RAG & Information Retrieval Guide for 2026

How RAG and information retrieval actually work.

Authors: Karol Pierkarski, Kacper Wlodarczyk
Contributing authors: Antoni Kozelski, Bartosz Gonczarek

Who is this e-book for?

- **Decision-makers** evaluating strategic directions for their organizations: we distill the most important insights and trends into clear, accessible language.
- **Product Managers and Product Owners:** grounded in hands-on experience and real-world case studies, we provide step-by-step guidance—from high-level concepts to implementation details—so you can engage confidently with your engineering teams on technical specifics.
- **Engineers** seeking inspiration and battle-tested solutions to everyday challenges: we share case studies, actionable recommendations, and findings from our experiments.

What will you learn?

- How RAG and information retrieval actually work, explained through plain language, concrete examples, and visual diagrams.
- The most significant challenges in designing RAG systems and proven strategies to address them.
- How to navigate the evolving landscape of retrieval technologies and best practices.

Why this guide matters now

The AI landscape is shifting rapidly. Retrieval-Augmented Generation has moved from a research curiosity to a production backbone in just a few years. Yet despite its widespread adoption, RAG systems continue to fail in predictable and avoidable ways—hallucinating answers, surfacing irrelevant documents, or becoming prohibitively expensive at scale.

This guide bridges the gap between high-level conceptual understanding and the engineering realities of building reliable RAG systems. Whether you are evaluating a vendor, scoping a project, or debugging a production pipeline, you will find actionable guidance grounded in real deployments. We cover not just what the best practices are, but why they work and when to apply them.

The ebook is structured so you can read it cover to cover for a complete foundation, or jump directly to whichever chapter or case study addresses the problem you are facing right now. Cross-references throughout make it easy to navigate between related topics.

Table of contents

Table of contents

Table of contents

How to use this guide

This ebook is written for three distinct audiences and you can tailor your reading accordingly. Decision-makers may focus on Chapters 1 and 5, plus the executive summaries at the start of each case study. Product Managers and Product Owners will find the most value in Chapters 2, 3, and 4, where the pipeline mechanics and failure modes are explained in detail. Engineers should read Chapter 4 in full—each case study includes concrete implementation notes, evaluation metrics, and code-level observations.

Chapters are largely self-contained. If you are already familiar with how RAG pipelines work, you can skip Chapter 2 and move directly to Chapter 3. If you are debugging a specific issue—say, poor retrieval precision—jump to Case 1 or Case 2 in Chapter 4. The index at the back maps common symptoms to the relevant sections.

A note on the case studies

All five case studies in Chapter 4 are drawn from real projects. Where client confidentiality permits, we have named the projects and provided detailed metrics. In cases where we have anonymised the client, the technical details, evaluation results, and lessons learned are presented in full.

Each case study follows a consistent structure: problem statement, root cause analysis, solution approach, results, and key takeaways. This makes it easy to scan for relevance and to apply the lessons to your own context.

01

Is RAG Dead? A Case for Context Engineering over Massive Context Windows

In the wake of massive context windows reaching and exceeding one-million tokens, many wonder if Retrieval-Augmented Generation (RAG) has been rendered obsolete. But that is far from the case. This chapter reveals why RAG is more essential than ever. Discover how context engineering intelligently curates information for AI models, preventing the "lost in the middle" problem that plagues oversized contexts.

Learn why RAG's transparency and explainability give it a decisive edge over the black-box of large language models, especially in regards to highly regulated industries. Explore the hidden costs behind processing millions of tokens and how selective retrieval maintains both performance and affordability. Finally, see how agentic RAG architectures combine intelligent agents with diverse data sources to deliver reliable, context-aware responses that scale beyond simple context expansion.

Why RAG is not dead

Amid the current AI boom, you may have recently heard that RAG is dead as major AI labs compete with the capabilities of new models, offering context windows of 1M tokens and sometimes even more.

If a model can process the content of several Harry Potter novels at once, why use RAG? This increasingly popular argument suggests that RAG was only necessary as a workaround for earlier models with limited context capacity. But the reality is quite the opposite.

What is RAG and what is a context window?

Before diving into the details, let us establish what RAG and context window actually mean. In its simplest form, RAG (Retrieval Augmented Generation) is a technique that allows an AI model to search for necessary information in external databases before providing an answer.

RAG is not just a document database. Recently, we have started talking more broadly about context engineering. This is the art of intelligently managing what information gets delivered to AI models, including not only document retrieval, but also context compression techniques, data organization, and the dynamic adaptation of information to specific tasks. It is like being a master of packing: instead of stuffing everything into a suitcase, you carefully select only what is really needed for the trip.

So it is about delivering our own external knowledge—that which was not baked in during the training process—to the model. This includes closely guarded company databases or confidential private user information.

McKinsey on RAG

> *"RAG allows LLMs to access and reference information outside the LLM's own training data, such as an organization's specific knowledge base, before generating a response—and, crucially, with citations included. This capability enables LLMs to produce highly specific outputs without extensive fine-tuning or training, delivering some of the benefits of a custom LLM at considerably less expense."*

— Lareina Yee, Senior Partner, McKinsey & Company, October 30, 2024

A context window is the largest chunk of text a large language model can handle, or "keep in mind," in any given instance. It serves as the model's working memory, defining how much information it can use when generating a response. Think of a context window as a chat interface like GPT: everything you input into the model plus what the model returns counts toward the context window.

The use of a RAG pattern or its neglect represent fundamentally different philosophies for information access: the difference is similar to that of a smart assistant who knows where to look for answers (RAG), and an overzealous one who brings along all possible documents "just in case" (large context window). The second option feels reassuring, but is impractical in daily use.

01

RAG vs. Large Context Windows: Direct Comparison

Aspect	RAG Approach	Large Context Window
Information Selection	Retrieves only relevant data chunks based on query	Loads all available related data regardless of relevance
Cost Efficiency	Low processing costs, pay for what you use	High computational costs for processing entire datasets
Data Management	Requires vector store updates for new information	Direct loading of current data without preprocessing
Scalability	Unlimited data sources, no token constraints	Limited by maximum context window size
Response Speed	Fast, processes only relevant data and information	Slower, must process entire data context every time
Setup Complexity	Requires retrieval system orchestration	Simple, direct data loading into context
Privacy Control	Granular access control and data permissions	All-or-nothing data access within context

Context rot: the pitfall of larger context windows

Expanding context windows has indeed been impressive in recent years. GPT's context window has exploded from a modest 2,048 tokens in GPT-3 to a full 1,000,000 tokens in GPT-4.1. Gemini, through its Flash and Pro variants, has consistently led proprietary models with best-in-class million-token capacities, while Qwen2.5-14B-Instruct-1M was the first open-source LLM to reach the 1M-token mark.

A million tokens is roughly equivalent to 750,000 words—about 200 typical academic research papers.

Context window growth timeline

Release Date	Model	Context Window
2020/06	GPT-3	2,048 tokens
2022/11	GPT-3.5	4,096 tokens
2023/03	GPT-4	8,192 tokens
2023/03	GPT-4-32k	32,768 tokens
2024/05	GPT-4o	128,000 tokens
2024/12	o1	200,000 tokens
2025/04	GPT-4.1	1,000,000 tokens

However, larger context windows do not automatically translate to equally long outputs—maximum generation length remains separately capped by model-specific decoding limits. It is like a translator who can read an entire foreign novel from cover to cover, but when translating it back, is limited to producing just one chapter at a time.

Just because these models can technically handle a million tokens does not mean they do it well. AI models often "forget" important information buried in the middle of long texts, focusing mainly on what they read at the beginning and end. Processing such massive amounts of text also gets expensive fast—sometimes costing 100 times more than a standard query.

The "lost in the middle" problem gets worse with longer texts. Models optimized for ultra-long contexts often stumble on regular tasks, becoming so specialized for extreme lengths that they lose everyday agility. Even advanced models show cracks as context grows, with attention wandering and accuracy dropping.

01

Building better AI: transparency and explainability

One of the most important factors hindering broader AI adoption in companies is the quality and reliability of data produced by models. Nothing undermines the credibility of agentic systems or the companies deploying them more than erroneous, hallucinated answers.

Even if we send reliable organizational data to the model's context, and even if the model actually manages to extract information, this is not always sufficient. Users expect transparency and reliability, especially in critical areas related to law or medicine.

Language models with large contexts most often operate as black boxes, meaning they cannot reliably provide sources for generated answers. RAG systems, on the other hand, can show where information comes from and how answers are created, which is important for companies with strict rules.

For example, when a legal assistant powered by RAG answers, "The statute of limitations for this case is 3 years," it can immediately show you it pulled this from Section 2.3 of the Civil Code, last updated on May 5, 2022, with a confidence score of 97%. A regular large-context model might give the same answer but cannot reliably tell you whether it came from page 5 or page 500 of the document you uploaded. Or worse, if it is mixing information from different sources.

While LLM weights are difficult to inspect, RAG provides transparency into its information sources and can rate how reliable the returned data is. This openness helps companies follow regulations as these systems keep detailed records that regulators want to see, so companies can prove how they made decisions and what sources they used. This is crucial in industries with strict controls where AI choices must be backed up with evidence and easy to verify.

How RAG pipeline preserves source transparency

Step 1: Document Processing & Metadata Preservation

Original documents are split into trackable chunks while maintaining complete metadata including source file names, page numbers, timestamps, and section identifiers for full traceability.

Step 2: Semantic Retrieval with Source Mapping

User queries retrieve the most relevant chunks with confidence scores, preserving exact source references and creating clear mappings between retrieved content and original document locations.

Step 3: Answer Generation with Full Attribution

The language model generates responses while the system maintains explicit connections between each part of the answer and its source chunks, providing specific citations with document names, page numbers, and confidence scores.

Step 4: Transparent Source Verification

Users receive answers with complete citations showing exactly where each piece of information originated, enabling immediate verification against original sources.

Step 5: Complete Audit Trail

The system preserves detailed logs of which documents were accessed, what chunks were used, and how the final answer was constructed, ensuring full accountability and explainability.

RAG vs. large context: the cost reality

While RAG implementation requires upfront investment in indexing systems and retrieval infrastructure, wide context window processing can work ad hoc by simply loading entire documents into the model's context. One option: dump the entire manual database into a model with a wide context window. Would it work? Most likely yes. But a RAG-based solution is more reliable and cheaper long term—it narrows the search area first, then passes only the relevant fragments to the model with precise source references.

The process of reducing context, and thus latency and cost, is a natural stage in developing any agentic application. Initially during the proof of concept stage, we check if the general project idea is feasible, not worrying about costs. However, in subsequent iterations toward launching the production version, we consider how to use smaller and faster models: less but better-selected context, while maintaining accuracy.

01

Summary: Agentic RAG models vs context length

This is just one example. In practice, every application creator strives to limit, not expand, the context of processed information, because this guarantees lower latency, greater accuracy and reduced service costs. Even if wide model context is available, it is still worth investing in context engineering, prioritizing selective delivery of the most appropriate content to the model.

RAG is not disappearing, but is being replaced by more sophisticated agentic systems that combine LLM-based agents with various data sources. Instead of by default burying the agent's context with mostly irrelevant data, system creators strive to provide the agent with filtered and precise information from multiple sources. This approach maintains balance between performance and cost as well as precision and accountability based on company-specific data. It appears to be the key to success in AI implementations.

The future of RAG rests in agentic architectures where AI systems are not limited to simple retrieval and generation, but actively reason about which tools and retrieval strategies to deploy. Emerging patterns in multi-modal retrieval combine text, images, and structured data, while real-time data integration and user personalization are transforming RAG from static pipelines into adaptive, context-aware systems that evolve with each interaction.

Agentic RAG is not a replacement for traditional RAG. It is an architectural evolution that expands what retrieval-augmented systems can do. Where traditional RAG retrieves and generates, agentic RAG plans, investigates, and synthesizes.

Traditional RAG

- Fixed retrieval pipeline
- Single-stage vector search
- Static document index
- One retrieval strategy for all queries
- Retrieve-then-generate pattern
- Limited to text modalities

Agentic RAG

- Dynamic tool and strategy selection
- Multi-step reasoning about retrieval
- Real-time index updates
- Query-adaptive retrieval strategies
- Plan-retrieve-synthesise pattern
- Multi-modal: text, image, structured data

What stays the same

The core principle of RAG—delivering the right information to the model at the right time, rather than flooding it with everything—remains constant across all architectural generations. Context engineering is not a temporary workaround. It is a permanent discipline.

Chapter 1 key takeaway

> *RAG is not dead. It has evolved. The question is no longer "RAG or large context windows?" but "how do we engineer context intelligently for each specific task?" Every production AI system in 2026 answers this question with selective retrieval.*

The evolution of RAG in the age of Agentic AI

Within this ebook, we will cover precisely what RAG is, how RAG systems work, the most common issues which arise and how to overcome them, present best practices through hands-on case studies, and examine the future of Information Retrieval, covering context engineering and agentic RAG, as they emerge onto the 2026 market.

02

How RAG Systems Work

Within this chapter you will find a comprehensive introduction to the typical RAG pipeline with clear explanations of each stage. We begin by breaking down the process in brief, followed by a much more detailed breakdown of each step.

In short, the TL;DR: Retrieval-Augmented Generation systems operate through a straightforward five-stage workflow pipeline that ensures reliable, data-grounded responses:

- First, **data collection:** the gathering and cleaning of information from various sources, creating a foundation of accessible content.
- Second, the **preparation stage** is comprised of structuring this data into searchable chunks with metadata and embeddings for efficient retrieval.
- Then, during the **retrieval phase,** user queries are transformed into searches that identify the most relevant information from the knowledge base.
- The **generation stage** uses AI models to synthesize answers based solely on the retrieved context, avoiding hallucinations.
- Finally, **verification and improvement** are required to validate accuracy, provide source attribution, and incorporate feedback to enhance future performance.

In this chapter, we will pull back the curtain and share the tools, methodologies, and tricks we use to build robust and reliable RAG systems so you can get a firm handle on what is going on under the hood.

The five-stage RAG pipeline

Stage 1: COLLECT YOUR DATA
Gather documents from all your sources. Clean and organize the content. Make sure the data is readable.
Stage 2: PREPARE FOR SEARCH
Break documents into smaller pieces. Add labels and descriptions (metadata). Create searchable index.
Stage 3: FIND RELEVANT INFORMATION
Convert user question into search format. Search through your indexed data. Rank and select the best matches.
Stage 4: GENERATE THE ANSWER
Feed relevant info to the AI. Generate response based on your data. Ensure answer stays grounded in sources.
Stage 5: VERIFY & IMPROVE
Check if answers are accurate. Link answers back to original sources. Learn from feedback to get better.

Not every RAG system follows these exact five stages—your implementation may combine or split these differently. But this framework helps to visualize the conceptual flow and recognize where things can typically go wrong.

You can think of RAG like a research assistant working through your company's knowledge base. First, it collects and cleans your data from all sources. Then it prepares everything for search by breaking documents into chunks and creating an indexed library. When you ask a question, it finds the relevant information by searching and ranking the best matches. Next, it generates an answer by feeding those specific passages to the AI—keeping responses grounded in your actual documents rather than invented facts. Finally, answers are verified and improved by checking accuracy, citing sources, and learning from feedback.

02

What is IR in RAG?

Information Retrieval (IR) takes place in Stage 3 and is the process of extracting relevant information from various sources. In the context of RAG systems, this typically means retrieving data from a **knowledge base**—most often a **vector database**. Here is how it works step by step:

- **Prompt encoding** — Your input question (the prompt) is converted into a numerical representation by an embedding model.
- **Similarity search** — This embedding vector is compared to entries in a vector database, which stores knowledge chunks (documents, paragraphs, text snippets) as vectors.
- **Result selection** — The retriever finds the most similar entries and returns the most relevant documents to be passed to the generator.

Several factors significantly impact the final retrieval results:

- **Relevance** — Only documents closest to the query in vector space should be returned. Closeness is computed with a similarity score, most often cosine similarity, which measures how closely two vectors point in the same direction rather than how long they are. It is commonly used because direction tends to reflect meaning more reliably than raw distance, especially when text length varies.
- **Controlling result count** — Returning thousands of documents is inefficient and wastes tokens in the LLM's context window.
- **Avoiding under-retrieval** — Returning just a single document may cause loss of valuable context.
- **Determinism** — Many retrieval algorithms are non-deterministic, meaning results may vary slightly between identical queries.

McKinsey on RAG

> *"RAG allows LLMs to access and reference information outside the LLM's own training data, such as an organization's specific knowledge base, before generating a response—and, crucially, with citations included. This capability enables LLMs to produce highly specific outputs without extensive fine-tuning or training, delivering some of the benefits of a custom LLM at considerably less expense."*

— McKinsey & Company

Jorge Amar, McKinsey Senior Partner

> *"I do think of it as a workforce. This is a workforce that will conduct end-to-end processes, replacing many tasks being performed today by the human workforce."*

— Jorge Amar, McKinsey Senior Partner, The future of work is agentic

Retrieval strategy overview

- **Keyword search** — matches exact words; fast and deterministic; best for precise terms, codes, and identifiers
- **Semantic search** — understands meaning; handles synonyms and vague queries; uses vector embeddings
- **Hybrid search** — combines both via fusion (e.g., RRF); best practice for production systems
- **Metadata filtering** — pre-filters by attributes (date, author, category) before search is applied

02

Keyword Search

The first and simplest retrieval method is **keyword search**, which looks for documents that contain specific words mentioned in the user query. This approach is based on traditional text-matching techniques and is widely used in search engines. Two popular algorithms within this category are **TF-IDF** and **BM25**.

What is TF-IDF?

TF-IDF (Term Frequency–Inverse Document Frequency) is a classical statistical method for measuring the importance of words in a document relative to a collection (corpus) of documents. Instead of using dense vectors like in modern neural networks, TF-IDF uses **sparse vectors** where each feature represents a specific word in the vocabulary. It consists of two main components:

Term Frequency (TF) — measures how frequently a term appears in a document:

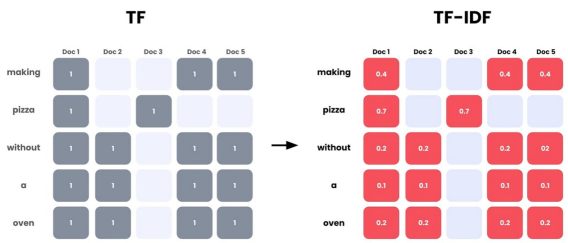
$$TF = \frac{\text{Number of occurrences of a term in a document}}{\text{Total number of words in the document}}$$

This alone, however, favors longer documents that naturally contain more keywords. To address this, we introduce the second component:

Inverse Document Frequency (IDF) — measures how rare a term is across all documents:

$$IDF = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing the term}}\right)$$

The final **TF-IDF score** is the product of these two values: **TF × IDF**. This ensures that commonly used words (e.g., "the", "and") receive lower importance, while more distinctive terms contribute more to the overall document score.



What is BM25?

BM25 is an advanced ranking function that builds upon TF-IDF. It introduces improvements to better handle document length and term saturation. Key enhancements over TF-IDF:

- **Saturation effect** — prevents a term that appears very frequently from disproportionately dominating the score.
- **Length normalization** — avoids favoring longer documents just because they include more words.

While the IDF part remains similar to TF-IDF, these additional terms allow for fine-tuning how much term frequency and document length influence the final score. This makes BM25 more robust and effective for real-world information retrieval tasks.

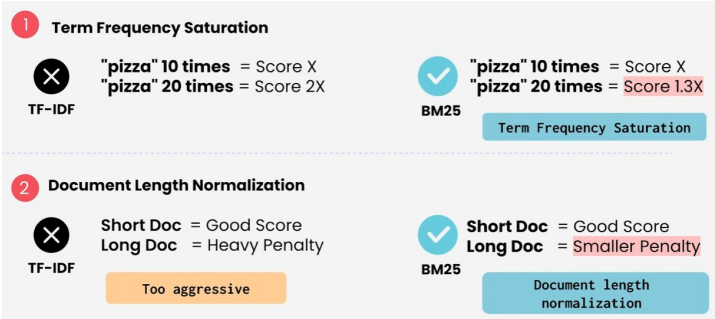


Figure: BM25 improves upon TF-IDF by applying term frequency saturation (reducing the weight of repeated terms) and balanced document length normalization (avoiding overly harsh penalties on longer documents).

Semantic Search

Semantic search is a more advanced retrieval technique compared to the keyword-based search. Instead of relying on exact word matches, it aims to understand the **meaning** behind a user's query and the content of documents. It does this by mapping text into a **high-dimensional vector space**, where the **semantic similarity** between pieces of text can be measured mathematically. This allows a semantic search system to recognize that phrases like "buying a house" and "purchasing a home" refer to the same concept, even though they do not share the exact wording.

02

How does semantic search work?

- **Text-to-vector conversion:** Both the query and documents are transformed into fixed-length vectors (embeddings) using a neural embedding model trained to capture the meaning of language.
- **Vector similarity calculation:** The system compares the query vector to each document vector using cosine similarity (angle between vectors), dot product (projection of one vector onto another), or Euclidean distance (straight-line distance between vectors).
- **Retrieval of semantically relevant documents:** The most similar vectors are returned as the top relevant results, regardless of whether they share any specific keywords with the query.

Even if two documents are "about the same thing," these metrics can rank them differently. For example, imagine your query is "refund policy for annual plans." **Cosine similarity** tends to favor a short focused snippet because it mainly cares whether the vectors point in the same direction, not how large they are. **Dot product** can favor a longer page if its embedding has a larger magnitude. **Euclidean distance** can also penalize magnitude differences. This is one reason why many systems normalize embeddings before using distance-based retrieval.

This method is especially powerful in cases where:

- The vocabulary between the query and the documents differs (e.g., synonyms)
- The wording is imprecise or vague
- More abstract or conceptual information is being sought

What are embedding models?

Modern RAG systems use sentence- or document-level embedding models that encode both queries and knowledge-base chunks into dense vectors for similarity search. These models are trained specifically for retrieval and semantic similarity, producing a single vector per chunk that can be efficiently indexed and compared using nearest-neighbor search. In practice, teams use either hosted API models such as OpenAI's text-embedding-3-small or text-embedding-3-large, or open-weights alternatives like the BGE, E5, or JinaAI families. Smaller models are often selected for lower latency, cost, or on-device deployment, while larger models typically improve recall, robustness, and multilingual performance on complex queries.

Coming next: vector space explained

At the core of semantic search lies the concept of a **vector space** — a mathematical abstraction where texts become points in high-dimensional space, with similar meanings positioned closer together. Turn to the next page for a full explanation and visual diagram.

02

What is a vector space?

At the core of semantic search lies the **vector space**: a mathematical abstraction where each piece of text is represented as a point in a **high-dimensional space**. In this space, semantically similar texts are positioned **closer together**, while unrelated ones are **further apart**. The number of dimensions typically ranges from **128 to 4096**, depending on the embedding model used.

To understand what this means in practice, consider a simple two-dimensional example. If we project word embeddings onto a plane representing "food-related" and "animal-related" concepts, words like **"food"** and **"cuisine"** cluster together—their vectors point in similar directions. Meanwhile, **"cat"** sits near other animal terms, and an unrelated word like **"trombone"** lands far from both clusters.

This geometric property is what makes semantic search powerful. A query about "what to eat in Tokyo" will retrieve documents about **"Japanese cuisine"** and **"street food"** even if those exact words never appear in the query—because their vectors are nearby in the space. This is impossible with pure keyword matching.

How similarity is measured

Once texts are represented as vectors, similarity between a query and a document is calculated using one of three distance metrics:

- **Cosine similarity** — measures the angle between two vectors. It is scale-invariant, meaning a short focused snippet and a long document can score equally if they point in the same semantic direction.
- **Dot product** — measures both angle and magnitude. Longer documents with larger embedding magnitudes may score higher, which can be useful when document length correlates with relevance.

