RAG

Team:
Sri Iyengar (si2468)
Radhika Patel (rpp2142)
Anushka Pachaury (ap4617)

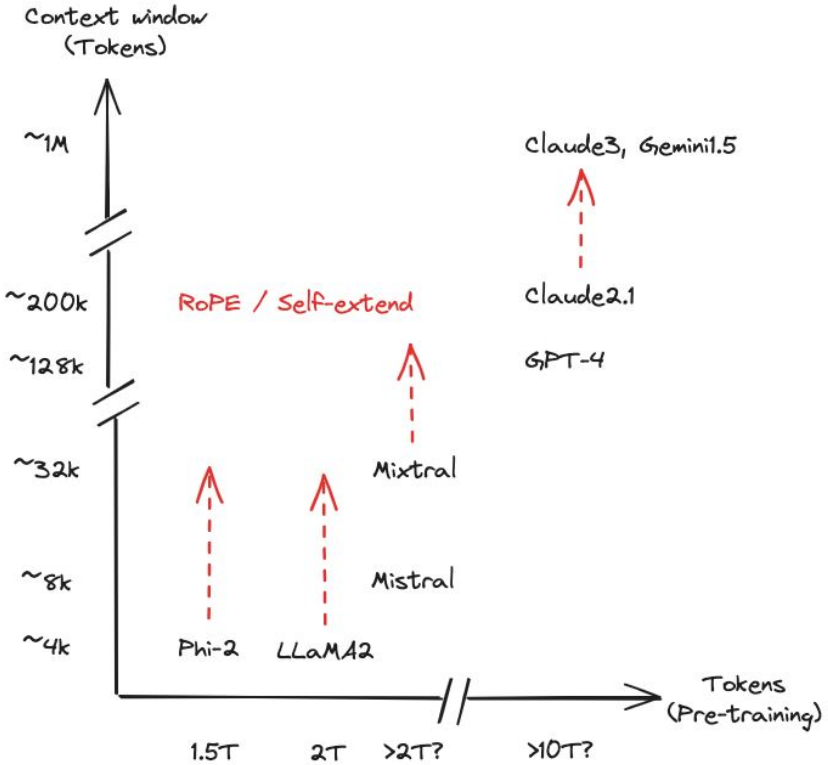
EECS E6691 Advanced Deep Learning, 2025 Spring

Outline of the Presentation

- Definitions / Motivation for RAG
- Basic RAG steps
  - Indexing
  - Retrieval
  - Generation
- Query Translation
- Routing
- Query Construction
- Indexing
- Retriever
- Agentic Retrieval Workflows (Optional)
- Memory and Modality in RAGs

LLMs

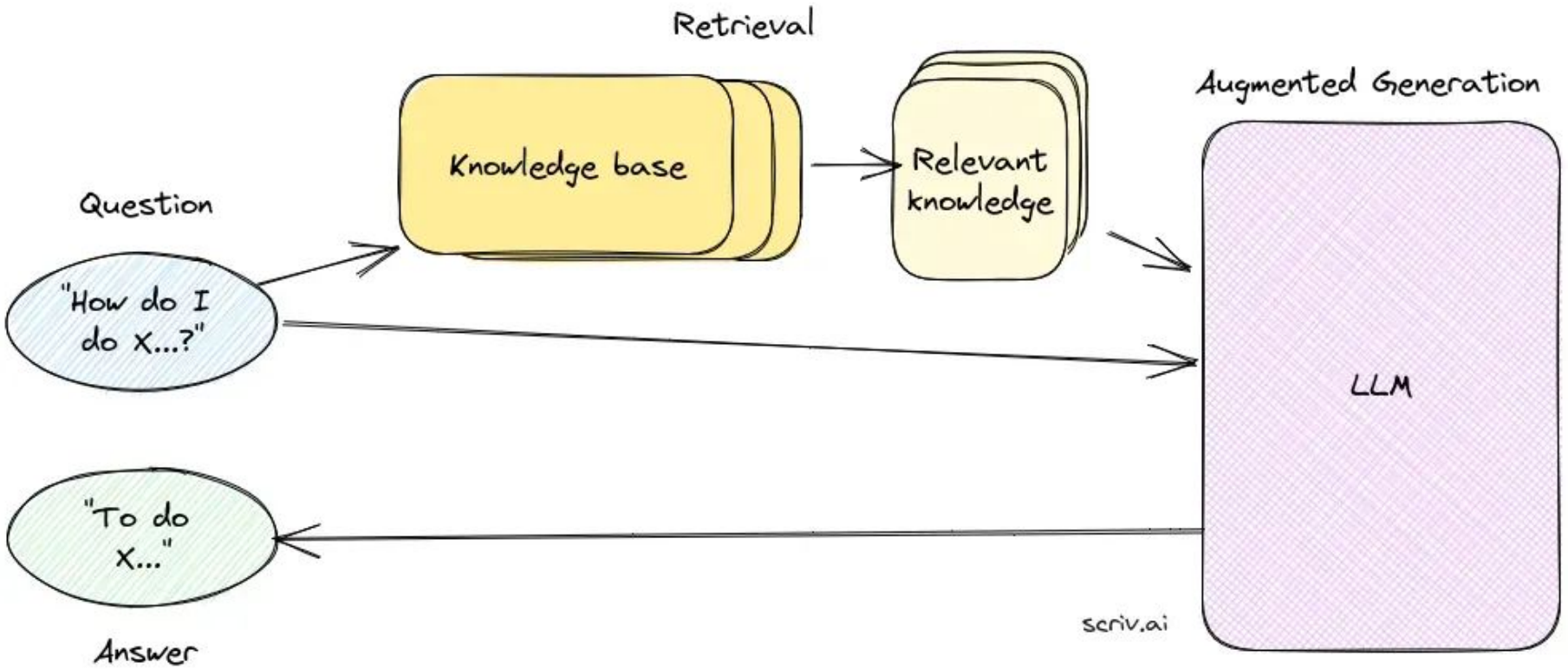
- Available Data
  - ~95% private
  - ~5% public -> used to train LLMs
- Increasing Context Window Sizes
  - Ability to feed more information per inference



