International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 4, July – August 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

The Evolution of Retrieval-Augmented Generation: From Traditional Information Retrieval to Intelligent Multi-Stage Retrieval Systems

Varun Reddy Maram[1]
Keshav Memorial Institute of Technology, Narayanaguda, Hyderabad, India[1]
varunreddymaram29@gmail.com

Nandan Reddy Maram[2]|
Manipal Institute Of Technology, Eshwar Nagar, Manipal, Karnataka, India[2]
nandanreddy886@gmail.com

----------**********----------

Abstract:

Retrieval used to mean one thing: match the words in a query against the words in a document, count how often they overlap, and rank accordingly. That approach carried search engines and enterprise document systems for the better part of four decades, and it still runs underneath a surprising number of production systems today. What has changed is what sits on top of it. Once large language models became capable enough to synthesize answers rather than just list documents, the retrieval problem quietly turned into a different problem: not "which documents contain these words" but "which passages, out of millions, actually let a model answer this question correctly." That shift in the question being asked is what has driven the last decade of retrieval research, and it is the throughline this paper follows. We trace the path from Boolean and TF-IDF retrieval through BM25, into dense embedding-based semantic search, approximate nearest neighbor indexing, hybrid lexical-semantic fusion, cross-encoder reranking, and finally into the multi-stage, permission-aware retrieval pipelines that sit inside modern Retrieval-Augmented Generation (RAG) systems. Rather than treating each of these as an isolated technique, we look at why each one appeared when it did, which failure of its predecessor it was responding to, and what new problem it introduced in the process — because none of these transitions were free. We also present a generalized architecture, distilled from patterns observed across enterprise-grade retrieval deployments, that separates ingestion, indexing, retrieval, reranking, and generation into layers that can be reasoned about independently. Along the way we discuss chunking strategy, embedding dimensionality, index structures such as HNSW and IVF, similarity functions, metadata filtering, and the evaluation metrics — Recall@K, MRR, nDCG — that let teams actually measure whether a retrieval change helped or hurt. The paper closes by looking at where the field appears to be heading: adaptive, query-dependent retrieval strategies and agentic systems that decide for themselves how much retrieval a question actually needs.

Keywords — Retrieval-Augmented Generation, Information Retrieval, Dense Embeddings, Semantic Search, Vector Databases, Approximate Nearest Neighbor Search, Hybrid Retrieval, Reranking, Cross-Encoders, Large Language Models

I. I. INTRODUCTION

Ask someone who has worked with search systems for more than a few years what "retrieval" means, and you will get a different answer depending on when they started. To someone who built enterprise search in the 2000s, retrieval means an inverted index, a ranking function, and a lot of tuning around stopwords and stemming. To someone who joined a machine learning team in 2022, retrieval more likely means a vector database, an embedding model, and a call to a similarity search API. Both are describing the same underlying task — given a query, find the most relevant pieces of text out of a large collection — but the tools, the assumptions, and the failure modes are almost unrecognizable from one era to the next.

This paper is an attempt to trace that path honestly, including the parts that do not flatter the newer techniques. Dense embeddings did not simply replace lexical search; they solved a specific problem (vocabulary mismatch) while introducing a new one (numerical hallucination of similarity where none really exists, particularly for rare terms, identifiers, and exact phrase matches). Vector databases did not make search "smart"; they made approximate similarity search fast enough to run at scale, which is a different achievement. Reranking did not fix retrieval; it added a second, more expensive stage that trades latency for precision. Every step in this history is a trade-off, and understanding the trade-off is more useful to a practitioner than knowing the name of the technique.

The immediate trigger for renewed interest in retrieval is, of course, the large language model. LLMs are fluent but not reliable narrators of fact — they will produce confident, well-formed sentences that are simply wrong, a failure mode usually called hallucination. Retrieval-Augmented Generation, first formalized by Lewis et al. [1], addresses this by grounding generation in retrieved text: instead of asking a model to recall facts from its parameters, you hand it the relevant passage and ask it to reason over that passage instead. This reframes the retrieval problem once more. It is no longer enough for a retrieved document to be topically related to the query — it needs to contain the specific fact, number, or clause the model needs, packaged into a chunk small enough to fit a context window without diluting it with irrelevant text. That requirement is what has pushed retrieval engineering from a single-stage lookup into the multi-stage pipelines discussed in this paper.

