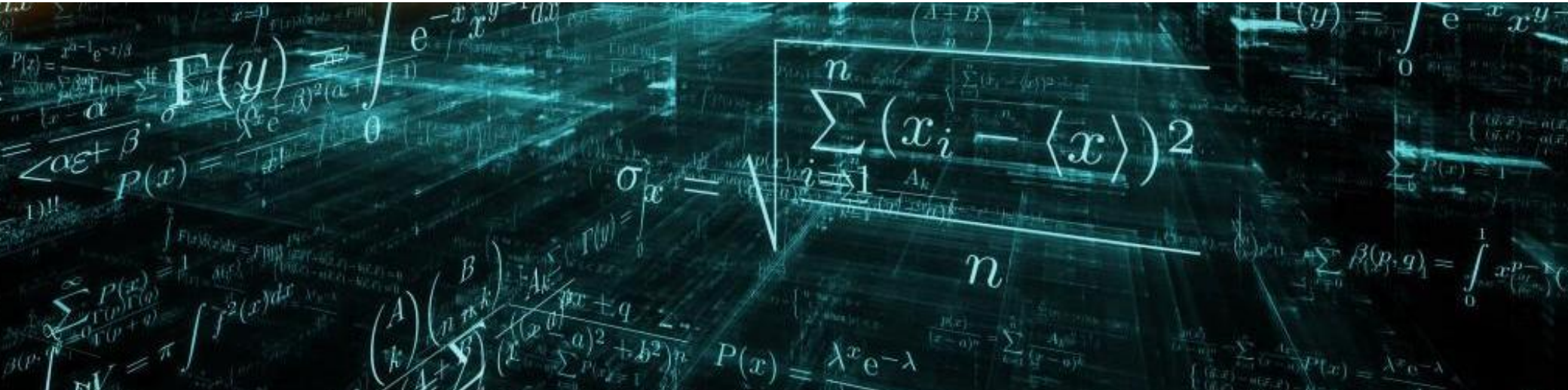
Universität Basel

Computer Science / 15731-01 / 2025

Multimedia Retrieval

Chapter 8: Retrieval Augmented Generation

Dr. Roger Weber, roger.weber@gmail.com

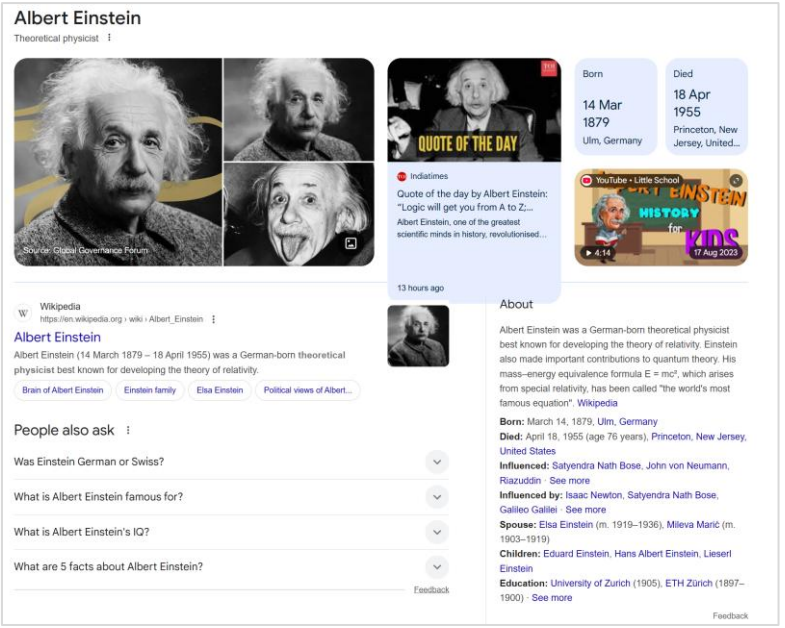


8.1 Introduction

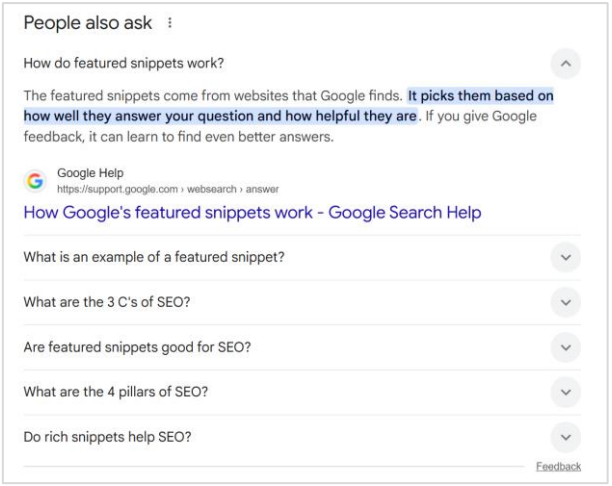
- Information retrieval is not just about finding documents. It is about responding accurately to the user's underlying intent. Without understanding that intent, there is a real risk of giving irrelevant or incomplete answers. For example, when someone asks, "Who won the Nobel Prize for Physics in 2025?" they expect a clear, direct answer with names and context, not just a list of links to sort through.
- Companies such as Google have invested heavily to provide direct answers using tools like the Knowledge Graph, featured snippets, and highlighted answers that show concise, authoritative information. Different types of search queries determine how retrieval systems respond:
  - **Navigational queries** try to find a specific site or brand, so systems favor exact matches and trusted sources.
  - **Transactional queries** show commercial intent, so systems combine product data and language understanding to display relevant offers and availability.
  - **Informational queries** seek clear, reliable answers and often use document search together with answer extraction or generation.
  - **News queries** emphasize freshness and real-time updates, integrating time-sensitive ranking and summarization to keep users informed.
  - **Research or exploratory queries** seek broad, authoritative coverage and may involve synthesizing information across multiple documents.
  - **Local queries** hinge on geographic context, using proximity and location data to deliver relevant results.
  - **Comparative queries** target structured comparisons and reviews.
  - **Personal queries** focus on privacy and tailoring results based on user profiles.
  - **Entertainment queries** highlight popularity, freshness, and multimedia content.
  - **Feed based queries** (often without explicit query) to explore more content on a social platform.
- To serve user intent well, it is important to distinguish different query types because each needs a different strategy. For example, informational queries require clear, authoritative answers, while news queries require recent, up to date coverage. This chapter focuses on informational queries and shows how **Retrieval Augmented Generation, or RAG**, improves their handling. RAG lets models answer questions that go beyond the static knowledge in LLM training data, such as current facts, product details, company policies, or personal meeting notes.

- The **Knowledge Graph** introduced by Google in 2012 is an early form of retrieval augmentation. Rather than returning raw documents, it supplies authoritative, structured facts that can be combined with retrieved data to produce concise answers. It helped shift search from lists of links to clear, context-aware responses.
- The earliest form of RAG in web search is Google's **Featured Snippets** (2014). These snippets pulled short, relevant answers from web pages and showed them above regular search results, synthesizing information instead of just listing links. Typical steps include:
  - **Query Understanding:** The system interprets the user's intent, recognizing that the query calls for a direct answer rather than just a list of links.
  - **Document Retrieval:** It pulls relevant pages using ranking algorithms based on keyword matching, relevance signals, and other ranking factors.
  - **Passage Extraction:** Within those documents, the system scans for passages or sentences that best address the query (so-called readers).
  - **Answer Selection:** The best candidate passage is selected as the snippet.
- While not generation in the modern LLM sense, featured snippets represented an important step toward answering user intent with extracted, precise information. This approach was paving the road for more advanced RAG techniques that integrate retrieval with natural language generation today.