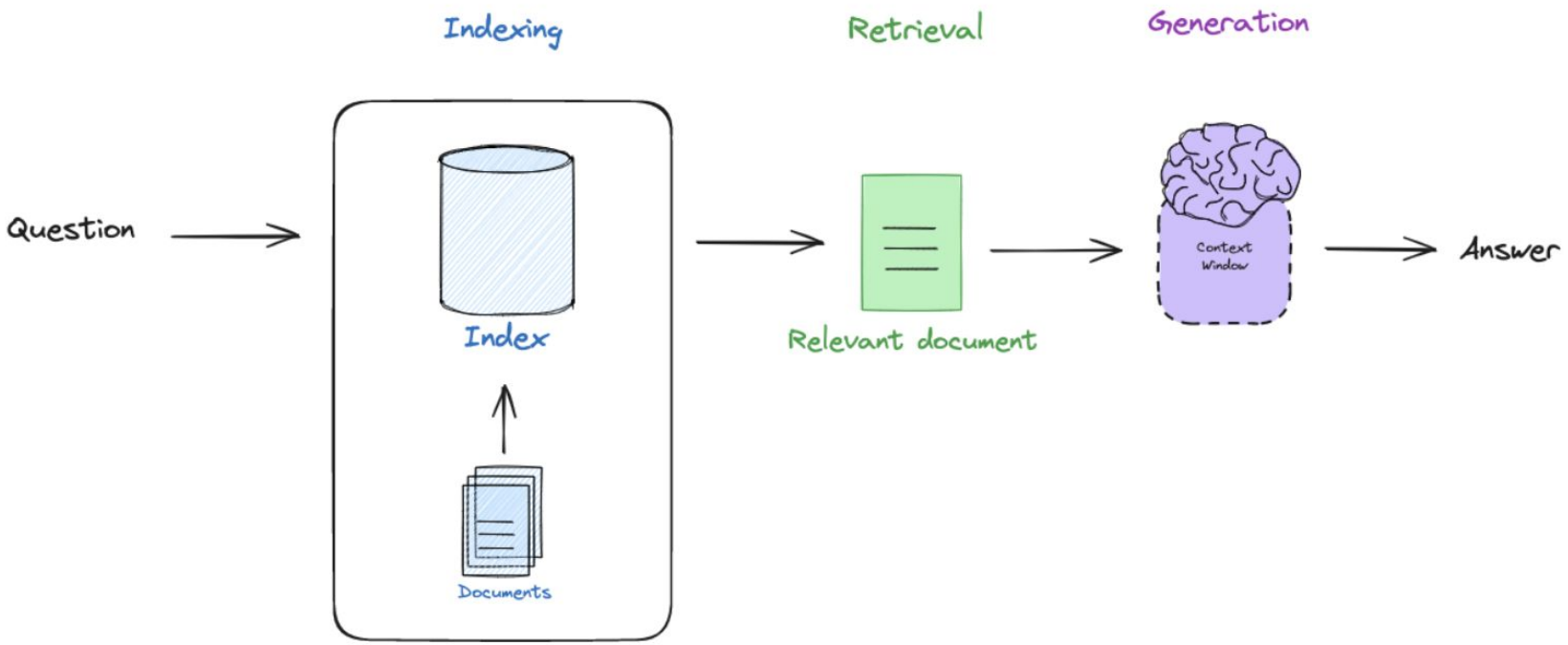
RAG Motivation

- Retrieval-Augmented Generation
- Limitations of Parametric Models
  - Difficult to update/revise knowledge in parameters
  - Hallucination
  - Explainability (why is something true)
- Generalization
  - Domain specific knowledge without fine tuning
- Best of both worlds
  - Extractive models (reading documents)
  - Parametric models (rely on internalized knowledge)

Abstraction



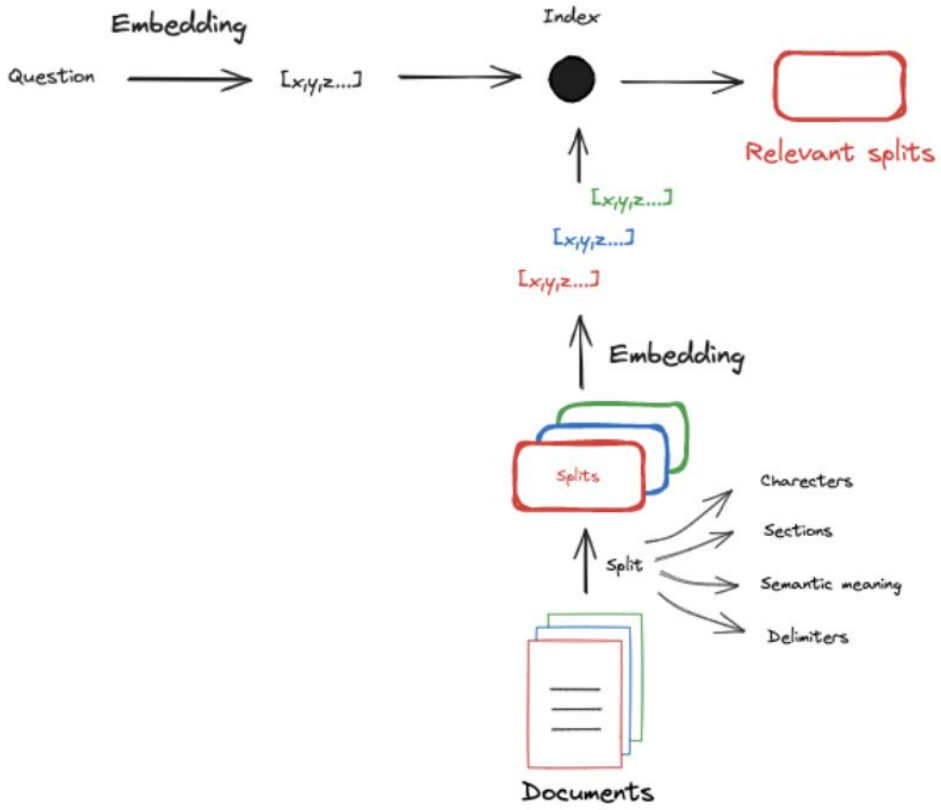
3 Main Steps


Indexing
Retrieval
Generation
Question
Index
Documents
Relevant document
Context Window
Answer

Indexing

- Pre-processing step - done before any inference
- Create a searchable representation of a large text corpus
- Retriever should be able to fetch most relevant documents for an input query
- This is typically fixed and not updated during training
- Pipeline
  - Corpus -> m-token documents
  - documents -> document encoder (i.e. BERT) -> d-dimensional vector embedding representing semantic context of document
  - Built to efficiently extract documents with high relevance to input query from user

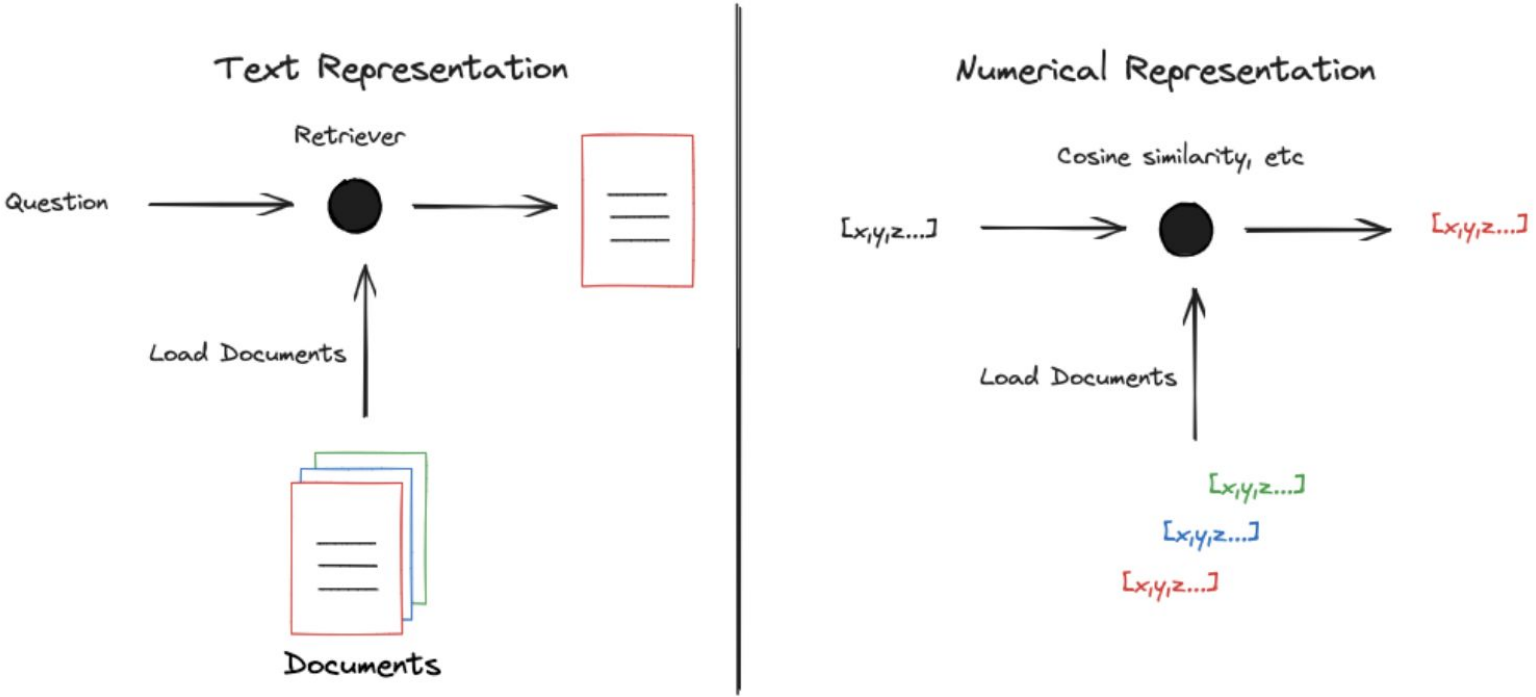
Indexing


Embedding
Question
[x,y,z...]
Index
Relevant splits
[x,y,z...]
[x,y,z...]
[x,y,z...]
Embedding
Splits
Charecters
Sections
Split
Semantic meaning
Delimiters
Documents