Vector space diagram

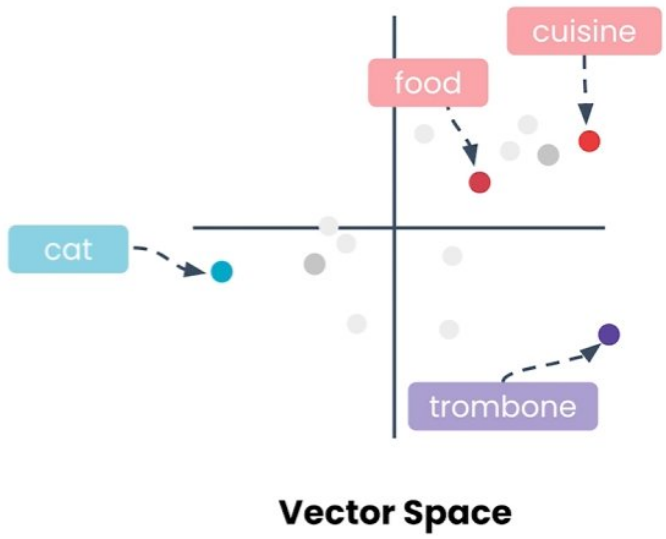


Figure: In a vector space, semantically similar words cluster together. "Food" and "cuisine" are nearby; "cat" sits in a different cluster; "trombone" is an outlier with no semantic neighbours in this domain.

Why dimensions matter

Real embedding models operate in hundreds or thousands of dimensions simultaneously—not just two. Each dimension captures a different latent aspect of meaning: topic, tone, formality, domain, entity type, and more. No single dimension maps to a human-readable concept; it is the combination of all dimensions together that encodes meaning.

Higher-dimensional models generally capture finer semantic distinctions and perform better on complex or multilingual queries, but require more storage and compute. Lower-dimensional models are faster and cheaper, making them suitable for large-scale production deployments where latency and cost are priorities.

Nearest-neighbour search

At query time, the system must find the vectors in the index that are closest to the query vector. Doing this exhaustively over millions of documents would be too slow, so production systems use **Approximate Nearest Neighbour (ANN)** algorithms—such as HNSW or IVF—that trade a small amount of recall for dramatically faster search. Libraries like FAISS, Qdrant, Weaviate, and Pinecone implement these algorithms at scale.

02

Hybrid Search

Hybrid search combines the strengths of both the **keyword-based** and **semantic search** approaches to deliver results that are both **lexically accurate** and **semantically relevant**. This method allows for more robust and flexible retrieval, especially in complex queries where either keywords or semantics alone may fall short. In a typical hybrid setup, both search methods are executed **independently**. Each returns its own set of top-ranked results (commonly referred to as top_k), which are then passed into a **fusion mechanism**, which merges and reorders them to produce a unified, optimized output. One popular and effective method for this merging process is **Reciprocal Rank Fusion (RRF)**.

What is Reciprocal Rank Fusion (RRF)?

Reciprocal Rank Fusion is a simple yet powerful algorithm used to combine results from multiple ranked lists. It works by assigning a **combined score** to each document based on its position (rank) in the individual search result lists.

Suppose you run two retrieval methods over the same documents: a keyword-based search and a semantic (embedding-based) search. One document ranks **2nd** in the keyword results and **15th** in the semantic results, while another document ranks **5th** in both lists. Under RRF, the second document (5th/5th) typically ranks higher overall, because it receives strong contributions from *both* retrieval methods, whereas the first document relies heavily on a single high ranking. This illustrates how RRF favors documents that perform well across multiple lists, while still rewarding documents that rank very highly in at least one.

What makes RRF robust?

Unlike weighted score combination, RRF requires no calibration between retrieval methods. You do not need to normalise scores from BM25 and a vector index onto the same scale — a notoriously difficult problem when the two systems use completely different scoring functions. RRF sidesteps this entirely by working only with rank positions, making it both simple to implement and surprisingly effective in practice.

When to use hybrid search

- **Mixed query types** — users ask both precise factual questions and open-ended conceptual ones
- **Domain-specific vocabulary** — product codes, legal citations, or medical terms that embeddings may not encode well
- **Long documents** — where keyword anchoring helps prevent semantic drift
- **Multilingual corpora** — where embedding quality varies across languages

RRF in plain terms

For each document, RRF sums 1 / (k + rank) across all ranked lists, where k is a constant (typically 60) that dampens the outsized influence of top-ranked documents. A document ranking 1st in one list and 30th in another scores lower than one ranking 5th in both — RRF rewards consistency across methods over dominance in a single one.

Coming next: metadata filtering

Hybrid search finds the right documents — metadata filtering makes sure you are only searching the right documents in the first place. The next page covers how attribute-based pre-filtering reduces noise before retrieval begins.

> *The business world moved from trying out chatbots with credibility level adjusted using model temperature settings. The systems of today can have hallucinations structurally reduced to zero if they are engineered to do so. It is the case when they operate within a range of possibilities that is contextualised and validated by business logic. When such circumstances can be created — and they are created in our projects — the LLM becomes a flexible tool for both customers and internal users in a variety of scenarios.*
>
> *— Bartosz Gonczarek, Chief Transformation Officer, Vstorm*

02

Metadata Filtering

Metadata filtering is a complementary retrieval mechanism that narrows the candidate document set based on structured **attributes** — such as title, author, publication date, category, language, or any custom tag attached to a document at ingestion time. Unlike semantic or keyword search, metadata filtering does not analyse the content of a document at all. It applies **strict logical conditions** to eliminate documents that do not match specified criteria before any content-based search takes place.

A practical example: a legal knowledge base containing contracts spanning 2010 to 2025. A query about GDPR obligations is only relevant to documents from 2018 onwards. Without metadata filtering, the retrieval system would search the entire corpus and risk surfacing pre-GDPR contracts that are factually correct but legally irrelevant. With a date filter applied first, the search space shrinks dramatically — and so does the risk of misleading output.

What metadata can be filtered

Any structured attribute attached to a document at ingestion time can serve as a filter. Common examples in production RAG systems include:

- **Temporal attributes** — publication date, last updated, version number, fiscal year
- **Categorical attributes** — document type (contract, FAQ, policy, report), topic, department
- **Source attributes** — author, team, data source, system of origin
- **Access attributes** — security clearance level, user role, jurisdiction
- **Language and locale** — filtering to documents in the user's language or region

Limitations

- **Not a search method** — Metadata filtering does not evaluate content relevance. A document can pass all filters and still be completely off-topic.
- **Discards potentially useful content** — Strict filters may exclude high-quality documents that fall just outside the criteria (e.g., an earlier document that introduced a concept later formalised in regulations).
- **Requires quality metadata** — Inconsistent or missing tags make filters unreliable. A "date" filter is useless if half the documents have no date attached.
- **Ineffective alone** — Must always be paired with semantic or keyword search to actually rank and retrieve relevant content.

Pre-filtering vs post-filtering

Metadata filtering can be applied at two points in the retrieval pipeline. **Pre-filtering** reduces the candidate set before search runs, improving both precision and speed. **Post-filtering** removes results after search has completed, preserving recall at the cost of potentially discarding top-ranked results.

In most production systems, **pre-filtering is preferred** — it is faster, cheaper, and prevents the language model from ever seeing irrelevant documents. Post-filtering is occasionally used when the filter criteria are derived from the query itself and cannot be evaluated until after retrieval.

IR in RAG systems in short

The four retrieval mechanisms — keyword search, semantic search, hybrid search, and metadata filtering — work best in combination. Pre-filter with metadata to narrow the candidate pool, then apply hybrid search to rank what remains, fused with RRF to balance lexical and semantic signals. Each layer is independently tunable.

Coming next: when RAG fails

Even a well-designed retrieval stack can break down. The next page identifies the five pipeline stages where RAG systems most commonly fail — and what each failure looks like in practice.

02

When RAG fails: the five-stage breakdown

Each stage of the RAG pipeline is a potential failure point where hallucinations can creep in, sabotaging both efficiency and reliability. The problems covered here are not exhaustive — every dataset and use case brings unique challenges. What we share comes from real-world experience building these systems in production.

Stage 1 — Data Extraction & Ingestion: accessible does not mean retrievable

Successfully extracting data does not mean it is in a form that supports meaningful retrieval. Even perfectly accessible content may be corrupted, fragmented, inconsistent, or encoded in ways that embeddings cannot understand. These are universal problems that occur regardless of source type.

What happens here:

- Collect data from diverse sources: documents (PDFs, scans), websites, databases, APIs
- Extract raw content into clean, accessible text or structured formats
- Preserve source context and metadata where available

Stage 2 — Data Structuring & Preparation: from clean data to retrieval-ready chunks

This stage transforms clean data into retrieval-ready content and every structural decision has consequences. Arbitrary chunking breaks context mid-thought. Missing metadata eliminates your ability to filter by date, version, or category. Generic embeddings trained on web text struggle to capture specialized domain terminology. Structure your data poorly here and relevant information becomes effectively invisible to search, even though it exists somewhere in your knowledge base.

What happens here:

- Transform extracted data into formats optimised for retrieval
- Chunk documents into logical, retrievable segments
- Extract and enrich metadata (dates, categories, entities, relationships)
- Generate embeddings using appropriate models
- Index content for retrieval

Why these two stages matter most

Stages 1 and 2 are upstream of retrieval and generation, yet they determine the quality of everything that follows. Flawed data at ingestion cascades through every downstream component — producing retrieval misses and generation errors that are difficult to trace back to their root cause.

Continued on next page

Stages 3, 4, and 5 — where retrieval, generation, and evaluation can each break down — are covered on the following page.

02

Stage 3 — Retrieval Layer: when relevant documents go unretrieved

No model can provide an answer that it cannot see. When retrieval fails, even the most capable language model must choose between refusing to answer, inventing information from its training data, or piecing together responses from fragments that miss the point. Poor retrieval makes correct information effectively invisible: semantic mismatches miss specialized terminology, outdated indexes return stale facts, and misclassified intent searches for the wrong documents entirely. Context quality determines answer quality.

Stage 4 — Generation & Alignment Layer: when components work in isolation but fail together

Even with perfect retrieval, generation can fail. Language models are trained to complete text plausibly, not necessarily accurately. Retrieval and generation can each work perfectly in isolation but fail when combined — misalignment creates subtle errors that are hard to detect because both subsystems appear functional. The model may ignore retrieved context entirely, blend it with training data, or misinterpret what it sees.

Stage 5 — Evaluation & Feedback: the blind spot problem

If you cannot measure hallucinations, you cannot fix them. Evaluation is often the weakest link, with teams measuring what is easy (fluency, speed, user satisfaction) instead of what matters (accuracy, faithfulness, factual consistency). Without proper evaluation, errors accumulate silently, biases go undetected, and subtle failures compound until user trust collapses. This is not a one-time checkpoint but an iterative cycle.

The five stages: a summary

Stage 1 — Data Extraction & Ingestion

Accessible does not mean retrievable. Content quality at ingestion determines the ceiling of everything downstream.

Stage 2 — Data Structuring & Preparation

Chunking, metadata, and embedding decisions made here shape what the retrieval layer can and cannot find.

Stage 3 — Retrieval

Context quality determines answer quality. The model cannot answer what it cannot see.

Stage 4 — Generation & Alignment

Retrieval and generation can each work in isolation and still fail when combined. Misalignment is subtle and hard to detect.

Stage 5 — Evaluation & Feedback

You cannot fix what you do not measure. Evaluation is not a one-time checkpoint — it is an iterative loop.

From stages to systems

Many accuracy problems do not respect stage boundaries. Temporal conflicts span ingestion, retrieval, and generation. Intent mismatches require coordination between retrieval and generation. The next page draws these stages together and sets the direction for the chapters that follow.

02

From stages to systems: what comes next

We have walked you through the five stages of the RAG pipeline, taken an in-depth look at Information Retrieval, and examined where RAG systems break — from corrupted source data to blind evaluation. Each stage has its own failure modes, and fixing them in isolation is necessary but not sufficient.

In reality, many accuracy problems do not respect stage boundaries. Temporal conflicts span ingestion, retrieval, and generation. Garbage data at stage 1 cascades through every downstream component, passing faithfulness checks while being fundamentally wrong. Intent mismatches require coordination between retrieval and generation. Multi-hop reasoning failures emerge from the interaction between how you index, what you retrieve, and how the model synthesises information.

This is why building reliable RAG systems requires a systems perspective — not just optimising each stage in isolation, but understanding how decisions at one layer propagate through all others. A retrieval strategy that works well for short queries may degrade silently on complex multi-part questions. A chunking approach optimised for one document type may corrupt context for another.

What the rest of this ebook covers

The chapters that follow take each failure mode seriously and offer concrete, tested responses drawn from production implementations.

- **Chapter 3** introduces the most common RAG failure patterns — with real examples, root cause analysis, and remediation strategies for each.
- **Chapter 4** covers advanced retrieval techniques including query rewriting, re-ranking, and multi-vector indexing.

Chapter 2 key takeaway

> *RAG is not a single component — it is a pipeline. Each stage introduces its own failure modes, and improving overall system reliability requires understanding how these stages interact, not just how each performs in isolation.*

The five stages at a glance

Stage 1 — Data Extraction & Ingestion

Accessible does not mean retrievable. Content quality at ingestion determines the ceiling of everything downstream.

Stage 2 — Data Structuring & Preparation

Chunking, metadata, and embedding decisions made here shape what the retrieval layer can and cannot find.

Stage 3 — Retrieval

Context quality determines answer quality. The model cannot answer what it cannot see.

Stage 4 — Generation & Alignment

Retrieval and generation can each work in isolation and still fail when combined. Misalignment is subtle and hard to detect.

Stage 5 — Evaluation & Feedback

You cannot fix what you do not measure. Evaluation is not a one-time checkpoint — it is an iterative loop.

03

Common Issues and How to Overcome Them

RAG is one of the most effective tools to tackle hallucinations in information retrieval — especially where expert knowledge is required and a model's built-in knowledge is far from sufficient. The most visible outcome of a model lacking the right information is hallucination: responses that are confidently wrong.

The problem proved immense enough that Cambridge Dictionary named "hallucinate" its word of the year for 2023. Left unchecked, culinary AI systems have suggested chlorine gas as a cooking ingredient, and studies show fabricated citations appear in up to one in five academic papers generated by LLMs.

The origin of hallucinations

Hallucinations are not exactly errors — they are a natural consequence of how LLMs are designed. These systems were built to be both helpful and creative, producing a unique response to every query. When interacting with a vanilla LLM like a locally-run Mistral or Llama instance, responses are frequently fabricated.

In some contexts, this is a feature. When a user asks for a draft email or a fairy tale about unicorns with lasers, the response is entirely constructed from no verifiable facts — technically hallucinated, functionally correct. The problem arises when that same generative tendency is applied to factual queries where accuracy is non-negotiable.

How RAG mitigates hallucinations

RAG can be compared to using a search engine. The user prompts the system, the engine returns a set of results of varying accuracy, and system architects are responsible for making those results as relevant as possible.

RAG-related risks

Properly tuned RAG systems are among the most effective tools to mitigate hallucinations, context loss, retrieval irrelevance, and gaps in LLM knowledge. Yet their complex, multi-stage nature means that "hallucinations" are often just symptoms of predictable pipeline breakdowns.

Not every RAG system follows the same five stages precisely — implementations may combine or split them differently. But this framework helps contextualise the flow and identify where things typically go wrong. The problems covered here are not exhaustive: every dataset and use case brings unique challenges. What follows comes from real-world experience building these systems in production.

The five stages at risk

Stage 1 — Data Collection

Things can break before a single line of RAG code is written. Data quality, format, and structure all determine the ceiling of what is retrievable downstream.

Stage 2 — Data Structuring & Preparation

Chunking strategy, metadata consistency, and embedding model choice shape what the retrieval layer can and cannot find.

Stage 3 — Retrieval

The core of RAG operations. Retrieval failures cascade directly into generation failures — the model cannot answer what it cannot see.

Stage 4 — Generation & Alignment

Even with perfect retrieval, models can blend facts with fiction, ignore context, or fill gaps with plausible-sounding fabrications.

Stage 5 — Evaluation & Feedback

The trickiest stage. Hallucinations cannot be reduced if they are not measured. Most teams measure fluency rather than accuracy.

03

Stage 1 — Collecting the data

This is where things may break even before the actual RAG starts to operate — sometimes before the first line of code is written. At this point the team needs to gather documents from all sources, clean them, and organise them in a form the system can read.

Data quality

The data may be corrupted, missing, or flawed in a multitude of ways — from mundane typos and missing fragments to more systemic structural damage.

How to mitigate: Sanity checks and diligent ongoing control of the database.

Outdated data

The data itself may be correct and truthful, yet no longer relevant. For example, a document may still show Bill Gates as CEO of Microsoft when the current CEO is Satya Nadella. Inserting outdated data may be accidental or intentional — for example, to allow users to analyse historical context.

How to mitigate: Timestamp the data, enable the system to filter by time, and regularly update time-sensitive records in the dataset.

Format inconsistencies

Data may use multiple formats across documents — confusing European and US decimal notations, or mixing Imperial and metric units. The system may be unable to determine whether 01.03.2026 refers to the third of January or the first of March.

How to mitigate: Standardise all documents to ISO 8601 for dates, and normalise currencies and units across the dataset.

Encoding and character issues

The dataset may include special characters, mathematical notation, or non-internationalized names that pose challenges for the parser. These can enter the system without the team's awareness and cause unexpected failures at retrieval time.

How to mitigate: Enforce UTF-8 encoding consistently across the entire database.

Data redundancy and duplication

Depending on data practices within the organisation, multiple copies or near-duplicates at varying levels of relevance may exist. Conflicting versions confuse both users and the system. Different versioning of the same information can result in contradictory content being retrieved with identical relevance scores.

How to mitigate: Cleanse the dataset and remove duplicates before ingestion.

Structural issues

Data in its original format may include tables, formatting, hierarchies, or other structured elements that carry meaning. When transferred into a flat knowledge base, the system may lose this nuanced information — making retrieval significantly harder.

How to mitigate: Preserve data as JSON, Markdown, or a comparable structured format that maintains all properties and supports multiple search scenarios.

Information fragmentation

The context and key pieces of information may be split across multiple locations or documents, hampering the system's ability to form a complete picture of available knowledge.

How to mitigate: Implement link resolution for paginated content, parent-child chunk relationships, and rich metadata to preserve cross-document context.

03

Stage 1 in short

Data collection failures are upstream failures. Every flaw introduced at this stage — corrupted records, outdated facts, missing structure, duplicate conflicts — propagates through every subsequent stage of the pipeline. The most reliable way to prevent downstream hallucinations is to invest seriously in data quality before retrieval ever begins.

Stage 2 — Data structuring and preparation

Once data is collected and cleansed, the next phase transforms raw pieces of information into a workable dataset — legible and formatted to the needs of both the RAG and LLM components. Machines perceive structure differently from humans. For maximum efficiency, the dataset needs to be sliced into processable chunks that follow a logical structure and are easy to retrieve. Metadata must be added and embeddings generated.

Arbitrary or flawed chunking strategies

Chunks need to preserve the logical flow and context of the retrieved content. A common pitfall is slicing by token count, which results in cutting chapters or sentences in half and breaking chains of thought mid-argument.

How to mitigate: Chunk according to the structure of the data. Semantic chunking is one of the most effective approaches. Leaving 10–20% overlap between consecutive chunks helps the system maintain context across boundaries.

Missing or inconsistent metadata

Every chunk needs proper metadata including timestamps, document type, source, and version number. Overlooking this step may result in the system delivering contradictory results or missing relevant chunks entirely. Metadata drives citations, filtering, and chunk selection far more efficiently than scanning the entire dataset.

Generic embeddings for specialised domains

Most available embedding models were trained on general web data, making them effective for common knowledge and generic reasoning. In specialised domains — healthcare, finance, legal — domain-specific jargon and terminology may be misrepresented or confused.

How to mitigate: Fine-tune and validate embedding models on domain-specific vocabulary before deployment.

Embedding-model data mismatch

The chosen embedding model may not be aligned with the content it is encoding. For example, the data may be multilingual while the model operates in English only. This mismatch results in poor semantic representations and retrieval failures that can be difficult to diagnose.

How to mitigate: Match the embedding model to the content type and language profile of the dataset.

Embedding dimension trade-offs

Choosing an embedding size without considering performance and cost balance is a common oversight. Higher-dimensional embeddings capture more semantic nuance but increase computational cost and query latency. Lower-dimensional embeddings are faster and cheaper but may sacrifice recall on complex queries.

How to mitigate: Profile embedding dimensions against the actual dataset and query types. Select the configuration with the best cost-performance ratio for the specific use case.

Stage 2 in short

Data structuring decisions define the ceiling of retrieval quality. Arbitrary chunking breaks context. Missing metadata eliminates filtering capability. Generic embeddings produce semantic drift in specialised domains. None of these failures announce themselves loudly — they surface later as retrieval misses and generation errors that are hard to trace to their root cause.

03

Stage 3 — Retrieval

Retrieval is the core of RAG operations — the moment the system accesses the documents needed to deliver an answer. The challenge is ensuring the model receives the right content. Without it, the model may answer using mismatched context, fall back on general training knowledge, or hallucinate an answer entirely.

Irrelevant or low-recall retrieval

The most obvious failure: the system fails to deliver relevant chunks for the LLM to use. Missing context forces the model to answer from other available information, return "no information available," or hallucinate entirely.

How to mitigate: Reinforce retrieval with hybrid search techniques. In enterprise settings, having the LLM rewrite the query to better fit the knowledge base is common practice. Re-rank retrieved chunks iteratively for better results. Evaluate recall independently of the generation model.

Not enough information retrieved

Despite perfect dataset preparation, the right answer may not appear within the top results returned by the system — particularly for complex or multi-part queries.

How to mitigate: Increase retrieval breadth by raising the number of fetched chunks for complex queries. Apply query decomposition — split complex questions into simpler sub-queries and aggregate their retrieved context before generation.

Embedding drift or domain mismatch

The embedding model may not capture specialised vocabulary or domain semantics, causing the system to miss relevant documents despite exact vocabulary matches in the source.

Query intent misclassification

The system may misinterpret user intent — especially when prompts are ambiguous or unclear. This typically occurs when there is an imbalanced interplay between keyword and semantic search, with the system ignoring domain-specific terms in favour of more general ones.

How to mitigate: Identify the type of information being requested. Clarify or refine the query by recognising key names, terms, and topics. Rewrite the query in a clearer form or route it to the correct subject area before searching.

Poor metadata filtering

Even with well-applied metadata — timestamps, categories, document types — the system may fail to apply those filters effectively at retrieval time. With incorrect content retrieved and irrelevant categories applied, the model generates suboptimal and potentially misleading responses.

How to mitigate: Pre-filter by metadata before vector search runs, to avoid semantic drift dominating retrieval. Partition data by category where appropriate. Apply query augmentation techniques to extract relevant metadata attributes for each specific query.

Stage 3 in short

Retrieval is where the pipeline's upstream investments pay off — or fail to. Good data preparation and embedding quality are necessary conditions for good retrieval, but not sufficient ones. Query intent, metadata filtering, and retrieval breadth all require their own tuning. Retrieval failures are the most direct cause of generation hallucinations and the most tractable to diagnose and fix.

03

Stage 4 — Generation and alignment

Despite all the reinforcements described above, challenges can still emerge in the final step before response delivery. Language models are designed to sound plausible, not necessarily accurate. Without proper constraints, they blend facts with fiction. Unlike humans, they have no connection to the real world outside their training data.