The remainder of the paper is organized as follows. Section II gives background on the retrieval problem and the evaluation metrics used throughout. Section III walks through the historical evolution of retrieval methods in roughly chronological order. Section IV presents a generalized architecture for modern multi-stage RAG systems, distilled from patterns common to production-grade retrieval deployments. Section V discusses the retrieval pipeline in more operational detail — ingestion, chunking, indexing, and reranking. Section VI covers performance and scalability considerations. Section VII discusses limitations that current architectures still have not solved. Section VIII looks at where the field seems to be heading, and Section IX concludes.

II. II. BACKGROUND AND EVALUATION

A. A. What Retrieval Is Actually Being Asked to Do

At its core, retrieval is a ranking problem: given a query q and a corpus of documents D = {d1, d2, ..., dn}, produce an ordering of D such that the documents most relevant to q appear first. Almost every technique discussed in this paper is a different answer to the same underlying question — how do you define and compute "relevant"? Lexical methods define it in terms of shared vocabulary. Dense methods define it in terms of proximity in a learned embedding space. Hybrid methods refuse to pick one definition and combine both. Rerankers define it with a model expensive enough to look at the query and document jointly rather than independently.

B. B. Evaluation Metrics

Comparing retrieval methods requires metrics that capture both whether relevant items were found and how well they were ranked. Precision measures the fraction of retrieved items that are relevant, while recall measures the fraction of all relevant items that were successfully retrieved:

Precision = |Relevant ∩ Retrieved| / |Retrieved| (1)

Recall = |Relevant ∩ Retrieved| / |Relevant| (2)

The two pull in opposite directions — retrieving more documents tends to raise recall and lower precision — so the F1 score is used to summarize both as a single number:

F1 = 2 · (Precision · Recall) / (Precision + Recall) (3)

For top-K retrieval specifically, Recall@K restricts the calculation to the first K results, which better reflects how retrieval is actually used in a RAG pipeline, since only the top few chunks are ever passed to the language model. Mean Reciprocal Rank (MRR) captures how early the first relevant result appears:

MRR = (1/|Q|) · Σ (1 / rank_i) (4)

where rank_i is the position of the first relevant document for query i. Normalized Discounted Cumulative Gain (nDCG) goes further by rewarding relevant documents that appear earlier in the ranked list more than those appearing later, and by accounting for graded relevance rather than a binary relevant/not-relevant judgment:

DCG@K = Σ (rel_i / log2(i + 1)), for i = 1 to K (5)

nDCG@K is then DCG@K normalized by the ideal DCG achievable for that query (IDCG@K), producing a score between 0 and 1 that is comparable across queries with different numbers of relevant documents. In production RAG systems, these ranking metrics are almost always reported alongside latency percentiles (p50, p95) and end-to-end answer quality metrics such as faithfulness (how well the generated answer is grounded in the retrieved context) and context recall (whether the retrieved set actually contained the information needed to answer correctly), since a retrieval pipeline that scores well on Recall@K but retrieves the right information padded with distracting text can still produce a poorly grounded answer.

III. III. EVOLUTION OF RETRIEVAL SYSTEMS

C. A. Boolean and Term-Frequency Retrieval

The earliest production retrieval systems worked on pure Boolean logic: a document either contained a query term or it did not, and queries were expressed with AND, OR, and NOT operators over an inverted index mapping terms to the documents (postings) that contained them. This was fast and predictable, but it had no concept of relevance ranking — a document mentioning a term once and a document built entirely around that term were treated identically as long as both matched.

Term Frequency-Inverse Document Frequency (TF-IDF) fixed the ranking problem by scoring how important a term is to a specific document relative to the whole corpus. A term that appears often in one document but rarely elsewhere is a strong signal of what that document is about; a term that appears everywhere (like "the" or "system") tells you almost nothing and should be down-weighted:

TF-IDF(t, d) = tf(t, d) · log(N / df(t)) (6)

where tf(t, d) is the frequency of term t in document d, N is the total number of documents, and df(t) is the number of documents containing t. This was a genuine improvement, but it still treated every occurrence of a term as equally valuable regardless of document length, and it had no way of saturating the score for terms that appear an unreasonable number of times (a document that repeats a keyword fifty times is not necessarily fifty times more relevant than one that mentions it once).

D. B. BM25 and the Peak of Lexical Retrieval

BM25 (Best Matching 25) addressed both weaknesses and remains, decades later, a surprisingly strong baseline — strong enough that most hybrid retrieval systems still keep it around as one half of the search.