Retrieval

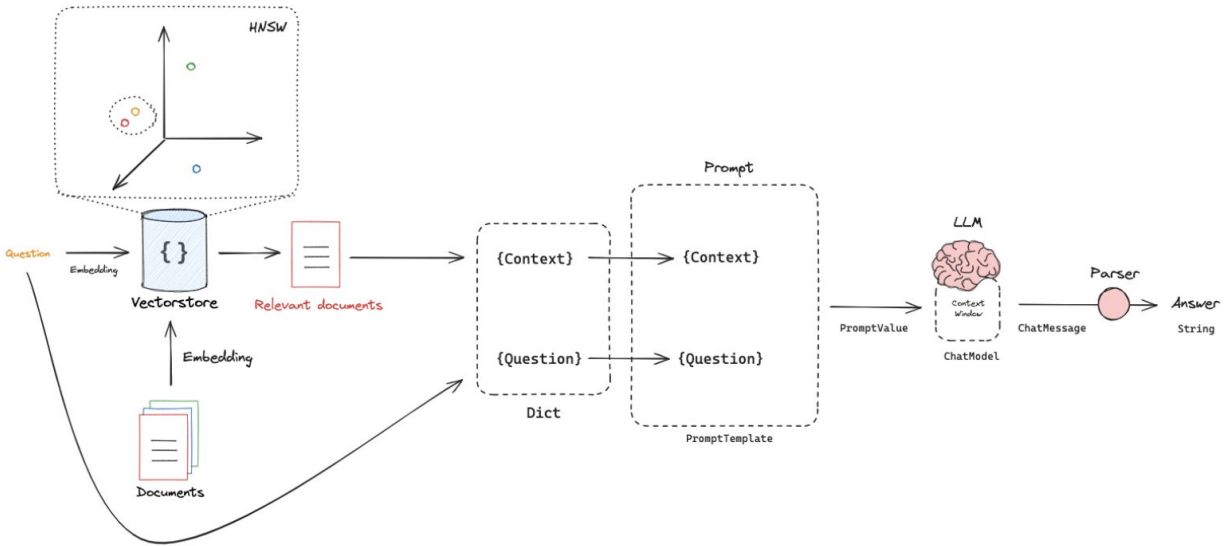
- Fetch the k most relevant documents from our database for a query
- Similarity search against indexed corpus
  - Documents (N of them) Z encoded as $d(Z) = \text{BERT}_d(Z) \in \mathbb{R}^{DxN}$
  - Query x encoded as $q(x) = \text{BERT}_q(x) \in \mathbb{R}^{Dx1}$
  - Document scores P are computed for each document as follows: $P = (d(Z)^T q(x)) \in \mathbb{R}^{Nx1}$
  - Get k most relevant documents (knn)
- Top-k?
  - Tunable parameter
  - Consider accuracy, efficiency, and context length -> higher k means less context available for previous query tokens as document_size * k + query_size <= context length

Retrieval


Text Representation
Retriever
Question
Load Documents
Documents
Numerical Representation
Cosine similarity, etc
[x,y,z...]
[x,y,z...]
Load Documents
[x,y,z...]
[x,y,z...]
[x,y,z...]

Generation

- We have our query and its K most relevant documents
- Formulate a new prompt that consists of our documents and the query -> input to LLM (BART) for augmented generation



Generation: RAG-Models

RAG-Sequence

$$\hat{y} = \arg\max_y \sum_{i=1}^{k} p(z_i \mid x) \cdot p(y \mid x, z_i)$$

- Computationally cheaper
- Returns sequence with maximum expected score across documents
- Used more in production

RAG-Token

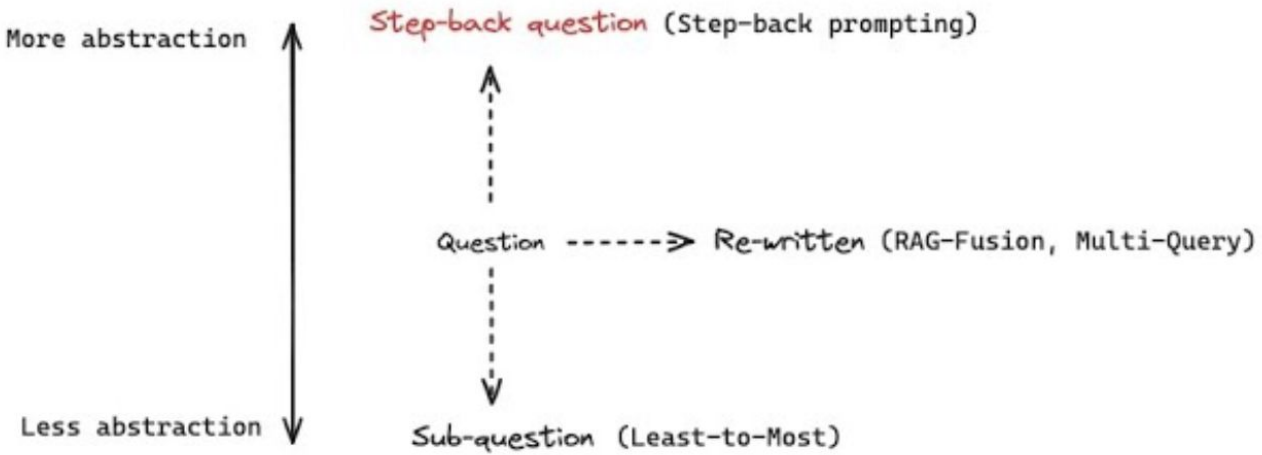
$$\hat{y} = \arg\max_y \prod_{t=1}^{T} \left(\sum_{i=1}^{k} p(z_i \mid x) \cdot p(y_t \mid y_{<t}, x, z_i) \right)$$

- Computationally expensive
- Returns sequence with max token-wise expectation across documents
- Used in benchmarking

Query Translation Motivation

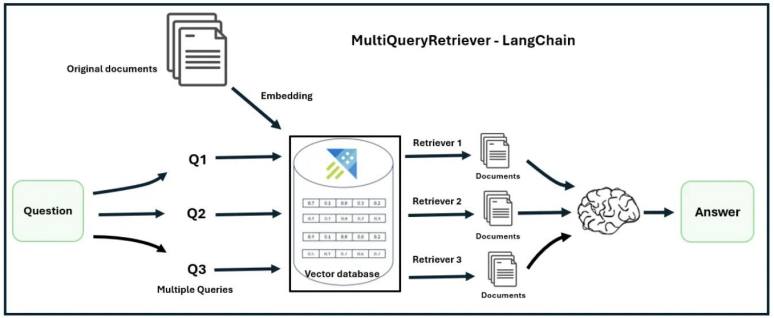
- Queries are created by the user -> room for ambiguity / misleading information
- Motivated to “translate” the input query into a representation better suited for pulling relevant documents from the database
- Examples
  - 1)
    - Query: “Tell me about whoever invented the telephone”
    - Appropriate Conversion: “Alexander Graham Bell Biography”
  - 2)
    - Query: “What substance did Fleming discover that kills bacteria?”
    - We might receive a document like: “Bacteria are commonly killed by ….” -> irrelevant
    - We want documents like: “Penicillin was discovered by Alexander Fleming in 1928…”

Query Translation



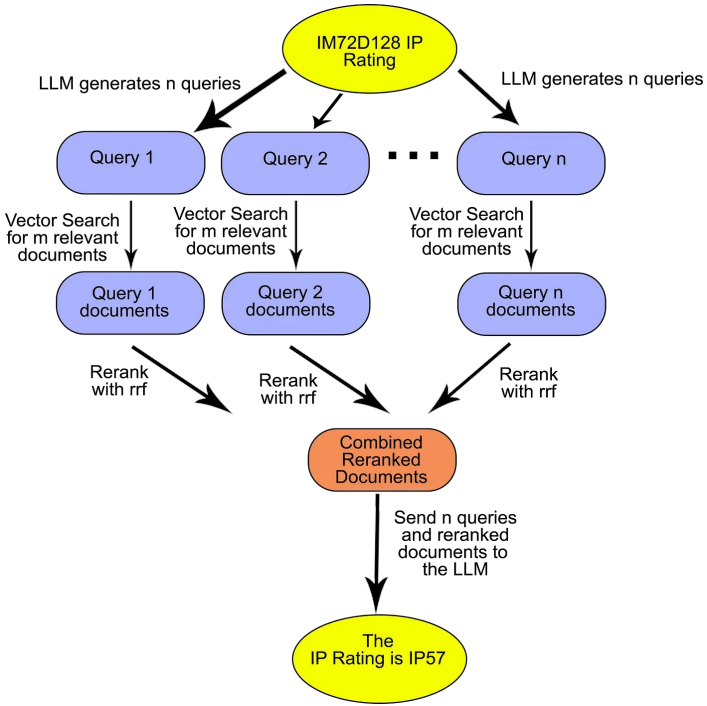
Query Translation - Rewriting Queries (Multi-Query)