Knowledge Graph, Google (2012)

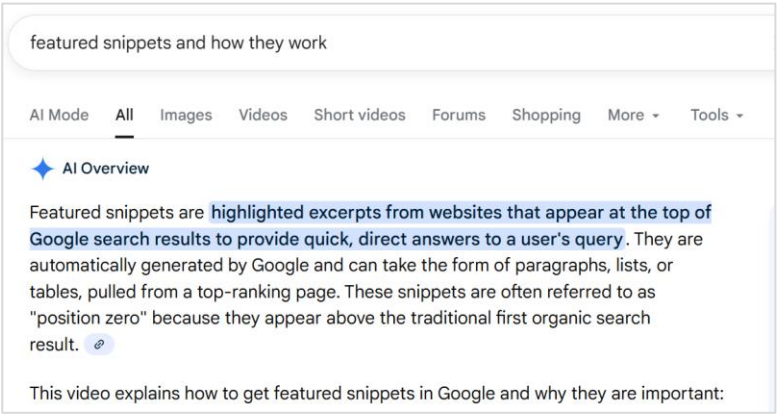


Featured Snippet, Google (2014)



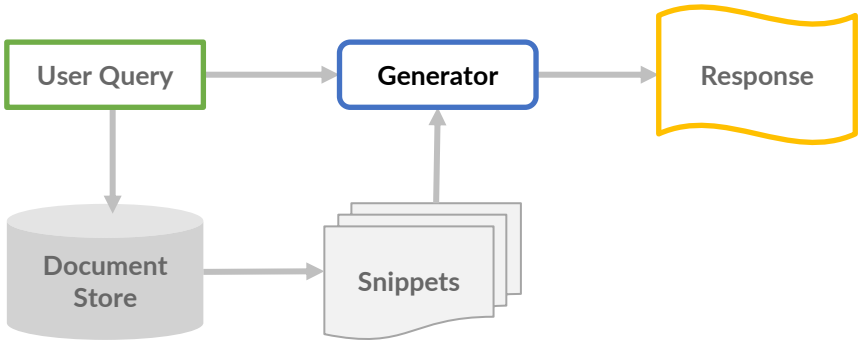
- Recently, Google integrated large language models to generate answers by combining retrieved information with generative AI. This produces responses that are accurate and contextually rich, going beyond simple extraction to include summarization and clarification.
  - In certain cases, Google combines AI generation with a reader approach, producing text and highlighting the direct answer to the user's question, as shown in the example on the right.
- **Retrieval-Augmented Generation (RAG)** is useful beyond traditional web search. For questions that require non-public or very specific information, such as "Which insurance is best for me?", responses should be tied to a clear context, for example an individual insurance company's policies, rather than to general pre-learned facts.
  - Fine-tuning a large language model to add this context is possible but costly, slow, and inflexible. RAG is a more efficient option because it retrieves relevant, up-to-date information from specific sources when a question is asked.
  - Hallucinations happen when a language model tries to answer without enough information or context, causing it to give plausible but incorrect or made-up responses. By explicitly grounding answers in retrieved documents or data, RAG significantly reduces these hallucinations and ensures the output is tied to real, verifiable sources.
  - RAG implementations often combine generation and reader approaches and link back to sources.

AI Overview, Google (2024)



Generation-only is not always successful

Retrieval Augmented Generation



- In the past two years, **Retrieval-Augmented Generation (RAG)** has advanced significantly as model capabilities improved and the need to handle larger, more complex information spaces grew.
  - At first, a major limitation was the small context window of large language models. For example, OpenAI's GPT-2 had a maximum context of 1,024 tokens, about 750 to 800 English words. Multi-language support was also basic.
  - Because LLMs can process only a limited amount of text at once, documents are split into manageable chunks. This lets retrieval systems break large sources into pieces (**chunks**) that fit the model's input size, so they can retrieve and generate from relevant segments instead of overwhelming the model with entire documents.
  - With the rise of transformer models, vector search grew more popular because it captures meaning better and can find relevant documents even when they use different keywords than the query.
  - Using the same chunks for vector search and generation at first seemed practical for fitting documents into early generators' small context windows, but it caused major problems. Splitting documents into smaller pieces made it possible to embed and search manageable segments, yet vector search on these isolated chunks often returned fragments that were semantically incomplete or lacked enough context to be relevant. As a result, the retrieved pieces did not fully answer queries and sometimes even confused the generator.
  - **Hierarchical chunking** is a promising approach to the trade-off between retrieval and generation. It uses small chunks for vector search to keep strong semantic matches, since vector search works better on concise, focused text. It then combines those small pieces into larger, coherent units for the generation step. For example, recent Gemini models have context windows of up to 1,000,00 tokens, allowing the generation step to process these larger, aggregated chunks and produce more comprehensive, context-rich responses without losing coherence.
- Recently the field has moved toward **agentic RAG** systems that combine retrieval, reasoning, and generation in a more interactive and flexible way. Rather than a simple retrieve then generate pipeline, these models can perform multiple retrieval steps, query external knowledge bases, or interact dynamically with structured data sources. This lets them handle more complex queries that need multi-step reasoning to combine data from different sources.
- Moreover, there is a growing shift to include generic structured data alongside unstructured text. By integrating databases, APIs, or knowledge graphs as retrievable sources, RAG systems can give precise answers based on up-to-date, authoritative data. This change makes RAG more versatile and useful in fields where accuracy and context matter. In the past two years, RAG has moved from limited, chunk-based retrieval combined with keyword search to semantically rich, agent driven systems that dynamically integrate structured and unstructured data to produce deeper, more reliable information.

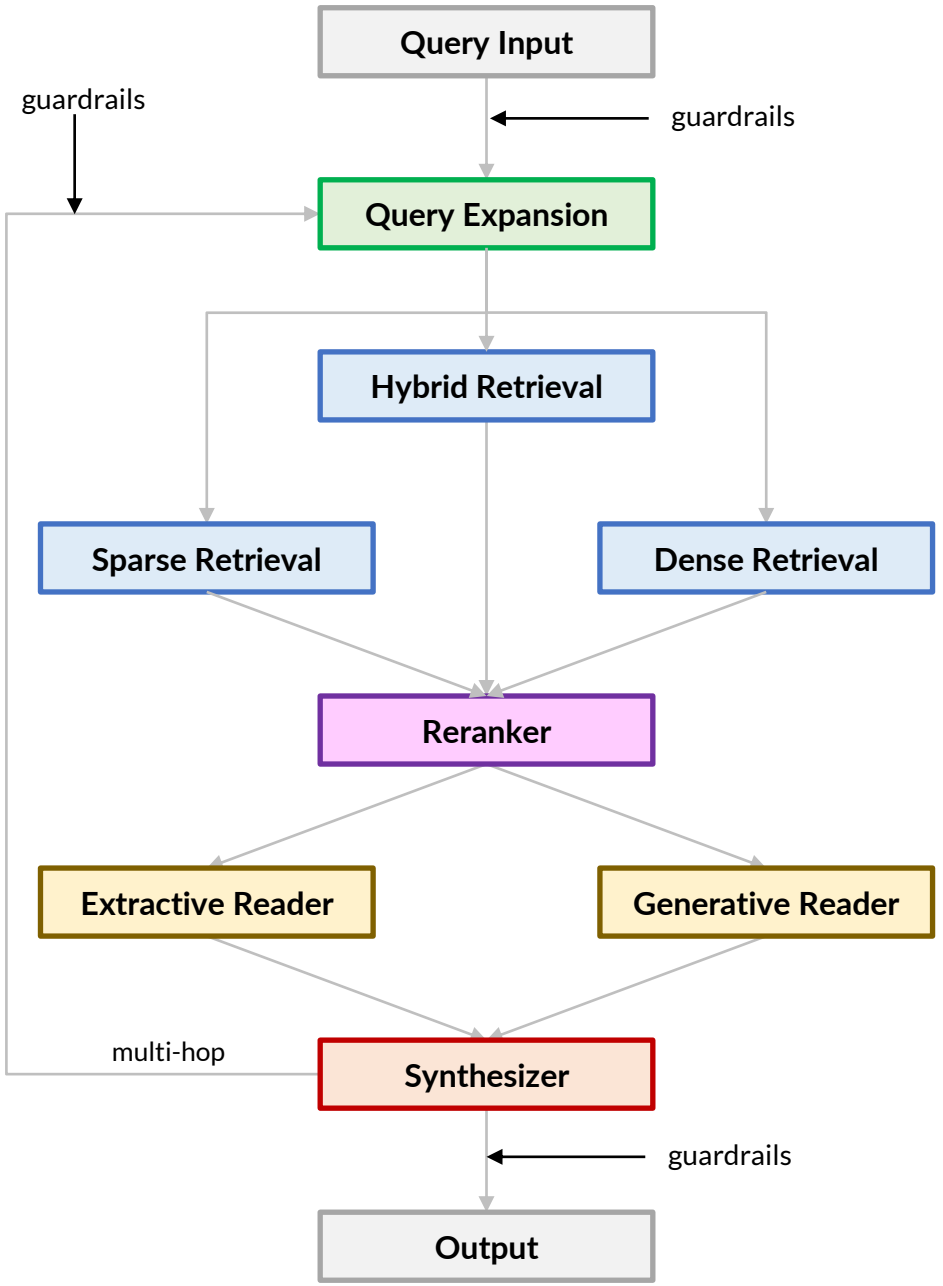
8.1.1 Pipeline Overview

- The tolerance for latency in search systems has shifted significantly. Users are more willing to accept longer latency if it means getting better answers. The experience of receiving responses as a continuous text stream has also made slight delays feel more natural and less frustrating. In specialized areas like research, search sessions can last minutes, but this is only acceptable when the results are truly more valuable and accurate. Because large language models inherently take time to generate responses, users have grown more accepting of slightly longer retrieval times. This has relaxed the previously strict “sub-10ms recall at all costs” mindset that dominated traditional search, allowing systems to prioritize more thorough, higher-quality retrieval over ultra-fast but potentially less accurate results. This shift enables more complex retrieval strategies without sacrificing user experience.
- In the following, we discuss the retrieval pipeline that are typically used for Retrieval Augmented Generation (RAG). Over the past two years, the pipeline has grown from a 2-step approach (retrieve, then generate) to a complex multi-step pipeline. Let us use the following query as a running example:

  Who won the Nobel Prize for Physics in 2025?

