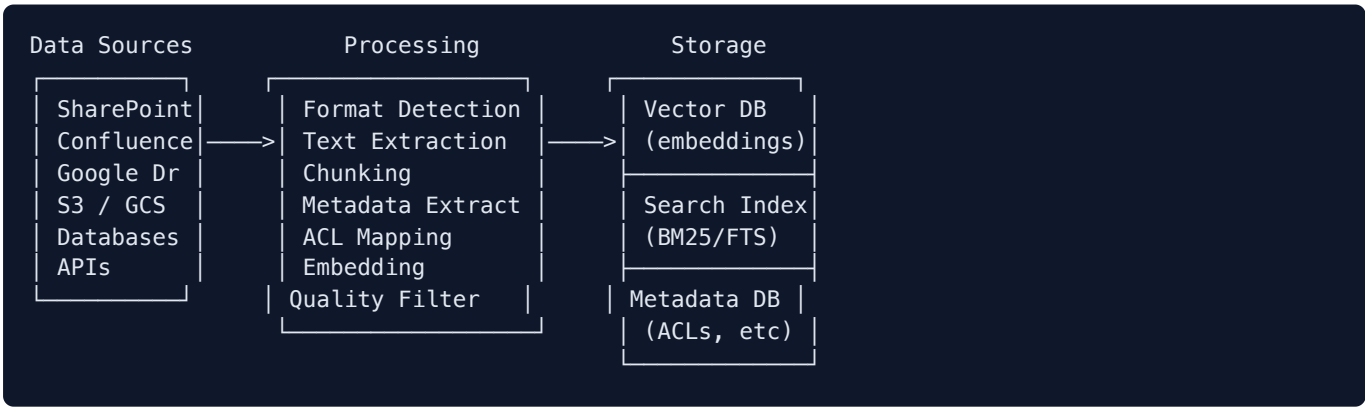
## Enterprise RAG vs Basic RAG

A demo RAG pipeline and a production enterprise RAG system are fundamentally different in scope. This comparison shows exactly where basic RAG falls short and what you need to add for real-world deployment.

| Aspect | Basic RAG | Enterprise RAG |
|---|---|---|
| Data sources | Single source | Multi-source, multi-format |
| Access control | None | Per-document ACLs |
| Search | Dense vector only | Hybrid (dense + sparse + metadata) |
| Ranking | Top-k by similarity | Reranking + business rules |
| Freshness | Static index | Incremental updates, TTL |
| Citation | None or approximate | Exact source + page/section |
| Scale | Thousands of docs | Millions of docs |
| Monitoring | None | Retrieval quality, latency, cost |
| Data lineage | None | Full provenance tracking |

## Ingestion Pipeline

The ingestion pipeline is the foundation of your entire RAG system -- garbage in means garbage out at query time. Getting document processing, metadata extraction, and ACL mapping right during ingestion prevents every downstream problem.




### Document Processing by Format

| Format | Extraction Tool | Notes |
|---|---|---|
| PDF | PyMuPDF, Unstructured, Amazon Textract | OCR for scanned docs |
| DOCX/PPTX | python-docx, python-pptx, Unstructured | Preserve structure |
| HTML | BeautifulSoup, Trafilatura | Strip boilerplate |

| Format | Extraction Tool | Notes |
|---|---|---|
| Markdown | Direct parse | Preserve headers as metadata |
| CSV/Excel | pandas | Row-level or table-level chunks |
| Images | OCR (Tesseract, Textract) + VLM | Vision model for diagrams |
| Audio/Video | Whisper transcription | Timestamps as metadata |
| Confluence/Wiki | API extraction | Page hierarchy as context |
| Code | AST parsing | Function/class level |

### Metadata to Extract and Store

| Field | Purpose | Example |
|---|---|---|
| `source_id` | Unique document identifier | `confluence://page/12345` |
| `title` | Display and search | "Q3 Revenue Report" |
| `source_type` | Format/origin | `pdf`, `confluence`, `slack` |
| `created_at` | Freshness sorting | `2025-11-15T10:00:00Z` |
| `updated_at` | Freshness, re-indexing | `2026-01-20T14:30:00Z` |
| `author` | Attribution | "jane.doe@company.com" |
| `acl_groups` | Access control | `["engineering", "all-staff"]` |
| `acl_users` | Access control | `["user123"]` |
| `department` | Filtering | "Engineering" |
| `chunk_index` | Ordering within doc | `3` (of 12) |
| `parent_id` | Parent-child linking | `doc_abc_chunk_0` |
| `confidence` | Extraction quality | `0.95` |

## Access Control (ACL) Filtering

In enterprise environments, returning a document the user should not see is a security incident, not a UX bug. ACL filtering must be enforced at the retrieval layer, not as a post-processing step.