- Multi-Query
  - Rewrite query into N different queries with nuanced differences using an LLM
  - Each question yields M documents in retrieval
  - unique(N*M) documents are passed as context for generation step
  - Rewritten questions might be closer to relevant documents in the search space
  - Cons?
    - O(N*M) is large for context window size
    - Humans need to make design choices for (N, M), and also decide how to synthesize documents
    - Reliant on generally-tuned LLM for translating initial query (Does it have the required domain-specific knowledge)
    - High Cost / Latency
    - Equal weight to all unique documents is suboptimal for performance



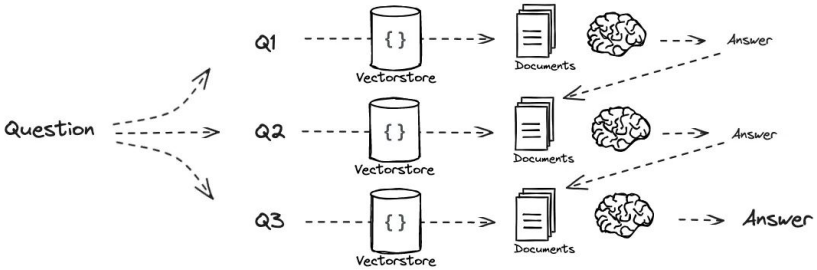
Query Translation - Rewriting Queries (RAG-Fusion)

- Rag Fusion
  - Smarter version of Multi-Query
  - N queries from LLM yield M documents each like before
  - N*M documents are ranked according to RRF function
    - RRF(d) = $\sum 1 / (k+\text{rank}_q(d))$ across all generated queries q
    - k is hyperparameter smoothing constant
    - $\text{rank}_q(d)$ - rank of document d in subquery q's retrieved document list
  - Choose highest ranking documents to send to the LLM



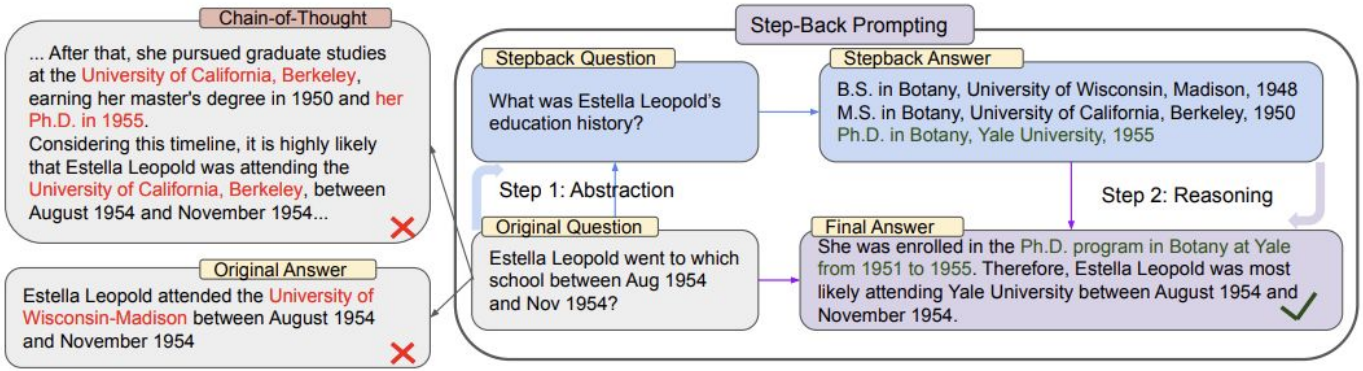
Query Translation - Sub-Queries (Query Decomposition)

- Query Decomposition
  - Don’t rewrite query, break it down (with LLM)
  - What is the population of the capital of the country where the Great Wall is located?
    - Where is the Great Wall? (Q1) -> China (A1)
    - What’s China’s capital (Q2) -> Beijing (A2)
    - What’s the population of Beijing (Q3) -> Answer (A3)
  - Retrieval/Generation for each subquestion
    - One at a time? (sub-questions are dependent)
    - Parallel (If sub-questions are independent)
  - Synthesizing Final Answer
    - Substitute A1 in Q2, A2 in Q3, to get A3 (CoT)
    - Use A1, A2 as “document” for Q2, Q3
    - Use LLM: Given the A1, A2, A3, to Q1, Q2, Q3, answer original question



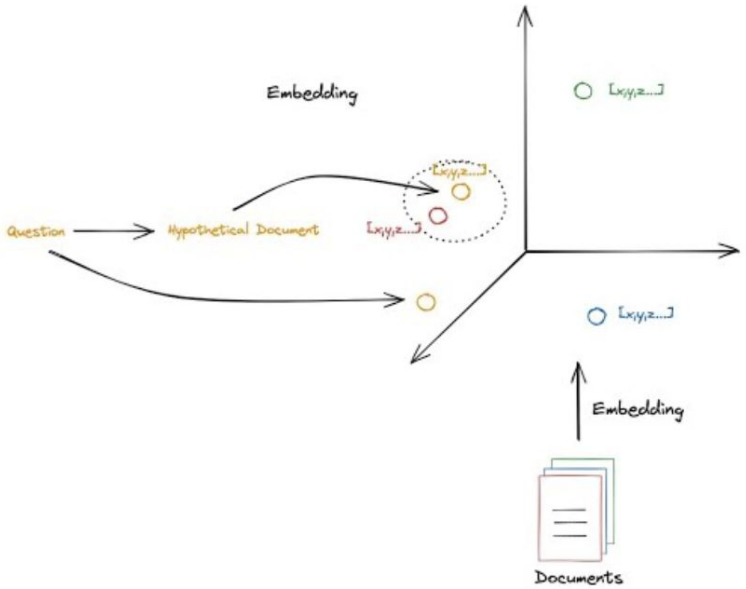
Query Translation - Stepback Prompting

- Use an LLM to ask a more abstract, higher level query (step backwards)
- Retrieve relevant documents for higher level query and combine it with the documents from the original query to get a more informed answer
- In some cases, can retrieve more important information than we can by breaking down the question



Query Translation - HyDE

- Hypothetical Document Embeddings
- Documents are usually well-phrased, informed, and dense. Queries are not well-written and subject to irrelevant information and falsities
  - Embedding a query and a document into the same vector space might not make much sense practically if they are so different at their core
  - Can we transform the query into a document using an LLM, and then use that document to generate the embedding?

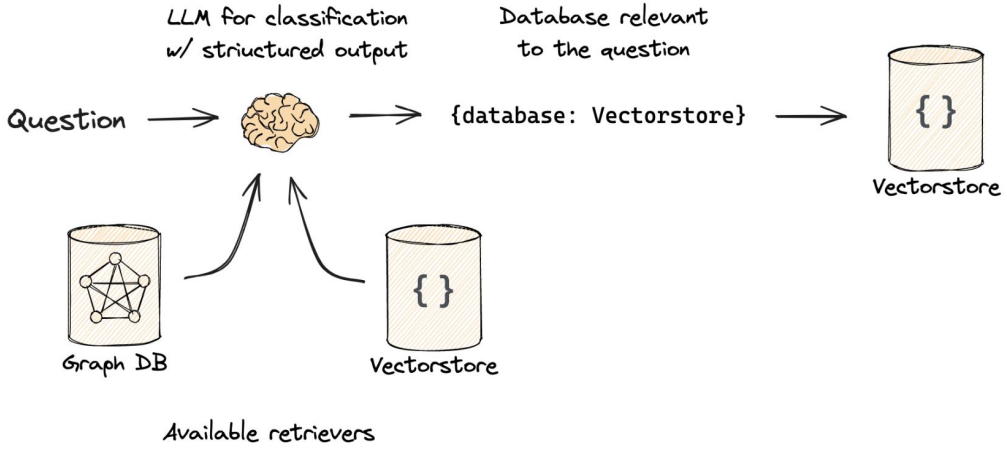


Routing

- Dynamically determines the best component (retriever, tool or prompt) to handle a query
- Enables task-specific handling within multi-retriever RAG systems
- Supports semantic, logical, or metadata-based decision making
- Replaces hard coded logic with embedding similarity or LLM-based classification
- Outputs structured routing decisions (eg., **{target: Vectorstore}** or **{prompt_id: 2}**)
- Reduces retrieval latency and compute by avoiding irrelevant pathways
- Forms the foundation for advanced strategies like Logical Routing and Semantic Routing