BM25 introduces term-frequency saturation (diminishing returns for repeated terms) and length normalization (penalizing documents that are relevant only because they are long):

$$BM25(q, d) = \Sigma\ IDF(qi) \cdot [\ f(qi, d) \cdot (k1 + 1)\ ] / [\ f(qi, d) + k1 \cdot (1 - b + b \cdot |d| / avgdl)\ ] \quad (7)$$

Here f(qi, d) is the frequency of query term qi in d, |d| is the document length, avgdl is the average document length across the corpus, and k1 and b are tunable parameters controlling term-frequency saturation and length normalization respectively. In practice, k1 is usually set somewhere between 1.2 and 2.0 and b around 0.75, though these are worth tuning against a labeled evaluation set rather than trusting the defaults blindly — we have seen retrieval quality shift noticeably just from adjusting b on a corpus with unusually long or short documents.

The reason BM25 refuses to go away is that it does something dense retrieval genuinely struggles with: exact matching on rare, specific tokens. Product codes, error messages, part numbers, legal citation formats, acronyms — anything where the literal string matters more than the semantic gist — tends to be retrieved more reliably by a lexical method than by an embedding model, which was trained to generalize past exact wording, not preserve it.

E. C. The Vocabulary Mismatch Problem

The limitation that eventually motivated a different approach entirely is vocabulary mismatch: a user asking "how do I get my money back" and a document titled "refund policy and eligibility" share almost no words, yet a human reader immediately sees they are related. Lexical methods, no matter how well-tuned, cannot bridge this gap because they operate purely on surface-level token overlap. Query expansion techniques (adding synonyms, stemming aggressively) were an attempt to patch this within the lexical framework, but they are brittle and language-specific, and they tend to introduce as much noise as they resolve.

F. D. Dense Embeddings and Semantic Search

The alternative that eventually won out was to stop matching words and start matching meaning. Sentence and passage embedding models, built on transformer encoders such as those in the Sentence-BERT line of work [8], map a piece of text to a fixed-length dense vector such that semantically similar text ends up close together in that vector space, regardless of whether the words themselves overlap. "How do I get my money back" and "refund eligibility" can end up neighbors in embedding space even though they share zero content words.

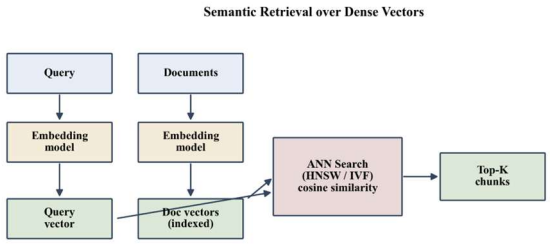

Fig. 1. Semantic retrieval pipeline: queries and documents are mapped into the same dense vector space by an embedding model, and relevance is computed via approximate nearest neighbor search over that space.

Given two embeddings, similarity is most often computed with cosine similarity, which measures the angle between vectors rather than their magnitude — useful because it makes the score insensitive to document length effects that crept into raw dot products:

$$\cos(\theta) = (A \cdot B) / (\|A\|\ \|B\|) \quad (8)$$

Some systems use raw dot product instead, particularly when the embedding model was trained with a dot-product objective and vector norms already carry meaningful information (longer, more detailed passages sometimes end up with larger norms, which dot product will reward and cosine similarity will normalize away — whether that is desirable depends on the corpus). Euclidean distance is used less often for text retrieval but remains common in other embedding domains:

$$d(A, B) = \sqrt{(\Sigma (Ai - Bi)^2)} \quad (9)$$

The choice of similarity function is not a cosmetic detail — it needs to match the objective the embedding model was actually trained under, or relevance rankings will be systematically distorted in ways that are hard to notice from spot-checking a handful of queries.

Dense retrieval is not free of trade-offs. Embeddings compress a passage into a fixed number of dimensions, and that compression is inherently lossy — an embedding model has to decide what to preserve and what to discard, and it will preserve topical similarity far more reliably than it preserves exact numbers, dates, or identifiers. This is why production systems rarely rely on dense retrieval alone; the honest framing is that dense retrieval solved vocabulary mismatch at the cost of precision on exact-match queries, which is exactly the failure mode BM25 does not have.

G. E. Approximate Nearest Neighbor Search and Vector Indexing

Finding the nearest vector to a query embedding by brute force means computing similarity against every vector in the corpus, which is linear in corpus size and becomes impractical past a few hundred thousand documents, let alone the tens of millions common in enterprise deployments. Approximate Nearest Neighbor (ANN) search trades a small amount of recall for a large reduction in latency by indexing the vector space so that only a small, promising subset of vectors needs to be examined per query.