  - Traditional keyword-based search systems have a major limitation: they rely on exact word matches with the query. For example, a relevant document that says "The 2025 Nobel Physics laureate was awarded to..." instead of "won the Nobel Prize" might be missed. The system does not recognize that "won" and "was awarded to" mean the same thing or that "2025" means the year, so it can miss relevant information, return unrelated documents that only match keywords, and downgrade highly-relevant documents that use wording similar to the query.
  - Vector search improves results by turning queries and documents into embeddings that capture overall meaning. This helps the system match different wording, for example recognizing that "recipient of the 2025 Nobel Prize in Physics" refers to the same idea. However, it can miss fine details. A word like "who" which signals a search for a person, becomes part of the general vector and loses that explicit role. The system may then return pages about the Nobel Prize event rather than the specific person who won in 2025.
  - To address these issues, retrieval systems should combine semantic search with deeper language understanding and answer extraction. For example, after a vector search narrows down documents related to the 2025 Nobel Prize, a specialized layer should recognize that the word "who" requires a person's name and extract that fact. This hybrid approach ensures the system does not just find broadly related information but provides the exact answer.

- The pipeline on the right shows all the components of a possible retrieval strategy for informational queries. We will break down the role of each component:
  - **Query Expansion**: As discussed earlier, the query can be transformed by several methods: LLM rewrites, breaking it into multi-hop questions, or semantic expansion with embeddings. These changes help to better align the query with the documents.
  - **Sparse Retrieval:** BM25 is a sparse retrieval method that relies on keywords and how often they appear. With effective query expansion, it can find many documents relevant to the query.
  - **Dense Retrieval:** Vector search can find semantically similar documents without query expansion. However, the context must be short to produce useful embeddings. The common approach splits a long document into smaller parts, called chunks, and searches those chunks instead of the full document.
  - **Hybrid Retrieval**: Combines sparse and dense retrieval in a two-stage framework. Stage 1 performs efficient sparse retrieval, and Stage 2 refines the results with more advanced models.
  - **Reranker:** Bi-encoders embed the query and document separately. Cross-encoders combine the query and document during inference, allowing richer interaction and letting the model directly judge query-document relevance. For example, it can match "who" to people instead of other meanings. Because this requires expensive inference, rerankers are usually applied to only a small set of candidates.



- **Extractive Reader:** An extractive reader answers questions by finding the exact span of text in a passage that contains the answer. It encodes the query and the passage together, then checks each token in the passage to decide where the answer starts and ends. Because it uses the original text directly, the output is not a paraphrase or interpretation but the exact snippet from the passage.
  - For example, if the passage says "John Clarke, Michel Devoret, and John Martinis won the 2025 Nobel Prize in Physics", an extractive reader finds the start and end tokens around the names and extracts that exact span: "John Clarke, Michel Devoret, and John Martinis".
- **Generative Reader:** A generative reader answers questions by generating responses one token at a time using a large language model. It takes both the query and the retrieved passages as input. This lets the model produce a synthesized, natural sounding answer.
  - For example, after reading several passages about the 2025 Nobel Prize in Physics, a generative reader might say: "The 2025 Nobel Prize in Physics was awarded to John Clarke, Michel Devoret, and John Martinis for their discoveries in quantum tunneling and energy quantization in electric circuits." Unlike extractive models, it does not just copy this sentence from one source; it builds the answer from the available evidence.
- **Synthesizer**: For more thorough answers in research mode, repeat the retrieval steps several times. At each step, a trained reasoning model asks for more information to complete the generative answer. For example:
  - Hop 1: Retrieve documents about the 2025 Physics Nobel winners. This identifies Clarke, Devoret, and Martinis.
  - Hop 2: Use names to retrieve data about their research focus. This finds work on quantum computing applications.
  - Hop 3: Retrieve documents that connect that research focus to specific teams or organizations.
- **Guardrails** are rules that make sure the system provides answers that are accurate, safe, and match the user's intent. They control how the system selects, generates, and delivers information after retrieval so the final answer is trustworthy and appropriate. They act at several stages of the process, from input to output, ensuring the final response is accurate, safe, and aligned with the user's expectations. Guardrails are implemented through a combination of:
  - Prompt instructions and fine-tuned model behavior
  - Rule-based filters and classifiers
  - Evidence grounding and verification mechanisms
  - Safe output formatting and fallback strategies

- Step by step execution:

1. **Query Input:** "**Who won the Nobel Prize in Physics in 2025?**"
2. **Query Expansion**: generate variants like "**2025 Physics Nobel winners**", "**physics Nobel laureates 2025**"
3. **Retrieval** (parallel or sequential):
   - **BM25**: returns ~500 documents with exact keyword matches (**Nobel**, **Prize**, **Physics**, **2025**, **winner**, **laureates**)
   - **Bi-Encoder**: returns ~500 documents with semantic similarity to query embeddings
   - **Hybrid Merging**: combine results from both retrievers and select ~100 top candidates
   - **Alternative**: BM25 for top-500, then bi-encoder over top-500 to select ~100 top candidates
4. **Reranker**: score all 100 candidates with a cross-encoder → top 20 reranked documents
5. **Reader** (one of the following):
   - **Extractive Reader**: finds a span with "**Clarke, Devoret, and Martinis**" in top documents
   - **Generative Reader**: generate answer: "**The 2025 Nobel Prize in Physics was awarded jointly to John Clarke, Michel Devoret, and John Martinis for their discoveries in quantum computing.**"
6. **Synthesizer**: performs multiple hops (reasoning) and produces final response with top-ranked documents: "**The 2025 Nobel Prize in Physics was awarded to John H. Clarke, Michel H. Devoret, and John M. Martinis for their groundbreaking discoveries on quantum tunneling and macroscopic quantum effects in electric circuits. Clarke, from UC Berkeley, Devoret from Yale University, and Martinis, now at Google Quantum AI, developed key technologies for quantum computing, including quantum error correction and superconducting qubits.**"

- Modern RAG systems are modular pipelines that can be tuned for speed, accuracy, or both. Individual components, such as sparse retrieval like BM25, dense retrieval using embeddings, or cross-encoder reranking, can be swapped, skipped, or run in parallel. Simple queries may skip dense retrieval, while complex or domain-specific queries often use cross-encoders for the best results.
- These systems trade off speed and quality. Sparse retrieval is fast but captures less meaning. Dense retrieval finds semantic matches but uses more compute. Cross-encoders raise accuracy by reranking results. Using all three gives the best output but at a higher cost. Because of this modular design, RAG systems can power anything from very fast search tools to precise, domain-specific question answering.