### Pattern: Pre-filter at Query Time

```
def search_with_acl(query: str, user: User, top_k: int = 10):
    # Get user's access groups
    user_groups = get_user_groups(user.id)

    # Build metadata filter
    acl_filter = {
        "$or": [
```

```
            {"acl_groups": {"$in": user_groups}},
            {"acl_users": {"$in": [user.id]}},
            {"acl_groups": {"$in": ["public"]}}
        ]
    }

    # Vector search with filter
    results = vector_db.query(
        embedding=embed(query),
        filter=acl_filter,
        top_k=top_k
    )
    return results
```

### ACL Sync Strategy

| Strategy | Description | Freshness | Complexity |
|---|---|---|---|
| Ingest-time ACL | Copy ACLs at indexing | Stale until re-index | Low |
| Query-time ACL check | Verify against source at query | Real-time | High latency |
| ACL cache + sync | Cache ACLs, sync periodically | Minutes-stale | Medium |
| Event-driven ACL | Update on permission change events | Near real-time | Medium-high |

**Recommendation:** ACL cache with 5-15 minute sync interval for most enterprise use cases.

## Hybrid Search

Enterprise queries range from exact error codes to broad conceptual questions -- no single search method handles both well. Hybrid search combines dense and sparse retrieval with metadata filtering to cover the full spectrum.

### Architecture

```
Query
  |
  ├──> Dense Search (embedding similarity)  ──> Scores A
  |
  ├──> Sparse Search (BM25 / keyword)       ──> Scores B
  |
  └──> Metadata Filter (date, source, ACL)   ──> Scores C

  Fusion (RRF or weighted sum) ──> Reranker ──> Top K results
```

### Reciprocal Rank Fusion

```
def rrf_fuse(result_lists: list[list], k: int = 60) -> list:
    """Fuse multiple ranked lists using Reciprocal Rank Fusion."""
    scores = {}
    for result_list in result_lists:
        for rank, doc in enumerate(result_list):
```

```
            scores[doc.id] = scores.get(doc.id, 0) + 1.0 / (k + rank + 1)

    sorted_docs = sorted(scores.items(), key=lambda x: x[1], reverse=True)
    return sorted_docs

# Usage
dense_results = vector_search(query, top_k=50)
sparse_results = bm25_search(query, top_k=50)
fused = rrf_fuse([dense_results, sparse_results])
```

### Search Type Strengths

| Query Type | Dense Best? | Sparse Best? | Example |
|---|---|---|---|
| Conceptual | Yes | No | "How to improve employee retention" |
| Exact term | No | Yes | "Error code ERR_SSL_PROTOCOL" |
| Acronym | No | Yes | "EBITDA calculation" |
| Synonym-rich | Yes | No | "laptop" finding "notebook computer" |
| Mixed | Both | Both | "Python memory leak fix" |

## Reranking in Production

Production reranking is a multi-stage funnel that balances retrieval breadth against latency constraints. Over-fetch candidates cheaply, then progressively narrow with more expensive models.

| Stage | Candidates | Latency Budget | Method |
|---|---|---|---|
| Initial retrieval | 50-100 | < 100ms | ANN search |
| Rerank pass 1 | 20-50 | < 200ms | Cross-encoder (light) |
| Rerank pass 2 | 5-10 | < 500ms | LLM reranker (optional) |
| Final context | 3-5 | N/A | Top results to LLM |

## Citation and Source Attribution

Enterprise users need to verify answers against source documents -- an answer without a citation is an answer nobody trusts. Enforce citation in your prompt template and validate that cited sources actually exist.

### Citation Pattern

```
PROMPT_TEMPLATE = """Answer the question using ONLY the provided sources.
For every claim, cite the source using [Source N] format.
If the answer is not in the sources, say "I don't have information about that."

Sources:
{formatted_sources}
```

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

def format_sources(chunks):
    formatted = []
    for i, chunk in enumerate(chunks, 1):
        formatted.append(
            f"[Source {i}] ({chunk.title}, {chunk.source_type}, "
            f"updated {chunk.updated_at})\n{chunk.text}"
        )
    return "\n\n".join(formatted)
```

### Citation Verification

