, factual corpora, a general embedding model is usually enough to start. For legal, medical, financial, scientific, or code-heavy corpora, evaluate a domain embedding model or a stronger general model against real questions. The vector database can be FAISS for local testing, or Qdrant, Pinecone, Weaviate, Milvus, Elasticsearch/OpenSearch, or PostgreSQL with pgvector for production. The important requirement is that the database can filter by access metadata and return the parent chunk pointer with the proposition.

Here is the practical implementation table.

Step	Developer task	Production warning
Parent chunking	Split source documents into manageable sections.	Bad parent chunks create bad propositions.
Proposition generation	Ask an LLM for atomic, self-contained facts.	The LLM may invent or over-compress claims.
Quality grading	Score accuracy, clarity, completeness, and conciseness.	Low-quality propositions should not be indexed.
Proposition indexing	Embed accepted propositions and store metadata.	Keep parent references for context hydration.
Query retrieval	Retrieve proposition candidates first.	Fine-grained matches can miss broader context.
Context hydration	Add parent chunk or neighboring source text.	Too much hydration can undo the precision benefit.

15.2.2 Query-time flow

At query time, embed the user's question, apply metadata and permission filters before or during retrieval, and retrieve the top proposition candidates only from sources the user is allowed to access. Then, hydrate the best propositions with their parent chunk, heading path, source title, or neighboring propositions.

The final prompt should show the model both the exact proposition and enough permitted source context to avoid brittle answers. For example, if a user asks, "Are contractors eligible for health benefits?", the proposition may answer the direct question, while the parent chunk may contain exceptions, dates, or jurisdiction rules.

Evaluate proposition chunking against both normal chunks and semantic chunks. Use factual queries where the expected answer is a specific statement. Track precision at top five, whether the answer is grounded in the original source, whether the parent context changes the answer, extraction cost, and freshness after document updates. Common failures include propositions that omit qualifiers, propositions that turn suggestions into rules, duplicate propositions that crowd out better evidence, and retrieval that answers narrowly when the user needs a broader explanation.

15.3 GraphRAG: knowledge-centric RAG

GraphRAG adds a relationship layer to retrieval. Standard RAG usually retrieves chunks that are similar to the user's question. GraphRAG also represents entities and relationships, such as customers, products, policies, incidents, people, teams, dependencies, or events. This helps when the answer depends on connections across documents rather than one obvious passage. For example, a product support assistant may need to connect a bug report, a release note, a feature flag, and a customer environment before it can answer safely.

The theory behind GraphRAG is that knowledge is often relational. A vector search index is good at finding nearby meaning, but it does not naturally know that Product A depends on Service B, Service B was changed in Release C, and Customer D is affected by Release C. A graph can store those links explicitly. Retrieval can then start with similar chunks or entities, expand through related nodes, collect supporting passages, and give the model a context set that explains the chain of evidence.

15.3.1 Implementation architecture

A practical GraphRAG system has two indexes that work together. The vector index stores chunks and embeddings for semantic search. The graph store stores entities, chunk nodes, document nodes, and relationship edges. For a prototype, developers can use NetworkX plus FAISS. For production, common choices include Neo4j, a relational database with relationship tables, a graph-capable search system, or a managed graph database, alongside a vector database such as Qdrant, Pinecone, Weaviate, Milvus, Elasticsearch/OpenSearch, or PostgreSQL with pgvector.

The ingestion pipeline should parse documents, create chunks, extract entities and relationships, normalize names, and link each graph node back to source evidence. Entity extraction can use named entity recognition, rules, an LLM, or a combination. Relationship extraction should be conservative: a wrong edge can mislead retrieval more deeply than a missing edge. Store fields: entity_id, entity_type, canonical_name, aliases, relationship_type, source_chunk_id, confidence_score, created_at, and access_scope. The NirDiamant GraphRAG pattern builds chunk nodes, extracts concepts, adds edges based on semantic similarity and shared concepts, weights edges, then traverses the graph from query-relevant seed nodes.

Here is the practical implementation table.