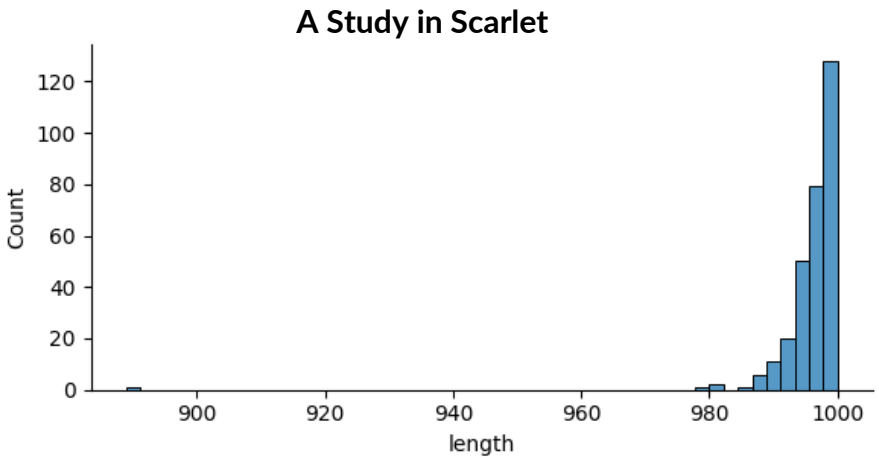
8.2 Chunking Strategies

- Classical information retrieval used chunking to improve search. Long documents were split into smaller sections so search engines could find where a query matched and make sure the search terms were near each other. Instead of returning the whole document, retrieval systems showed the most relevant passage, giving the user both location and proximity context.
- When Retrieval-Augmented Generation (RAG) first appeared, it used chunking for a practical reason: limits in both embedding models and language models. Early embedding models could handle only short pieces of text, often a few hundred tokens, before losing semantic clarity. At the same time, early language models had small context windows and could not accept long documents as input. For these reasons, splitting documents into manageable chunks was essential to enable retrieval and generation.
- Today, chunking is no longer just a workaround for technical limits but a deliberate design choice that needs balance. Larger chunks give the language model more context and help it reason and produce accurate answers. But embeddings work best with shorter segments that preserve precise meaning and avoid blurring. Research and practice show that about 512-1024 tokens form the sweet spot: large enough for coherent ideas but small enough to yield high-quality embeddings.
- The best chunk size depends on the task. For factual lookups or direct question answering, shorter chunks often work because the answer is usually in a small piece of text. For summarizing, extracting lists (for example, common retrieval models), or combining ideas across a document, you need longer context. If the system only retrieves short fragments, the model can miss important links between ideas.
- Newer retrieval designs use hierarchical chunking. Documents are stored at multiple levels. Small, fine-grained chunks produce dense, accurate embeddings for retrieval. Larger sections or merged chunks give the language model enough context to reason and compose an answer. Additionally, newer methods use query-aware chunk selection. The system adjusts how much and what kind of context it sends to the language model based on the query. For a precise factual question, it sends only the most relevant small chunk. For a summary request or a question spanning several topics, it gathers and sends a much larger section or multiple related chunks.
- Many frameworks include built-in chunking features, and LangChain is one of the most popular. LangChain lets developers choose simple fixed-size token or character chunking, recursive chunking that follows document structure, or overlap-based chunking to keep continuity between segments.

- **Method 1:** Splitting the text into fixed-sized chunks
  - A simple chunking method divides a document into fixed-size pieces. First split the text into words, then group words until each chunk reaches the set size. To reduce cutting through sentences or paragraphs, add overlaps between chunks. If a sentence or paragraph is split, most of it will still appear in the next chunk, preserving context.
  - The `langchain` library has a convenient function for this kind of chunking:

```
from langchain_text_splitters import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
      separator = " ",
      chunk_size = 1000,
      chunk_overlap  = 200,
      add_start_index = True
)
text_splitter.split_text("…a very long text…")
```

  - In this example:
    - `separator=" "` tells the splitter to break the text along spaces (word boundaries).
    - `chunk_size=1000` sets the maximum length of each chunk in characters.
    - `chunk_overlap=200` defines an overlap of about 200 characters between subsequent chunks.
    - `add_start_index=True` stores the starting position of each chunk in the original document.
  - When this splitter is used on a sample document, the chunk length distribution shows most chunks are close to the 1,000-character limit. Small variations arise because the splitter avoids breaking words. The single short outlier is the final chunk, which contains the remaining text.
  - This method is practical and easy to implement. It works well when rough locality is enough. However, it has a major drawback: chunks do not line up with meaningful boundaries. A new topic, paragraph, or section can be split across two chunks or mixed with the end of an unrelated one. Retrieval systems may return passages that lack coherent context, and embeddings can merge unrelated concepts.

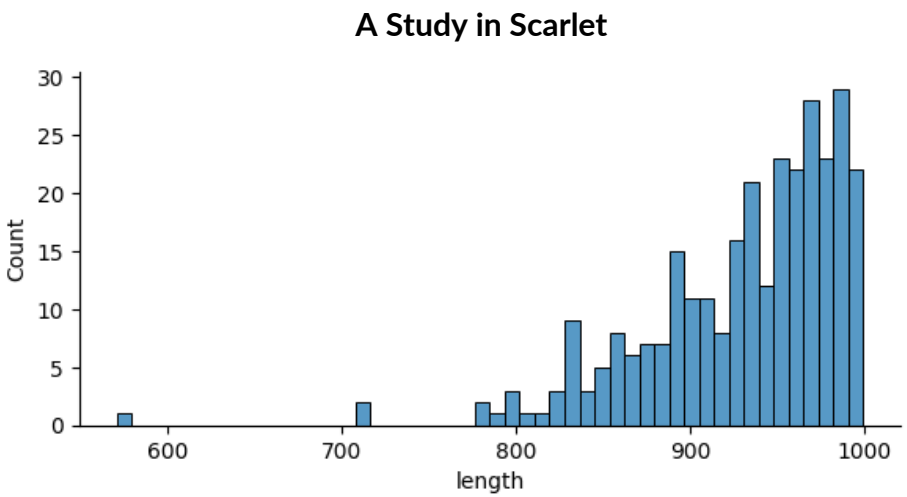


- **Method 2**: Splitting at sentence boundaries
  - This approach is like fixed-size chunking, except that sentences, not raw character counts or word groups, are the smallest unit of text. By combining full sentences into chunks until a size limit is reached, this method avoids awkward breaks in the middle of sentences while still producing manageable chunks. Chunk sizes can vary slightly as a result, but the difference is usually negligible.
  - Sentence boundary detection may sound trivial, but it's more complex than simply splitting at every period. To handle this accurately, libraries provide robust models for sentence tokenization:
    - **NLTK's Punkt Sentence Tokenizer:** Punkt uses an unsupervised algorithm that learns how sentences start and end from a training corpus in the target language. It builds internal models of abbreviations, common sentence starters, and frequent boundary patterns. NLTK includes a pre-trained English Punkt model, and you can train custom models for other languages. It is fast and usually very accurate, but it can struggle with complex narrative dialogue where punctuation appears both inside and outside quotation marks.
    - **spaCy Sentence Segmentation:** spaCy uses machine learning and deep linguistic parsing to detect sentence boundaries. It supports many languages built in and handles complex sentence structures more consistently than rule-based systems. It is heavier than Punkt, requiring more CPU and GPU resources, but it is well suited for production systems where accuracy and extensibility matter.
  - LangChain supports sentence-aware text chunking using both NLTK and spaCy:

```
from langchain_text_splitters import SpacyTextSplitter, NLTKTextSplitter
text_splitter = SpacyTextSplitter(    # or: NLTKTextSplitter
      separator = " ",
      chunk_size = 1000,
      chunk_overlap  = 200,
      add_start_index = True
)
text_splitter.split_text("…a very long text…")
```

  - Sentence-based chunking balances structure and simplicity. Compared with raw character splitting, it produces more coherent chunks that map naturally to meaning. However, sentence boundaries do not always match topic shifts or logical sections, so this method can still mix unrelated ideas within a chunk.

- The plot on the right displays the distribution of chunk lengths obtained by applying the NLTK splitter to the sample document ("A Study in Scarlet"). The results for spaCy are very similar:
  - Most chunks are between 800 and 1000 characters long, with variations resulting from sentence lengths causing splits. Long sentences can result in much higher variance than the simple word splitting approach.
  - The left outlier represents the final chunk of the document, which is generally smaller.
  - The approach is a bit better than word-based splitting, however, we still have a misalignment between chunk size and semantic changes in the document.

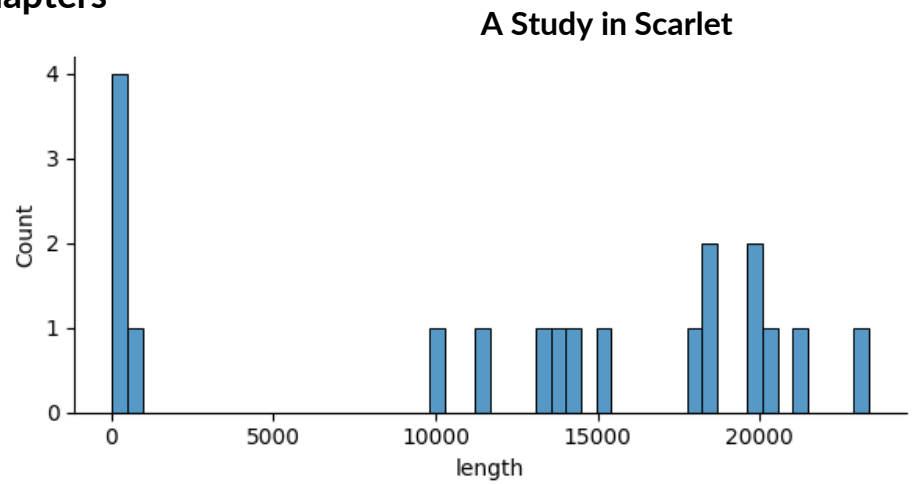


- **Method 3:** splitting on structure
  - The concept is to divide text based on its structural elements. For instance, in books, we can split at pages, parts, chapters, sections, and paragraphs. Authors commonly use these structural elements to separate content, making them strong indicators of topic or aspect changes.
  - Detecting these structural elements relies on the document's format. In our running example, we utilized text from Project Gutenberg, where paragraphs, chapters, and parts are structured with increasing numbers of newlines before their start. For instance, 4 consecutive newlines indicate chapters, and 2 consecutive newlines indicate paragraph breaks. In HTML or Markdown formats, we can split by identifying headers (`<h1>`, `<h2>`, …), while treating the text within `<p>` and `<div>` tags as paragraphs.