Loose or missing grounding

The model may introduce completely irrelevant information, blending retrieved facts with hallucinations. This occurs when the system lacks explicit instructions to stay within the retrieved context.

How to mitigate: Use prompt engineering to explicitly instruct the model to answer only from retrieved information, use citations, and show sources.

Gap-filling with hallucinations

Retrieval may work correctly, yet no database contains all knowledge. When the answer is genuinely absent, the model may fill the gap with plausible-sounding fabrications.

How to mitigate: Set temperature to minimal creativity. Force the model to show sources. Explicitly instruct the system to state when it does not know the answer rather than invent one.

Information overload

Too much retrieved context can overwhelm the model's effective attention span, resulting in suboptimal or incoherent generation.

How to mitigate: Ensure the best possible filtering and ranking of chunks passed to the model. Limit context to the most relevant and recent chunks rather than maximising volume.

Semantic mismatch

The model may deliver information that is contextually wrong yet semantically close — a consequence of the gap between ground truth and the knowledge embedded in the LLM. For example, a system designed as a chess instructor, when asked about queens attacking kings, may return a story about the Byzantine Empire instead.

How to mitigate: Test and validate answers against known ground truth. Use prompt engineering to constrain the model strictly to its intended domain and task type.

Cross-document reasoning failure

The system may confuse entities from different sources — incorrectly splitting or fusing them in ways that produce factually wrong composite answers. This is particularly common in knowledge bases covering multiple organisations, products, or time periods.

How to mitigate: Track separate entities in the dataset and include entity identifiers in metadata. Ensure retrieved chunks are not mixed unpredictably — sort or filter by document year, title, or source before passing context to the model.

Stage 4 in short

Generation is where all upstream decisions materialise as user-visible output. Even a well-tuned retrieval system can produce hallucinated responses if the model is not grounded, constrained, and validated. Prompt engineering, temperature control, citation enforcement, and domain scoping are not optional refinements — they are structural requirements for production-grade RAG systems.

03

Stage 5 — Evaluation and feedback

The last stage is arguably the trickiest, because this is where actual hallucinations and mistakes surface and must be measured. It is impossible to reduce the number of hallucinations if they are not counted. System failures range from critical hallucinations to subtle misinterpretations — and no amount of prompt engineering bypasses the need for rigorous testing.

Omitting accuracy in evaluation metrics

Most evaluation metrics focus on fluency and readability, not factual accuracy. Yet grounded accuracy is what users actually depend on.

How to mitigate: Include fact-checking metrics such as RAGAS faithfulness or TruLens groundedness. Evaluation tooling must include answer-source overlap scoring.

No human or expert validation

Automated validation alone is insufficient for complex domains like healthcare or finance, where failures carry legal, compliance, or safety implications.

How to mitigate: Domain experts must review samples in repeatable cycles to keep the system observed and under control.

No source traceability

If the system delivers answers without citing sources, it becomes impossible to determine whether results are factual, hallucinated, or manipulated.

How to mitigate: Include structured logging of retrieval-to-generation mappings so every answer can be traced back to its source.

No continuous feedback mechanisms

Without an active, ongoing error-detection process, errors accumulate over time — compounded by new unverified data or obsolete duplicates entering the system.

Confidence calibration issues

The system may be overconfident in incorrect responses, with no mechanism for distinguishing reliable from unreliable outputs. Overconfidence is more dangerous than uncertainty — users are less likely to question a confident wrong answer.

How to mitigate: Design and enforce confidence scores. Surface uncertainty explicitly in system outputs where appropriate.

No A/B testing

In production systems, any change to retrieval strategy, embedding model, chunking approach, or prompt template should be validated through controlled experimentation before full rollout.

How to mitigate: Roll out every new version gradually, observing defined metrics across performance, cost, and user satisfaction before committing to changes.

Missing cost-performance tracking

It is possible to build a system that is highly accurate yet prohibitively expensive — making the effort unprofitable. Accuracy and cost must be tracked together.

How to mitigate: Track cost per query and accuracy per dollar spent. Make deliberate trade-offs where necessary.

Stage 5 in short

Evaluation is not a one-time checkpoint — it is an ongoing discipline. You cannot fix what you do not measure, and you cannot improve what you do not track. The most reliable RAG systems combine automated metrics, expert review, source traceability, and cost-performance monitoring in a continuous feedback loop.

03

Chapter 3 summary

RAG delivers reliable augmentation for LLMs — making outputs more trustworthy and grounded in company-specific data. But RAG is not a magic wand. Each pipeline stage carries its own failure modes, and every failure is addressable with the right approach.

The failures covered here are not exhaustive, but they represent the most common, highest-impact breakdowns encountered when building these systems in production.

The five failure stages at a glance

Stage 1 — Data Collection

Corruption, outdated records, encoding errors, duplicates, and fragmentation are upstream failures that cascade through every stage below.

Stage 2 — Data Structuring

Flawed chunking, missing metadata, and generic embeddings set a hard ceiling on retrieval quality that no downstream improvement can overcome.

Stage 3 — Retrieval

Low recall, embedding drift, and query misclassification are the most direct cause of hallucinations — and the most tractable to fix.

Stage 4 — Generation & Alignment

Gap-filling, context overload, and cross-document confusion require prompt engineering, temperature control, and entity-aware metadata as safeguards.

Stage 5 — Evaluation & Feedback

Without accuracy metrics, source traceability, and continuous feedback, errors accumulate silently until user trust collapses.

Chapter 3 key takeaway

> *RAG failures are predictable. Each stage of the pipeline has known failure modes with known mitigations. The goal is not to eliminate all risk — it is to make failures visible, measurable, and addressable before they compound.*

What comes next

The next chapter provides case studies, practical examples, and best practices for implementing the principles above in real-world, hands-on business applications.

Common patterns across stages

Looking across the five stages, several cross-cutting themes emerge in production RAG systems:

- Upstream failures are the hardest to detect and the most expensive to fix — invest in data quality and structuring before optimising retrieval or generation.
- Metadata is leverage — every attribute attached at ingestion time enables faster filtering, more precise retrieval, better citations, and easier debugging downstream.
- Evaluation cannot be bolted on at the end — accuracy metrics, source traceability, and expert review must be built into the pipeline from the start.
- Cost and accuracy are not opposites — with proper profiling, most systems can achieve high accuracy at reasonable cost through targeted optimisation rather than brute-force scaling.

04

Best Practices Explained: Hands-On Case Studies

Let us now move from theory into grounded practice. This chapter dissects the most common failure points that undermine even the most promising RAG systems — presenting five detailed case studies that diagnose critical issues, from semantic search imprecision to the retrieval of irrelevant results.

Each case study is drawn from real implementations. Each one identifies a specific, recurring failure mode, explains why it occurs at the pipeline level, and provides proven, actionable strategies to overcome it.

What this chapter covers

Case Study 1 — When pure vector search fails

Why semantic search breaks down with specific identifiers — product codes, legal citations, names — and how hybrid retrieval strategies recover precision without sacrificing recall.

Case Study 2 — Two-stage retrieval with rerankers

How adding a reranking layer after initial retrieval fixes the "garbage in, garbage out" problem — delivering dramatically more relevant context to the generation model.

Case Study 3 — When clean text is not enough

Why text-only ingestion fails for structured, visual, or domain-specific content — and what enrichment pipelines are needed to make that content retrievable.

Case Study 4 — A data-driven guide to model selection

How to balance cost, privacy, and performance when choosing embedding and generation models — with a systematic framework for making that trade-off explicit.

Case Study 5 — Evaluating search strategies in dynamic tool discovery

A systematic evaluation of eight search strategies in environments where the tool landscape changes dynamically — and what that means for retrieval design.

From theory to practice

The previous chapters built the conceptual foundation: what RAG is, how its pipeline stages work, and where each stage can break down. This chapter puts that foundation to work.

Each case study follows the same structure: a real-world failure scenario, a root-cause diagnosis mapped to the five-stage framework, and a concrete remediation approach with measurable outcomes. This format is intentional — it mirrors how production teams actually debug and improve RAG systems.

How to read this chapter

If you are building a new RAG system, read the case studies in order — they follow the pipeline from retrieval through to evaluation. If you are debugging an existing system, use the failure descriptions as a diagnostic lens: find the case study whose symptoms most closely match what you are observing, then apply the remediation pattern.

A note on case studies

The case studies in this chapter are based on composite patterns from real production implementations. Specific client details have been anonymised. The technical findings, failure modes, and remediation strategies reflect actual observed outcomes across multiple engagements.

Chapter 4 key question

> *The theory tells you what can go wrong. The case studies show you what it actually looks like — and what to do about it.*

04

Case Study 1

When Semantic Search Fails: the Precision Problem

With all the AI hype, semantic search powered by language models has become the default choice for RAG systems. But is it always the best approach? While semantic search excels at understanding meaning and different phrasing, it often stumbles on precise identifiers — confusing similar product names or mixing up date ranges. This case study explores how hybrid search combines semantic embeddings with keyword matching to handle the conceptual understanding and literal precision that real-world queries demand.

Why your RAG might confuse similar products or dates

Clients often reach out in frustration: their RAG systems produce hallucinated or inaccurate results despite clean data and properly configured vector databases. The culprit is frequently the retrieval strategy itself, which must match the type of content to what the user is actually searching for.

Pure semantic search can struggle with precise identifiers and exact matches. A query about "IBM 5150" might return results about the similar "IBM 5100," or searching for "Q3 2024 revenue" could pull up Q4 2023 or Q2 2024 results instead. This happens because semantic embeddings capture conceptual similarity rather than exact terminology.

To overcome these pitfalls, modern RAG systems increasingly adopt hybrid search approaches that combine semantic understanding with keyword matching (like BM25) to catch both the conceptual meaning and the literal precision that users demand.

Understanding dense vs. sparse search

Semantic search is "dense" because it represents documents as rich numerical vectors to capture both meaning and context. Sparse search treats documents as simple keyword lists — much like a chef who marks only the specific ingredients each recipe contains rather than describing its overall flavour profile.

BM25 (Best Match 25) is one of the most commonly used sparse search algorithms. It scores matches by counting how often search terms appear, while accounting for document length — finding "Tesla" three times in a 200-word article is far more significant than the same count in a 10,000-word history of inventors.

Note on sparse algorithms

While this case study focuses on BM25, other popular sparse retrieval algorithms exist — TF-IDF, BM11, and Dirichlet language models among them. BM25 is used here as a representative example of the category.

Key takeaways at a glance

- Pure semantic search struggles with exact identifiers such as product codes, model numbers, and date ranges.
- BM25 keyword search excels at precision but misses conceptual meaning and paraphrased queries.
- Hybrid search combines both, consistently keeping the correct answer in the top results across different query types.
- The optimal weight split varies by corpus — technical, identifier-heavy content favours more sparse weighting.

04

Three real queries — how different strategies compare

To understand where hybrid search truly shines, we examine three real queries from a vintage computer database. These examples reveal how different retrieval approaches handle the same information and why combining both methods often yields the best results.

Example 1: semantic search finds the answer when keywords fail

Query: Which computer did UK schools use?

Expected answer: BBC Microcomputer System

Dense (semantic) — top 5 results

Rank	Computer	Score
1 ★	**BBC Microcomputer System**	**0.5560**
2	Bell & Howell Computer	0.4665
3	VideoBrain Family Computer	0.4554
4	Tomy Tutor	0.4378
5	Commuter 1083	0.4351

Sparse (BM25) — top 5 results

Rank	Computer	Score
1	Sinclair ZX Spectrum	7.0326
2 ★	**BBC Microcomputer System**	**6.5383**
3	Apple I	5.8093
4	NEC APC	5.5303
5	TRS-80 Color Computer	1.8753

Analysis

This example demonstrates where semantic search excels. The query contains no direct reference to "BBC Microcomputer System," yet semantic search places it first because the system's description includes contextual information about educational use.

Notice something interesting about the BM25 results: Sinclair ZX Spectrum ranks first despite not mentioning schools at all. The keyword algorithm latched onto "UK" — a term strongly associated with the popular British computer. This shows keyword search's core limitation: it cannot distinguish between a computer that was popular in the UK versus one specifically used in UK schools.

Hybrid search result (50% semantic + 50% BM25)

1. [0.9548] BBC Microcomputer System ★

2. [0.5717] Sinclair ZX Spectrum

3. [0.3883] Apple I

4. [0.3628] NEC APC

5. [0.2322] Bell & Howell Computer

The hybrid approach surfaces the correct answer decisively — with a significant score gap between the first and second results. Neither method alone achieves this clarity.

04

Example 2: when queries explicitly mention alternatives

Query: Which computer was used by UK Schools, was it Spectrum?

Expected answer: BBC Microcomputer System

Dense (semantic) — top 5 results

Rank	Computer	Score
1	Sinclair ZX Spectrum	0.61
2 ★	**BBC Microcomputer System**	**0.51**
3	VideoBrain Family Computer	0.43
4	Commuter 1083	0.43
5	Spectravideo CompuMate	0.43

Sparse (BM25) — top 5 results

Rank	Computer	Score
1 ★	**BBC Microcomputer System**	**16.89**
2	VideoBrain Family Computer	9.41
3	Sinclair ZX Spectrum	8.66
4	NEC APC	8.31
5	Commodore Amiga 600	7.87

Analysis

This result reveals an intriguing pattern. Semantic search gets distracted by the direct mention of "Spectrum" in the query, ranking it first — the embedding model captures the semantic context of the ZX Spectrum's popularity in the UK market. Meanwhile, BM25 correctly weighs the combination of "UK," "Schools," and the comparative structure of the question.

Hybrid search result (50% semantic + 50% BM25)

1. [0.80] BBC Microcomputer System ★

2. [0.69] Sinclair ZX Spectrum

3. [0.36] VideoBrain Family Computer

4. [0.23] Commuter 1083

5. [0.18] NEC APC

The hybrid approach finds the sweet spot — correctly identifying BBC Microcomputer System as the primary answer while still acknowledging the mentioned alternative. Neither method alone handles the ambiguity introduced by naming a rival directly in the query.

Pattern observed across examples 1 and 2

When a query contains no direct keywords pointing to the correct answer, semantic search compensates. When a query names a plausible but incorrect alternative, keyword search's structural weighting compensates. Hybrid retrieval benefits from both directions simultaneously.

Semantic search excels when:

- The query uses natural language without matching the document's exact vocabulary
- Intent and context matter more than literal term matching

Keyword search excels when:

- Queries include precise identifiers, model names, or date references
- The correct document contains the exact phrasing the query uses

04

Example 3: precision matters for technical specifications

Query: Which Apple II clone shipped with 128K RAM?

Expected answer: Franklin ACE 2100

Dense (semantic) — top 5 results

Rank	Computer	Score
1	Apple IIc	0.61
2	Apple IIc Plus	0.60
3	Apple II	0.59
4	Apple IIGS	0.59
5	Franklin Ace 100	0.56

Sparse (BM25) — top 5 results

Rank	Computer	Score
1 ★	**Franklin ACE 2100**	**13.78**
2	Franklin Ace 100	13.19
3	Franklin Ace 1000	12.24
4	Apple I	10.48
5	Apple IIGS	10.43

Analysis

This represents a common real-world scenario where specific details are the distinguishing factor — similar to searching for "silver iPhone SE 3rd gen with 128GB storage" in an e-commerce context. Keyword search performs better because the exact phrases "Apple II clone" and "128K RAM" are crucial discriminators. Semantic search focuses on general similarity within the Apple product family, missing the specific technical requirements.

Hybrid search result (50% semantic + 50% BM25)

1. [0.86] Franklin Ace 100

2. [0.78] Apple IIc Plus

3. [0.77] Franklin ACE 2100 ★ (expected)

4. [0.76] Apple IIGS

5. [0.73] Apple IIc

The hybrid approach places Franklin ACE 2100 in third position. While not perfect, the hybrid system keeps the correct answer visible — whereas semantic search alone would bury it outside the top results. Interestingly, rephrasing the query to "Which Apple II fork came with 128K RAM?" returns Franklin ACE 2100 first in keyword search and third in hybrid, suggesting the retrieval pattern is not coincidence.

Three examples — the consistent pattern

Across all three queries, hybrid search never produces the worst result. It consistently keeps the expected answer within the top 3 results, combining the complementary strengths of both approaches.

- Semantic search alone: correct answer rank 1, 2, and outside top 5 across three examples.
- BM25 alone: correct answer rank 2, 1, and 1 — strong on technical queries, weaker on conceptual ones.
- Hybrid: correct answer rank 1, 1, and 3 — consistent top-tier performance across all query types.

04

The full evaluation: 169 computers, 507 queries

The three examples above illustrate individual query behaviour. To measure overall retrieval quality across diverse query types, we ran a comprehensive evaluation using a dataset of 169 vintage computer descriptions and 507 LLM-generated test queries.

How the evaluation works

- Each of the 169 computer documents is converted into embeddings using the jina-embeddings-v3 model via JinaAI's API and indexed for retrieval.
- For each of the 507 test queries, embeddings are generated using the same method.
- HNSW indexing with cosine similarity handles semantic retrieval; BM25 handles keyword matching.
- Results from both methods are normalised and combined using configurable weights.
- We examine the top 5 ranked results and compare them against the expected document name.
- We repeat the process for different dense/sparse weight ratios to find the optimal balance.

Evaluation results: dense vs. sparse weight configurations

Best values shown in red. Acc = Accuracy, MRR = Mean Reciprocal Rank.

Dense / Sparse	Acc@1	Acc@3	Acc@5	MRR@1	MRR@3	MRR@5
100% / 0%	66.47%	83.04%	86.19%	66.47%	74.10%	74.85%
70% / 30%	78.30%	89.35%	93.89%	78.30%	83.23%	84.28%
50% / 50%	85.40%	96.84%	**98.82%**	85.40%	90.66%	91.14%
30% / 70%	**89.15%**	**97.24%**	98.03%	**89.15%**	**92.93%**	**93.13%**
0% / 100%	87.57%	96.25%	97.63%	87.57%	91.55%	91.87%

Understanding the metrics

Accuracy@k measures the percentage of queries where the correct answer appears anywhere in the top k results. Accuracy@3 means "how often is the right answer in the top 3?"

MRR@k (Mean Reciprocal Rank) measures how high the correct answer ranks, giving more credit for a rank-1 result than rank-3. Calculated as the average of 1/rank across queries.

Key observations

Sparse (keyword) search significantly outperforms pure dense (semantic) search for this dataset — 87.57% accuracy versus 66.47% for pure embeddings. This gap reflects the corpus's heavy concentration of technical identifiers, model numbers, and exact specifications.

The optimal configuration — 70% sparse / 30% dense — reaches 89.15% top-1 accuracy. A balanced 50/50 split delivers strong performance at 85.40%, demonstrating robustness across different query types.

Worth noting

This system could be further improved with metadata filtering. For queries like "computers from 1981 with 128K RAM," pre-filtering by production year and memory specifications before running hybrid search dramatically reduces the search space and improves both speed and accuracy.

04

Weighing the hybrid approach: what your RAG gains and what it costs

Hybrid search is not a silver bullet — it is a deliberate design choice that trades operational simplicity for more robust relevance. The core advantage is complementary coverage: sparse search catches exact identifiers reliably, while dense search captures intent and handles paraphrasing. Each method compensates for the other's blind spots.

What you gain	What it costs
Coverage Catches both exact terms (IDs, dates, SKUs) and fuzzy intent (synonyms, paraphrases)	**Dual systems** Two retrieval pipelines to run, monitor, and keep synchronised instead of one
Explainability Shows which words and fields matched — easier debugging and user trust	**Synchronisation overhead** Keyword analyzers and embeddings must stay aligned as content changes
Precision Exact queries like "IBM 5150" or "2024-Q3" hit reliably without semantic drift	**Query complexity** Result merging and score fusion add processing steps to every query
Rich filtering Facets and filters (date, category, entity) work naturally via keyword fields	**Schema maintenance** Fields such as titles, entities, and tags must be designed and kept up to date

The operational trade-off

Running hybrid search means maintaining two retrieval pathways in parallel. Both must stay synchronised: tokenisation rules, text analyzers, keyword field structure, and the fusion weights that balance the two scoring systems. This adds engineering overhead that compounds as the corpus scales.

For a static corpus with stable content, this is a one-time configuration cost. For a high-volume, rapidly updating knowledge base, it is an ongoing operational commitment that requires dedicated bandwidth.

When the trade-off is worth it

The gains justify the costs when your users ask mixed query types — precise technical lookups alongside natural-language exploratory questions. A corpus heavy with model numbers, product codes, legal citations, or date ranges will see the largest accuracy improvements.

When your corpus is predominantly narrative text and queries are consistently conceptual, pure dense search may be sufficient — and operationally simpler to sustain.

04

Finding the right fit: when hybrid search makes sense

The decision to adopt hybrid search depends less on theoretical benefits and more on whether it solves problems you are actually experiencing. Evaluate these four factors before committing:

Query diversity

Do your users ask both precise questions ("find invoice INV-2024-0847") and exploratory ones ("how do I resolve payment disputes")? Mixed query patterns favour hybrid. Homogeneous queries may work perfectly with a single retrieval method.

Failure pattern analysis

Instrument your system and observe where it breaks. Wrong date quarters, confused product models, or incorrect model numbers signal a need for keyword precision. Users who constantly rephrase the same query suggest the semantic layer needs reinforcing. Let observed failures drive the decision — not assumptions.

Content characteristics

Documents heavy with technical identifiers, model numbers, legal citations, or precise dates benefit more from hybrid systems than narrative text. Assess what percentage of your corpus contains exact-match elements that semantic embeddings are likely to blur.

Scale and operational burden

A static archive of 10,000 documents is a one-time setup cost. Millions of rapidly updating documents require continuous reprocessing of both keyword indexes and embeddings. Consider whether your team has the bandwidth to keep dual systems synchronised as content changes.

How to start

If you are uncertain whether hybrid retrieval will help, start simple. Deploy pure semantic search first, log every query that returns wrong or missing results, and build a failure catalogue over real usage. That catalogue will tell you whether your failures are semantic (paraphrase misses) or lexical (identifier confusion).

Once you identify a clear pattern, test hybrid search on a representative sample of those failing queries. Measure the improvement against your specific pain points, and adopt hybrid only when the accuracy gains justify the added complexity your team can sustainably maintain.

Case Study 1 key finding

> *For 169 technical documents and 507 diverse queries, the optimal dense/sparse split was 30/70 — reaching 89.15% top-1 accuracy versus 66.47% for pure semantic search alone. A balanced 50/50 split delivered 85.40%, demonstrating robust performance across mixed query types.*

Applicable to your context when:

- Your corpus contains technical specifications, product codes, or date-sensitive records.
- Your users ask a mix of exploratory and precision-lookup queries.
- You have observed retrieval failures that semantic search alone cannot resolve.
- Your team can commit to maintaining synchronised dual indexes as content evolves.

Case Study 2

The Retrieval Quality Gap: Fixing Irrelevant Results

Why this case

Standard RAG systems face a common problem: they can quickly find documents, but often surface irrelevant files that lead to low-quality answers. This case tackles that "garbage in, garbage out" problem at the retrieval stage — showing how adding a reranking step resolves it, using real-world regulatory documents to demonstrate clear, measurable improvement.

Our testing ground: the EurLex project

We chose the **EurLex** dataset — a collection of European Union regulations spanning more than 700 different documents. This dataset presents many of the same challenges encountered in real-world enterprise RAG:

- **Domain complexity** — Legal language is precise, nuanced, and contains subtle distinctions critical for accurate retrieval.
- **Document length variability** — Relevant documents range from brief amendments to comprehensive regulatory frameworks.
- **Semantic similarity challenges** — Multiple documents may discuss similar topics but with vastly different legal implications.
- **High accuracy requirements** — Retrieving the wrong document or missing a regulation can have serious consequences.