GraphRAG component	Developer task	Production warning
Entity extraction	Identify people, products, systems, policies, events, or concepts.	Entity names need normalization and aliases.
Relationship extraction	Create edges such as depends-on, owns, affects, supersedes, or cites.	False edges can create persuasive but wrong context.
Vector index	Store chunk embeddings for seed retrieval.	Vector search still needs metadata and permissions.
Graph store	Store entities, chunks, documents, and relationship edges.	Keep source evidence for every important edge.
Graph traversal	Expand from seed chunks/entities to related evidence.	Unbounded traversal can add noisy context.
Context assembly	Send selected paths and supporting chunks to the model.	The model needs citations, not just graph labels.

15.3.2 Query-time flow

At query time, embed the user query and retrieve seed chunks or entities. Then expand through the graph using strict limits: relationship types, max hops, edge confidence, access permissions, and token budget. For a compliance assistant, the query might retrieve a regulation entity, expand to internal policy nodes that cite it, then collect the policy sections that apply to the user's region. For a product assistant, the query might retrieve a feature entity, expand to dependent services and recent release notes, then return evidence that explains both the direct answer and the dependency chain.

Evaluate GraphRAG with questions that require relationships, not simple lookup. Test multihop completeness, relationship correctness, source traceability, answer groundedness, and graph freshness after documents change. A good GraphRAG answer should show why connected evidence matters. Common failures include noisy entity extraction, duplicate entities, stale relationships, over-expansion into unrelated nodes, and answers that cite graph-derived claims without showing the source passage that supports the edge.

15.4 Hybrid search—fusion retrieval

Hybrid search combines dense semantic search with lexical search, usually BM25 or another keyword-based method. Dense search finds text with similar meaning even when the wording differs. Lexical search finds exact terms such as error codes, product names, legal phrases, IDs, function names, and policy labels. Fusion retrieval combines the result sets so end users get the benefit of both. This matters because real users mix natural language with exact tokens: "How do I fix ERR_AUTH_042 after SSO migration?" needs both meaning and exact matching.

The theory is that dense and lexical retrieval fail in different ways. Dense retrieval can find paraphrases but may miss exact identifiers. BM25 can catch exact words but may miss relevant passages that use different wording. A hybrid system creates candidates from both paths, normalizes or ranks them, combines scores or ranks, and returns the best final set. The NirDiamant implementation pattern builds a FAISS vector index, creates a BM25 index over the same chunks, normalizes both score sets, combines them with an alpha weight, and returns the top combined results.

15.4.1 Implementation architecture

A production hybrid system needs two retrieval paths for the same content. One path stores dense embeddings in a vector index. The other path stores lexical tokens in a BM25, sparse-vector, or full-text index. You can run these in separate systems, such as a vector database plus Elasticsearch/OpenSearch, or in one system with native hybrid support: Qdrant, Weaviate, Pinecone, Milvus, Elasticsearch/OpenSearch with vectors, or PostgreSQL with pgvector plus full-text search. The key is to keep document IDs and chunk IDs stable so results from both paths can be merged without losing source metadata.

Choose the dense embedding model for semantic meaning and the lexical index for exact terms. For most handbook, support, and policy corpora, start with a general embedding model plus BM25. For code, security, scientific, or multilingual corpora, evaluate domain embeddings or multilingual embeddings against real queries. Store metadata such as doc_id, chunk_id, heading_path, source_url, product, version, updated_at, and access_scope. Apply permission and product filters before fusion when possible, so restricted candidates never enter the merged result set.

Here is the practical implementation table.

Fusion choice	How it works	Use when
Weighted score fusion	Normalize dense and lexical scores, then combine with an alpha weight.	You can trust score normalization and want simple tuning.
Rank fusion	Combine based on result positions rather than raw scores.	Dense and lexical scores are not comparable.
Reciprocal rank fusion	Reward documents that rank well in either or both result lists.	You want a robust default across mixed query types.
Native hybrid search	Let the vector database or search engine combine dense and sparse retrieval.	Your database supports hybrid search and filtering well.
Hybrid plus reranking	Retrieve broadly with hybrid search, then rerank candidates.	User-facing quality matters more than the extra latency.