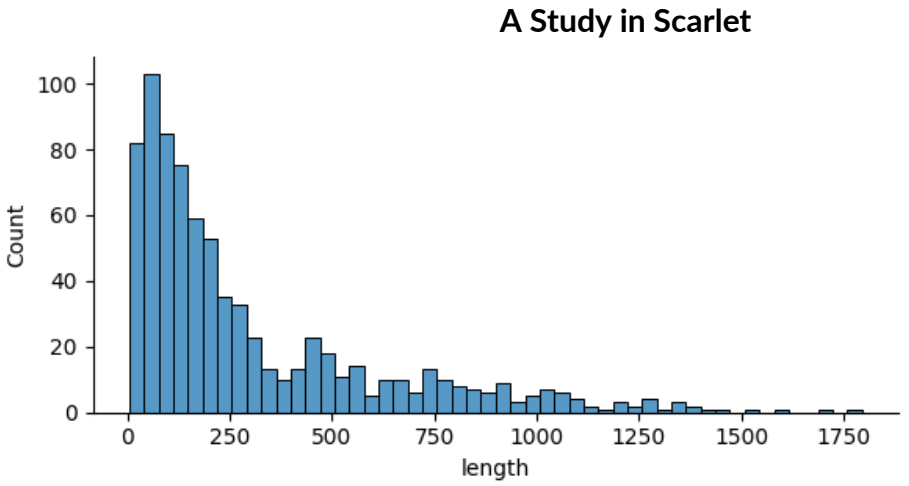
```
from langchain_text_splitters import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
      separators = ["\n\n"],    # use 4x\n for chapters, and 2x\n for paragraphs
      chunk_size = 100,
      chunk_overlap  = 20,
      add_start_index = True
)
text_splitter.split_text("…a very long text…")
```

- To prevent adjacent paragraphs or chapters from merging, make the chunk size and overlap small enough that each chunk holds only one paragraph or one chapter. This splitting approach then produces separate chunks for every paragraph or chapter, depending on the separator you choose.
- The figures on the right show chunk size distributions for chapters and for paragraphs, which contrast strongly with the word- and sentence-based approaches. Chunks now follow chapter or paragraph boundaries and give more accurate semantic coherence than the fixed-size splits used before.
- Unlike earlier splitting methods, the chunks now vary in size. For chapters, small chunks can come from formatting at the start of the text or from parts that can be ignored. These coarse splits usually yield coherent pieces, but their sizes are hard to control and can be too large for embedding and language models, for example when several chapters are added to the RAG context.
- Splitting by paragraphs (lower figure on the right) lets us control chunk size better. Most chunks stay below 1,500 characters, although this varies by author. But this method creates many very small chunks. This happens because the book is written as dialogue and different speakers are separated by blank lines. Those tiny chunks are not good for generating responses because they provide too little context and information to extract or synthesize an answer. So in this case, we want to merge the smaller chunks again.

Chapters

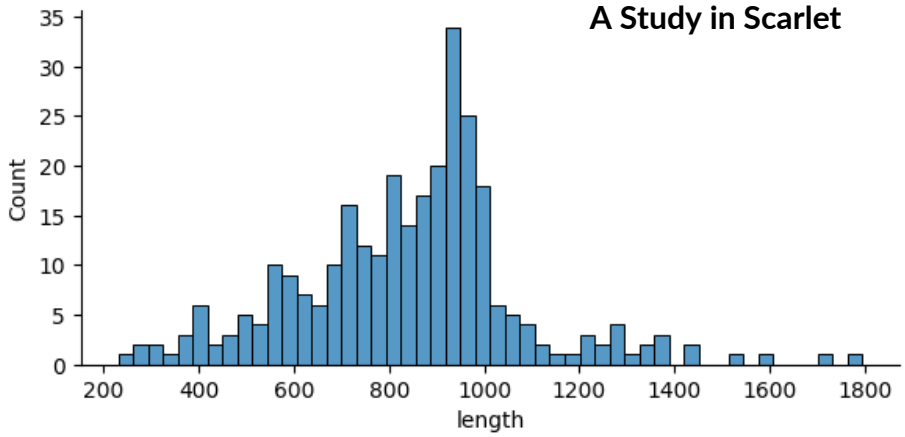


Paragraphs

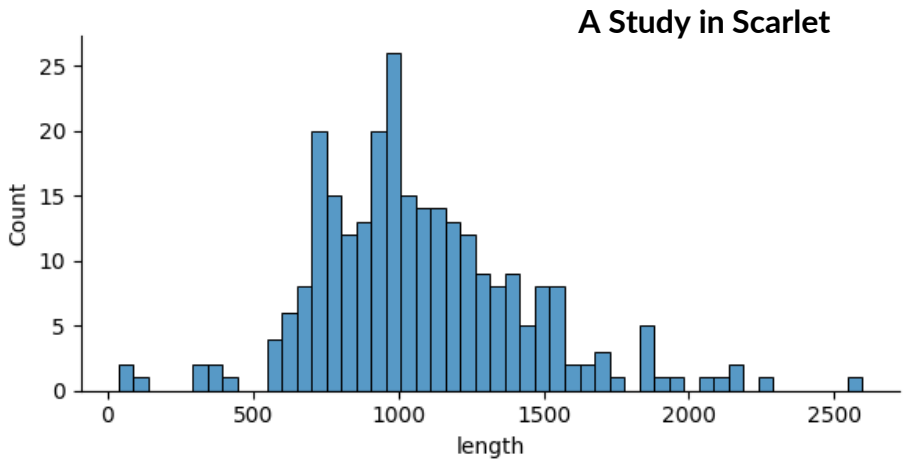


- To improve paragraph-based chunking, we increase chunk sizes and merge small paragraphs that are likely related. The example on the right shows this. Small chunks were removed, and most chunks are now between 800 and 1,000 characters, though some are longer because of the author's preference for long paragraphs. For excessively long paragraphs, we can split them further with the `RecursiveCharacterTextSplitter`. It accepts a list of separators and can break oversized chunks into smaller ones in stages.
- Trying to control overlap of paragraph-based chunks by setting a fixed number of characters is difficult. Whole paragraphs can be longer than the overlap limit, so there may be no overlap. A better approach is to create paragraph chunks as described earlier and merge any small ones. For each paragraph, add the last sentence from the previous chunk and the first sentence from the next chunk. These added sentences often capture the meaning of nearby paragraphs and give better descriptions and context.

Paragraphs (merged)



Paragraphs (merged with overlap)



- **Method 4**: semantic splitting
  - Splitting text by document structure (section headers, paragraphs, etc.) often produces coherent chunks with little effort. However, structural metadata is not always available or reliable. For example, scraped web pages may have inconsistent formatting, PDFs can have broken layouts, and scanned documents often lack clear headings. In such cases, we can use semantic splitting: group sentences by meaning rather than by formatting.
  - The general approach builds on sentence-based splitting and extends it with semantic similarity:
    - Define a similarity measure between sentences
    - Specify minimum and maximum chunk sizes
    - Split the text into sentences using NLTK or spaCy
    - Merge neighboring sentences or chunks if they are semantically similar not exceeding the maximum chunk size
  - **Sentence Similarity:** A naive approach uses term-based vector space models. Sentences are converted into word-frequency vectors and compared with measures such as cosine similarity. A major flaw is that two sentences that express the same idea with different words can receive a misleadingly low similarity score. To fix this, we use a bi-encoder that encodes sentences into dense vectors and measures similarity with cosine similarity. A more advanced approach uses models that detect whether the next sentence belongs in the current chunk.
  - **Merging Chunks Based on Similarity:** Once sentence embeddings are generated, we can begin merging:
    - Start with each sentence (or sentence group) as its own chunk.
    - Compute similarities between adjacent chunks
    - Merge the top-k most similar neighboring chunk pairs as long as maximum chunk size is not exceeded
    - Use a relatively small k, often a fraction of the current number of chunks
    - Repeat the process iteratively. Over multiple passes, this encourages the formation of longer, semantically coherent chunks that reflect topic continuity

    This iterative merging process clusters related sentences while preserving boundaries between distinct topics. It also allows the flexibility to control how granular or expansive the final chunks should be.
  - Semantic splitting yields higher-quality chunks than fixed-size or structural methods, especially in narrative or technical documents where topic flow matters more than surface formatting. It is more computationally expensive, but produces chunks that align with meaning and improve embeddings, retrieval accuracy, and downstream responses in RAG systems.