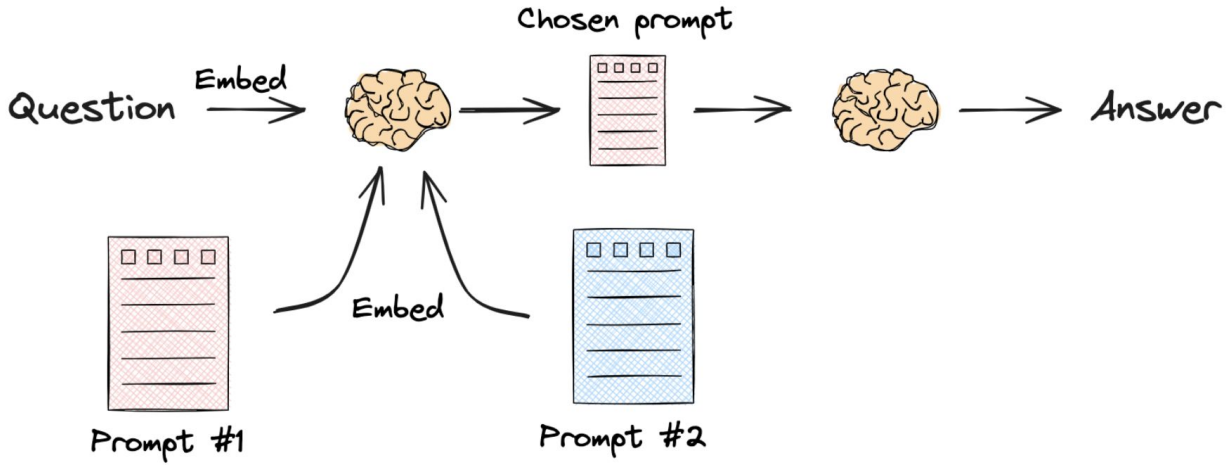
Logical Routing

- Routes queries to optimal retrievers using LLM-based classification
- Outputs structured routing decisions (eg., {database: Vectorstore})
- Supports heterogeneous backends: vector DBs, graph DBs, APIs
- Reduces latency and boosts retrieval accuracy by filtering pathways
- Enables modular, multi-retriever RAG architecture at scale



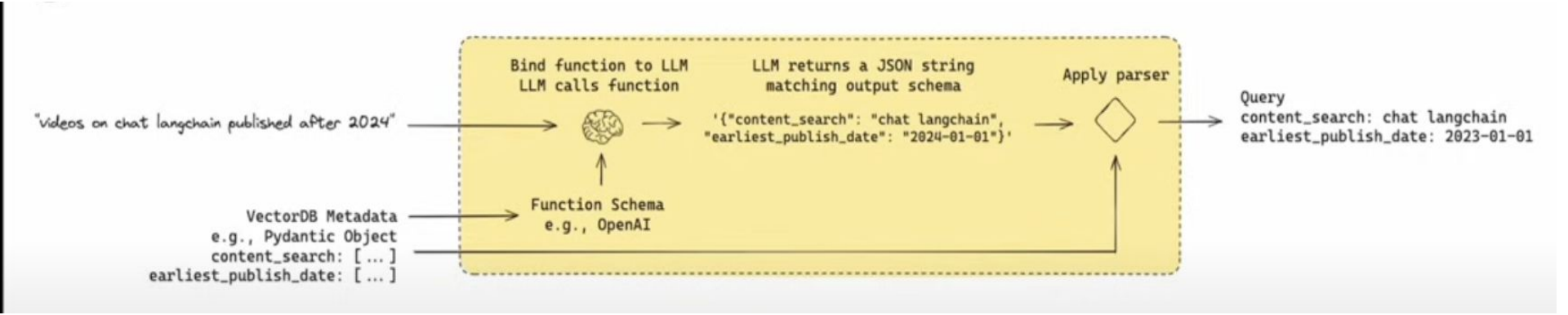
Semantic Routing

- Routes queries to the most semantically aligned prompt template or chain
- Embeds both the query and available prompts using a shared encoder
- Selects the prompt with the highest embedding similarity to the query
- Enables context-aware prompt selection without hardcoded rules
- Ideal for multi-task agents and RAG systems with specialized behaviors



Query construction

- Converts user-friendly language queries into structured retrieval instructions
- Uses an LLM to extract fields aligned with the retriever's metadata schema
- Outputs a JSON-formatted query matching the expected filter format (eg., {"content_search": …., "earliest_published_date": …})
- Enables precise filtering over vector DB metadata (eg., date ranges, tags, domains)
- Improves retrieval quality by aligning the query format with how the vector DB was indexed

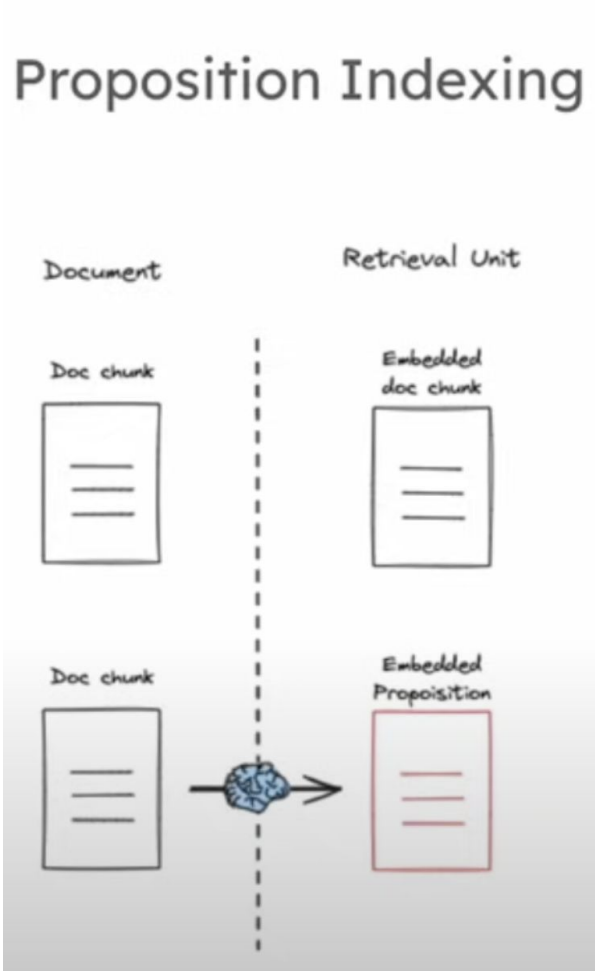


Indexing (Multi-Representation)

- Stores multiple semantic views of the same document to improve retrieval quality
- Replaces or supplements raw chunk embeddings with summaries, propositions, or captions
- Each representation is embedded and indexed separately in the vector store
- Enables queries to match the most relevant abstraction, not just surface level text
- Supports downstream techniques like proposition-leve matching and summary-based retrieval
- Improves retrieval recall and precision by exposing different aspects of document meaning

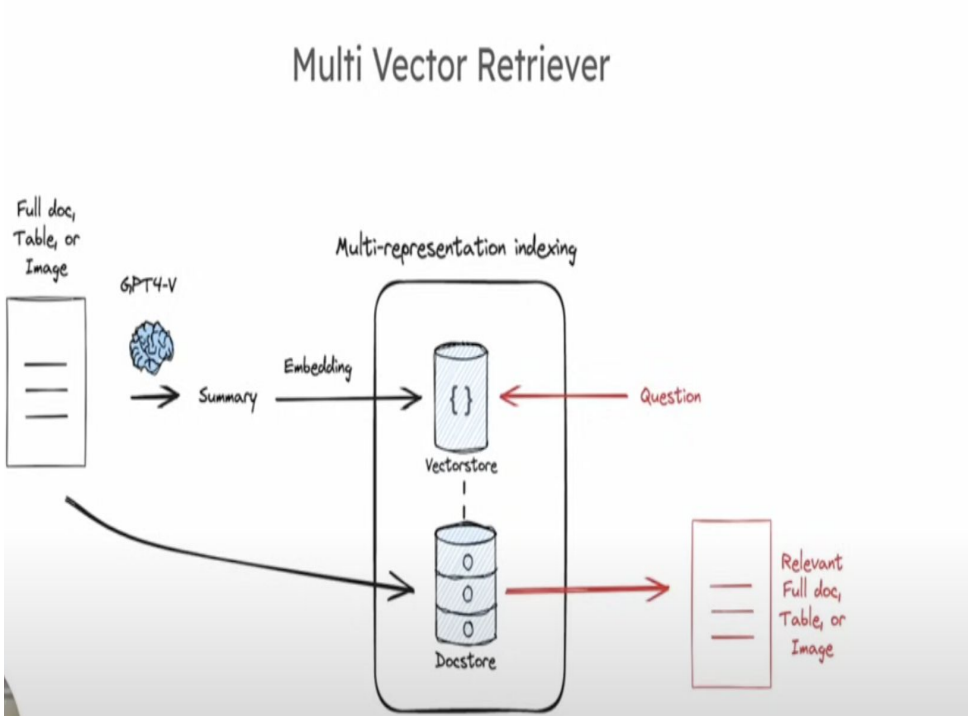
Multi-Representation : Proposition Indexing