Working with EurLex lets us demonstrate technique performance under demanding conditions that mirror the sensitive, domain-specific datasets our clients work with regularly.

Key takeaways

- Why initial retrievers return too many irrelevant documents.
- How reranking (cross-encoders) provides precision.
- Speed vs. accuracy trade-offs.
- Practical performance metrics from real evaluation data.

What are Cross-Encoders (aka Rerankers)?

A common pain point in RAG is irrelevant documents appearing in the retrieval step, which leads to poor-quality answers. One of the most effective fixes is adding a second step: **reranking**.

A reranker — also known as a cross-encoder — takes a user's query and a single retrieved document, and outputs a relevance score. Its only job is to determine how relevant that specific document is to that specific query.

Two-stage retrieval process

Stage 1 — Broad Retrieval. A fast retriever scans the entire database and pulls a large set of potentially relevant documents (e.g. the top 50). This stage prioritises speed over accuracy.

Stage 2 — Precise Reranking. The cross-encoder examines each of those 50 documents alongside the user's query, assigning a precise relevance score to each pair. Documents are re-sorted based on this score, pushing the most relevant to the top. This stage prioritises accuracy.

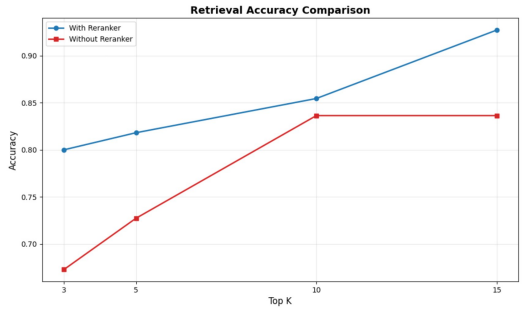
We use two stages because rerankers are computationally intensive and slow, while initial retrievers are fast and efficient. Using a reranker on a full database would be far too slow and costly — so we use a fast retriever to create a smaller, more manageable list for the reranker to work with.

Putting it to the Test: Reranking in action

To measure the real-world impact of a reranker on our EurLex project, we ran a direct comparison. We built an evaluation dataset of 55 questions for which we know the exact document containing the correct answer.

The test is simple: for each question, does our retrieval system fetch the correct document? We measure this as **retrieval accuracy** — the percentage of questions for which the correct document was successfully retrieved.

In the experiment we vary the **Top K**: the number of documents initially retrieved from the vector store. We then plot the accuracy with and without our cross-encoder reranker.



As the chart shows, the reranker provides a significant and consistent boost in accuracy. When retrieving a smaller number of documents (Top K = 3), the reranker improves accuracy by **over 10 percentage points** — from 67% to 80%.

This is crucial: sending fewer, higher-quality documents to the LLM is more efficient and typically yields better final answers.

Retriever vs. Reranker: the key difference

Retrievers (Bi-encoders) — Fast but less accurate. They create generic document embeddings without comparing them to the query. The query and document are encoded independently and similarity is computed afterwards.

Rerankers (Cross-encoders) — Slow but highly accurate. They directly compare the query and document text together, allowing the model to capture rich contextual relevance signals that bi-encoders miss.

Why not just use a reranker on the full database?

Rerankers require a forward pass for every query-document pair. For a database of 100,000 documents, that means 100,000 model inferences per query — which is orders of magnitude too slow for production use.

The two-stage approach solves this elegantly: a fast retriever reduces the candidate set to a manageable size (e.g. top 50), and the reranker then applies its heavier computation to only those 50 pairs.

Key finding

Adding a reranker is not just a theoretical improvement — it is a practical, high-impact technique for enhancing the quality and reliability of any serious RAG system, raising retrieval accuracy by over 10% at low Top K values.

In summary: the power of advanced retrieval techniques

Incorporating a reranking step in your RAG implementation can significantly enhance the effectiveness of your information retrieval process. This advanced technique, combined with proper embedding models and vector stores, forms the foundation for building tailored AI assistants capable of leveraging domain-specific knowledge with high accuracy and reliability.

The 4 key benefits of rerankers in RAG

1. Improved accuracy

By using a two-stage retrieval process with reranking, we can raise the relevance of retrieved documents by over 10 percentage points versus a retriever-only pipeline.

2. Reduced hallucinations

With more accurate document retrieval, we provide LLMs with better context, reducing the likelihood of generating incorrect or irrelevant information.

3. Efficient processing

The combination of swift initial retrieval and precise reranking allows for efficient handling of large document collections without sacrificing answer quality.

4. Versatility

This approach is applicable across a wide variety of RAG use cases — from question-answering systems to chatbots and agentic AI workflows — particularly wherever domain-specific accuracy matters.

How this fits the broader RAG picture

The EurLex case demonstrates that retrieval quality is not just about the embedding model or the index structure — it is about the full pipeline. A fast retriever paired with a precise reranker outperforms any single-stage approach at realistic Top K values.

As you continue reading this chapter, we will explore additional advanced RAG techniques: hybrid search methods, chunking strategies, and the use of dense vectors for improved similarity scoring. We will also discuss how RAG integrates with other machine learning techniques to create even more efficient and flexible systems.

When to adopt this pattern

- Your corpus is large and diverse (100+ documents with variable lengths).
- Answer quality degrades when retrieving a small Top K.
- Your domain requires high precision (legal, medical, technical manuals).
- You have observed that the LLM is frequently misled by irrelevant retrieved context.

Case Study 2 key finding

On a 700+ document EU legal dataset with 55 evaluation questions, adding a cross-encoder reranker improved retrieval accuracy by more than 10 percentage points at Top K = 3 — without any changes to the underlying embedding model or vector store.

Case Study 3

The Data Preparation Trap: When Clean Text Is Not Enough

Why this case

Garbage in, garbage out. Yet there is a gap between having clean text and having retrieval-ready content. While raw text in a vector database may suffice for basic use cases, this case explores how LLMs can restructure that text into retrieval-ready formats for more refined search.

Is perfectly parsed text good enough for RAG?

Real-world data quality extends beyond file formats or OCR accuracy. Even clean text, perfectly parsed from a PDF, may not be in the best shape for an effective RAG system. The key lies in how you **structure** that extracted data — making it comprehensible and easy to leverage in different retrieval contexts.

Data processing pipeline for RAG

1. Document ingestion

Collect and prepare source files — PDFs, scans, documents.

2. Raw text extraction

Extract content using OCR or PDF parsers to get plain, unstructured content.

3. Structured extraction

Use LLMs to extract metadata, keywords, summaries, and entities from raw text.

4. Chunking

Divide structured content into logical segments optimised for retrieval.

5. Vector database storage

Embed chunks with their metadata and index for semantic search.

Key takeaways

- The gap between parsed text and retrieval-ready content.
- When to invest in structured extraction.
- Batch processing for cost efficiency.
- Using smaller models for extraction tasks.

What semantic search does well — and where it falls short

In the simplest vector search implementation, you split text into chunks, create embeddings, and load them into a vector database. Queries are then semantically compared to those chunks.

In many cases this works well — but stumbling blocks hide in the details. A RAG system handles "revolutionary IBM computer from the 80s" beautifully, but may struggle to distinguish "IBM 5100" from "IBM 5150." Classic keyword search handles that easily.

Similar problems arise with "IBM computer released in 1981" when the database contains other models from around that time — similar dates become confusing when relying purely on semantic search.

What does each retrieval technique require?

Retrieval Technique	Structured Elements Required
Metadata Filtering	dates, categories, document types, tags
Hybrid Search	keywords, exact identifiers, product codes
Query Understanding	keywords, intent markers, domain terms
Contextual Retrieval	doc title, section summaries, key concepts
Graph RAG	entities, relationships, attributes

Making your retrieval system more robust

To address the limitations of pure semantic search, engineers employ various techniques. What they all have in common: they benefit from — or outright require — structured information extracted from your raw text.

- **Metadata extraction and filtering** — narrow down chunks by specific attributes (year, document type) before semantic search even begins.
- **Hybrid search** — combine semantic vectors with traditional keyword matching (BM25, exact matches) to catch both conceptual and literal queries.
- **Query understanding** — analyse the user's question to route it appropriately or reformulate it for better retrieval.
- **Contextual Retrieval** — Anthropic's approach, which enriches each chunk with surrounding context before embedding.
- **Summary-based indexing** — embed condensed document versions rather than raw text, filtering out noise while preserving key information.
- **Graph RAG** — structure knowledge as interconnected nodes and relationships (patient → diagnosed_with → condition → treated_by → medication), then use semantic search to navigate the graph.

Why structured extraction is the critical foundation

Think about what each technique actually requires:

- Metadata filtering needs dates and categories.
- Hybrid search needs keywords and exact identifiers.
- Contextual Retrieval needs document titles and summaries.
- Graph RAG needs entity relationships.

None of these exist in your raw extracted text — you have to create them through structured extraction. This is not just about making your data prettier; it is about creating the foundation that allows advanced retrieval techniques to work at all.

From raw text to retrieval-ready content

Imagine building a search engine for thousands of PDF articles — walls of text with no clear headers, tables scattered throughout. After PDF extraction you get unstructured prose. The next page shows how structured extraction transforms that into a queryable, filterable JSON record.

The core insight

> *Structured extraction is the bridge between "I have clean text" and "I have a retrieval system that actually works well."*

Structured extraction in action: the IBM PC 5150 example

Each article is a wall of text without clear headers, with tables scattered throughout. After PDF extraction you end up with unstructured prose like this:

> *The IBM Personal Computer, model 5150, revolutionized the computing industry when it launched in the early 1980s. This machine featured an Intel 8088 processor running at 4.77 MHz, starting with 16KB of RAM... Storage options included cassette tape or 5.25-inch floppy drives with 160KB capacity. The base configuration started at $1,565... IBM-PC-5150-SYS-001-1981-US*

After structured extraction, the same content becomes:

```
{ "name": "IBM PC 5150",
"category": "Personal Computer",
"keywords": ["IBM","PC","5150",
"Intel 8088","open architecture","1980s"],
"summary": "IBM PC model 5150, launched 1981.
Intel 8088 CPU, 16KB–256KB RAM, 5 expansion
slots, CGA graphics, floppy drives.",
"year_start": 1981, "year_end": 1987,
"identification_number":
"IBM-PC-5150-SYS-001-1981-US",
"tables": [{"title":"IBM PC 5150 Timeline",
"data":[{"Event":"Initial Release",
"Year":"1981"},
{"Event":"Peak Sales","Year":"1982-1984"},
{"Event":"Discontinuation","Year":"1987"}]
}]
}
```

What the transformation unlocks

What was once buried prose is now a queryable dataset:

- **Metadata filtering by year** — filter to documents where year_start ≤ 1981 ≤ year_end instantly, without scanning all embeddings.
- **Graph RAG** — the IBM 5150 entity is now linkable to its specifications, timeline events, and competitor entities (Apple II).
- **Contextual Retrieval** — the pre-generated summary can be embedded alongside each chunk, dramatically improving retrieval of related passages.
- **Hybrid search** — the keywords list feeds directly into BM25 indexing, catching exact model number queries that pure semantic search misses.
- **Searchable identifiers** — "IBM-PC-5150-SYS-001-1981-US" becomes a filterable field rather than an orphaned string buried in text.

But this power comes with a cost… literally.

When working with hundreds of thousands of documents, processing each one through an LLM adds up quickly. The next page explores how to manage that cost effectively.

Tradeoffs involved in structured extraction

When working with hundreds of thousands of documents, the cost of processing each one through an LLM can add up quickly. You need to carefully weigh whether improved retrieval accuracy justifies the extraction costs.

The good news: you can extract everything in a **single pass**. Rather than making multiple API calls for summaries, then keywords, then metadata separately, modern LLMs can pull all structured elements at once — titles, dates, categories, entity relationships, and summaries in one operation. This dramatically reduces both cost and processing time.

Batch processing: a set-it-and-forget-it solution

Major providers — OpenAI, Anthropic, Google Gemini, and Mistral — offer cost discounts of up to **50%** for batch API calls.

Instead of managing thousands of individual API calls with retry logic and rate limit handling, you package all requests into a single JSONL file, upload it, and let the provider handle everything. Most batch jobs complete within 24 hours — many finish much faster.

For large document collections this means processing your entire archive overnight — upload in the evening, retrieve structured results in the morning. You also get significantly higher throughput limits, enabling hundreds of thousands of requests in a single batch.

Why batch processing matters at scale

Real-time API: 1 million documents at $0.01/doc = $10,000. Batch at 50% discount = $5,000. For a static archive this is a one-time cost. For continuously updated datasets, the savings compound over time.

Beyond cost, batch eliminates operational complexity: no retry logic, no queuing systems, no rate-limit management. Upload your JSONL, walk away, collect results.

Single-pass extraction prompt structure

A well-designed prompt extracts all required fields in one call:

```
Extract from this text and return JSON:
{
"name": str,
"category": str,
"keywords": [str, ...],
"summary": str,
"year_start": int, "year_end": int,
"entities": [{name, type}],
"identification_number": str
}
```

Key principle

> *Structured extraction is not about making your data prettier. It is about creating the foundation that allows advanced retrieval techniques — metadata filtering, hybrid search, Graph RAG — to function at all.*

Open-weight models for structured extraction

You do not need expensive frontier models to test whether structured extraction improves your RAG system. We tested five popular open-weight models on the same IBM PC 5150 extraction task to compare their output quality and accuracy.

Model ID	Model Name
openai/gpt-oss-20b	**GPT-OSS 20B**
openai/gpt-oss-120b	**GPT-OSS 120B**
moonshotai/kimi-k2-instruct	**Kimi K2 Instruct**
meta-llama/llama-4-maverick	**Llama 4 Maverick (17B)**
meta-llama/llama-4-scout	**Llama 4 Scout (17B)**

One standout: **gpt-oss-20b** delivers impressive extraction quality and can run locally on a good laptop — giving you complete data control at zero API cost.

How to test without spending money

All five models are available through Groq's free tier — no credit card required. A simple Python script is enough to run the IBM PC extraction task on any of them and compare JSON outputs side by side before committing to any production infrastructure.

What to look for when evaluating model outputs

- **Field completeness** — does the model populate every field in your schema, or does it skip optional ones?
- **Entity accuracy** — are extracted entities (names, dates, product codes) correct and unambiguous?
- **Summary faithfulness** — is the generated summary faithful to the source, without hallucinating details?
- **Format consistency** — does the model reliably return valid JSON, or does it sometimes wrap output in prose?

Model size vs. quality tradeoff

For simple schemas (name, date, category, keywords), a 17B parameter model typically performs comparably to a 120B model at a fraction of the cost. The gap widens for complex schemas requiring multi-hop reasoning or long-range entity resolution.

Next: making the right choice

With the technical foundations clear, the final question is how to decide which approach — and how much structure — your specific RAG system actually needs.

Making the right choice for your RAG system

Structured extraction is not all-or-nothing — the question is how much structure you actually need. Start by examining where your retrieval system falls short: imprecise date searches, missing product codes, inability to filter by type. These pain points reveal exactly which structured elements will deliver value.

Test on a small scale first. Extract structure from a representative sample — 100 or 1,000 documents — and measure the impact. Does metadata filtering reduce irrelevant results? Does hybrid search catch model numbers that semantic search misses? Clear improvements validate the approach before committing to large-scale processing.

Key factors to evaluate

Volume

1,000 documents is a negligible cost. 10 million requires careful cost justification. Know your scale before committing.

Query patterns

Broad semantic searches need less structure. Precise filtering and exact matching need more. Audit your actual query logs.

Update frequency

Static archives are one-time costs. Continuously updated datasets require ongoing processing — factor this into your TCO.

Model selection

Frontier models are not always necessary. Smaller open-weight models handle simple extraction at lower cost.

Batch processing

For large archives, batch APIs offer 50% savings and eliminate operational complexity.

The pragmatic path forward

The goal is pragmatic optimisation: extract enough structure to enable the retrieval techniques that solve your actual problems, without over-engineering for theoretical gains.

Start with your pain points, validate on a subset, then scale with confidence once you know which structured elements actually improve your users' search experience.

Structured extraction decision guide

- **You need it** if users search by date, product code, category, or named entity and semantic search keeps returning wrong results.
- **Start small** — 100 to 1,000 documents. Measure precision improvement before committing to full-corpus extraction.
- **Use batch APIs** for archives over 10,000 documents — 50% cost saving and simplified operations justify the 24-hour turnaround.
- **Test open-weight models** first. gpt-oss-20b handles most tasks well and can run locally at zero API cost.
- **Extract in a single pass** — name, category, keywords, summary, entities, and dates all in one LLM call.

Case Study 3 key finding

Clean text alone is not enough for advanced RAG. Structured extraction — transforming raw content into queryable JSON records with metadata, keywords, summaries, and entities — is the foundation that makes metadata filtering, hybrid search, and Graph RAG technically possible. The cost is manageable through single-pass extraction and batch processing.

Case Study 4

The Model Selection Dilemma: Frontier vs. Lightweight LLMs

Why this case

Structured extraction is one of the most effective ways to enhance RAG — but it carries a pre-processing cost: running every document through an LLM before indexing. Do you really need expensive frontier models to get good results?

This case presents evaluation results for smaller variants of DeepSeek, Gemma, Llama, Mistral, Phi, and Qwen — demonstrating a systematic evaluation methodology using both deterministic tests and LLM-as-a-judge approaches.

Why use smaller open-weight models?

While individual consumers rely on ChatGPT web sessions, businesses often face stricter requirements. Public sector and NGO organisations especially need to run models locally — as a necessity rather than a preference. Three factors drive this need:

Data privacy and security

Sensitive documents and extracted information stay on-premise. No data leaves your infrastructure through third-party APIs, ensuring compliance with data protection regulations and internal security policies.

Cost efficiency

API costs disappear entirely. Operating expenses become predictable and controllable — especially important when processing large document volumes at scale, where per-token pricing quickly adds up.

Independence and control

No vendor lock-in. Full control over model deployment and customisation. Service availability does not depend on external providers or their pricing decisions.

Key takeaways

- When smaller models (3B–8B) match frontier performance.
- Privacy and cost considerations for on-premise deployment.
- Systematic evaluation methodology you can replicate.
- Fine-tuning opportunities for specialised extraction tasks.

The critical question

These advantages make smaller models compelling for many organisations. But must you sacrifice accuracy to gain them?

Our evaluation suggests you do not — but there is no shortcut to finding the right model. You need systematic evaluation with your specific data and tasks.

What we evaluated

- 13 models from DeepSeek, Gemma, Llama, Mistral, Phi, and Qwen families.
- 168 historical computing documents.
- GPT-4.1-generated baseline as the reference standard.
- All models run via Ollama with quantization.
- PydanticAI Evals framework for reproducible benchmarking.
- Combined deterministic metrics + LLM-as-a-judge scoring.

The question of whether you need a frontier model is one we have tested extensively. Bielik is a general-purpose multilingual large language model with a strong focus on Polish — at 11 billion parameters, it sits well below the frontier tier. Yet in our evaluations, it consistently outperformed models with two to six times more parameters across tasks that matter for RAG workflows, including question answering over retrieved context and extractive comprehension.

We took this further by compressing the 11B model into a 7B variant through structured pruning and knowledge distillation. That compressed model retained strong performance — evidence that even within the lightweight category, there is room to optimise without meaningful loss of accuracy.

Our experience confirms what this case study demonstrates: the gap between frontier and smaller models is narrower than most organisations assume, and for structured, well-defined tasks it can disappear entirely. The key is not to guess — it is to evaluate systematically against your own data. There is no universal ranking of models; there is only the ranking that matters for your specific workload.

— Krzysztof Wróbel, Co-author of the Bielik LLM

Our evaluation approach: testing small language models for document extraction

Metric	Schema Attribute	Eval Method	What We Are Testing
Summary length & ending	summary	Deterministic	Checks summary is 100–300 characters and ends with proper punctuation. Scores 1.0 if both met, else 0.0.
Name similarity	name	Deterministic (RapidFuzz)	Normalises both names and compares using fuzz.ratio(). Returns 0.0–1.0, tolerating minor formatting differences.
Manufacturer similarity	manufacturer	Deterministic (RapidFuzz)	Same fuzz.ratio() approach as name. Captures cases where models expand or rephrase manufacturer names.
Release year accuracy	release_year	Deterministic	Exact integer match between expected and extracted year. 1.0 for match, 0.0 otherwise.
Keyword coverage	keywords vs summary	Deterministic (fuzzy set)	Fuzzy matching and set coverage measuring how many key concepts appear in model output, even when phrased differently.
Coverage	full extraction	Deterministic	Checks whether model produced any meaningful extraction (non-empty fields). Returns 1.0 if at least one field populated.
Summary quality	summary	LLM judge (kimi-k2)	Evaluates on three dimensions: fidelity to facts (0–4), comparison to baseline (0–3), narrative richness (0–3). Normalised to 0.0–1.0.

Key design principles

- **Character-based fuzzy matching** (fuzz.ratio) for name and manufacturer fields handles punctuation and formatting variations gracefully.
- **Keyword coverage** uses substring matching for efficiency and interpretability.
- **Release year matching** is exact (no tolerance) to ensure precision in temporal data.
- **Coverage evaluator** checks for name presence as the minimum viable extraction signal.
- **LLM judge** focuses on summary quality — the dimension where deterministic metrics are insufficient.
- **All evaluators** return 0.0–1.0 scores for consistent aggregation and comparison.

Evaluation and metrics design challenges

Our approach prioritises deterministic evaluation. We keep LLM-as-a-judge limited to cases where deterministic metrics are insufficient — to reduce ambiguity and additional evaluation costs. However, even comparing product names can be tricky depending on your dataset.

Manufacturer mismatches: when "wrong" is actually right

Consider these evaluation results, where Gemma3-12b scored poorly on manufacturer extraction:

Expected (GPT-4.1)	Actual (Gemma3-12b)	Score
Apple (manufactured by Sharp)	Sharp	0.29
Poqet Computer Corporation	Poqet	0.32
NCR Corporation	NCR	0.33
Altos Computer Systems	Altos	0.37
Tano Corporation	Tano	0.40

Though they achieve low scores, the Gemma3 results are not actually bad. After investigation, we found "Poqet Computer Corporation" was not present in the source data. GPT-4.1 expanded "Poqet" to the full corporate name and we used those results as our baseline.

This illustrates a common challenge: you rarely have ideal reference data and often must create it synthetically. The apparent discrepancies reflect the baseline model going beyond the source text — not the evaluated model failing.

The baseline problem

When your reference data is model-generated (as ours was from GPT-4.1), you inherit the baseline model's biases and expansion patterns. This is a necessary tradeoff when no human-verified ground truth exists at scale.

The same tradeoffs between strict metrics and nuanced judgment become even more pronounced when evaluating summarisation quality — where there is no single correct answer.

Our evaluation setup

- **168 documents** — historical computing articles.
- **13 models** tested across six model families.
- **GPT-4.1 baseline** — consistent reference, not ground truth.
- **Ollama + quantization** — all models run via local inference.
- **PydanticAI Evals** — reproducible framework for systematic comparison.
- **Composite scoring** — z-score normalisation with: Summary Quality 30%, Coverage 10%, 12% each for other metrics.

A note on composite scoring

The composite score weights summary quality heavily at 30%. Z-score normalisation can amplify small metric differences, and the judge model significantly shapes quality scores. The rank-based view on the following page offers an alternative by treating all metrics equally.

Evaluation results across 13 models

Top-tier cluster (scores 0.97–1.00)

Four models cluster at the top separated by only 3 percentage points: gpt-oss-20b (1.00), llama3.1-8b (0.98), llama3.2-3b (0.97), and qwen3-8b (0.97). A significant gap follows — Phi-4 drops to 0.91.