1) 1) HNSW

Hierarchical Navigable Small World graphs [6] build a multi-layer graph structure where higher layers contain progressively fewer nodes and act as express lanes for coarse navigation, while the bottom layer contains every vector and supports fine-grained search. A query starts at the top layer, greedily walks toward the nearest node, drops a layer, and repeats, converging on a good answer in roughly logarithmic time relative to corpus size. HNSW tends to offer very good recall for its speed, at the cost of a larger memory footprint than some alternatives, since the graph structure itself has to be kept in memory alongside the vectors.

2) 2) IVF

Inverted File indexes take a different approach: they cluster the vector space into a fixed number of partitions (via something like k-means), and at query time only the handful of partitions closest to the query vector are searched, rather than the entire index. IVF indexes are more memory-efficient than graph-based approaches and scale well with quantization (compressing vectors to reduce their size, at some further cost to precision), which is why they show up frequently in very large-scale deployments where memory, not latency, is the binding constraint. FAISS [5] popularized much of the engineering around IVF and product quantization at scale.

Neither index type is strictly better — the choice is a genuine engineering trade-off between memory, latency, recall, and index build time, and the right answer changes depending on corpus size and how often the index needs to be rebuilt as new documents arrive.

H. F. Vector Databases and Metadata Filtering

As dense retrieval moved from research prototypes into production, a new category of infrastructure emerged around it: purpose-built vector databases that combine ANN indexing with the operational concerns any real system needs — persistence, replication, incremental updates, and, critically, metadata

filtering. A pure vector search has no native concept of "only search documents from this tenant" or "only search documents updated in the last month"; metadata filtering bolts structured constraints onto the similarity search so that the ANN index only considers vectors that also satisfy a filter predicate. This turns out to matter enormously in multi-tenant or access-controlled systems, where retrieving the semantically closest document is worthless — or actively a security problem — if that document belongs to a different tenant or the requesting user is not permitted to see it.

I. G. Hybrid Retrieval

Given that lexical and dense retrieval fail in complementary ways — lexical struggles with vocabulary mismatch, dense struggles with exact-match precision — the natural next step was to stop choosing between them. Hybrid retrieval runs both searches independently and merges their results, typically using a rank-fusion technique such as Reciprocal Rank Fusion, which combines rankings without needing the two systems' raw scores to be on comparable scales (a real problem, since a BM25 score and a cosine similarity score are not directly comparable numbers):

$$RRFscore(d) = \Sigma\ 1 / (k + rank\_s(d)) \quad (10)$$

summed over each retrieval system s that returned document d, where rank_s(d) is d's rank in that system's result list and k is a small constant (commonly 60) that dampens the influence of very high ranks. In practice, hybrid retrieval is one of the highest-leverage changes a team can make to an existing dense-only pipeline, because it recovers exact-match precision without giving up any of the semantic recall gains dense retrieval already provided.

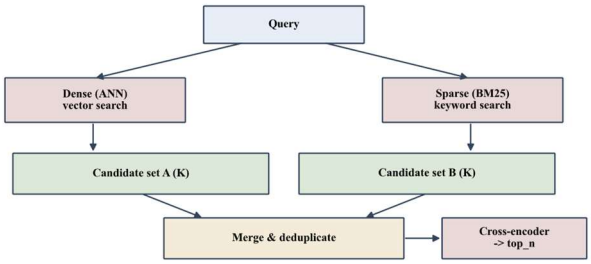

Fig. 2. Hybrid retrieval: dense and sparse search run in parallel over the same query, and their candidate sets are merged before being handed to a reranking stage.

J. H. Cross-Encoder Reranking

Both lexical and dense retrieval share a structural limitation: they score documents independently of the query at index time (for dense retrieval) or with a relatively shallow scoring function (for lexical retrieval), which is what makes them fast enough to search millions of documents. Cross-encoders give up that speed in exchange for accuracy — they take the query and a candidate document together as a single input and let a transformer model attend across both simultaneously, producing a relevance score that captures interactions no independent embedding comparison can.

Because this joint scoring is too expensive to run against an entire corpus, cross-encoders are used as a second stage: an initial retriever (dense, lexical, or hybrid) pulls back a modest candidate set — commonly somewhere between 20 and 100 documents — and the cross-encoder reranks only that shortlist down to the handful actually passed to the language model. This retrieve-then-rerank pattern is, in our experience, the single change most likely to move retrieval quality metrics noticeably in a production system, precisely because it corrects errors introduced at every earlier stage rather than depending entirely on getting the first-stage retrieval right.