- Enhances tradition chunk based indexing by extracting finer-grained semantic units
- Each document chunk is processed to identify individual propositions or claims
- Propositions are independently embedded and stored as distinct retrieval units
- Improves retrieval precision by matching queries to specific claims, not generic chunks
- Supports reasoning-intensive queries more efficiently than full chunk embeddings



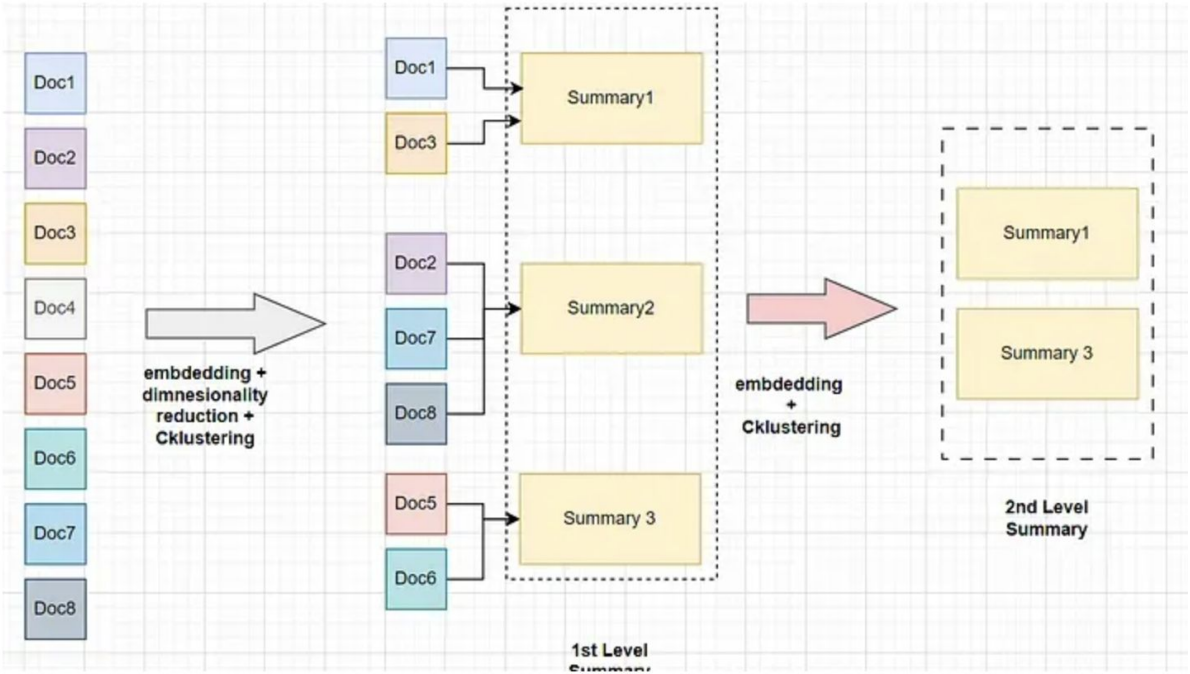
Multi-Representation: Multi-Vector Retriever

- Stores multiples representations (eg summaries, captions) for the same document
- Summarization is done through an LLM, embedding only the abstracted representation
- The vectorstore holds the summary embeddings for semantic search
- The doctstore holds the full original content (eg full doc, table, image) for context
- Retrieval returns full docs by matching query embeddings to summary vectors then fetching the linked source



Indexing (RAPTOR)

- Organizes raw documents into a hierarchy of semantic clusters
- Applies recursive summarization: each level summaries the level beneath it



Indexing (RAPTOR) continued

- Starts from raw document “leaf nodes” and builds up to cluster and root summaries
- Summaries at each level are embedded and indexed independently
- Embeddings represent aggregated meaning across multiple related documents
- Stored in the vectorstore for efficient, high-level semantic analysis retrieval
- Root summary captures the global abstraction of full corpus
- Ideal for retrieving multi-documents insights, not just local texts

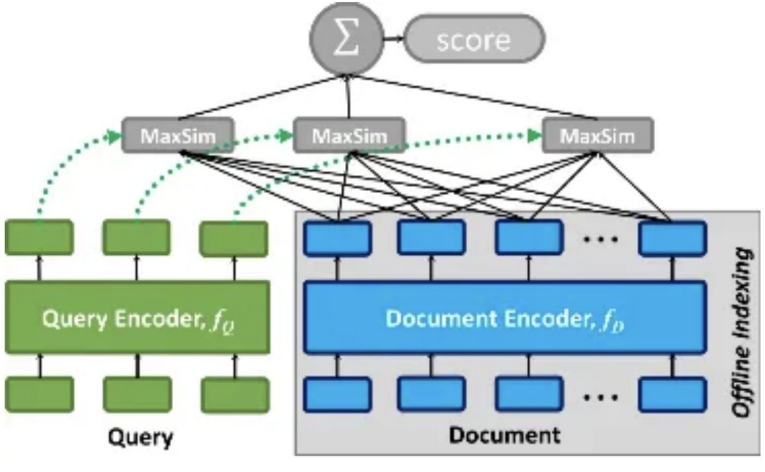
Indexing (ColBERT) - Phase 1 (Offline)

- Stores token-level embeddings for each document instead of whole doc-vectors
- Splits documents into tokens → each token is contextually encoded with BERT
- Produced a vector for each token using ColBERT encoder (eg 768-dim)
- Associates embeddings with token positions and document IDs
- Stores them in a ColBERT index for fast late interaction
- Enables fine-grained token access instead of coarse document-level similarity
- Great for semantic precision in retrieval, especially for long-form content

Phase-2 (Online)

- Query is tokenized and passed through same ColBERT encoder
- Each query token gets its own vector embedding (context aware)
- Late interaction: each query token finds most similar doc taken
- Score = sum of max similarities across all query tokens
- Enables highly granular semantic alignment between query and docs

$$\text{Score}(\text{doc}) = \sum_{i=1}^{m} \max_{j} (q_i \cdot d_j)$$



Retrievers

- This is the core component of Retrieval-Augmented Generation (RAG) systems
- Takes an embedded query and returns the most relevant documents from a large corpus
- Operates over a pre-indexed set of document representations (vectors)
- Enables the LLM to access external knowledge beyond its training data
- Two main types of retrievers:
    - Non-neural: Based on static embeddings + vector similarity search
    - Neural: End to end trainable models that learn retrieval behavior
- Choice of retriever impacts latency, retrieval accuracy and domain adaptability
- Can be combined with rerankers, metadata filters, or hybrid (sparse+dense) strategies or optimal performance

Neural Retriever - How it works

- Query encoder: Converts a user question into a dense vector (often BERT-based)
- Document encoder: Converts each document (eg chunk, proposition) into its own vector
- Can be:
  - Same model (shared weights) or
  - Separate encoders (like in DPR)
- Training objective: Contrastive Learning:
  - Given a positive pair: (q, p) → relevant query-doc
  - And several negative par: (q, n)
- Loss function: InfoNCE:

$$\mathcal{L} = -\log\left(\frac{e^{q\cdot p}}{e^{q\cdot p} + \sum_{i=1}^{k} e^{q\cdot n_i}}\right)$$

Neural Retriever - Inference Time

- User query is passed through the trained query encoder
  - Gives query vector $\mathbf{q} \in \mathbb{R}^{\mathbf{d}}$
  - The system has an index of embedded documents, built offline
    - $\rightarrow \mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n] \in \mathbb{R}^{\mathbf{dxN}}$
- Compute Similarity:
  - $P = D^T q$
- This gives us N x 1 vector of similarity scores
- The system returns the Top-k documents with the highest similarity scores
- Common models: DPR (Dense Passage Retriever), ColBERT, GTR, ANCE