The most striking finding: **llama3.2-3b at just 3B parameters nearly matches the 8B llama3.1** (0.97 vs 0.98 composite) and achieved a perfect 100% success rate (168/168 vs 167/168).

Summary quality: the universal challenge

Most models excel at field extraction — average coverage across all 13 models is 95%. But summary quality proved universally challenging: no model exceeded 46% on judge scores (average: 32%). This dramatic gap confirms that summarisation is the hardest aspect of structured extraction.

qwen3-4b paradox: achieved perfect coverage and topped several metrics, yet ranked 9th overall due to frequently cutting summaries mid-sentence — triggering zero judge scores.

Rank-based view: treating all metrics equally

Rank	Model	AvgRank	Name	Manuf.	Year	Keywords	Coverage	Sum.Length	Sum.Quality
1	llama3.2-3b	**3.71**	**1**	7	**2**	7	**1**	3	5
2	phi4-14b	**3.86**	2	**2**	**1**	3	5	7	7
3	llama3.1-8b	**4.29**	3	5	3	6	4	5	4
4	gpt-oss-20b	4.57	8	4	5	5	6	**2**	**2**
5	gemma3-12b	4.71	6	3	4	9	7	**1**	3
6	qwen3-8b	5.57	5	8	8	**2**	9	6	**1**
7	qwen3-4b	5.71	4	**1**	6	**1**	**2**	13	13
8	deepseek-r1-7b	6.86	7	6	7	8	3	8	9
9	gemma3-1b	8.86	13	10	11	4	10	4	10
10	gemma3-4b	9.14	9	9	9	11	11	9	6
11	qwen3-1.7b	11.14	11	11	13	12	13	10	8
12	deepseek-r1-1.5b	11.29	12	13	10	13	8	11	12
13	mistral-7b	11.29	10	12	12	10	12	12	11

Size efficiency findings

1–3B range

llama3.2-3b (3B) delivers elite performance rivalling much larger models. Tiny models like gemma3-1b (1B) and qwen3-1.7b struggle with consistency despite decent coverage.

4–8B tier

Shows the strongest overall performance: llama3.1-8b and qwen3-8b both achieve composite scores ≥0.97, balancing coverage reliability with respectable summary quality.

12–20B parameters

More nuanced. Phi-4 (14B) ranks 5th composite — excellent at structured extraction but weaker on summaries. gpt-oss-20b and gemma3-12b show that larger models excel at narrative generation without improving field extraction.

Notable findings

Mistral-7b surprise: last place (0.00 composite) despite its 7B parameter count. These results reflect only this specific task and prompting approach. **We make no claims about overall model quality.**

Phi-4 specialisation: excels at structured fields (year rank 1, name rank 2, coverage rank 5) while struggling at narrative generation (summary quality rank 7).

> ***The core insight:*** *well-tuned lightweight LLMs consistently outperform poorly-optimised larger ones. The 3–8B range is the optimal choice for production extraction — balancing quality, efficiency, and resource requirements.*

Fact-focused vs. contextual summaries

Aspect	Fact-focused (smaller models)	Contextual (larger models)
Primary focus	Technical specifications and features	Technical details with historical significance
Language	Neutral verbs: "had", "included", "featured"	Narrative language: "pioneering", "revolutionized", "first to"
Content	What it was and what it had	What it was plus why it mattered
Context	Minimal or absent	Market position, company history, competition
Impact	Rarely discussed	Often discussed: success, failure, influence, legacy
Tone	Dry technical catalog	Informative narrative grounded in facts

How different models handle summarisation: three examples

Computer	Llama3.2-3B (local, 3B params)	GPT-4.1 (frontier baseline)
IMSAI 8080	The IMSAI 8080 is an early S-100 based system designed to be highly expandable. It uses the Intel 8080A CPU at 2.0 MHz and has a maximum RAM of 64K.	*The IMSAI 8080, released in 1975 by IMSAI Manufacturing Corporation, is an Intel 8080A-based computer famous as the first successful Altair 8800 clone and for its expandable S-100 bus architecture.*
Atari 800	The Atari 800 is a home computer designed by Jay Miner and released in November 1979. It features a MOS 6502 CPU, 8K base RAM, and 48K max RAM, as well as a 24x40 text display and 320x192 monochrome display.	*The Atari 800 is a modular 8-bit home computer introduced by Atari in late 1979, notable for its expandability, advanced graphics, and gaming capabilities.*
Commodore 64	The Commodore 64 was a groundbreaking personal computer released in late 1982 by Commodore Business Machines, Inc. It built upon the success of the VIC-20 and went on to become one of the most popular computers of all time.	*The Commodore 64 is an 8-bit home computer released in late 1982 by Commodore, selling about 17 million units — more than any other single model.*

Key findings

No universal solution exists.

One model may excel at a task while another, similar model fails. Everything depends on your specific use case and requirements.

Responsiveness to prompts improves with model size.

Smaller models respond poorly to prompt instructions and few-shot learning. Medium-sized models (7B) show some improvement; larger models handle these techniques more reliably.

Fine-tuning extends smaller model capabilities.

For simple tasks with limited token usage (processing one-page documents), you can fine-tune smaller models on consumer hardware (24GB RAM) using quantised models via Ollama or MLX. These often outperform larger general-purpose models.

Summarisation is harder than extraction at every size.

Smaller models (2–4B and 7B) handle basic data extraction adequately but struggle with quality summaries. Models around 12B parameters begin to produce consistent summaries that properly account for context. The notable exception: Llama3.2-3b.

Note: contextual summaries are not always preferable. If you want maximally objective, fact-based summaries, specify this explicitly in your prompt.

Recommendations: run your own evaluations

Our evaluation covers a relatively small dataset and your results will vary with different documents and extraction tasks. The goal is to demonstrate a systematic approach you can apply to your own use case.

- **Establish systematic evaluation.** Without proper test cases and reproducible benchmarks, you cannot reliably compare models. Anecdotal testing will mislead you.
- **Test with your actual data.** These results reflect our specific documents and schemas. Build a representative test set from your real-world documents.
- **Prompt engineering matters as much as model selection.** How you structure your extraction prompt significantly affects accuracy across all model sizes.
- **Consider fine-tuning for specialised needs.** If your task has specific patterns or domain language, a fine-tuned smaller model may outperform a larger general-purpose one.

Case Study 4 key finding

You do not need frontier models for structured extraction. With systematic evaluation, the 3–8B parameter range delivers production-quality results while maintaining data privacy, predictable costs, and full deployment control. Llama3.2-3b at 3B parameters nearly matches 8B models — challenging every assumption about parameter count requirements.

Case Study 5

Agentic RAG: Evaluating Tool Selection Strategies

Why this case

In a traditional RAG pipeline, the retrieval strategy is fixed at design time. Agentic RAG changes this fundamentally: the LLM itself decides which retrieval tool to invoke for each query, dynamically selecting the most appropriate method based on the nature of the question.

This case evaluates eight distinct retrieval strategies — from simple regex matching to hybrid vector search — across two interaction modes: **One-Shot** (single message) and **Conversation** (multi-turn dialogue). It asks a deceptively simple question: does giving an agent more tools make it better?

The eight strategies tested

- **all_tools** — agent has access to all retrieval tools simultaneously.
- **bm25** — classic keyword search (Best Match 25 ranking).
- **cross_encoder** — reranking model for precision retrieval.
- **hybrid_milvus** — hybrid search via Milvus vector database.
- **hybrid_pinecone** — hybrid search via Pinecone vector database.
- **milvus** — pure dense vector search via Milvus.
- **pinecone** — pure dense vector search via Pinecone.
- **regex** — pattern-based exact matching (lightweight baseline).

Two interaction modes

One-Shot: the agent receives the query and must answer in a single message exchange. Fast and token-efficient, but no opportunity to clarify or retry.

Key takeaways

- More tools does not always mean better accuracy — the all_tools strategy achieves 99% accuracy but at 3× the token cost of focused strategies.
- Regex, the simplest strategy, achieves 95% accuracy at the lowest token cost — making it the efficiency leader for structured queries.
- One-Shot mode is generally preferable: it outperforms Conversation in accuracy for 6 out of 8 strategies while consuming significantly fewer tokens.
- Conversation mode adds 33–73% more tokens across strategies, with only marginal or negative accuracy impact in most cases.
- Query complexity matters: accuracy drops significantly for all strategies when queries require 3 simultaneous tools, revealing the limits of agentic tool selection.

What we measured

Accuracy was defined as correct tool selection and retrieval per query. We tested across query complexity levels (1, 2, and 3 tools required simultaneously) and tracked both total token consumption and accuracy across message turns in conversation mode.

Nine charts, nine perspectives

The following nine pages each present one visualisation from this evaluation, with analysis of what it reveals about strategy design, efficiency tradeoffs, and the practical implications of agentic tool selection in production RAG systems.

Charts labelled G1–G7 correspond to the original evaluation notebook series. Additional charts provide strategy-type breakdowns and efficiency frontier analysis.

Chart 1 — Accuracy by Query Complexity (One-Shot Mode)

This grouped bar chart reveals how accuracy degrades as query complexity increases. Each group represents queries requiring 1, 2, or 3 simultaneous retrieval tools. At 1 tool, most strategies achieve 100% accuracy — the task is straightforward. At 2 tools, performance begins to diverge, with most strategies settling in the 89–99% range. The real test arrives at 3 tools: accuracy drops significantly for every strategy, with some falling to 79%.

The dashed 90% threshold line highlights which strategies hold above it as complexity grows. **all_tools, milvus,** and **regex** maintain or approach the 90% line at 3-tool queries, while **bm25, milvus,** and **hybrid_pinecone** fall to 79%. This exposes a clear ceiling effect: no strategy perfectly handles queries demanding concurrent multi-tool reasoning, and the gap between best and worst widens substantially at higher complexity.

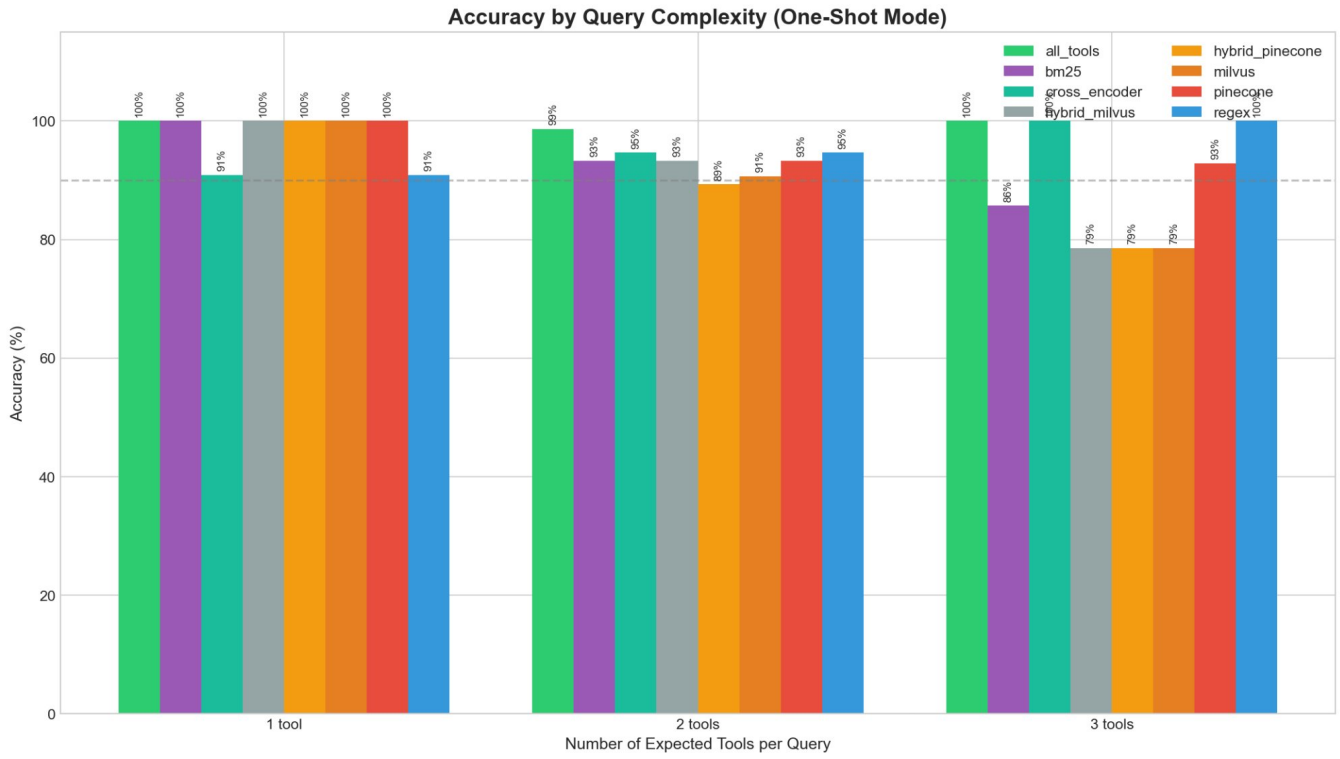


Chart 2 (G1) — Accuracy Comparison: One-Shot vs. Conversation Mode

This side-by-side bar chart provides the clearest summary of the One-Shot vs. Conversation tradeoff across all eight strategies. Blue bars (One-Shot) and red bars (Conversation) are shown together for each strategy, sorted by One-Shot accuracy from best to worst.

all_tools leads at 99% (One-Shot) and 98% (Conversation) — both modes excellent, with One-Shot marginally better. **regex** and **cross_encoder** both achieve 95% One-Shot with divergent Conversation performance. Most striking is **milvus**: 90% One-Shot rising to 93% Conversation — one of only two strategies that actually improves in multi-turn mode. The chart confirms that One-Shot is typically the safer default, with Conversation offering benefits only for specific strategies in constrained scenarios.

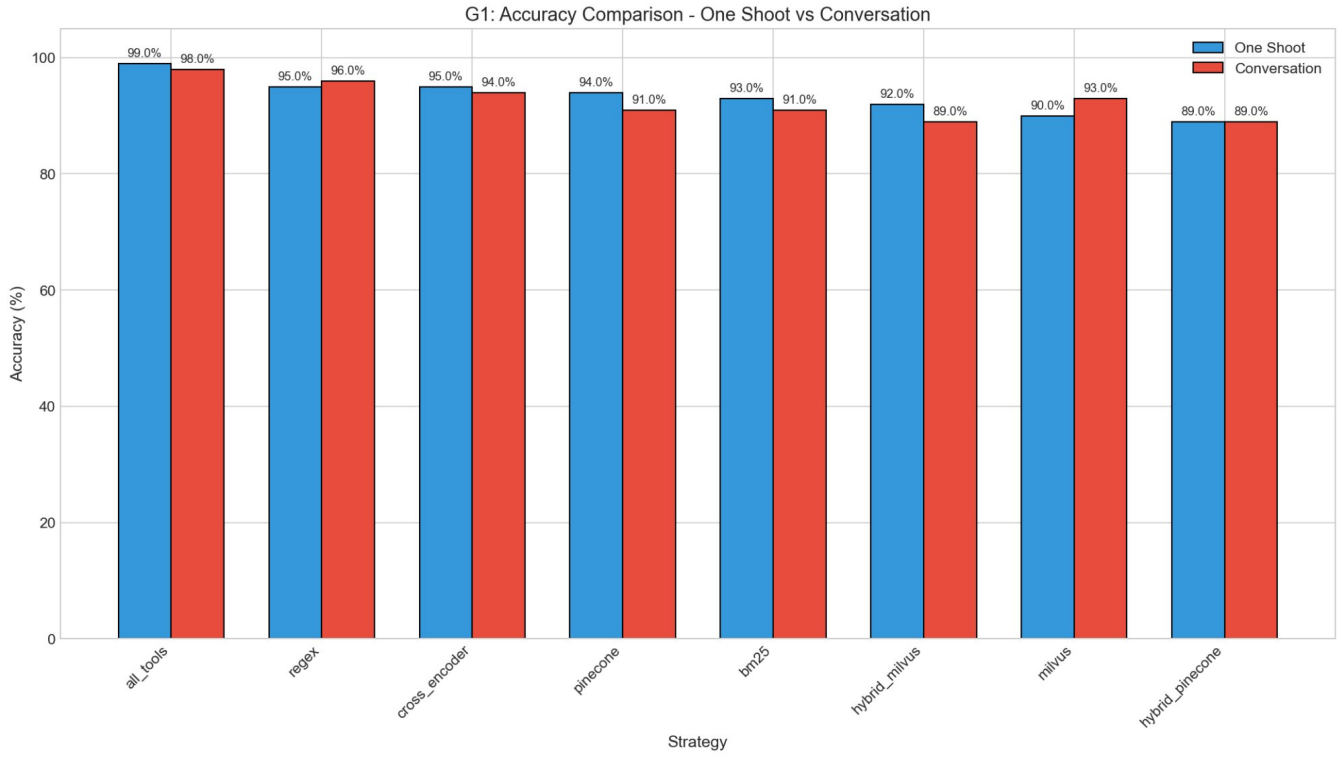


Chart 3 — Accuracy Change: Conversation vs. One-Shot Mode (percentage points)

This waterfall-style chart isolates the delta — the pure gain or loss in accuracy when switching from One-Shot to Conversation mode. Green bars indicate improvement; red bars indicate degradation. The pattern is striking: only two strategies benefit from conversation, while six lose accuracy.

milvus gains the most (+3pp), suggesting that multi-turn dialogue genuinely helps the agent when using dense vector search — perhaps allowing it to refine ambiguous queries. **regex** gains +1pp; **hybrid_pinecone** holds flat at 0pp. But **pinecone** and **hybrid_milvus** both lose 3 percentage points, while **bm25** drops 2pp. The practical implication is direct: before choosing Conversation mode, verify that your specific strategy actually benefits from it — for most, it is a net negative on accuracy while also consuming significantly more tokens.

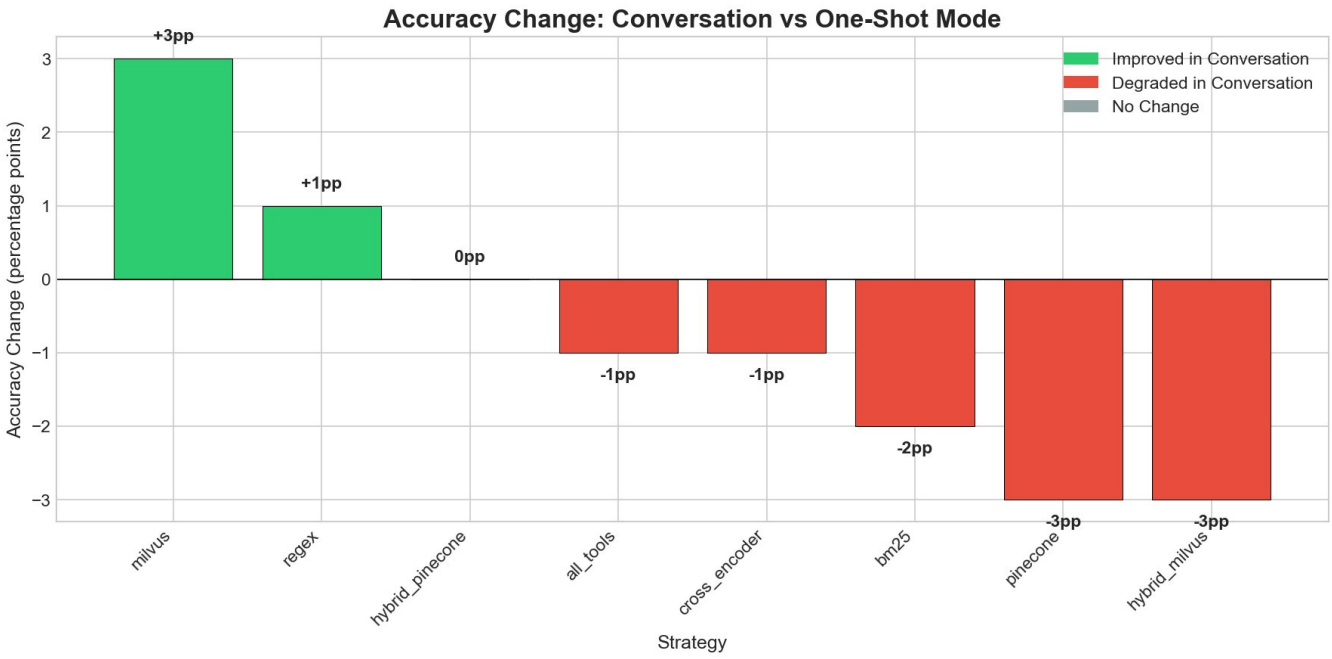


Chart 4 (G2) — Token Usage per Strategy: One-Shot Mode

This horizontal bar chart breaks down average token consumption per query in One-Shot mode, split into input tokens (blue) and output tokens (red). The contrast between strategies is dramatic: **all_tools** consumes 18,492 tokens per query — more than 3× the next closest strategy (pinecone at 7,459). This overhead comes from the agent's system prompt describing all available tools, which grows proportionally with the number of tools.

The most efficient strategies are tightly clustered: **regex** (4,986), **cross_encoder** (5,643), **hybrid_pinecone** (5,886), **hybrid_milvus** (5,961), and **milvus** (5,967) all consume under 6,000 tokens. Output tokens are minimal across all strategies — the cost is almost entirely in input context. This chart makes the case for focused, single-strategy agents over all-tools configurations when token budget matters.

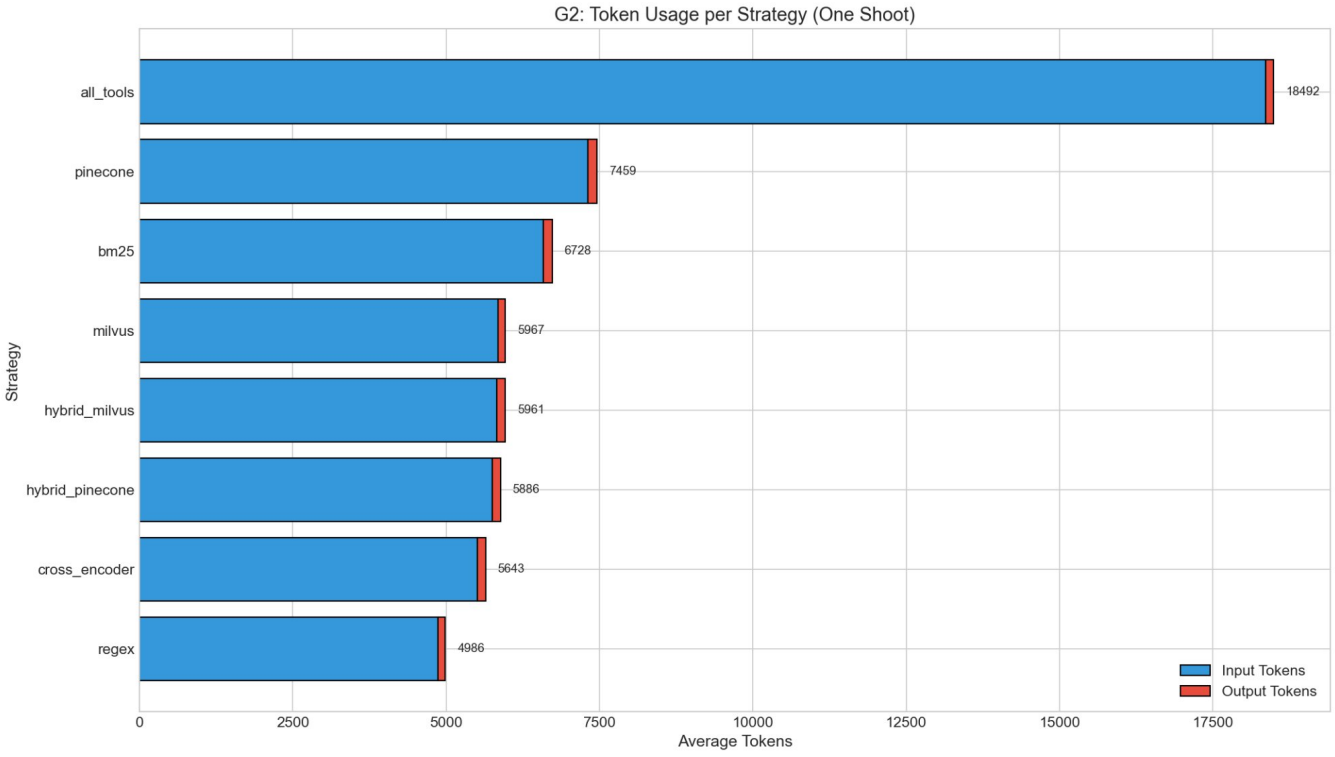


Chart 5 — Token Usage: One-Shot vs. Conversation Mode (per 100 queries)

This horizontal comparison chart shows total token consumption per 100 queries for each strategy, contrasting One-Shot (blue) and Conversation (red). The red percentage labels quantify the overhead of Conversation mode. The message is unambiguous: multi-turn dialogue is expensive.