K. I. Retrieval-Augmented Generation

RAG closes the loop by feeding the reranked context into a language model alongside the original query, typically as part of a structured prompt that instructs the model to answer using the provided context and, ideally, to say so explicitly when the context does not contain an answer rather than falling back on its own possibly-inaccurate training knowledge. The quality of a RAG system's output is bottlenecked by whichever stage is weakest — a perfect generation model fed irrelevant context will still produce a wrong or unfounded answer, and a perfect retrieval stage feeding a model that ignores its context will still hallucinate. This is why RAG evaluation increasingly separates retrieval-quality metrics from generation-quality metrics (faithfulness, in particular) rather than judging the system only on final answer correctness, which conflates the two.

IV. IV. A GENERALIZED MULTI-STAGE RAG ARCHITECTURE

Pulling the preceding techniques together, most production-grade RAG systems converge on a similar shape, regardless of the specific vendor or open-source components used underneath. It is useful to describe this shape in terms of layers, since each layer can be reasoned about, scaled, and replaced somewhat independently of the others.

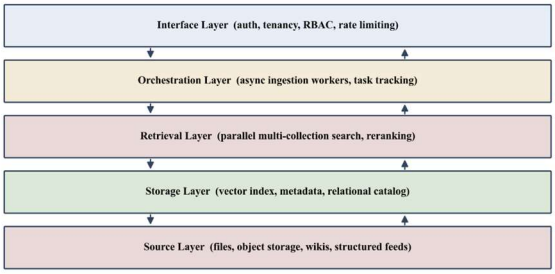

Fig. 3. Layered view of a generalized enterprise RAG architecture. Each layer can be scaled and replaced independently, and the same retrieval and storage layers typically serve many downstream applications.

L. A. Source Layer

The lowest layer is simply where content lives before it enters the system — files, object storage, internal wikis, structured data feeds. A design decision worth making explicitly here is whether ingestion pulls from these sources on a schedule or reacts to change events; polling is simpler to reason about but introduces staleness, while event-driven ingestion is more responsive but couples the RAG system to the reliability of upstream change notifications.

M. B. Storage Layer

Above the source layer sits persistent storage: the vector index itself, a metadata store (often a conventional relational database, since metadata filtering, access control, and collection bookkeeping are relational problems even when the payload being filtered is a vector), and a catalog describing what collections exist and which embedding model each one was built with. This last point matters more than it sounds — mixing vectors from two different embedding models inside a single index silently corrupts similarity search, since the two models' vector spaces are not comparable, so collections need to be bound to a specific model at creation time and kept that way.

N. C. Retrieval Layer

This layer implements the actual multi-stage retrieval discussed in Section III: query understanding, optional metadata filtering, first-stage candidate retrieval (dense, lexical, or hybrid), and reranking. A detail that matters for latency at scale: when a query needs to search multiple collections or multiple retrieval methods, those searches should run concurrently rather than sequentially, since the total latency

then equals the slowest single search rather than the sum of all of them. This sounds obvious in principle but is easy to get wrong in practice if a system grows organically from supporting one collection to supporting many.

O. D. Orchestration Layer

Ingestion — extracting text, cleaning it, chunking it, embedding it, and writing it into the index — is comparatively slow and should not block whatever triggered it. Asynchronous task workers with status tracking let ingestion run in the background while immediately returning a task identifier the caller can poll, which is a much better experience than holding a request open for however long it takes to process a large document set. This layer also often handles atomic collection replacement patterns — building a new version of a collection in the background and swapping it in only once ingestion succeeds fully, so that a partially-completed refresh never becomes visible to queries.

P. E. Interface Layer

The topmost layer handles authentication, tenancy, role-based access, and rate limiting. In any system serving more than one team or customer, this layer is not an afterthought — a retrieval system that returns the semantically best-matching document regardless of who is asking is a data leak waiting to happen. Permission checks belong as close to the retrieval layer as possible, ideally applied as filters within the search itself rather than as a post-hoc step that discards results after they have already been computed, since the latter wastes compute and can leak information through timing or result-count side channels.