- **Method 5**: hierarchical chunking (for RAG use cases)
  - Hierarchical or hybrid chunking uses multiple splitting methods so the strengths of some methods offset the weaknesses of others. Instead of one chunk size or rule, the document is split at several levels of detail, for example into sections, then paragraphs, then sentences or token-based chunks. This approach is especially useful for retrieval-augmented generation (RAG), because it gives high quality embeddings for retrieval and larger context windows for synthesis.
  - The core idea is simple:
    - Split the document using large structural units such as chapter titles, headings (H1, H2), or sections.
    - Within each section, apply a finer splitter such as sentence-based, fixed-token, or semantic splitting.
    - Store both levels of chunks:
      - Small chunks → optimized for embeddings and similarity search.
      - Large chunks or merged segments → provide richer context to LLMs during answer generation.
  - Here is an example of how LangChain supports this:

```python
from langchain_text_splitters import MarkdownHeaderTextSplitter, RecursiveCharacterTextSplitter

header_splitter = MarkdownHeaderTextSplitter(["#", "##", "###"])
sections = header_splitter.split_text(long_markdown_text)

token_splitter = RecursiveCharacterTextSplitter(chunk_size=512, chunk_overlap=50)

final_chunks = []
for sec in sections:
    sub_chunks = token_splitter.split_text(sec.page_content)
    for idx, sc in enumerate(sub_chunks):
        final_chunks.append({
            "content": sc,
            "metadata": {**sec.metadata, "sub_chunk": idx}
        })
```

- **Method 6**: Query-Aware or Adaptive Chunking
  - Most chunking strategies split documents before any query is made. This is called static chunking. Not every user query needs the same context. For example, "When was Albert Einstein born?" needs a short, fact-centered chunk, while "Summarize how vector search works" needs much more context. Query-aware chunking changes chunk size and content to match the query type and intent. Instead of returning the same chunk size for all queries, the system dynamically selects, expands, or combines chunks depending on how much context is needed.
  - How It Works:
    - Preprocess the document using small, embedding-friendly chunks (e.g., 512 tokens).
    - Interpret the query intent (factual, analytical, summarization, list generation, multi-hop reasoning).
    - Retrieve top-k relevant chunks using vector search or sparse retrieval.
    - Adapt chunk assembly before passing to the LLM:
      - Factual queries: return only the highest-scoring chunk or two.
      - Summarization/explanation queries: Combine several neighboring chunks or return entire sections.
      - Multi-hop questions: Collect related chunks from different sections and stitch them together with metadata or reasoning prompts.
  - This approach reduces noise for simple questions and gives enough context for complex ones. It also improves efficiency by sending fewer tokens to the LLM when they are not needed, and it increases accuracy for tasks that cover multiple parts of a document.
  - Frameworks like LangChain, LlamaIndex, and Haystack already use early versions of this idea, retrieving small chunks to score relevance and sending larger parent sections to the LLM when needed. As models gain larger context windows, query-aware chunking becomes less about overcoming limits and more about optimizing relevance, cost, and reasoning quality.

8.3 Query Transformation

- Retrieving the right information depends not only on what users want to know but also on how they ask. Users often pose queries in informal, vague, or incomplete ways, while documents use different words or phrasing. This mismatch is known as the **vocabulary gap**. Query transformation is an umbrella term for methods that modify, expand, or reformulate a query to close this gap. It includes simple term-level changes and more advanced rephrasing driven by large language models. The main goal is to improve recall by finding more relevant documents while keeping precision by preserving the original intent.
- To show how these methods work, consider the query: "**Who won the Nobel Price in Physics in 2025?**" Although clear in everyday speech, an information retrieval system must treat it as a search for recipients, their names, institutions, scientific contributions, and other related terms that may appear in relevant documents.
- **Method 1: Spell-checking**
  - The first change to a user's query is often correcting spelling mistakes or typing errors. This may seem simple, but it is essential because even a small misspelling can prevent a system from finding relevant documents.
  - In the example query "Who won the Nobel Price in Physics in 2025?" the phrase "Nobel Price" is misspelled: Price should be Prize. If not corrected, the system could search for documents containing the word price, which is not about awards and could return irrelevant results.
  - Why this step matters:
    - Improves matching accuracy: Most retrieval systems rely on exact or near exact term matches. Spelling errors make it less likely that query terms will match words in documents.
    - Prevents recall loss: A misspelled word can reduce matches and cause many relevant documents to be missed.
    - Supports further transformation: Later steps, such as rephrasing, depend on clean and accurate input terms.
  - How spell-checking works: The system compares query terms to a list of correctly spelled words. If a term is not found, it suggests the closest match based on string similarity, e.g., by using edit distance between Price and Prize.
  - After spell-checking, the query becomes:

    Who won the Nobel Prize in Physics in 2025?

  - Now the system can more confidently proceed to the next transformation tasks, such as identifying relevant entities, expanding terms, or generating alternative phrasings.

- **Method 2: Knowledge-Based Expansion**
  - Once the query is clean and spelled correctly, expand it with structured knowledge sources. This helps when key terms link to several related concepts in curated resources. The aim is not to change the query intent but to cast a wider net and find documents that use different but related terms.
  - Documents do not always use the same words as the query. A newspaper might say "The Nobel was awarded to" instead of "Who won the Nobel Prize". By expanding the query with related terms from trusted knowledge sources, systems can match more of these variations, improving recall and keeping results relevant to the user's intent.
  - Types of knowledge sources used
    - Thesauri and lexical databases (e.g., WordNet) provide synonyms or closely related words. For example:

      won → received was awarded secured
      prize → award honor

      This helps match documents that describe the same event using different wording.
    - Ontologies and knowledge graphs (e.g., ConceptNet, Wikidata, DBpedia) go beyond synonyms by revealing relationships between ideas and entities. For instance:

      Nobel Prize → scientific achievement award ceremony Royal Swedish Academy of Sciences
      Physics → quantum mechanics particle physics theoretical physics
      Laureate → winner

      These structures are human-validated, which helps maintain high precision while broadening the search vocabulary.
  - With knowledge-based expansion, our running example becomes something like:

    Nobel Prize, Nobel award in Physics, physical sciences, won, received, was awarded, laureate, scientific achievement, Royal Swedish Academy of Sciences, quantum mechanics

  - Internally, the system uses knowledge-based connections to recognize documents that express the same ideas in different words or with more technical terms, without showing this complexity to the user. This helps deliver more relevant results by capturing phrasing variations, while relying on carefully curated knowledge to avoid irrelevant matches. However, there is a balance. Adding too many related terms can introduce noise and weaken the original intent, so expansions must be managed carefully to stay focused and to avoid query drift.

- **Method 3: Embeddings**
  - After cleaning the query and adding structured knowledge, the next step uses semantic representations learned by neural networks trained on large text collections. Unlike keyword matching or synonym expansion, these models capture subtle meanings and the relationships present in normal language.
  - Why embeddings and neural methods matter: Traditional expansion methods use fixed word lists or rigid relationships. Language is fluid, and words can mean different things depending on context. Embeddings place words in a multi-dimensional space where closer words have similar meanings. This lets retrieval systems find documents that do not use the same words but cover the same ideas.
  - Word embeddings like Word2Vec and GloVe learn vector representations of words by analyzing the words that appear near them in sentences. In this vector space, quantum and mechanics will be close, as will award and prize.
    - For the query “Who won the Nobel Prize in Physics in 2025?”, embeddings might suggest adding terms such as “**laureate**”, “**quantum**”, or “**breakthrough**”, based on semantic proximity.

- **Method 4: Large Language Models**
  - Large language models such as GPT improve queries by using deep reasoning, paraphrasing, and creativity to rewrite them. Rather than only expanding or embedding a query, they understand the user's intent, infer what information is needed, and offer several ways to phrase the same question. They can also think through a likely answer internally to guide retrieval.
    - Multi-query generation: The LLM produces various paraphrases and alternative queries, such as “**Physics Nobel winners 2025**", “**Who received the 2025 Nobel Prize in Physics?**”, “**John Clarke Nobel Prize quantum research**”
    - Step-back prompting: The LLM first broadens the scope by abstracting the query to a general question, like “**What are recent Nobel Prize winners in Physics?**” This wider context helps the system reason before homing in on the 2025-specific answer.
    - Chain-of-thought decomposition**:** The model breaks the complex query into smaller parts: "**What is the Nobel Prize in Physics?**", "**When is it awarded?**", and "**Who won it in 2025?**". Each answer builds on the previous, allowing more precise and structured retrieval.
  - These LLM-driven changes go well beyond matching keywords. They create a richer understanding of user needs, narrative context, and logical structure, enabling more accurate and insightful information retrieval.