Token overhead ranges from +33% (bm25) to +73% (milvus) when switching from One-Shot to Conversation. Only **all_tools** defies this pattern with a –6% reduction in Conversation mode — an anomaly likely explained by the agent converging on fewer tool calls across turns when it already has full tool access. For all other strategies, Conversation mode costs substantially more with, as the accuracy chart confirms, little corresponding accuracy benefit. This chart is the strongest argument for defaulting to One-Shot mode in production agentic RAG deployments.

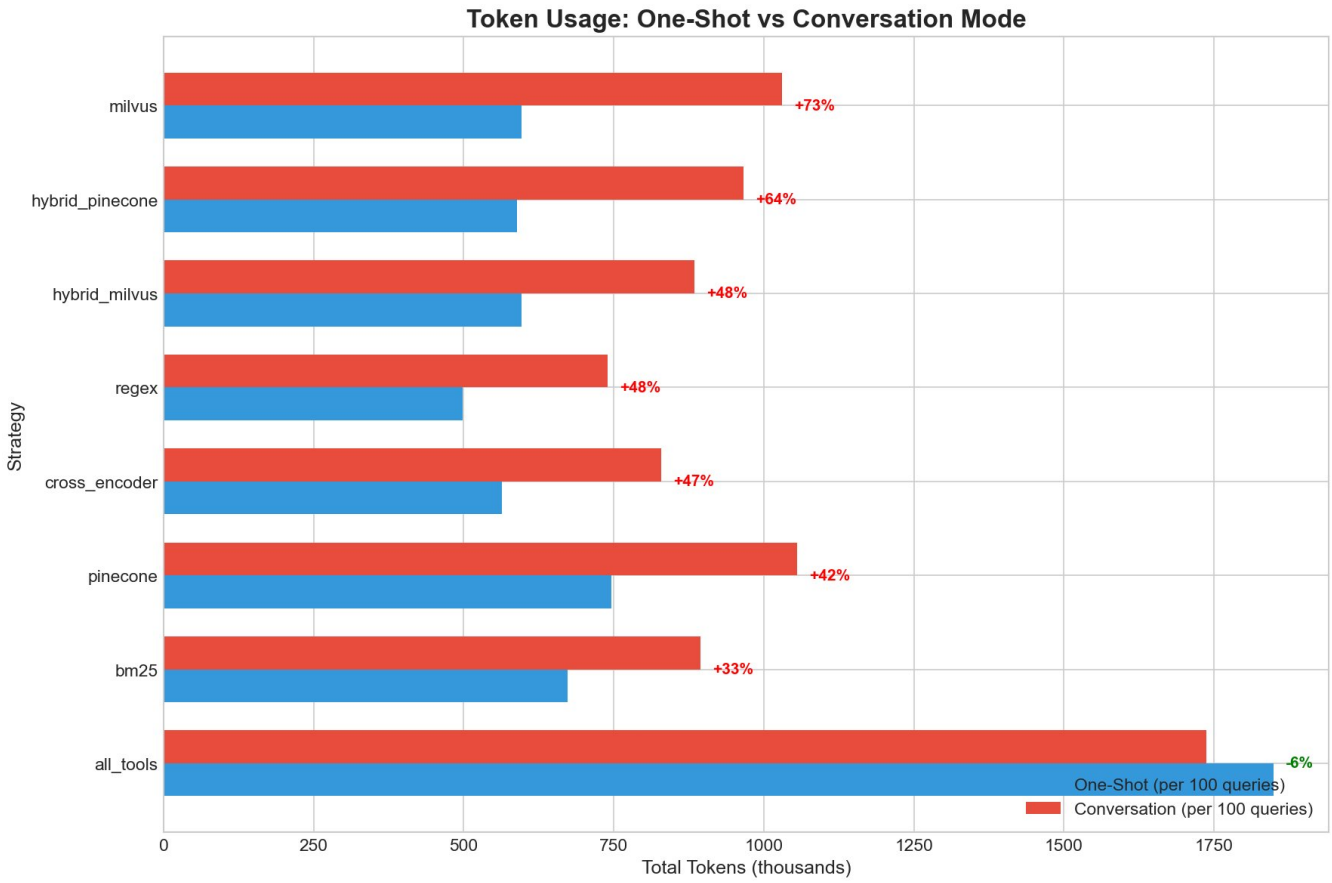


Chart 6 (G5) — Accuracy vs. Token Cost: The Efficiency Frontier (One-Shot)

This scatter plot is the most strategically important chart in the evaluation. Each bubble represents one retrieval strategy plotted on two axes: accuracy (Y) vs. average tokens per query (X). The top-right quadrant is ideal — high accuracy, higher cost. The top-left is the true sweet spot: high accuracy at low cost.

regex and **cross_encoder** dominate the efficiency frontier — both achieving ~95% accuracy while consuming only ~5,600–5,700 tokens per query. They sit firmly in the top-left. **all_tools** achieves the highest accuracy (99%) but at 18,500 tokens — a 3× premium over the next cluster. The remaining strategies cluster between 88–94% accuracy at 5,900–7,500 tokens. The clear recommendation: if 95% accuracy is acceptable, regex or cross_encoder deliver it at a fraction of the cost. Only when marginal accuracy gain from 95% to 99% is business-critical does the all_tools premium become justifiable.

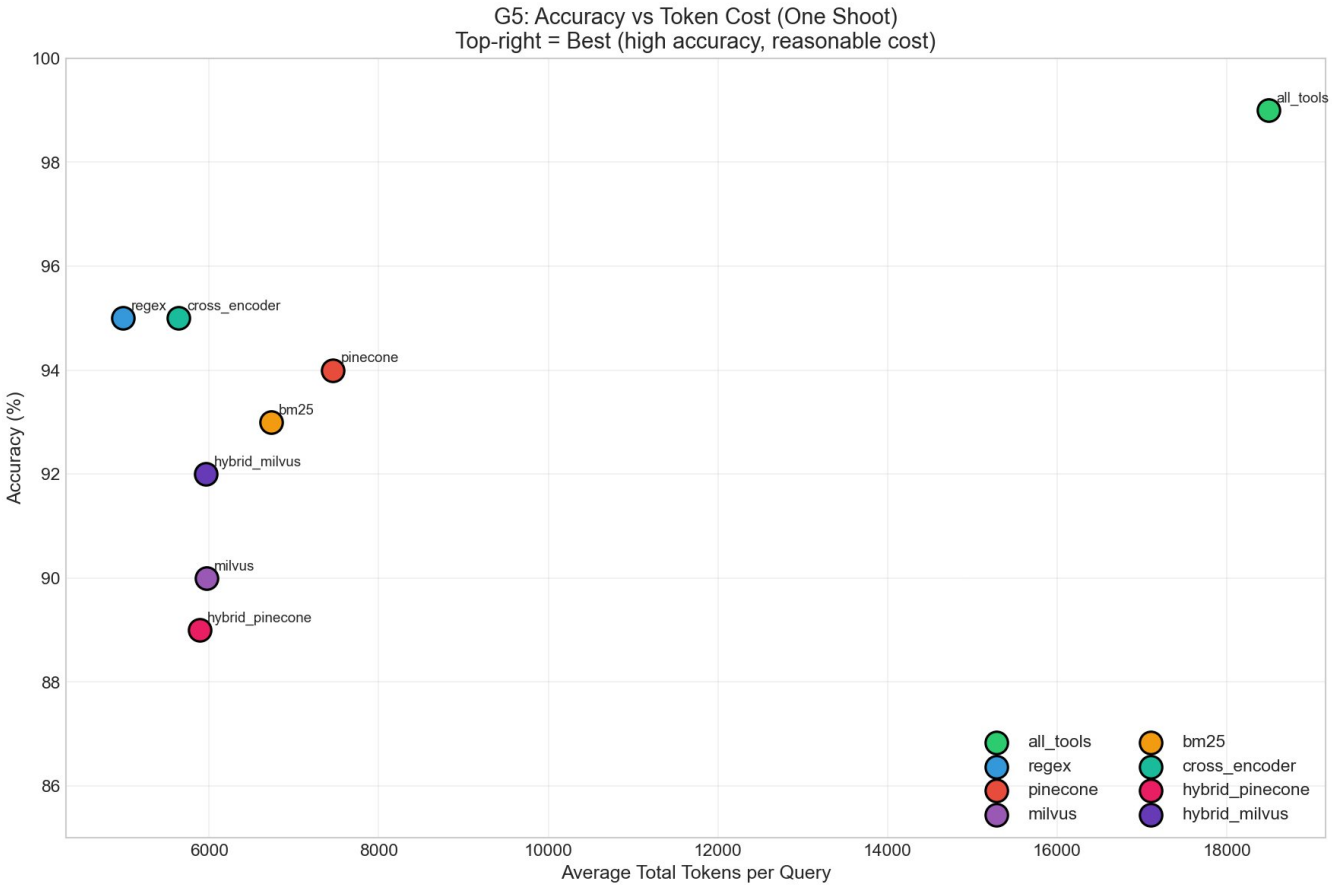


Chart 7 — Accuracy and Token Usage by Strategy Type (One-Shot vs. Conversation)

This dual-panel chart groups strategies into four types — Baseline (all_tools), Hybrid, Keyword (bm25, regex), and Semantic (milvus, pinecone, cross_encoder, reranker variants) — and compares their average accuracy and token consumption across both modes.

The **Baseline (all_tools)** category stands alone: 99% accuracy (One-Shot), 98% (Conversation), but consuming ~18–17k tokens per query — roughly 2–3× more than any other category. **Keyword** strategies average 94% / 93.5% across modes at just 8k tokens. **Semantic** strategies cluster at 93% / 92.7% at ~6–10k tokens. **Hybrid** strategies deliver the lowest accuracy at 90.5% / 89.0% — a surprising result suggesting that combining search methods without careful calibration can introduce noise rather than reducing it. This grouping confirms that strategy architecture matters as much as individual tool selection.

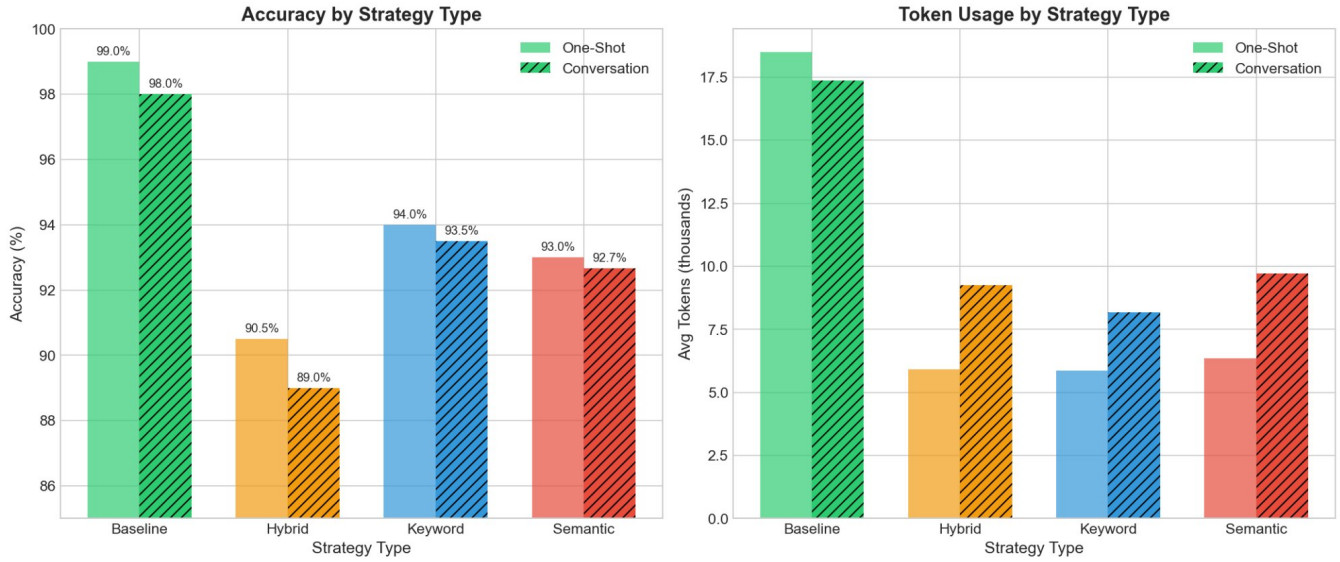


Chart 8 (G6) — Strategy Performance Heatmap: Normalised Multi-Metric View

This normalised heatmap provides the most comprehensive single-view comparison of all strategies across five metrics simultaneously: Accuracy, Avg Duration, Avg Requests, Avg Tool Calls, and Avg Total Tokens. Green cells indicate better performance; red cells indicate worse. Each metric is normalised to 0.0–1.0 so that disparate scales can be compared directly.

all_tools dominates three metrics (Accuracy 1.00, Avg Requests 1.00, Avg Tool Calls 1.00) but scores a stark red 0.00 on Avg Total Tokens — the worst token efficiency of any strategy. **regex** is the inverse: perfect 1.00 on token efficiency, strong 1.00 on Avg Duration (fastest), but mid-range on other metrics (0.46–0.60). **pinecone** shows the most unbalanced profile — reasonable accuracy (0.50) but last place on duration, requests, and tool calls (all 0.00). **cross_encoder** and **milvus** offer the most balanced profiles across all five metrics, making them the safest all-round choices when no single metric dominates. The heatmap confirms that no strategy wins on every dimension simultaneously — the right choice always depends on which trade-offs matter most for your deployment.

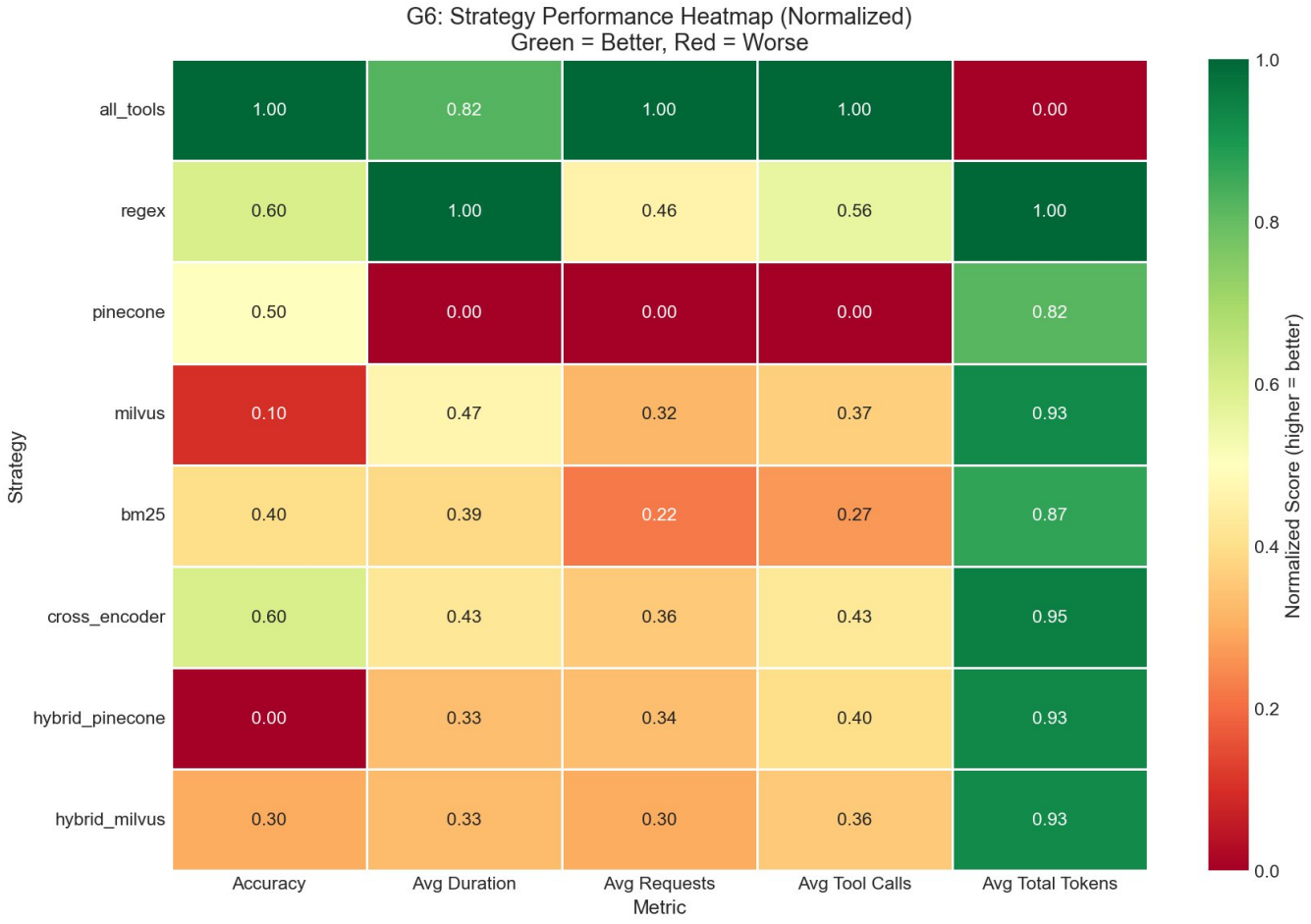


Chart 9 (G15) — Cost Projection at Scale (GPT-4o-mini pricing, log scale)

This log-scale bar chart projects total cost (USD) for four representative strategies — all_tools, regex, bm25, and rag (semantic) — across five volume tiers from 100 to 1,000,000 queries. The logarithmic Y-axis compresses the absolute differences, but the relative gaps between strategies remain constant: at every scale, all_tools costs roughly 3× more than the focused alternatives.

At 100 queries the difference is negligible ($0.28 vs $0.08). At 10,000 queries it becomes meaningful ($28 vs $8). At 1,000,000 queries, **all_tools reaches ~$2,800** while regex stays at ~$800 — a $2,000 saving per million queries on GPT-4o-mini alone. For more expensive models (Claude 3.5 Sonnet, GPT-4o), the savings scale proportionally: at Claude 3.5 Sonnet pricing, the same million queries would cost ~$57,100 with all_tools vs ~$16,400 with regex — a **$40,700 saving**. This chart makes the business case unambiguous: dynamic tool selection is not an optimisation — it is a cost control necessity at scale.

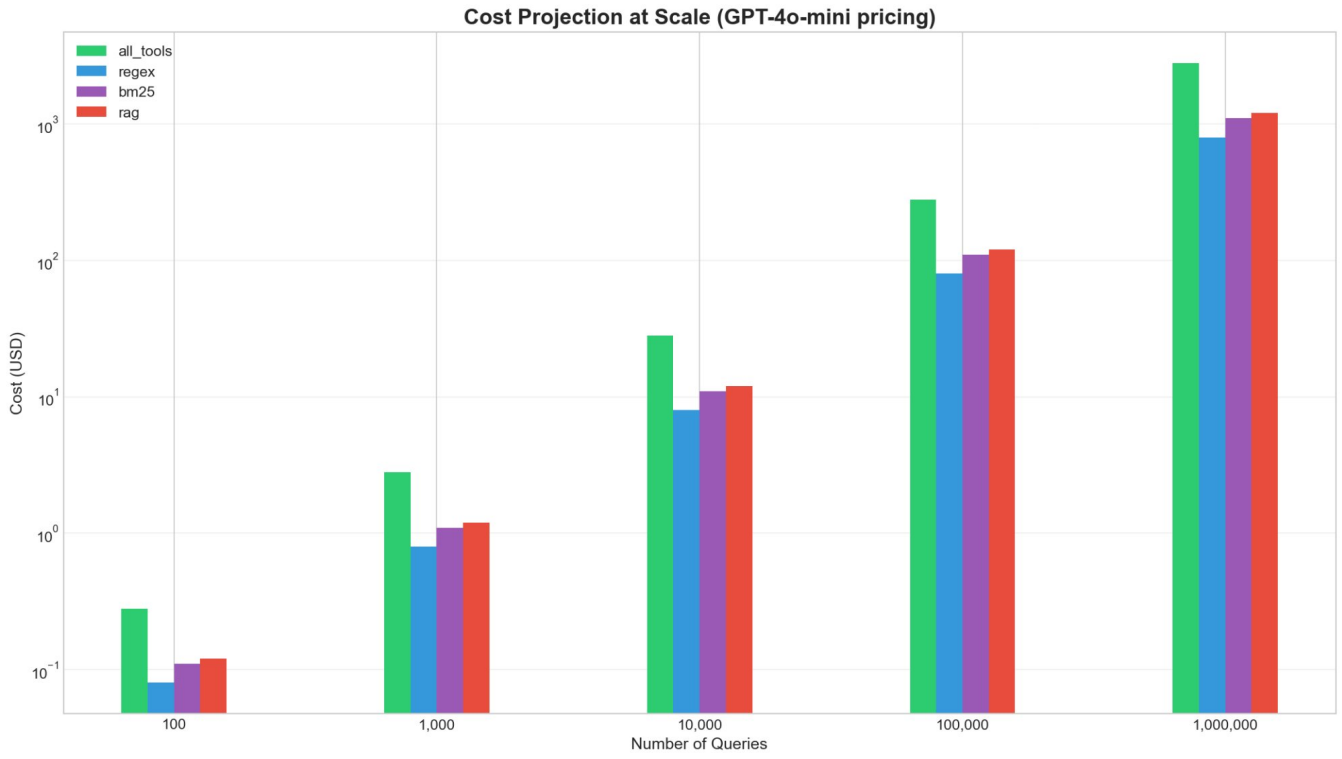


Chart 10 (G3b) — Accuracy vs. Message Index in Conversation Mode

This line chart tracks per-strategy accuracy at each turn of a multi-turn conversation (message indices 1–5). Rather than showing a smooth improvement, accuracy is volatile — spiking and dipping unpredictably as the conversation progresses. This reveals a fundamental challenge with Conversation mode: the agent's performance is not monotonically improving with more context.

all_tools (green squares) is the standout exception — after an initial dip at message 1, it stabilises at 100% from message 2 onward, demonstrating that broad tool access enables reliable convergence in multi-turn dialogue. **pinecone** (red) shows the worst instability, dropping to 80% at message 3 before partially recovering. Most other strategies oscillate in the 85–100% band. This chart cautions against assuming that more dialogue rounds improve retrieval quality — for most focused strategies, accuracy is highest in early turns and degrades with compounding context.