15.4.2 Query-time flow

At query time, run dense retrieval and lexical retrieval against the same filtered corpus. Retrieve more candidates than you plan to send to the model, such as 20 dense and 20 lexical candidates, then fuse, deduplicate by chunk_id, and return the top 5 to 10. For exact-token-heavy queries, lower the dense weight or boost lexical matches. For broad conceptual questions, raise the dense weight. Some teams tune this with query classification: code and error queries lean lexical, explanatory questions lean dense, and mixed questions use balanced fusion.

Evaluate hybrid search by comparing three systems: dense-only, lexical-only, and hybrid. Use exact-code questions, paraphrased questions, and mixed questions. Track Recall@K for expected source chunks, Precision@K after fusion, duplicate rate, latency, and final answer groundedness. Common failures include adding raw scores that are not comparable, letting one retrieval path dominate all results, failing to deduplicate chunks, and retrieving exact keyword matches that mention the term but do not answer the user's question.

15.5 Reranking

Reranking is a second-stage retrieval technique. The first stage retrieves a broad candidate set quickly. The reranker then scores those candidates more carefully against the user's query and returns the best few for context assembly. This is useful because fast retrievers are optimized for broad recall, not final answer quality. A vector search might return 50 plausible chunks, but the model should only see the 5 to 10 chunks that best answer the user.

The theory is that a reranker can compare the query and document together instead of embedding them separately. A bi-encoder embedding model creates one vector for the query and one vector for each chunk, then compares vectors. A cross-encoder reranker reads the query and candidate text as a pair, which is slower but often more precise. Developers can also use hosted rerankers, late-interaction models such as ColBERT-style systems, or an LLM-as-judge reranker for small, high-value workloads. The NirDiamant implementation pattern retrieves candidates from a vector store, then reranks them with either an LLM relevance scorer or a sentence-transformers cross-encoder.

15.5.1 Implementation architecture

Reranking sits after candidate retrieval and before prompt assembly. It does not replace the vector database. A typical flow: retrieve top_k=30 to 100 candidates using vector search, BM25, hybrid search, GraphRAG, or proposition retrieval; pass candidates to the reranker; return top_n=5 to 10; deduplicate and trim; then send the final context to the generator. The candidate set must be broad enough for recall, but small enough that reranking cost and latency stay acceptable.

Choose the reranker based on the workload. Cross-encoders are a strong default when you can host or call a model that scores query-document pairs. Hosted reranking APIs reduce infrastructure work but add provider dependency. LLM-as-judge reranking is flexible and can consider instructions, metadata, and business intent, but it is usually slower and more expensive. For low-latency consumer apps, use a lightweight reranker or rerank fewer candidates. For legal, compliance, research, or customer support systems, the added latency may be worth the quality gain.

Here is the practical implementation table.

Reranker type	Best for	Watch out for
Cross-encoder	High-quality query-document relevance scoring.	Slower-than-vector similarity and needs model hosting or API calls.
Hosted reranker	Teams that want quality without managing inference.	Provider limits, cost, and data-handling requirements.
LLM-as-judge reranker	Small candidate sets where instructions and nuance matter.	Higher latency, higher cost, and prompt sensitivity.
Late-interaction reranker	Large-scale retrieval where token-level matching helps.	More complex indexing and serving architecture.
Rule-assisted reranking	Boosting metadata, freshness, product version, or access-specific relevance.	Rules can hide good evidence if they are too aggressive.

15.5.2 Query-time flow

At query time, retrieve broadly, rerank narrowly, and assemble carefully. A support assistant might retrieve 50 chunks with hybrid search, rerank them with a cross-encoder, keep the top eight, remove near-duplicates, and then send the top four chunks to the model. The reranker input should include the user query, candidate text, source title, heading path, and important metadata. Avoid sending huge chunks to the reranker. If a candidate is long, rerank a concise passage or passage summary while keeping the original source pointer for citation.