```
def verify_citations(answer: str, sources: list[str]) -> dict:
    """Check that cited sources exist and claims are grounded."""
    cited = re.findall(r'\[Source (\d+)\]', answer)
    valid_range = set(str(i) for i in range(1, len(sources) + 1))

    invalid = [c for c in cited if c not in valid_range]
    uncited_sentences = [s for s in split_sentences(answer)
                         if not re.search(r'\[Source \d+\]', s)
                         and len(s.split()) > 5]

    return {
        "citations_found": len(cited),
        "invalid_citations": invalid,
        "uncited_claims": uncited_sentences,
        "all_valid": len(invalid) == 0
    }
```

## Freshness Management

Stale data produces wrong answers -- and in enterprise contexts, outdated policy or compliance information can create real liability. Your freshness strategy must match how frequently your source data changes.

| Strategy | Mechanism | Use Case |
|---|---|---|
| Full re-index | Rebuild entire index periodically | Small corpus (< 50K docs) |
| Incremental update | Add/update changed docs only | Large corpus, frequent changes |
| TTL-based expiry | Remove chunks older than threshold | News, time-sensitive data |
| Change data capture | Stream changes from source | Database-backed sources |
| Webhook-triggered | Re-index on source system events | Confluence, GitHub |

### Freshness Boosting

```
def apply_freshness_boost(results, decay_factor=0.1):
    """Boost recent documents in search results."""
    now = datetime.utcnow()
```

```
    for result in results:
        age_days = (now - result.updated_at).days
        freshness_score = math.exp(-decay_factor * age_days)
        result.final_score = result.relevance_score * 0.8 + freshness_score * 0.2
    return sorted(results, key=lambda r: r.final_score, reverse=True)
```

## Data Lineage

When an answer is wrong, you need to trace it back to the exact document, chunk strategy, and embedding model that produced it. Data lineage metadata makes debugging possible instead of guesswork.

| Field | Purpose |
|---|---|
| `pipeline_version` | Which pipeline version processed this |
| `embedding_model` | Which model created the embedding |
| `chunk_strategy` | How the doc was chunked |
| `extraction_method` | PDF parser, OCR, etc. |
| `processing_timestamp` | When it was processed |
| `source_hash` | Detect if source changed |

## Scaling Considerations

Your architecture must match your document scale -- what works for 10K documents will collapse at 10M. Plan your infrastructure tier based on where you are today and where you expect to be in 12 months.

| Scale | Docs | Architecture | Notes |
|---|---|---|---|
| Small | < 10K | Single node, ChromaDB/pgvector | Simple, low cost |
| Medium | 10K-1M | Managed vector DB + BM25 index | Pinecone, Weaviate, or Qdrant |
| Large | 1M-100M | Distributed vector DB + Elasticsearch | Sharding, replication |
| Very large | 100M+ | Multi-tier: cache + distributed search | CDN caching, query routing |

### Performance Optimization

| Technique | Impact | Complexity |
|---|---|---|
| Query caching | High (avoid repeat searches) | Low |
| Embedding caching | Medium (avoid re-embedding) | Low |
| ANN index tuning (nprobe, ef) | Medium (speed vs recall) | Medium |
| Quantized vectors (binary, PQ) | High (memory reduction) | Medium |
| Async retrieval | Medium (parallel search) | Low |

| Technique | Impact | Complexity |
|---|---|---|
| Pre-computed aggregations | High for common queries | Medium |

## Monitoring Metrics

Enterprise RAG systems degrade silently -- retrieval quality drops, indexes go stale, and ACL filters miss updates. These metrics are your early warning system for catching problems before users report them.

| Metric | Target | Alert If |
|---|---|---|
| Retrieval latency (P95) | < 200ms | > 500ms |
| Reranking latency (P95) | < 300ms | > 800ms |
| End-to-end latency (P95) | < 3s | > 5s |
| Retrieval recall@10 | > 0.8 | < 0.6 |
| Faithfulness score | > 0.9 | < 0.7 |
| Index freshness | < 1 hour | > 4 hours |
| ACL filter accuracy | 100% | Any miss |
| Guardrail trigger rate | < 5% | > 15% |

## Common Pitfalls

Enterprise RAG failures tend to be more severe than basic RAG failures -- data leakage, compliance violations, and stale answers affecting business decisions. Audit your system against this list before going to production.

| Pitfall | Problem | Fix |
|---|---|---|
| No ACL filtering | Data leakage between users | Implement ACL at retrieval time |
| Stale index | Wrong/outdated answers | Incremental re-indexing pipeline |
| Single search method | Misses keyword or semantic matches | Hybrid search (dense + sparse) |
| No reranking | Mediocre top-k quality | Add cross-encoder reranker |
| Missing citations | Users can't verify answers | Enforce citation in prompt |
| No monitoring | Silent quality degradation | Track retrieval + generation metrics |
| Flat chunking | Loses document structure | Use section-aware chunking with hierarchy |
| Ignoring document relationships | Missed cross-references | Add links/parent-child in metadata |
| No fallback for empty retrieval | "I don't know" without guidance | Add web search or escalation path |