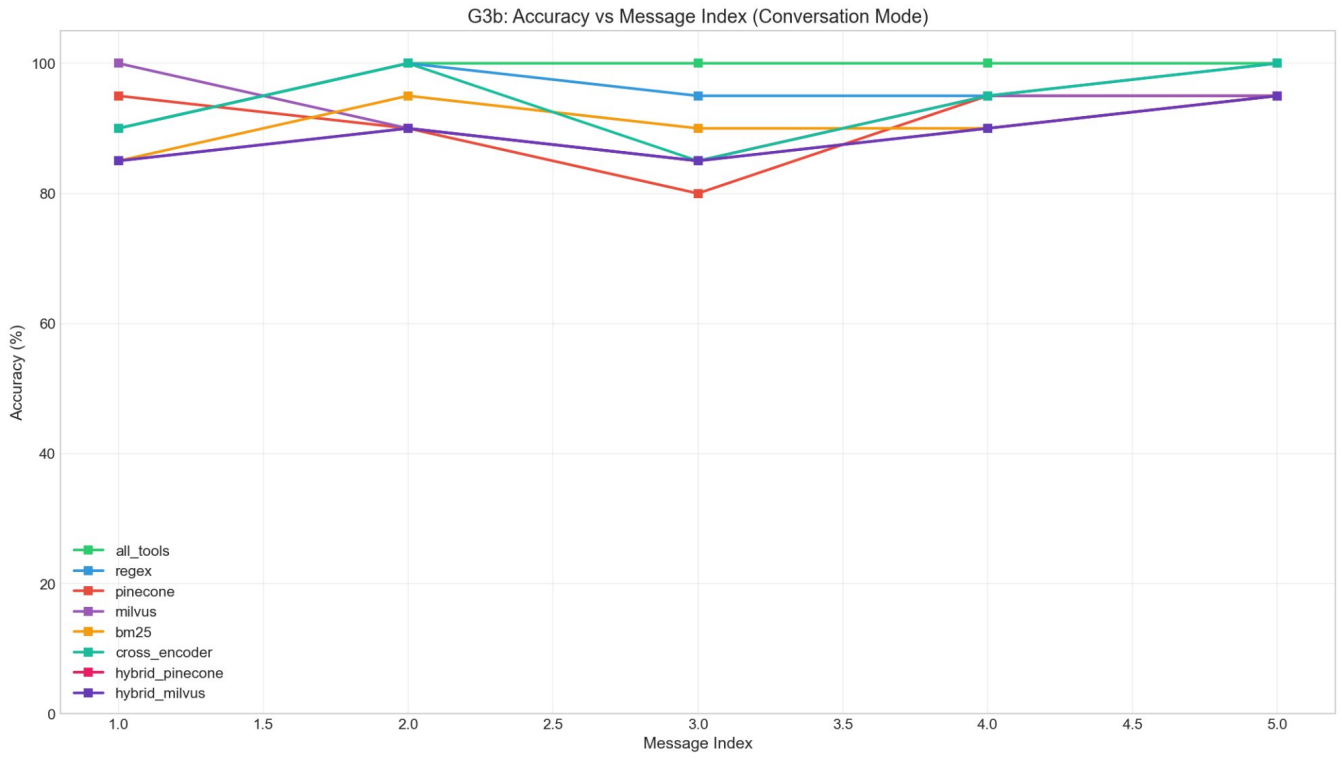
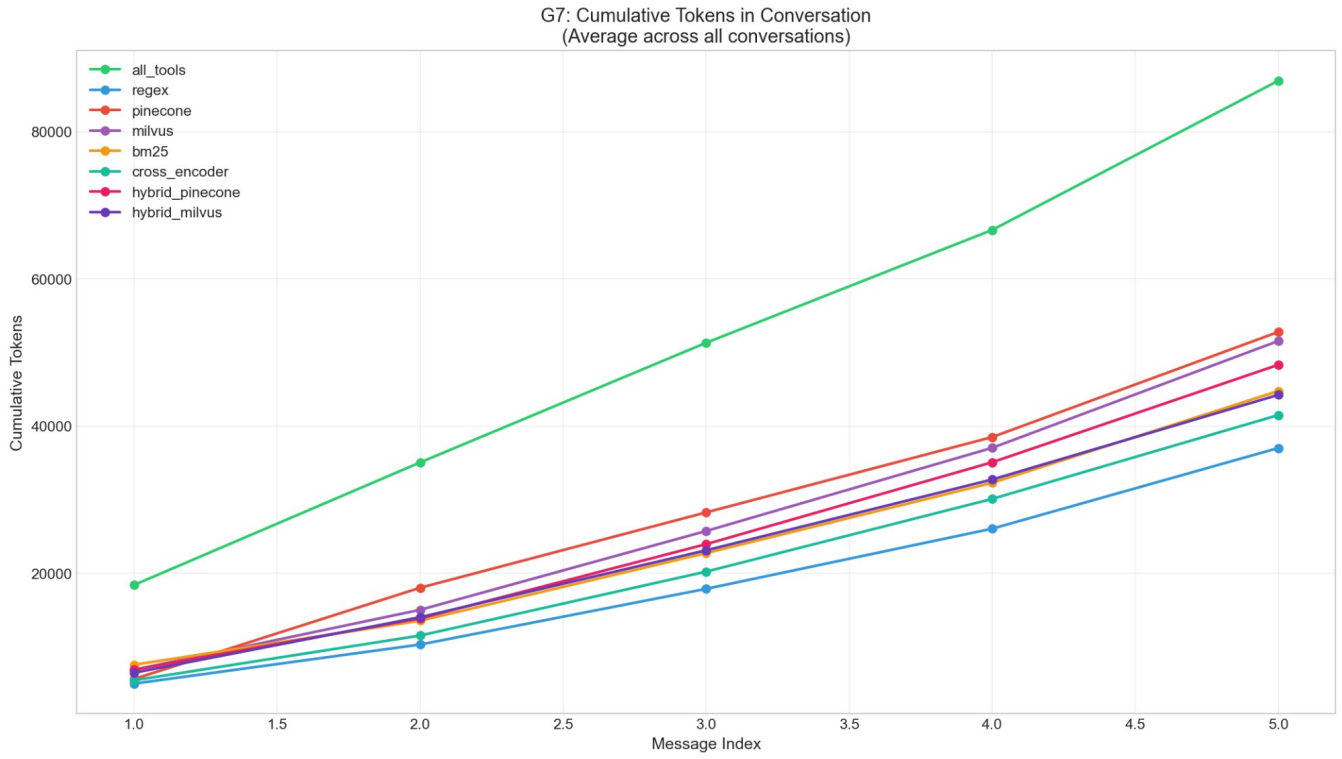


Chart 11 (G7) — Cumulative Token Accumulation in Conversation Mode

This line chart shows how total token consumption accumulates across the 5 message turns of a conversation, averaged across all test conversations. The separation between strategies becomes increasingly pronounced as the conversation extends — a direct consequence of the expanding context window in multi-turn exchanges.

all_tools (green) diverges dramatically from all other strategies, reaching ~87,000 cumulative tokens by message 5 — more than double the next highest (pinecone at ~52,000). The high initial token cost of the all-tools system prompt compounds with each turn, creating runaway context growth. By contrast, **regex** (blue) accumulates the most slowly, reaching only ~37,000 tokens at message 5. The cluster of focused strategies (bm25, milvus, hybrid variants, cross_encoder) grows in parallel in the 41,000–51,000 range. This chart illustrates why long conversations with all_tools agents are impractical at scale — context accumulation creates a cost trajectory that is unsustainable for high-volume production use.



Key findings

More tools ≠ better accuracy.

all_tools achieves 99% accuracy — the highest of any strategy — but at 18,492 tokens per query: more than 3× the cost of focused alternatives. For most applications, the 4–9 percentage point accuracy gain over regex or cross_encoder does not justify the 3× token premium.

One-Shot dominates for efficiency.

One-Shot mode outperforms Conversation mode in accuracy for 6 out of 8 strategies while consuming 33–73% fewer tokens. Only milvus and regex genuinely benefit from multi-turn dialogue. The default should be One-Shot unless your specific strategy and use case demonstrate measurable improvement from conversation.

Regex punches far above its weight.

Pattern-based exact matching achieves 95% accuracy at the lowest token cost of any strategy (4,986 tokens/query). For structured, predictable query patterns, regex is not a fallback — it is a deliberate first choice. Reserve complex retrieval strategies for query types that genuinely require them.

Query complexity is the real ceiling.

All strategies degrade at 3-tool queries. No current agentic strategy handles simultaneous multi-tool reasoning reliably. System design should minimise queries requiring 3+ concurrent tools, or decompose them into sequential single-tool calls.

Hybrid strategies disappoint.

Combining keyword and semantic search without careful calibration produces the lowest average accuracy of any strategy type (90.5% One-Shot). Hybrid approaches require significant tuning investment to outperform their individual components.

Recommendations

- **Start with regex or cross_encoder.** Both deliver ~95% accuracy at under 6,000 tokens. Justify the cost of more complex strategies with evidence, not assumption.
- **Default to One-Shot mode.** Multi-turn conversation adds cost and typically reduces accuracy. Validate Conversation mode benefits empirically before deploying it in production.
- **Reserve all_tools for high-stakes queries.** If near-perfect accuracy is genuinely required and cost is secondary, all_tools delivers 99%. Otherwise, the efficiency cost is unjustifiable.
- **Design around query complexity.** Evaluate the distribution of 1-tool vs. 2-tool vs. 3-tool queries in your use case. If 3-tool queries are common, consider query decomposition strategies rather than relying on single-pass agentic selection.
- **Benchmark token accumulation in long conversations.** For conversational agents, cumulative token growth is the primary cost driver — not per-query cost. Monitor and cap conversation length for strategies with high per-turn overhead.

Case Study 5 key finding

Agentic RAG does not require frontier complexity. The most efficient path to 95% accuracy costs under 6,000 tokens per query using pattern-based or cross-encoder retrieval in One-Shot mode. All-tools agents deliver marginal accuracy gains at a 3× token premium — a tradeoff that is rarely justified in production. Design your agentic RAG system around the simplest strategy that meets your accuracy threshold, then add complexity only where evidence demands it.

05

What Comes Next? Future Trends in Context Engineering & Agentic RAG

The future of RAG rests in agentic architectures where AI systems are not limited to simple retrieval and generation, but actively reason about which tools and retrieval strategies to deploy. Platforms like Claude Code orchestrate multiple retrieval methods — dynamically selecting between web search, knowledge base queries, and hybrid approaches based on query intent. Frameworks like PydanticAI, LangChain, and LlamaIndex enable agents to compose retrieval pipelines, maintain conversation memory, and coordinate multi-agent systems where specialised retrievers work together.

In this chapter, we explore emerging patterns in multi-modal retrieval, real-time data integration, and user personalisation — and how these forces are transforming RAG from static pipelines into adaptive, context-aware systems that evolve with each interaction.

Introduction to RAG Paradigms

Retrieval-Augmented Generation (RAG) equips a language model with an external knowledge source so it can "look things up" before answering. A user query is expanded with relevant content retrieved from a database or document index, and the LLM then generates a response grounded in that content. This improves factual accuracy and keeps answers current by using non-parametric memory instead of relying solely on fixed training data.

Illustrative example: customer support

Consider a customer support chatbot asked: *"I'm unable to log in to my account after the latest update. What should I do?"*

Traditional RAG approach

The chatbot treats this as a single-turn query. It searches a knowledge base for keywords like "log in," "account," and "latest update," retrieves a relevant FAQ article, and feeds it to the LLM. The LLM generates an answer: a step-by-step from the article on resetting the password or clearing the cache. The interaction ends once the answer is given.

This assistant delivers an answer grounded in documentation, but it will not go beyond the given articles. If the first retrieved document is insufficient, a basic RAG system may give an incomplete answer — because it does not autonomously perform additional searches or actions.

Agentic RAG approach

An agentic assistant handles this more like a support agent solving a case. First, the agent might clarify or decompose the problem — check what the "latest update" refers to, or ask "Are you seeing an error message?" Then it performs a series of retrievals and actions.

The agent searches internal docs for "login issues after update" and, if the first article proves insufficient, automatically refines the query or searches another source. It retrieves multiple pieces of information over several steps, each time analysing whether the new information answers the question. It could even call an API to check the user's account status.

After gathering enough evidence — a known bug description and a workaround — the agent composes a helpful answer: *"It looks like this is a known issue after the update. I've checked your account — no flags there. The solution is to clear your app cache and log in again."*

This illustrates how agentic RAG not only fetches the book, but also reads it, cross-checks other sources, and figures out the solution. Traditional RAG yields a straightforward, context-based answer. Agentic RAG delivers more interactive and autonomous help — adapting its strategy if the first attempt does not solve the problem, much like a human support agent.

05

Architecture and design differences

In a traditional RAG model, the process is **linear** — the system performs a single retrieval from a vector database or knowledge base, optionally re-ranks results, then passes relevant text to the LLM which generates an answer. The architecture is relatively simple: a single retrieval step followed by a single generation step.

The LLM's prompt is augmented with the top retrieved documents, and the model outputs an answer. This pipeline does not maintain state beyond each query: once an answer is produced, the process resets. There is no persistent memory of previous interactions, and reasoning is largely implicit in the model's final forward pass. Design focus centres on building a good retriever — indexing, embedding — and a prompt strategy for grounding the answer in retrieved facts.

An agentic RAG system has a more complex, **modular design**. It introduces a controller or agent module on top of the LLM. This agent uses the LLM not just for answering, but for reasoning in steps — often employing a prompt strategy like chain-of-thought or the ReAct framework (Reason+Act) to decide on next actions.

The architecture includes a loop where the agent can call a retrieval tool multiple times or invoke other tools: a vector database query tool, a web search tool, a calculator, or a code executor. The system must manage the sequence of actions — the agent observes the query, decides which tool to use, gets the result, and can then decide the next step based on that result. This requires maintaining a working memory: the agent remembers what it has found so far and what its current goal is.

Core mechanism and control flow

Traditional RAG

Autonomy and reasoning capabilities

A traditional RAG system does no explicit planning — it responds to each query by fetching relevant text and producing an answer from it. The system will not take any initiative beyond that single retrieval. It has zero autonomy: if the user's query does not explicitly ask for something, the system will not volunteer new actions. This makes traditional RAG highly controlled and predictable, but limits it to answering the question as posed.

Agentic RAG imbues the model with a goal-driven modus operandi. The agent has a notion of objectives and can pursue them through multiple steps, giving it partial autonomy. Once tasked, it can decide how to achieve its goal without constant user prompting. If asked for an in-depth analysis, an agentic system might on its own break the request into sub-tasks: gather background data, verify facts, then compose the analysis.

Techniques like the ReAct framework are often used: the model generates a "Thought" (reasoning) and an "Action" (like a tool call) iteratively. As a result, the agent can handle open-ended or complex tasks that require more than one step of thinking. The trade-off is that autonomy brings unpredictability — the agent may need careful constraints and guardrails to stay on track.

Decision-making: who is in charge?

Traditional RAG — the developer decides. The developer determines when and where the search happens. The logic is hard-coded: "Always query the vector DB when a user asks a question." The retrieval step is fixed at design time.

Agentic RAG — the model decides. The model determines if it needs to retrieve information at all, which specific tool to use (vector search vs. web search vs. file read), and how many times to invoke it. This flexibility is powerful, but it transfers control from the developer to the agent.

Relationship between the two paradigms

05

Orchestration and tool use

In traditional RAG, orchestration is minimal because the control flow is straightforward: the only "tool" is typically the retrieval mechanism, invoked just once per query. The logic — query, retrieve, then generate — is static and predetermined. There is usually a single knowledge source, so no dynamic routing between multiple sources is needed. This simplicity means latency is low and implementation can often be completed with a few API calls.

Agentic RAG requires a more sophisticated orchestration layer. The agent acts as a controller that can invoke multiple tools or data sources. An agentic system might need to decide whether to fetch from an internal knowledge base, perform a web search, query a SQL database, or call an external API — depending on the user's request. This dynamic tool use requires routing logic. In advanced designs, multiple agents are used: a master agent delegates to specialised sub-agents (one for documents, one for code, one for web search) and aggregates the results.

Each action-observation cycle is orchestrated through prompts — the agent's prompt includes a history of what has been done so far and what to do next. For example, an agentic coding assistant might retrieve docs, run a piece of code in a sandbox, read the output, and decide the next step — something a traditional RAG cannot do. Modern agent frameworks (PydanticAI, LangChain, LlamaIndex) provide the building blocks for this orchestration, but integrating them and ensuring reliable behaviour is a significant engineering task.

Implementation focus

Traditional RAG — Retrieval algorithms: vector similarity, re-ranking, and embedding strategies.

> *When an agent needs to decide between querying a knowledge base, searching the web, or calling an API, the orchestration logic is just software — and it deserves the same rigor. PydanticAI treats tool coordination as a first-class engineering problem: type-safe, testable, and observable, so you can trust what your agent is doing and fix it when it does not.*
>
> *— Douwe Maan, Lead Developer of PydanticAI*

Capabilities and use cases

Traditional RAG — Focused Q&A and knowledge lookup. Traditional RAG shines when you need factual, grounded responses to single-shot queries from a fixed body of knowledge. Ideal for FAQ chatbots, documentation assistants, or search engines on proprietary data.

These scenarios require accurate answers drawn from a known set of documents, and they benefit from RAG's ability to update the model's knowledge without retraining — simply by adding new documents to the index. Traditional RAG is also preferred when latency and cost are concerns, since it involves only one model invocation per query, making it faster and cheaper on average.

In summary: if your task is straightforward question-answering, summarisation of a given text, or any situation where a single round of retrieval can provide the needed information, traditional RAG is usually the most efficient choice.

Agentic RAG — Interactive problem solving and complex tasks. Agentic RAG unlocks capabilities beyond the scope of a simple lookup. Well-suited for multi-step tasks that involve planning, research, or tool use beyond text retrieval.

An analytical research agent can use agentic RAG to gather information from multiple sources, verify and synthesise it, and produce a comprehensive report — functioning like a diligent human researcher. In coding assistants, agentic RAG allows the AI to retrieve documentation, run code or tests, and refine its answer based on results.

Use cases that require dynamic decision-making and adaptation — legal research (finding and cross-referencing cases), data analysis (querying databases, then interpreting results), or autonomous AI ops assistants — benefit from an agentic approach. These systems combine retrieval with other capabilities (tool usage, long-form reasoning), making them far more flexible when the problem is open-ended. The trade-off is more moving parts and higher cost — chosen when the additional capability is worth the complexity.

Many real-world AI applications hybridise the two: an AI agent might use a traditional RAG step for factual lookup as one of its actions, marrying the reliability of RAG with the adaptability of an agent.

05

Challenges and considerations

Traditional RAG limitations. The system is only as good as the retrieval corpus and query. Traditional RAG pipelines can break down with unstructured or scattered data, or when a single retrieval is insufficient. Tuning is required for chunking documents and crafting queries — a rigid retrieve-one-and-done approach may pull in irrelevant chunks or miss needed information. Another limitation is the lack of reasoning: if a question requires deductive steps or combining facts, a single-pass RAG may falter. Traditional RAG systems also remain subject to hallucination if the retrieved context is insufficient.

On the upside, traditional RAG's simplicity makes it easier to evaluate and control. Ensuring quality involves monitoring

Attribute	Traditional RAG	Agentic RAG
Autonomy	None — reactive only, answers exactly what is asked	Partial — goal-driven reasoning over multiple steps
Data Sources	Single knowledge base or index (fixed corpus)	Multiple sources and tools (databases, web search, APIs)
Retrieval	One-shot retrieval then generation (no iteration)	Iterative retrieval with refinement; can re-query or switch sources
Tool Use	Typically just a vector DB or search API. No other tools	Web search, calculators, databases, code executors
Planning	No explicit planning; single-step prompt-based generation	Agent plans steps, decomposes tasks, decides on actions
State/Memory	Stateless — no memory beyond current query context	Maintains state within a session; working memory of facts found
Adaptability	Fixed logic — cannot adjust mid-course	Adaptive — can reformulate queries, try different tools mid-task
Cost	Fast & cheap — optimised for speed, single pass	Slow & expensive — iterates many times, higher token usage
Best Use Cases	FAQ bots, knowledge base Q&A, document search	Research assistants, analytical tools, workflow automation

Key differences at a glance

05

Agentic RAG in practice: an applied scenario

Your product team lead asks a question in the weekly review: *"Paid churn jumped 40% in December. What is driving it and what should we do next?"*

This is not a question you answer by searching documentation. It is not a single fact lookup. It is an investigation that requires synthesising evidence from multiple sources, testing hypotheses, and connecting disparate signals into a coherent narrative. And it reveals something important about the limitations of traditional RAG systems.

Traditional RAG follows a straightforward pattern: receive query, retrieve relevant documents, feed context to the model, generate answer. This works effectively for FAQ systems, document Q&A;, and scenarios where answers live in well-indexed knowledge bases.

Agentic RAG evolves this foundation by embedding autonomous reasoning into the pipeline. Instead of a single retrieve-then-generate pass, agentic systems plan investigations, dynamically select data sources, iteratively refine their understanding, and use specialised tools to gather and analyse information. The shift is from "search and summarise" to "plan, retrieve, reason, and act."

The data ecosystem

Most SaaS organisations have scattered evidence across multiple systems:

- **Product analytics** — user behaviour, feature adoption, session patterns.
- **Billing and subscriptions** — churn type, plan changes, payment failures.
- **Customer feedback** — exit surveys, satisfaction scores, support tickets.
- **Support signals** — ticket volume, response patterns, issue categories.

How traditional RAG approaches the question

A traditional RAG system treats this as a search-and-summarise task. It retrieves documents matching "churn" and "December," pulls recent exit survey responses, searches related support tickets, and generates a summary of themes.

The output tends to be generic: *"Users mentioned pricing concerns and missing features. Support tickets increased for onboarding issues. Some customers cited the complexity of the platform."*

This answer is not wrong. But it is incomplete. It lacks the investigation strategy that would reveal *why* churn increased specifically in December, which customer segments were affected, and what specific changes triggered the spike. Traditional RAG provides a summary of available information. It does not conduct an investigation.

How agentic RAG approaches the question

An agentic system recognises this as an investigation requiring multi-step reasoning.

Step 1 — Frame the investigation

The agent translates an ambiguous question into testable sub-questions: What exactly changed? (Segment, plan tier, region, customer type.) Is this voluntary churn or payment failure? What changed in the product or commercial terms? What patterns appear in customer feedback?

Step 2 — Establish ground truth from billing data

The agent pulls a churn decomposition: rate by month, voluntary vs. involuntary cancellations, and breakdown by plan tier. **Finding:** the spike is primarily voluntary cancellations concentrated in Pro plan customers — preventing a common failure mode of investigating product issues when the cause is billing-related.

Step 3 — Identify who is churning

The agent segments churned customers by tenure, acquisition channel, industry, and geography. **Finding:** the spike is heavily concentrated in new customers who churned within 14–21 days, most from a specific partner campaign. The narrative shifts from "churn increased" to "activation failure in a specific cohort."

05

Step 4 — Surface the why from qualitative signals

The agent extracts recurring themes from unstructured feedback and correlates with support ticket categories. Exit survey themes: *"Too expensive for what I got," "Setup was confusing," "Missing integration with X."* Support patterns: tickets tagged "onboarding" or "setup" increased 35% in the same period.

This is where traditional semantic search excels — but note that it is the fourth step, not the first. It is informed by the segmentation work that came before it.

Step 5 — Correlate with timeline and produce decision-ready analysis

The agent checks for temporal alignment: did anything change immediately before the churn increase? **Finding:** a packaging change moved "Integration X" from Pro to Enterprise tier on December 1. The partner campaign targeted users who specifically needed Integration X.

Conclusion: the churn spike is driven by new Pro customers from the partner campaign who encountered an early "value cliff" — Integration X is no longer included in their plan tier. This triggered feedback themes around cost and missing features, compounded by increased onboarding friction.