Evaluate reranking with retrieval metrics and final-answer metrics. Before reranking, check Recall@K: did the broad candidate set contain the right source? After reranking, check Precision@K, MRR, NDCG, citation usefulness, answer groundedness, latency, and cost. Common failures include retrieving too few candidates before reranking, reranking chunks that are too large, using an LLM judge with inconsistent scoring, ignoring metadata that matters to end users, and improving offline relevance scores without improving final answer quality.

Note: For more information on implementing RAG techniques, see this repository.

Section 16

Glossary

Section 16

Glossary

Access control: Rules that decide which users can see which documents or records. In RAG, access control should be applied before restricted content is retrieved.

Agentic RAG: A RAG pattern where an AI system can decide when to search, what to search for, and whether more retrieval steps are needed.

Augmentation: The step where retrieved information is added to the model's input so the model can answer with context.

Chunk: A smaller piece of a larger document. Chunking helps the retriever find and pass only the most relevant parts of a source.

Citation: A pointer to the source used for an answer. Citations help users verify important claims.

Cosine distance: A measure of how far apart two embeddings are in vector space. Semantic chunkers often use it to detect when neighboring sentences shift topic.

Cross-encoder: A reranking model pattern that reads the query and candidate text together to score their relevance.

Embedding: A numerical representation of text, images, or other content. Embeddings help systems search by meaning instead of only exact words.

FAISS: A local vector search library often used for prototypes and experiments. It is useful for learning and testing, but production systems may need persistence, permissions, scaling, and operations around it.

Fine-tuning: Training a model further on examples so it learns a style, format, behavior, or task pattern. Fine-tuning is different from RAG because it does not automatically retrieve fresh, external knowledge.

Generator: The language model that writes the final answer after context has been retrieved and added to the prompt.

GraphRAG: A RAG pattern that uses relationships between entities, such as people, products, or events, to help retrieve connected information.

Groundedness: The degree to which an answer is supported by the provided source context.

HR: Short for human resources, the team or function that manages employee policies, benefits, hiring, and related workplace processes.

Hybrid search: A retrieval method that combines keyword search and semantic search.

BM25: A keyword-search scoring method that ranks documents based on term frequency, document frequency, and document length.

Bi-encoder: A retrieval model pattern that embeds the query and document separately, then compares their vectors.

ID: Short for identification, a stable value used to refer to a document, chunk, record, user, product, or other object.

Index: A search-ready structure that stores content so it can be retrieved quickly.

LLM: Short for large language model, an AI model that can understand and generate language-like text.

Metadata: Labels about content, such as date, author, product, region, document type, or permission level.

MRR: Short for mean reciprocal rank, it’s a retrieval metric that rewards systems for placing the first relevant result near the top.

Multimodal RAG: RAG that retrieves from more than text, such as images, slides, tables, audio transcripts, or video metadata.

Named entity recognition: A technique for identifying names and categories such as people, organizations, products, places, or dates in text.

NDCG: Short for normalized discounted cumulative gain, it’s a retrieval metric that rewards relevant results appearing higher in the ranked list.

NLP: Short for natural language processing, it’s a field of computing

focused on working with human language.**OWASP:** The open worldwide application security project is a community that publishes security guidance, including the top 10 for large language model applications.

Prompt: The input sent to the model. In RAG, the prompt often includes instructions, the user's question, and retrieved context.

Prompt injection: A malicious or unsafe instruction hidden in user input or retrieved content that tries to make the model ignore its rules.

Proposition: A small, self-contained, factual statement extracted from a larger source. Proposition chunking indexes these statements so precise factual questions can retrieve precise evidence.

RAG: Short for retrieval-augmented generation, it’s a pattern that retrieves external context before a language model generates an answer.

Reranking: A second scoring step that reorders retrieved candidates before they are sent to the model.

Retrieval: The step where the system searches for source material relevant to the user's question.

Reciprocal rank fusion: A fusion method that combines ranked result lists by rewarding items that appear high in one or more lists.

Semantic search: Search based on meaning rather than only exact keyword matches.