- **Method 5: Feedback-Based Transformation**
  - Relevance feedback uses user input to improve the query based on real judgments. When users mark documents as relevant or irrelevant, the system updates the query vector to raise the weight of terms from relevant documents and lower the weight of terms from irrelevant ones. In the Nobel Prize example, if users mark documents that mention John Clarke, Michel Devoret, and John Martinis as relevant, those documents likely include terms such as "**macroscopic quantum tunneling**", "**energy quantization**", or "**quantum computing**". The system adds these frequent terms to the query and reduces terms from irrelevant documents that discuss prizes in other fields like Literature or Peace or awards from earlier years, keeping the search on topic.
  - Another useful type of feedback comes from using users' collective behavior by showing similar queries they have made before. Instead of relying only on explicit judgments about document relevance, this method uses patterns in how other people ask related questions. When users see alternative queries phrased differently but asking the same thing, such as "**Physics Nobel winners 2025**" or "**Who received the Nobel Prize in Physics this year?**", they can refine their search or choose a version that better matches the system's retrieval capabilities.
  - Pseudo-relevance feedback: Manual relevance feedback works well but is not always practical because it asks users to judge documents. Pseudo-relevance feedback automates this by assuming the top-ranked results from the initial search are relevant. The system extracts terms from those documents to expand the query.
    - For example, when the initial query "Who won the Nobel Prize in Physics in 2025?" yields a list of top documents, the system analyzes them to find terms that commonly occur with the query concepts and then adds those terms back into the query.
    - The main benefits are increased recall by adapting the query to the actual content returned. It requires no extra effort from the user and speeds up retrieval.
    - Risks include: if the first retrieval results are off-topic or noisy, the system may drift the query toward irrelevant content, a phenomenon known as query drift. This can cause the system to retrieve fewer relevant documents or to introduce unwanted biases.
  - Despite advances in neural and LLM-based methods, feedback-based approaches remain a cornerstone of many retrieval systems. They draw directly on the user's experience and retrieval results, so query modifications reflect practical relevance rather than just linguistic or statistical links.

- **Balancing Precision, Recall, and Intent**
  - Query transformation is powerful, but it must be managed carefully to avoid losing sight of the user's real goal. A major risk is **query drift**. Adding too many new terms, or terms only loosely related to the original query, can gradually pull the search away from the user's true intent. This can produce many irrelevant documents, reducing precision even as recall increases.
  - Detecting query drift often means checking how query expansion affects retrieval performance. For example, if expansion makes the top results less relevant or shifts them to unrelated topics, such as prizes in other fields or different years, the system flags possible drift. Another method is to measure semantic similarity between the original and expanded queries. If similarity falls below a set threshold, the expansion may be too broad.
  - Preventing drift usually means assigning weights to query terms. The original words keep the highest weight so they anchor the search, while added expansion terms receive lower weights according to their relevance or confidence. For the Nobel Prize query, the weights might look like this:
    - Original terms: "**Nobel Prize Physics 2025**" with full weight (1.0)
    - Named entities from feedback: "**John Clarke**", "**Michel Devoret**", "**John Martinis**" at slightly lower weight (0.8)
    - Related scientific concepts: "**quantum tunneling**", "**energy quantization**" at moderate weight (0.6)
    - Broader contextual terms: "**Royal Swedish Academy**", "**announcement**" at low weight (0.4)

    This hierarchy keeps the search centered while still exploring useful variations.
- **Hybrid Systems: Combining Methods for Best Results**
  - No single query transformation method works for every situation. Different use cases, query types, and document collections need specific strategies. Modern search engines combine several approaches in layered pipelines to use each method's strengths.
  - A typical hybrid system might begin by using a large language model to clarify or paraphrase the user's query, making it more precise. Next, contextual embeddings such as BERT expand the query to include related meanings that a keyword system might miss. Then a knowledge graph adds domain terms and relationships to enrich the search with human curated insights. Finally, pseudo-relevance feedback reviews the initial search results and fine tunes the query to the corpus language and to emerging topics.
  - Combining these strategies, the system balances user intent, semantic nuance, and real document language, while preventing drift through weighted term importance and relevance checks.

8.4 Result Presentation

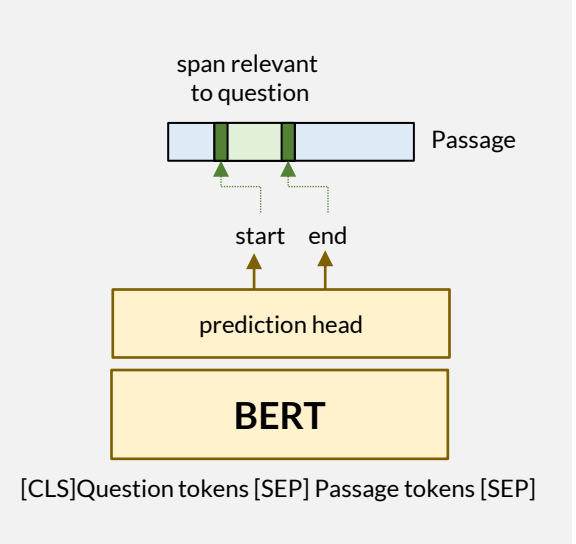
- After the query is transformed, the retriever and reranker work together to produce a short list of relevant results. Instead of just showing that list, it is more effective to give a direct answer taken from or built from those results. This saves time and gives a clearer, more immediate response that better meets the user's needs. Choosing an extractive reader, a generative reader, or a synthesizer depends on the question, the available context, and the user's intent.
  - An **extractive reader** is best when you need exact facts from a few documents or short passages. It selects the exact words from the source, so its answers match the original text and are factual. For example, if the question is "Who won the 2025 Nobel Prize in Physics?", the extractive reader returns "John Clarke, Michel Devoret, and John Martinis". This method is fast, reliable, and especially suitable when accuracy is essential and the context is small.
  - A **generative reader** excels when a question requires combining information from multiple sources. For example, if a user asks "What contributions led to the 2025 Nobel Prize in Physics?" a generative reader can combine details from several documents and merge facts about the winners' research into a clear, natural answer. This makes it a good choice for complex, open-ended questions or to provide a concise, fluent summary. However, that flexibility can lead the model to hallucinate, inventing details not present in the sources, so it needs careful control.
  - The **synthesizer** keeps the generative reader's strengths and adds a multi-step retrieval process. It is best for deep research or when an answer must be built from several layers of information. For example, answering "What teams and organizations were behind the 2025 Nobel-winning quantum computing research?" might require first identifying the winners, then finding their research areas, and finally tracing their affiliations. This step by step approach improves accuracy and completeness but increases complexity and processing time.
- A system deciding between these approaches can leverage several factors:
  - **Context Size and Quality:** Extractive readers work best with small, highly relevant passages. Generative readers or synthesizers are better suited to broad, diverse information.
  - **Question Complexity**: Simple fact-based questions call for extractive methods. Complex, multi-faceted questions benefit from generative or synthesizing methods.
  - **User Intent:** If the user wants exact quotes or citations, an extractive reader is best. For explanations or summaries, a generative reader works better. For thorough research, the synthesizer works best.
  - **Risk Tolerance**: When exact factual accuracy is essential, as in legal or medical questions, extractive readers reduce the risk of hallucination. When interpretation is important, generative readers provide value despite some risk.

- In practical systems, a specially trained model evaluates the query, chooses the best presentation strategy, and gives a plan based on the pipeline blueprint shown at the start of this chapter.

Aspect	Extractive Reader	Generative Reader	Synthesizer
Answer Type	Exact text span from source	Generated, paraphrased response	Multi-hop synthesized answer
Best for	Simple, factual queries with clear context	Complex, multi-source, explanatory	Deep research requiring multi-step reasoning
Speed	Fast	Moderate	Slower due to multiple retrieval steps
Risk of Hallucination	Minimal	Moderate	Reduced by iterative retrieval
User Intent Fit	Precise, verifiable answers	Natural, summarized explanations	Comprehensive, thorough research
Example Query	"Who won the Nobel Prize in 2025?"	"What discoveries led to the 2025 Nobel?"	"Which organizations supported the Nobel-winning research?"

- **Guardrails** are built into the pipeline to make sure answers from the generative reader and synthesizer stay accurate, safe, and relevant to their area of use. Instead of generating responses from everything the model has learned, these systems are guided to rely only on documents retrieved from approved sources. This matters because generative models can produce convincing but incorrect information, a problem known as hallucination.
  - To prevent this, guardrails make sure answers are based on retrieved evidence and stay within the application's scope. For example, if a user asks about insurance policies, the system must use only information from that specific insurance company's products, not from competitors or from general knowledge the model learned during training. This is enforced with prompt constraints, filtered retrieval sources, and checks that compare the generated answer to trusted documents.
  - In a synthesizer setup where multiple retrieval steps build a more complete answer, guardrails are even more critical. At each step the system checks that new information matches the domain and the user's original intent. If a reliable answer cannot be found, the system will indicate uncertainty instead of producing incorrect content.