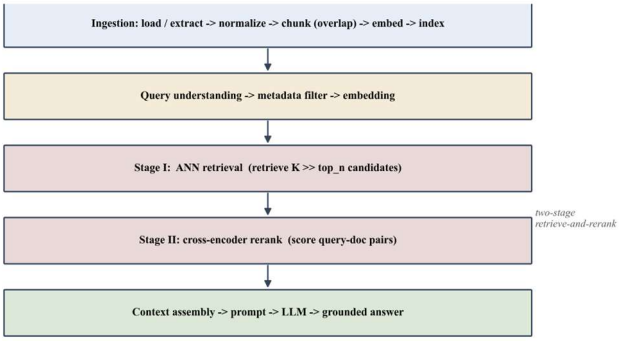

Fig. 4. Complete multi-stage retrieval pipeline: ingestion builds the index offline, while query time executes a two-stage retrieve-and-rerank sequence before context assembly and generation.

V. V. RETRIEVAL PIPELINE IN DETAIL

Q. A. Ingestion and Preprocessing

Raw source documents rarely arrive in a form directly usable by an embedding model. Ingestion typically involves extracting text from whatever container format the source used (documents, spreadsheets, presentations, markup), normalizing whitespace and encoding, and stripping structural noise (headers, footers, boilerplate) that would otherwise pollute every chunk derived from that document. Text normalization decisions made here — whether to preserve or collapse formatting, whether to keep tables as structured text or flatten them — have downstream consequences for retrieval quality that are easy to underestimate during initial implementation.

R. B. Chunking Strategy

Documents must be split into chunks before embedding, since embedding models have fixed input length limits and, more importantly, because a single embedding vector for an entire long document dilutes any specific fact into an average representation of the whole document's topic. Chunk size is a

genuine trade-off: chunks that are too small lose surrounding context needed to interpret them correctly, while chunks that are too large dilute the embedding and waste context-window space once retrieved. Overlap between consecutive chunks — repeating a small amount of text at each chunk boundary — mitigates the risk of a fact being split awkwardly across two chunks such that neither one, in isolation, actually answers the question a retriever needs it to answer.

There is no universally correct chunk size; the right value depends on the density of the source text and the kinds of questions the system needs to answer. Dense reference material with short, self-contained facts (specifications, glossaries) tolerates smaller chunks well, while narrative or procedural text often needs larger chunks to preserve the step-by-step logic that makes it useful.

S. C. Embedding Generation

Once chunked, each piece of text is passed through an embedding model to produce a fixed-dimensionality dense vector. Dimensionality itself is a trade-off: higher-dimensional embeddings can encode more nuance but cost more to store and search, and the marginal retrieval-quality benefit of going from, say, 768 to 1536 dimensions is often smaller than the corresponding increase in index size and search latency would suggest is worthwhile. Embedding quality is arguably the single highest-leverage component in the entire pipeline — a weak embedding model puts a ceiling on retrieval quality that no amount of reranking or clever indexing downstream can fully recover from, since reranking can only reorder candidates the first stage actually retrieved.

T. D. Indexing

Embeddings are written into an ANN index (Section III-E) alongside their associated metadata. Index build strategy interacts with the ingestion pattern chosen upstream: systems that ingest continuously need indexes that support efficient incremental updates, while systems that refresh large collections in bulk can afford to rebuild an index from scratch periodically, which sometimes yields a more optimally structured index than one built up through many small incremental insertions.

U. E. Query Understanding and Retrieval

At query time, the incoming query is itself embedded using the same model used for the corpus (a mismatch here silently breaks similarity search), any applicable metadata filters are applied, and the ANN index returns a candidate set considerably larger than what will ultimately be shown to the language model, since this candidate set still needs to survive reranking.

V. F. Reranking and Context Assembly

The candidate set is scored by a cross-encoder and truncated to the final top_n results, which are then assembled into a context block for the language model. Context assembly is not merely concatenation — chunk ordering, deduplication of near-identical chunks pulled from different sources, and clear delimiting of where one source ends and another begins all measurably affect how well the model actually uses the provided context rather than ignoring parts of it.

W. G. Caching

Because embedding generation and reranking are the most computationally expensive steps in the pipeline, caching at multiple levels pays off disproportionately: caching query embeddings for repeated or similar queries, caching retrieval results for identical queries within a short window, and caching generation results where the same question is asked frequently. None of this is exotic, but it is easy to skip during initial implementation and expensive to retrofit once a system is under real load.