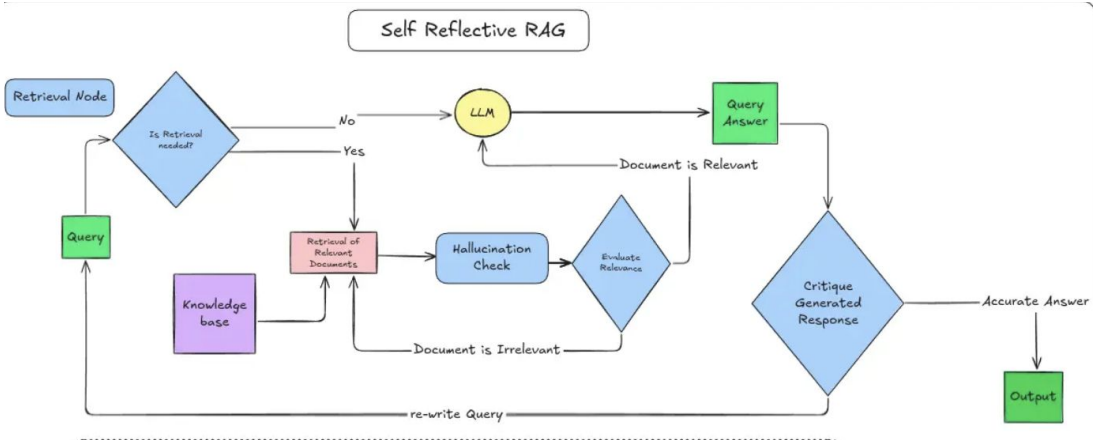
Non-Neural Retriever

- Uses pretrained embedding models (eg OpenAI, BGE, SBERT) to convert docs and queries into vectors
- Stores embeddings in a vector index (eg FAISS, Quadrant, Pinecone)
- Retrieval is performed using Approximate Nearest Neighbor (ANN)
- Common ANN algorithms: HNSW, IVF, PQ, Flat
- Does not require training - all logic happens after embedding
- Can be combined with rerankers for improved quality (eg BERT-based cross-encoders)

Agentic Retrieval Workflows

Motivation:

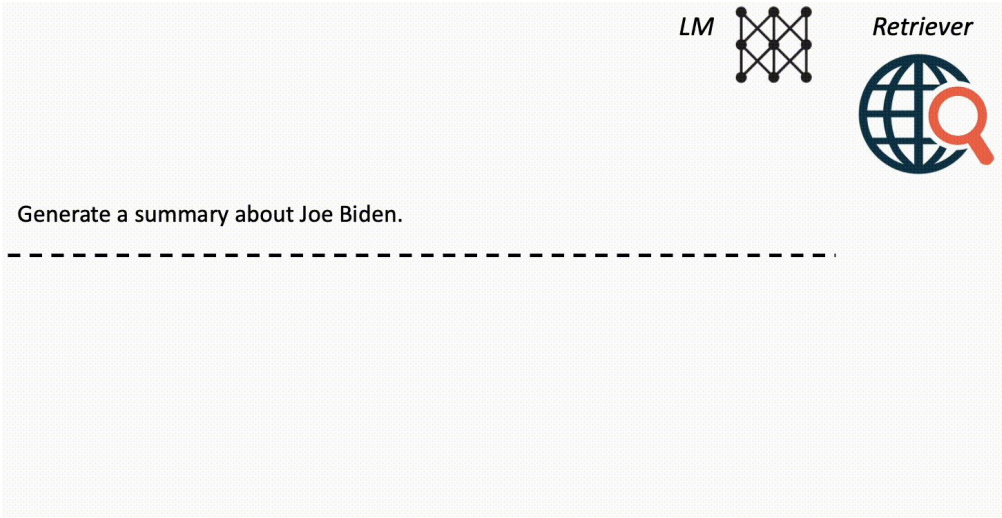
- Traditional RAGs:= Static Pipeline: Index ⟶ Retrieve ⟶ Generate
- Agentic Retrieval := Dynamic Retrieval
  - LLMs control :=
    - WHEN to retrieve
    - HOW MUCH to retrieve
    - WHETHER to retrieve again based on existing retrievals, generations and query complexity



- Let LLM choose between various tasks in pipeline by defining transitions to each task in agent's "State Machine" Graph Model
- Agentic systems think, act, observe and adapt based on graph state

Active RAGs

- "**Proactive Agency**": Agent actively decides when and what to retrieve using a prediction of the upcoming sentence to anticipate future content



1) Confidence Scoring Module: Input Query → calculate confidence estimation based on
   - entropy of logits
   - question typing
   - perplexity scoring

2) Conditional Retriever:
   - High confidence = skip retrieval
   - Low confidence = trigger retrieval
     - Adaptive Top-K retrieval: lower confidence = make k higher

Active RAGs

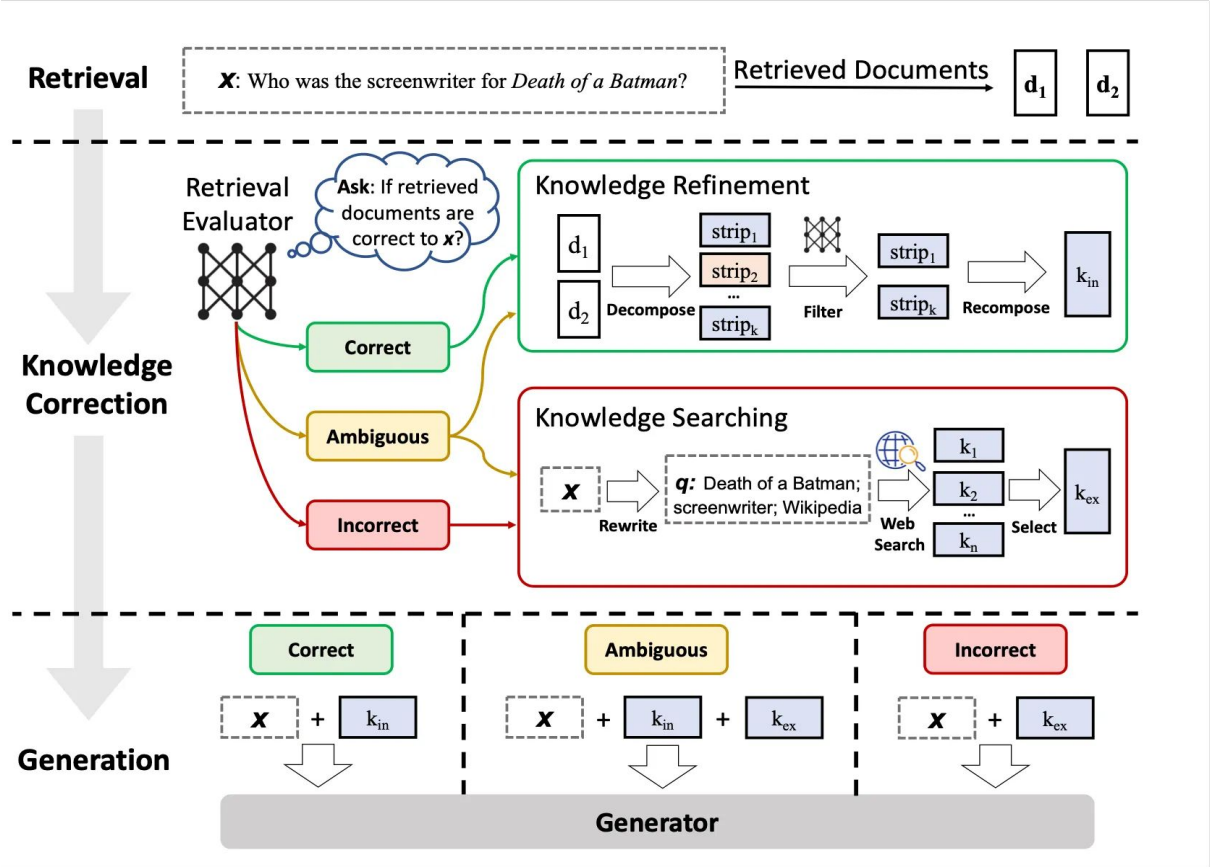
2) Retriever Typing:
    - Use fast retrievers (BM25) for low-cost vector search, for shallow queries
    - Deep retrievers like ColBERT, HNSW for semantic-heavy queries