- An **extractive reader based on BERT** finds the exact span in a passage that answers a question. It does not generate new text. Instead, it predicts the start and end positions of the answer in the context. Below are the steps it follows when using models such as DistilBERT or RoBERTa trained for this task:
  - **Input Format:** The question and passage are joined into one sequence before being fed into BERT (see figure on the right). `[CLS]` marks the start of the input. `[SEP]` separates the question and the passage. This arrangement lets BERT apply self-attention across both the question and the passage.
  - **Encoding:** BERT processes the combined sequence with its transformer encoder. Like the bi-encoder, it produces a contextual embedding that captures each token's meaning relative to all other tokens in the sequence.
  - **Prediction Head:** On top of BERT, a simple prediction head assigns each token in the passage a start score and an end score. These scores indicate how likely the token is to be the beginning or the end of the answer. The final answer is the span with the highest sum of start and end probabilities.
  - **Training with SQuAD (Supervised Fine-Tuning)**: The model is fine-tuned on datasets such as SQuAD (Stanford Question Answering Dataset). For each question and passage pair, SQuAD provides the correct start and end token positions. During training the model learns to reduce the error between its predicted positions and the true positions.
  - **Inference:** The model receives the question and the passage. The prediction head produces start and end scores for each token. The model extracts the answer by selecting the token span with the highest probability.




```python
from transformers import pipeline

# Load an extractive question-answering pipeline
qa_pipeline = pipeline("question-answering",
                       model="distilbert-base-uncased-distilled-squad")


# Define the question and context
question = "Who developed the theory of relativity?"

context = """
    Albert Einstein developed the theory of relativity in the
    early 20th century. It transformed our understanding of
    space, time, and gravity.
"""

# Run the extractive reader
result = qa_pipeline(question=question, context=context)
print(result)


↪{'score': 0.9970649480819702, 'start': 0, 'end': 15,
'answer': 'Albert Einstein'}
```

- A **generative reader uses a large language model such as GPT or Qwen** to create new text rather than extract it from the passage. After receiving the user's question and the retrieved documents, the model generates the answer one token at a time, based on patterns learned during training. This lets it produce natural, coherent responses instead of copying exact text spans.
- Effective prompt engineering is key to improving RAG systems. Clear prompts guide the model to stay grounded in retrieved documents, answer within the intended domain, and avoid hallucinations:
  - **Clear Task Instruction:** Explain what the model should do: answer a question, summarize, or explain.
  - **Context or Retrieved Evidence:** Include the relevant passages or documents after the instruction. The model should treat these as the only source of truth.
  - **Grounding Reminder:** Explicitly tell the model to base its answer only on the provided context and not on prior knowledge or assumptions.
  - **Domain or Scope Restriction:** Useful for business or specialized applications. It keeps the model from drifting into unrelated knowledge.
  - **Output Format Specification:** Specify how the answer should be structured: short answer, paragraph, bullet list, JSON, or other.
  - **Tone or Style Guidance (Optional):** If needed, specify formality, voice, or audience.
  - **Uncertainty Handling:** Tell the model what to do if the answer is unclear or missing.

```
Instruction to the Model:

You are given a user question and supporting
documents. Answer the question using only the
information from the documents. Do not use
outside knowledge. If the answer is not contained
in the documents, respond with "I don’t know."


User Question:

Who won the 2025 Nobel Prize in Physics, and why
were they awarded?


Retrieved Documents:

Document 1:"John Clarke, Michel Devoret, and John
Martinis won the 2025 Nobel Prize in Physics for
their groundbreaking work in quantum tunneling
and the development of superconducting circuits
used in quantum computing."

Document 2:"The Nobel Committee recognized
Clarke, Devoret, and Martinis for pioneering
experiments that enabled stable qubits and energy
quantization in electric circuits."


Response Format:

Provide a short paragraph answering the question.
```

- Implementing a **synthesizer** means coordinating multiple components in an iterative retrieval and reasoning loop so the final answer is supported and coherent. Here is how this is typically done:
  - The user's question is first transformed or expanded to optimize retrieval.
  - A retriever/reranker fetches a set of relevant documents or passages based on the original query.
  - A reader or reasoning module processes these documents to extract important entities, concepts, or partial answers that can guide the next retrieval step.
  - The system uses the extracted information to create refined or follow-up queries that retrieve additional documents providing context or details missing from the first pass.
  - After collecting several document sets, a generative model combines the information, merges facts, and clears up contradictions to produce a coherent answer. Prompt engineering often guides the model to base its answers on the retrieved evidence.
  - Apply guardrails throughout the process to ensure the model uses only supported information. Detect and reject hallucinated or unsupported claims. Handle uncertainty gracefully, and when evidence is insufficient say I do not know.
  - Finally, the synthesizer produces the answer in the required format, matching the user's intent and the domain constraints.


```
{
  "question": "What did John Clarke win the Nobel Prize for and
              where does he work?",
  "reasoning_steps": [
    {
      "hop": 1,
      "query": "Who won the Nobel Prize in Physics in 2025?",
      "answer": "John Clarke, Michel Devoret, and John Martinis",
      "confidence": 0.95
    },
    {
      "hop": 2,
      "query": "What did they win for?",
      "answer": "Macroscopic quantum mechanical tunneling and …",
      "confidence": 0.92
    },
    {
      "hop": 3,
      "query": "Where does John Clarke work?",
      "answer": "University of California, Berkeley",
      "confidence": 0.88
    }
  ],
  "synthesized_answer": "John Clarke won the 2025 Nobel Prize in
    Physics for his groundbreaking discovery of macroscopic
    quantum mechanical tunneling and energy quantization in an
    electric circuit. He is affiliated with the University of
    California, Berkeley, where he works as a professor in the
    Department of Physics.",
  "supporting_facts": [
    {
      "fact": "Won Nobel Prize for quantum tunneling discovery",
      "source": "Nobel Prize Official Website",
      "confidence": 0.95
    },
    {
      "fact": "Works at UC Berkeley",
      "source": "UC Berkeley Physics Department",
      "confidence": 0.88
    }
  ]
}
```

- **Citations:** The ability to provide citations is a key strength of Retrieval-Augmented Generation (RAG) systems compared to large language models alone. By linking answers to source documents, citation-aware RAG systems deliver more transparent, accurate, and trustworthy responses.
  - The citation process is integrated into the retrieval and reading pipeline to preserve source information. During retrieval, when relevant documents or passages are found, their metadata is kept. This metadata typically includes a unique passage ID for internal tracking, a source document identifier, the document URL or physical location, a retrieval relevance score, and the passage's original text.
  - As the reader component processes retrieved passages, either by extracting exact text spans or by generating synthesized answers, the system records which passages contributed to the final response. This tracing is essential for creating an auditable link between the answer and the underlying evidence.
  - The final output of a citation-aware RAG system usually includes the answer plus structured citation details (e.g., JSON format). For example, an answer naming the 2025 Nobel Prize in Physics laureates would include a citation to the official Nobel Prize website, the exact passage used, and the access date.
  - This architecture offers several key benefits. First, it enables source verification. Users can follow citations to confirm an answer and read the original material themselves. Second, citation transparency helps build user trust because answers are shown to be based on verifiable evidence rather than unsupported claims. Third, this traceability lets developers monitor and refine the retrieval and reading components step by step, so the system becomes more accurate and reliable over time.

```
{
  "answer": "John Clarke, Michel H. Devoret, and John M. Martinis",
  "confidence": 0.95,
  "citation": {
    "passage_id": "passage_001",
    "source": "Nobel Prize Official Website",
    "url": "https://nobelprize.org/prizes/physics/2025/",
    "original_text": "The 2025 Nobel Prize in Physics was awarded to John Clarke,
                      Michel H. Devoret, and John M. Martinis for their groundbreaking
                      discovery of macroscopic quantum mechanical tunneling and energy
                      quantization in an electric circuit.",
    "retrieved_date": "2025-10-07"
  }
}
```

8.5 Literature and Links

- A. T. Kalai, O. Nachum, S.S. Vempala, E. Zhang, **Why Language Models Hallucinate**, https://cdn.openai.com/pdf/d04913be-3f6f-4d2b-b283-ff432ef4aaa5/why-language-models-hallucinate.pdf?utm_source=chatgpt.com
- Lewis, Patrick, et al. **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks**. NeurIPS 2020. https://arxiv.org/abs/2005.11401
- Karpukhin, Vladimir, et al. **Dense Passage Retrieval for Open-Domain Question Answering.** EMNLP 2020. https://arxiv.org/abs/2004.04906
- Xiong, Linyong, et al. **Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval**. https://arxiv.org/abs/2007.00808
- Reimers, Nils, and Iryna Gurevych. **Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks.** EMNLP 2019. https://arxiv.org/abs/1908.10084
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.** NAACL-HLT, 2019. https://arxiv.org/abs/1810.04805
- Fernando Diaz, Bhaskar Mitra, Nick Craswell, **Query Expansion with Locally-Trained Word Embeddings.** https://arxiv.org/abs/1605.07891
- **Stanford Question Answering Dataset (SQuAD).** A benchmark dataset commonly used to train and evaluate extractive question-answering models like BERT. https://rajpurkar.github.io/SQuAD-explorer