VI. VI. PERFORMANCE AND SCALABILITY CONSIDERATIONS

Latency in a multi-stage pipeline is additive across stages unless stages are deliberately parallelized, which means the total time budget has to be allocated consciously across embedding the query, searching the index, reranking candidates, and generating the final response. Reranking and generation are typically the two most expensive stages, since both involve running a transformer model rather than a comparatively cheap vector similarity computation; retrieval-only latency is usually a small fraction of end-to-end latency once a language model is in the loop, which is worth remembering before over-optimizing the retrieval stage at the expense of ignoring generation.

Scalability introduces its own tension: as a corpus grows, ANN index search time grows sub-linearly (that is the entire point of using an ANN index rather than brute force), but recall for a fixed candidate size K tends to degrade somewhat as the corpus grows, since there are simply more plausible-looking distractor documents for the index to confuse with the true match. This is one of the more understated reasons hybrid retrieval and reranking matter more, not less, as a system scales — the safety margin that a small corpus affords disappears as the corpus grows.

TABLE I
COMPARISON OF RETRIEVAL METHODS

Method	Matching Basis	Handles Vocabulary Mismatch	Handles Exact Match
Boolean / TF-IDF	Term overlap	No	Yes
BM25	Weighted term overlap	No	Yes
Dense embeddings	Learned semantic similarity	Yes	Weak
Hybrid (dense + BM25)	Fused ranking	Yes	Yes
Cross-encoder rerank	Joint query-document scoring	Yes	Yes (2nd stage only)

TABLE II
COMPARISON OF VECTOR INDEXING STRATEGIES

Index Type	Structure	Memory Profile	Best Fit
HNSW	Multi-layer proximity graph	Higher (graph + vectors)	Latency-sensitive, moderate corpus size
IVF (+ PQ)	Clustered partitions, quantized vectors	Lower (compressed)	Very large corpora, memory-constrained
Brute-force / flat	No index, exhaustive scan	Lowest overhead, slowest search	Small corpora, exact recall required

The V1-to-V2 style evolution observed across iterations of enterprise retrieval pipelines is illustrative of the general pattern: moving from a single-stage dense retriever toward an agentic, multi-stage retrieve-and-rerank pipeline tends to raise faithfulness and context recall meaningfully, at the cost of measurably higher retrieval and generation latency and higher per-request token consumption. Whether that trade is worthwhile depends entirely on the application — a customer-facing chat assistant answering compliance-sensitive questions can usually justify the added latency; a low-stakes autocomplete-style feature usually cannot.

TABLE III
EVALUATION METRICS ACROSS THE RETRIEVAL PIPELINE

Metric	What It Measures	Typical Use
Precision@K	Fraction of top-K results that are relevant	First-stage retrieval tuning
Recall@K	Fraction of relevant docs captured in top-K	First-stage retrieval tuning
MRR	How early the first relevant result appears	Single-answer retrieval tasks
nDCG@K	Rank-weighted relevance with graded scores	Multi-relevance ranking quality
Faithfulness	How grounded the generated answer is in context	End-to-end RAG evaluation
Context Recall	Whether retrieved context contains the answer	End-to-end RAG evaluation

VII. VII. DISCUSSION AND LIMITATIONS

A pattern worth naming directly: every advance covered in this paper solved a specific, narrow failure mode of its predecessor rather than being a general improvement in every dimension. BM25 solved TF-

IDF's lack of saturation and length normalization, not its inability to bridge vocabulary mismatch. Dense embeddings solved vocabulary mismatch, not exact-match precision. Hybrid retrieval solved the exact-match regression dense retrieval introduced, not the latency cost of running two searches. Reranking solved the precision ceiling of independent scoring, not the fact that it is too slow to run against an entire corpus. None of this is a criticism of the field — it is simply how engineering trade-off frontiers move, and pretending otherwise leads teams to expect a single technique to solve problems it was never positioned to solve.

A second limitation worth being explicit about is that current RAG architectures are fundamentally reactive: a fixed retrieval strategy runs regardless of whether the incoming query actually needs deep retrieval, needs no retrieval at all, or needs retrieval across multiple, decomposed sub-questions. A simple factual query and a multi-hop reasoning query are, in most current systems, handled by the identical pipeline with the identical parameters, which is wasteful in the first case and insufficient in the second.

A third limitation is evaluation itself. Faithfulness and context recall are usually computed with an LLM acting as a judge, which introduces its own biases and blind spots into the very metric meant to catch a different model's mistakes. This is a solvable problem in principle but not yet a solved one in practice, and it means retrieval quality claims in this space should be read with some skepticism until independently reproduced against a held-out, human-labeled evaluation set.