Recommendation: either restore Integration X access for Pro customers from this partner cohort, adjust partner messaging to set correct expectations, or modify partner targeting to focus on Enterprise-tier buyers.

The agent activity log

```
[Agent: Clarify scope]
→ "Breaking down: which customer segment,
which churn type, what changed recently?"
[Agent: Query billing system]
→ Result: Voluntary churn ↑42%, concentrated
in Pro plan
[Agent: Segment churned cohort]
→ Result: 73% are new customers (14–21 day
window), 68% from partner channel
```

What makes agentic RAG different

The churn investigation demonstrates several key distinctions between the two paradigms:

Core pattern. Traditional RAG is one-shot: retrieve → generate. Agentic RAG is iterative: plan → retrieve → reason → act. The agent does not stop after one pass — it continues until it judges the task complete.

Data sources. Traditional RAG accesses a single knowledge base or vector index. The churn investigation required billing APIs, product analytics, customer feedback, support ticketing systems, and release notes — simultaneously.

Query strategy. Traditional RAG uses a static, single retrieval pass. The agent in this scenario reformulated its queries at every step based on what it had found — narrowing from "churn" to "Pro plan voluntary churn" to "new customer cohort from partner channel."

Planning. Traditional RAG has no explicit planning; it follows a fixed retrieve-answer flow. The agent explicitly decomposed the question into five investigation steps, each building on the previous finding.

State and memory. Traditional RAG is stateless. The agent maintained a working memory across all investigation phases — knowing at step 4 that it was looking for qualitative signal to explain a billing-verified, segment-specific cohort failure.

Output. Traditional RAG produces a summary of retrieved information. The agent produced a decision-ready analysis: a confirmed root cause, supporting evidence from five independent data sources, and a concrete recommendation.

The architectural shift

*From **retrieve-then-generate** to **plan-retrieve-reason-act.** This is not merely a technical upgrade — it is a change in what the system is capable of. Where traditional RAG answers questions, agentic RAG solves problems. The churn scenario required an investigation that no search-and-summarise system could have conducted. Only the iterative, multi-source, hypothesis-testing approach of agentic RAG revealed the root cause.*

05

Visualising the difference

Traditional RAG flow

```
User query: "Why did churn spike?"
↓
Retrieve relevant documents (exit surveys, support tickets)
↓
Feed context to LLM
↓
Generate summary: "Users mentioned pricing and features"
```

Agentic RAG flow

```
User query: "Why did churn spike?"
↓
Agent plans investigation
■■ Frame: What changed? Which segment? Churn type?
■■ Hypothesis: Need billing breakdown first
↓
Agent retrieves from billing API
■■ Finding: Voluntary churn in Pro plan
↓
Agent refines strategy
■■ Next: Who are these Pro churners?
↓
Agent queries analytics + CRM
■■ Finding: New customers, partner channel
↓
Agent gathers qualitative signals
■■ Exit surveys: "too expensive," "missing integration X"
■■ Support tickets: +35% onboarding issues
↓
Agent correlates with timeline
■■ Integration X moved tiers Dec 1
↓
Agent synthesises decision-ready output
■■ Root cause identified + recommendation
```

Note: Agentic flow includes feedback loops and adaptive strategy refinement at each step.

When to use which approach

Traditional RAG excels when:

- **Query type:** single fact lookups, defined questions with clear answers.
- **Data sources:** one knowledge base or well-indexed document collection.
- **Reasoning:** retrieve and summarise is sufficient.
- **Constraints:** low latency requirements, cost sensitivity, predictable load.

Examples: FAQ bots, internal documentation search, customer support for known issues.

Agentic RAG enables scenarios in:

- **Query type:** open-ended investigations, multi-part questions, root cause analysis.
- **Data sources:** multiple systems requiring different access patterns (APIs, databases, analytics platforms).
- **Reasoning:** requires hypothesis testing, evidence correlation, iterative refinement.
- **Outcome:** decision-ready insights, not just information summaries.

Examples: business intelligence investigations, diagnostic workflows, research assistants, complex customer issue resolution.

Decision criteria

When evaluating whether agentic RAG is appropriate, consider four factors:

- **Query complexity** — can this be answered in one retrieval pass, or does it require investigation?
- **Source diversity** — is the answer in one place, or scattered across systems?
- **Reasoning depth** — do we summarise information, or synthesise evidence into conclusions?
- **Value of outcome** — does the use case justify the additional orchestration complexity and cost?

05

Implications for teams

Architecturally

Traditional RAG systems are simpler to deploy and maintain. The flow is predictable: embed documents, store vectors, retrieve on query, generate response. Performance characteristics are well understood.

Agentic RAG demands more robust orchestration. You need state management across investigation steps, error handling for multiple API calls, strategies for when to stop iterating, and mechanisms to prevent runaway costs. If your agent can invoke five different data sources, you need to handle partial failures gracefully.

For decision-makers

Traditional RAG is the right choice when the task is straightforward retrieval. Agentic RAG becomes valuable when the alternative is either manual investigation by analysts or accepting incomplete answers.

The question is not "should we always use agentic RAG?" It is "where does investigation complexity justify the orchestration investment?"

For developers

Building agentic RAG systems requires skills beyond traditional RAG pipelines. You are orchestrating multi-step workflows, designing agent strategies, managing state across reasoning steps, and integrating diverse tools and APIs. The patterns resemble building autonomous systems more than building search interfaces.

Frameworks like PydanticAI or LangGraph provide orchestration primitives. But the hard work is in designing the agent strategy: when to retrieve, when to reason, when to call tools, and when to stop.

A note from Vstorm

Conclusion

Agentic RAG is not a replacement for traditional RAG. It is an architectural evolution that expands what retrieval-augmented systems can do. Where traditional RAG retrieves and generates, agentic RAG plans, investigates, and synthesises.

The churn investigation scenario illustrates this distinction clearly. A search-and-summarise system would provide a list of themes. An agentic system conducts an investigation, identifies root causes, and delivers decision-ready recommendations.

The question is not which approach is better. It is which approach matches the task. For FAQ systems and document search, traditional RAG is simpler and more cost-effective. For diagnostic workflows and multi-source investigations, agentic RAG enables capabilities that were previously the sole domain of manual analysts.

Both traditional and agentic RAG represent powerful paradigms for augmenting LLMs with external knowledge, but they cater to different needs. Traditional RAG is like a precision tool for answering questions with authoritative sources, prized for its simplicity and reliability. Agentic RAG transforms the system into a more autonomous problem-solver, capable of handling complex goals by planning, retrieving, and acting in a loop.

Product teams may start with traditional RAG for straightforward Q&A; and evolve toward agentic RAG as they seek greater intelligence and autonomy. The choice ultimately depends on the use case requirements: if you need quick, factual answers from a stable corpus, traditional RAG is often best. If you need an AI that can reason, use tools, and carry out multi-step tasks, agentic RAG provides the necessary framework — albeit with greater engineering effort to ensure it operates safely and effectively.

Chapter 5 key finding

Beyond Hallucinations

A Practical Index to Reliable and Relevant RAG Systems

After building RAG systems for dozens of organisations, we have learned this: hallucinations are rarely random. They happen at predictable points in your pipeline. The symptoms are familiar — inconsistent or irrelevant answers, and sometimes information being invented entirely. The typical response is to experiment with better embeddings, adjust chunking strategies, or upgrade to the latest model.

But what appears to be hallucination often stems not from the model itself. Instead, it is a visible signal that something has broken at a specific stage of your RAG pipeline. This article maps every stage where accuracy and relevance can break down — from corrupted source data to blind evaluation — and shows you how to diagnose and fix each failure mode systematically.

How RAG actually works: a five-stage journey

Think of RAG like a research assistant working through your company's knowledge base. Each stage is a potential failure point where hallucinations can creep in. Not every RAG system follows these exact five stages — your implementation may combine or split them differently. But this framework helps you recognise where things typically go wrong.

The five-stage RAG pipeline

Stage 1 — COLLECT YOUR DATA

Gather documents from all sources · Clean and organise content · Make data readable

Stage 2 — PREPARE FOR SEARCH

Break documents into smaller pieces · Add labels and metadata · Create searchable index

Stage 3 — FIND RELEVANT INFORMATION

Convert query into search format · Search indexed data · Rank and select best matches

Stage 4 — GENERATE THE ANSWER

Feed relevant info to the AI · Generate response from your data · Ensure answer stays grounded

Stage 5 — VERIFY & IMPROVE

How to use this appendix

The following pages walk through each stage in detail, covering the key failure modes, concrete examples, consequences, and mitigations. Feel free to read straight through or jump directly to whichever stage is causing you trouble right now.

Stage overview

Stage 1 — Data Extraction & Ingestion. Successfully extracting data does not mean it is in a form that supports meaningful retrieval. Even perfectly accessible content may be corrupted, fragmented, inconsistent, or encoded in ways that embeddings cannot understand.

Stage 2 — Data Structuring & Preparation. This stage transforms clean data into retrieval-ready content — and every structural decision has consequences. Structure your data poorly here and relevant information becomes effectively invisible to search, even though it exists in your knowledge base.

Stage 3 — Retrieval Layer. No model can provide an answer it cannot see. Poor retrieval makes correct information effectively invisible: semantic mismatches miss specialised terminology, outdated indexes return stale facts, and misclassified intent searches for the wrong documents entirely.

Stage 4 — Generation & Alignment. Even with perfect retrieval, generation can fail. Language models are trained to complete text plausibly, not necessarily accurately. Retrieval and generation can each work perfectly in isolation but fail when combined.

Stage 5 — Evaluation & Feedback. If you cannot measure hallucinations, you cannot fix them. Evaluation is often the weakest link. Without proper evaluation, errors accumulate silently, biases go undetected, and subtle failures compound until user trust collapses.

From stages to systems

> *Many accuracy problems do not respect stage boundaries. Temporal conflicts span ingestion, retrieval, and generation. Garbage data at Stage 1 cascades through every downstream component, passing faithfulness checks while being fundamentally wrong. Intent mismatches require coordination between retrieval and generation. Multi-hop reasoning failures emerge from the interaction between how you index, what you retrieve, and how the model synthesises information.*

Stage 1: Data Extraction & Ingestion

Accessible does not mean retrievable

Successfully extracting data does not mean it is in a form that supports meaningful retrieval. Even perfectly accessible content may be corrupted, fragmented, inconsistent, or encoded in ways that embeddings cannot understand. These are universal problems that occur regardless of source type.

What happens here:

Failure Mode	Description	Example	Consequence	Mitigation
Data quality & reliability	Extraction errors, missing fields, corrupted content	OCR misreads "$5M revenue" as "$55M revenue"	Incorrect data propagates through pipeline; missing info forces guessing	Confidence scores, sanity checks, spot-checks on extraction output
Format inconsistencies	Same information in different formats across sources	Dates: 2025-10-10 vs 10/10/2025 vs October 10, 2025	String comparisons fail, semantic search struggles, missed duplicates	Standardise to ISO 8601, normalise currency and units
Encoding issues	Character encoding problems; special characters mangled	Smart quotes become "???", non-Latin text unreadable	Text becomes semantically different; search fails; degraded embeddings	UTF-8 enforcement, encoding detection and conversion
Duplication & redundancy	Same content extracted multiple times; versioned docs without distinction	notes.pdf, notes-final.pdf, notes-final-v2.pdf all indexed	Redundant results dilute relevance; conflicting versions confuse users	Deduplication via hashing, version metadata (supersedes-id)
Structure loss	Tables, lists, hierarchies flattened during extraction	Pricing table becomes "Basic 10 Pro 20 Enterprise 50"	Embeddings miss relationships; table data becomes nonsensical	Preserve as markdown tables or JSON; maintain formatting
Information fragmentation	Context split across locations, requires joining pieces	Web article split across 5 pages; DB customer-orders relationship lost	Incomplete chunks, unclear relationships, failed cross-references	Link resolution for pagination, DB denormalisation, parent-child chunks
Temporal & versioning	Multiple versions without clear indicators; conflicting timestamps	Employee handbook 2020, 2023, 2025 all indexed without date markers	Mixing outdated with current info; contradictory answers	Extract created-at, valid-from, version-number, deprecation flags

Stage 2: Data Structuring & Preparation

From clean data to retrieval-ready chunks

This stage transforms clean data into retrieval-ready content and every structural decision has consequences. Arbitrary chunking breaks context mid-thought. Missing metadata eliminates your ability to filter by date, version, or category. Generic embeddings trained on web text struggle to capture specialised domain terminology. Structure your data poorly here and relevant information becomes effectively invisible to search.

What happens here:

Failure Mode	Description	Example	Consequence	Mitigation
Arbitrary chunking	Token-count or character-based splitting breaks logical units mid-sentence	"The company DID NOT meet targets" where "DID NOT" separates from "meet targets"	Loss of meaning, contradictory fragments, incomplete context	Semantic chunking based on structure; respect boundaries; 10–20% overlap
Structure flattening	Tables, lists, hierarchies converted to plain text lose meaning	Pricing table becomes "Basic 10 Pro 20 Enterprise 50"	Embeddings miss structured relationships; retrieval fails comparisons	Preserve as markdown tables or JSON; use LLM-based structured extraction
Missing metadata	No timestamps, source attribution, document types, or version control	Customer policy from 2020 and 2025 both indexed without date markers	Mixing outdated with current info; temporal conflicts; cannot filter	Systematic metadata extraction: dates, entities, categories, version info
Stale / duplicate content	Outdated documents not refreshed; same content indexed multiple times	Product specs from 2023 and 2025 models without distinction	Retrieval returns irrelevant historical versions; noise dilutes relevance	Version control with created-at, updated-at, valid-until; fuzzy deduplication
Generic embeddings	General-purpose models on domain-specific content; trained on web text	Medical retrieval using Wikipedia-trained embeddings	Semantic similarity fails for jargon; domain concepts collapse incorrectly	Domain-specific or fine-tuned models; validate on domain vocabulary; hybrid search
Model-data mismatch	Embedding dimensions, tokenisation not aligned with data characteristics	Multilingual data with English-only embeddings; code with natural language model	Poor retrieval quality; semantic mismatch; content "invisible" to search	Match model to content type: multilingual for mixed-language, code-aware for technical
Dimension trade-offs	Choosing embedding size without considering performance/cost balance	Using 1536-dim embeddings when 384-dim would suffice for the use case	4× higher storage/compute costs with marginal quality gains; slower retrieval	Profile 384/768/1536 dimensions against your data; measure recall vs. latency/cost

Stage 3: Retrieval Layer

When relevant documents go unretrieved

No model can provide an answer that it cannot see. When retrieval fails, even the most capable language model must choose between refusing to answer, inventing information from its training data, or piecing together responses from fragments that miss the point. Context quality determines answer quality.

What happens here:

Failure Mode	Description	Example	Consequence	Mitigation
Low recall / irrelevant retrieval	System fails to fetch relevant documents due to poor embeddings or vocabulary mismatch	"Who designed Datapoint 2200?" returns documents about Kenbak-1	Missing context forces model to guess, refuse, or hallucinate from training knowledge	Hybrid search (BM25 + embeddings), query expansion/rewriting, increase top-k, evaluate recall
Low top-k context	Not enough passages retrieved to capture complete information	Top-3 results omit the correct answer buried at position 7	Incomplete context leads to partial answers, hedging, or fabrication to fill gaps	Adaptive k based on query complexity; reranking before truncation; test with multi-chunk answers
Index staleness	Knowledge base does not reflect current information; changes not propagated	Index says "CEO = Jane Doe" though it is now "John Smith"; discontinued product still appears	System confidently returns wrong but once-correct information; temporal conflicts	Scheduled reindexing, change detection, timestamp-based filtering, recency weighting
Embedding drift	Embedding model does not capture specialised domain vocabulary or semantics	Legal term "estoppel" collapses with unrelated concepts in general-purpose embeddings	Semantic search misses relevant documents despite exact vocabulary matches in source	Domain-specific embeddings, vocabulary coverage analysis, hybrid search, fine-tuning
Intent misclassification	System misinterprets what user actually needs	"Apple earnings 2024" retrieves fruit harvest data instead of financial reports	Correct retrieval impossible when intent is wrong from the start	Intent classification layer, query preprocessing with entity recognition, domain routing
Weak hybrid search	Improper balance between keyword and semantic search components	Pure semantic search blurs "meeting on 2025-10-10" into "meetings in October"	Retrieved chunks semantically close but factually wrong on precise details	Dynamic query-aware balancing; boost keywords for exact terms; BM25 + embeddings + cross-encoders
Poor metadata filtering	Available metadata not used effectively during retrieval	Documents have valid-from/valid-until fields but retrieval ignores them	Wrong versions retrieved; irrelevant categories pollute results	Pre-filtering by metadata before vector search; temporal range queries; category-based partitioning

Stage 4: Generation & Alignment Layer

When components work in isolation but fail together

Even with perfect retrieval, generation can fail. Language models are trained to complete text plausibly, not necessarily accurately. Retrieval and generation can each work perfectly in isolation but fail when combined — misalignment creates subtle errors that are hard to detect because both subsystems appear functional.

What happens here:

Failure Mode	Description	Example	Consequence	Mitigation
Loose grounding	Prompt lacks strict instruction to stay within retrieved context	Model adds "IBM PC 5150 specs" not in retrieved text	Blends retrieval facts with hallucinations; plausible but unverifiable	Explicit "Answer ONLY from context"; require citations; negative examples in prompt
Gap-filling hallucinations	Model fills gaps using prior knowledge when context is incomplete	Invents "1971 release year" because it sounds right given surrounding context	Plausible but false details that are hard to detect without source tracing	Temperature 0–0.1 for determinism; structured output with source IDs; allow "unknown"
Context overload / dilution	Too many or too long chunks exceed the model's effective attention span	"HP-9830A" and "Wang 2200" merged incorrectly due to crowded context	Crucial information ignored or mixed; key facts lost in noise	Filter and rerank top chunks; compress context; maintain priority ordering
Retriever–generator mismatch	Embeddings differ, causing near-miss but wrong retrievals fed to generator	Context semantically close yet factually wrong; misleads generation	Generator builds answer on misleading text while appearing grounded	Use compatible models; validate retrieval-generation alignment; monitor quality
Prompt↔chunk format mismatch	Generator receives truncated or fragmented text without full context	Reads "He…invented Datapoint 2200" as a new fact due to truncation	Misattribution or false links; model fills in gaps incorrectly	Ensure full-sentence chunks; add headers; validate chunk boundaries
Conflicting evidence	Retrieved docs disagree with no conflict handling logic	One source says "launched 2023," another says "2024"	Model picks or blends arbitrarily; inconsistent answers across sessions	Source weighting; flag contradictions explicitly; prefer newer/credible data
Cross-document reasoning	Model fuses entities or facts from different sources incorrectly	Combines drug A from Patient 1 with disease from Patient 2	False associations in sensitive domains; dangerous in medical or legal contexts	Track entities across chunks; graph-based reasoning; conservative synthesis
Context position bias	Model attention favours beginning/end of context window ("lost in the middle")	Critical fact in middle of 10 chunks gets overlooked despite correct retrieval	Key information overlooked even when successfully retrieved	Place important chunks strategically; apply lost-in-middle mitigation techniques
Missing few-shot grounding	Prompt lacks examples of proper source attribution and citation format	Model does not understand expected citation format without examples	Inconsistent or missing source references; unreliable attribution	Include 1–2 examples of properly grounded answers in the system prompt

Stage 5: Evaluation & Feedback

The blind spot problem

If you cannot measure hallucinations, you cannot fix them. Evaluation is often the weakest link, with teams measuring what is easy (fluency, speed, user satisfaction) instead of what matters (accuracy, faithfulness, factual consistency). This is not a one-time checkpoint — it is an iterative cycle: measure performance, identify failure patterns, adjust your pipeline, and measure again.

What happens here:

Failure Mode	Description	Example	Consequence	Mitigation
Fluency over accuracy	Metrics focus on readability, coherence, satisfaction rather than factual correctness	Answer eloquent and confidently cited but contains invented statistics or wrong dates	High scores for hallucinated content; "looks good" passes testing	Fact-checking metrics (RAGAS faithfulness, TruLens groundedness); adversarial testing
Incomplete evaluation	"Looks plausible" ≠ "factually correct" without grounded metrics	Only measuring BLEU, ROUGE, or perplexity — metrics rewarding fluency, not accuracy	No signal when system fabricates information; false sense of quality	Grounded metrics: answer-context overlap (NLI-based), citation accuracy, factual consistency
No source traceability	Cannot verify which retrieved chunk led to which part of the answer	Answer claims revenue figures but audit trail shows no supporting chunk	Hallucinations undetected; no debugging path; compliance and legal risk	Structured logging of retrieval-generation mapping; citation validation; provenance tracking
No human validation	Automated metrics alone; no domain expert review of representative outputs	Medical RAG evaluated only by BLEU/ROUGE without clinician review	Critical errors missed: wrong dosage, incorrect legal precedent application	Expert review cycles on samples; adversarial red-teaming by domain specialists
No feedback mechanism	System deployed without ongoing monitoring; errors accumulate over time	User complaints about wrong answers but no logging or retraining pipeline	Repeated hallucinations on same topics; degrading trust; no learning from mistakes	Active learning from corrections; regular evaluation on held-out test sets; drift detection
Confidence calibration	System provides no uncertainty signal or is overconfident about wrong answers	RAG answers "Capital of Australia is Sydney" with 98% confidence (correct: Canberra)	Users cannot distinguish reliable from unreliable outputs	Retrieval confidence scores; answer-source similarity checks; explicit "I do not know" responses
Happy path testing only	Evaluation ignores edge cases and failure modes; focuses on queries system should answer	Test set contains well-formed queries with known answers in knowledge base	System appears to work until real users ask ambiguous or adversarial queries	Adversarial test sets: ambiguous queries, contradictory evidence, out-of-knowledge questions
No A/B testing	Changes deployed without controlled comparison against baseline	New retrieval strategy seems better in dev but degrades in production	Performance regression goes unnoticed; degraded user experience	Shadow mode testing; gradual rollout with metrics comparison
Missing cost tracking	Evaluation ignores economic factors alongside accuracy metrics	System achieves 95% accuracy but costs $5 per query at scale	Economically unviable despite technical success	Track cost per query, accuracy per dollar, latency-cost trade-offs

Sources & References

Major AI blogs and practitioner guides were referenced, including NVIDIA's technical blog on agentic RAG, industry analyses from Qodo.ai and Bitcot, as well as community knowledge from AI frameworks like PydanticAI, LangChain, and LlamaIndex. These sources reflect the latest developments (late 2023–2025) in retrieval-augmented AI system design, ensuring an up-to-date comparison.

Agentic RAG

Agentic RAG: Building a coding agent (no frameworks), ep #28
https://www.youtube.com/watch?v=grGSFfyejA0

Is Agentic RAG worth it? An experimental comparison of RAG approaches
https://arxiv.org/pdf/2601.07711

Traditional RAG vs. Agentic RAG — Why AI Agents Need Dynamic Knowledge to Get Smarter
NVIDIA Technical Blog
https://developer.nvidia.com/blog/traditional-rag-vs-agentic-rag-why-ai-agents-need-dynamic-knowledge-to-get-smarter/

What is Agentic RAG?
https://www.youtube.com/watch?v=0z9_MhcYvcY

Information Retrieval & Search

Keyword Search — OpenSearch Documentation
https://docs.opensearch.org/latest/search-plugins/keyword-search/

What is BM25 (Best Matching 25) Algorithm? — GeeksforGeeks
https://www.geeksforgeeks.org/nlp/what-is-bm25-best-matching-25-algorithm/

What is Semantic Search? — Elastic
https://www.elastic.co/what-is/semantic-search

Hybrid Search — Supabase Documentation
https://supabase.com/docs/guides/ai/hybrid-search