Semantic chunking: A chunking method that uses meaning shifts, often measured with embeddings, to decide where chunks should begin and end.

SSO: Short for single sign-on, this is an authentication approach where a user signs in once and accesses multiple connected systems.

Traversal: The process of moving through graph nodes and edges to find connected evidence.

Vector database: A database designed to store and search embeddings.

Section 17

References and Further Reading

Section 17

References and Further Reading

• Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks": https://arxiv.org/abs/2005.11401

• OpenAI Retrieval guide: https://platform.openai.com/docs/guides/retrieval

• Microsoft Foundry, "Retrieval augmented generation (RAG) and indexes": https://learn.microsoft.com/en-sg/azure/foundry/concepts/retrieval-augmented-generation

• Microsoft Azure Architecture Center, "Design and develop a RAG solution": https://learn.microsoft.com/azure/architecture/ai-ml/guide/rag/rag-solution-design-and-evaluation-guide

• LangChain retrieval documentation: https://docs.langchain.com/oss/python/langchain/retrieval

• Pinecone, "Retrieval-Augmented Generation": https://www.pinecone.io/learn/retrieval-augmented-generation/

• Google Cloud Architecture Center, "Generative AI with RAG": https://cloud.google.com/architecture/rag-reference-architectures

• Yu et al., "Evaluation of Retrieval-Augmented Generation: A Survey": https://arxiv.org/abs/2405.07437

• OWASP, "OWASP Top 10 for LLM Applications 2025": https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/

- NirDiamant RAG Techniques repository: https://github.com/NirDiamant/RAG_Techniques

- NirDiamant semantic chunking runnable script: https://github.com/NirDiamant/RAG_Techniques/blob/main-all_rag_techniques_runnable_scripts/semantic_chunking.py

- NirDiamant proposition chunking notebook: https://github.com/NirDiamant/RAG_Techniques/blob/main/all-rag_techniques/proposition_chunking.ipynb

- NirDiamant GraphRAG notebook: https://github.com/NirDiamant/RAG_Techniques/blob/main/all-rag_techniques/graph_rag.ipynb

- NirDiamant fusion retrieval runnable script: https://github.com/NirDiamant/RAG_Techniques/blob/main/all-rag_techniques_runnable_scripts/fusion_retrieval.py

- NirDiamant reranking runnable script: https://github.com/NirDiamant/RAG_Techniques/blob/main/all-rag_techniques_runnable_scripts/reranking.py

- Qdrant hybrid search with reranking tutorial: https://qdrant.tech/documentation/tutorials-search-engineering/reranking-hybrid-search/

- Weaviate hybrid search concepts: https://docs.weaviate.io/weaviate/concepts/search/hybrid-search

- Weaviate reranking concepts: https://docs.weaviate.io/weaviate/concepts/reranking

- OpenAI embeddings guide: https://platform.openai.com/docs/guides/embeddings

- Pinecone search overview: https://docs.pinecone.io/guides/search/search-overview

Section 18

Closing Note

Section 18

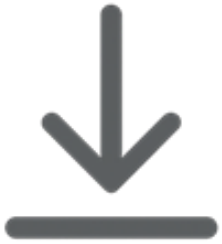
Closing Note

RAG is best learned as a practical workflow. Start with trusted source material, retrieve the right evidence, give that evidence to the model clearly, and check the answer against the source. The more serious the use case, the more important the surrounding system becomes: permissions, citations, evaluation, monitoring, and ownership.

For your first project, keep the design small enough to inspect by hand. A simple handbook assistant with a few real questions can teach more than an impressive architecture that nobody can debug. Once the basic chain works, improve it one step at a time.

Resources

Explore these resources to get started and stay up to date



Get Code Studio →

Download and install.

Video Tutorials →

Watch short, clear video demos.

Product Documentation →

Setup, usage, and integration guide.

Support →

Get dedicated assistance.

Book a Free Demo →

Schedule your 30-minute demo with a product specialist.

Feedback →

Submit suggestions and shape the roadmap.