VIII. VIII. FUTURE DIRECTIONS

The most active area of current work addresses the reactive-pipeline limitation directly through adaptive retrieval: systems that first decide how much retrieval a query actually needs, potentially decompose a complex question into sub-questions retrieved and answered independently, and iterate — retrieving again if an initial pass did not surface sufficient grounding — rather than running a single fixed retrieval step regardless of query complexity. This is often framed as agentic RAG, where the language model itself is given retrieval as a tool it can invoke repeatedly and selectively rather than being handed a pre-assembled context block once.

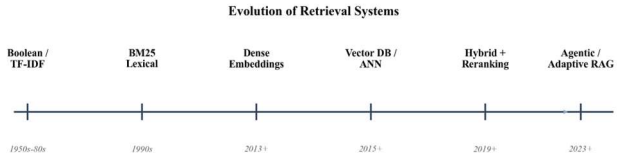

Fig. 5. Approximate evolution of retrieval systems from term-based methods through to adaptive, agentic retrieval.

A second direction worth watching is the move toward retrieval architectures that operate without a conventional pre-built vector index at all, instead performing retrieval-like reasoning directly over source structure at query time. Early results along these lines show promise on faithfulness and context recall, generally at the cost of materially higher latency, which suggests the field is trading retrieval speed for retrieval correctness as language models get better at directly reasoning over larger volumes of raw context rather than needing it pre-filtered by an external index.

Longer term, the boundary between "retrieval" and "reasoning" seems likely to keep blurring. A system that decides for itself what to retrieve, how much, and when it has enough grounding to answer is not cleanly a retrieval system or a reasoning system — it is both, and the evaluation frameworks and architectural patterns this paper describes will need to keep evolving alongside that blur rather than treating retrieval as a solved, static preprocessing step bolted onto generation.

IX. IX. CONCLUSION

Retrieval has evolved from exact term matching to dense semantic search to multi-stage hybrid pipelines with reranking, and each transition was driven by a genuine, specific limitation of what came before rather than by a general notion of technological progress. Understanding retrieval as a series of targeted trade-offs, rather than a ladder of strictly improving techniques, is what lets a team choose the right architecture for their actual constraints — corpus size, latency budget, precision requirements, and the shape of the queries their users actually ask — instead of defaulting to whatever the newest technique happens to be. Modern Retrieval-Augmented Generation systems are best understood not as a single technique but as a layered architecture assembled from decades of retrieval research, and the direction the field is heading — adaptive, agentic, reasoning-integrated retrieval — suggests that the layers themselves are likely to keep being redrawn.

----------**********----------

REFERENCES

[1] I. P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Adv. Neural Inf. Process. Syst. (NeurIPS), vol. 33, 2020, pp. 9459-9474.
[2] II. V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," in Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP), 2020, pp. 6769-6781.
[3] III. K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "REALM: Retrieval-augmented language model pre-training," in Proc. 37th Int. Conf. Mach. Learn. (ICML), 2020.
[4] IV. O. Khattab and M. Zaharia, "ColBERT: Efficient and effective passage search via contextualized late interaction over BERT," in Proc. 43rd Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval, 2020, pp. 39-48.
[5] V. J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with GPUs," IEEE Trans. Big Data, vol. 7, no. 3, pp. 535-547, 2021.
[6] VI. Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," IEEE Trans. Pattern Anal. Mach. Intell., vol. 42, no. 4, pp. 824-836, 2020.
[7] VII. S. Robertson and H. Zaragoza, "The probabilistic relevance framework: BM25 and beyond," Found. Trends Inf. Retrieval, vol. 3, no. 4, pp. 333-389, 2009.
[8] VIII. N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP), 2019, pp. 3982-3992.
[9] IX. LangChain, "Retrieval," LangChain Documentation. [Online]. Available: https://python.langchain.com/docs/concepts/retrieval/
[10] X. Pinecone Systems, "What is a vector database?" Pinecone Learning Center. [Online]. Available: https://www.pinecone.io/learn/vector-database/
[11] XI. C. D. Manning, P. Raghavan, and H. Schutze, Introduction to Information Retrieval. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[12] XII. Y. Gao et al., "Retrieval-augmented generation for large language models: A survey," arXiv preprint arXiv:2312.10997, 2023.
[13] XIII. G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics (EACL), 2021, pp. 874-880.
[14] XIV. T. Gao, H. Yen, J. Yu, and D. Chen, "Enabling large language models to generate text with citations," in Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP), 2023.