Key takeaway: **Retrieval is Conditional**
- Active RAG Agent decides when and if to retrieve BEFORE fetching docs
    - If current context has sufficient confidence score → skip
    - Saves time, cost and reduces over-retrieval

Query → Confidence Estimator ? Low: Retrieve; High: Generate → Feedback

Corrective RAGs

- **“Reactive Agency”:** Fix after feedback
- LLM self-critiques by calculating confidence score AFTER retrieving docs
- Knowledge Refinement: Filter out irrelevant chunks/strips from retrieved docs
- Knowledge Searching: Rewrite query and perform web search/external retrieval



Corrective RAGs

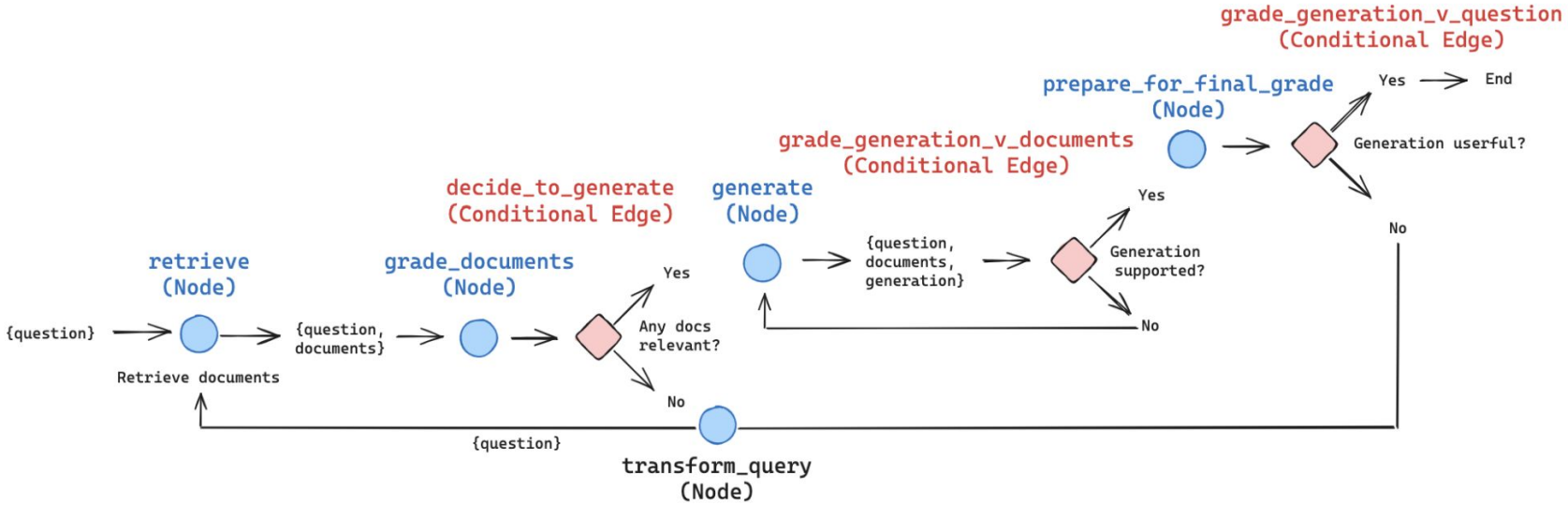
Algorithm 1: CRAG Inference

Require : E (Evaluator), W (Query Rewriter), G (Generator)
Input : x (Input question), $D = \{d_1, d_2, ..., d_k\}$ (Retrieved documents)
Output : y (Generated response)

$score_i$ = E evaluates the accuracy of each pair (x, d_i), $d_i \in D$
Confidence = Calculate and give a final judgment based on $\{score_1, score_2, ...score_k\}$
`// Confidence has 3 optional values: [CORRECT], [INCORRECT] or [AMBIGUOUS]`
if ***Confidence*** == *[CORRECT]* **then**
    Internal_knowledge = Knowledge_Refine(x, D)
    k = Internal_knowledge
else if ***Confidence*** == *[INCORRECT]* **then**
    External_knowledge = Web_Search(W Rewrites x for searching)
    k = External_knowledge
else if ***Confidence*** == *[AMBIGUOUS]* **then**
    Internal_knowledge = Knowledge_Refine(x, D)
    External_knowledge = Web_Search(W Rewrites s for searching)
    k = Internal_knowledge + External_knowledge
end
G predicts y given x, k

- Focuses on fidelity and completeness over efficiency of retrieval
- Query → Retrieve → Critique Retrieval→ Info missing ? Re-retrieve → Generate

Self-Reflection and Reasoning



- Reasoning in RAG is evolving: from static retrieval to intelligent, self-corrective, and adaptive cycles
- Example - Implementing cyclic reasoning with query rewriting in Self-RAGs for improved retrieval if low-quality results produced at any stage (diagram)

Self Reflection and Reasoning: ReAct Framework

```
[User Query] ← What is the total distance traveled by the
Mars Rover Perseverance as of today, and what is that in
miles?
     |
[Thought] → "I need the total distance Perseverance has
traveled, likely requires a web search."
     |
[Action] → Web Search ("Perseverance rover distance traveled
2025")
     |
[Observation] → "As of April 2025, Perseverance has traveled
20.15 kilometers."
     |
[Thought] → "Convert 20.15 kilometers to miles."
     |
[Action] → Calculator (20.15 km × 0.621371)
     |
[Observation] → "20.15 km = 12.52 miles"
     |
[Final Answer] → "Perseverance has traveled 20.15 kilometers
(12.52 miles) as of April 2025."
```

- Using ReAct Prompting:
  Enables dynamic reasoning in RAGs via Thought ⟶ Action ⟶ Observation chain-of-thought reasoning

- Instead of relying on confidence thresholds and predefined transitions in agent graphs, give LLM autonomy to decide what action to take next (perform web search, fetch calculator, invoke specific tools, etc) beyond just retrieval

Memory and Modality Challenges in RAGs

- LLMs are limited by the size of their context windows (4k-128k traditionally)
- Standard RAGs retrieve short passages (100-200 tokens), but:
  - Retrieves needs to scan millions of small chunks/units in vector score
  - The reader (LLM) processes small, fragmented contexts → weakens ability to capture long range contexts
  - Results in semantic incompleteness
- Emerging needs:
  - Need to handle longer documents, sessions,or conversation history
  - Integrate images, audio, videos or structured data
- Recent innovations aim to expand RAG's capacity beyond just short-term text retrieval

LongRAGs

Long Retriver:

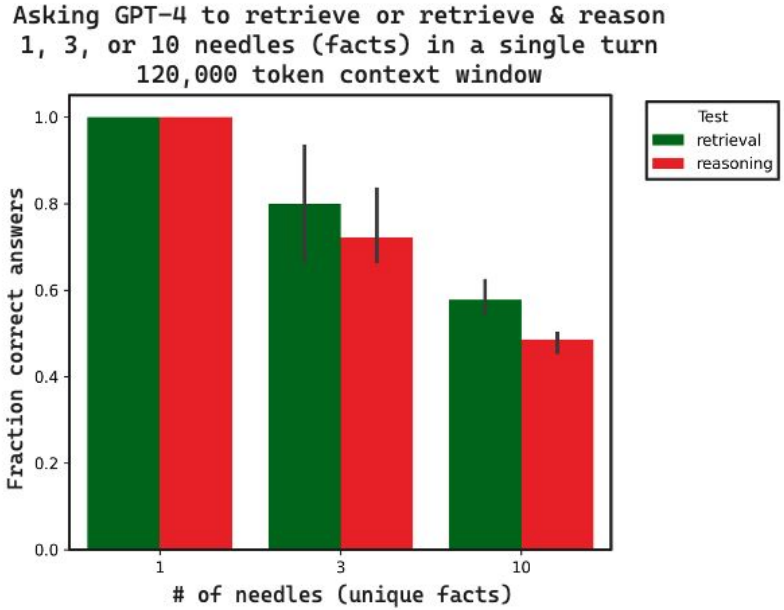
- Corpus divided into fewer chunks of larger sizes
  - Retrieves larger unit/chunk size (~4k or more)
  - Smaller search space with richer document representation (22M → 6M)
- Uses (Maximum Passage Score) MaxP scoring: Only the most relevant part of a long document determines its match score

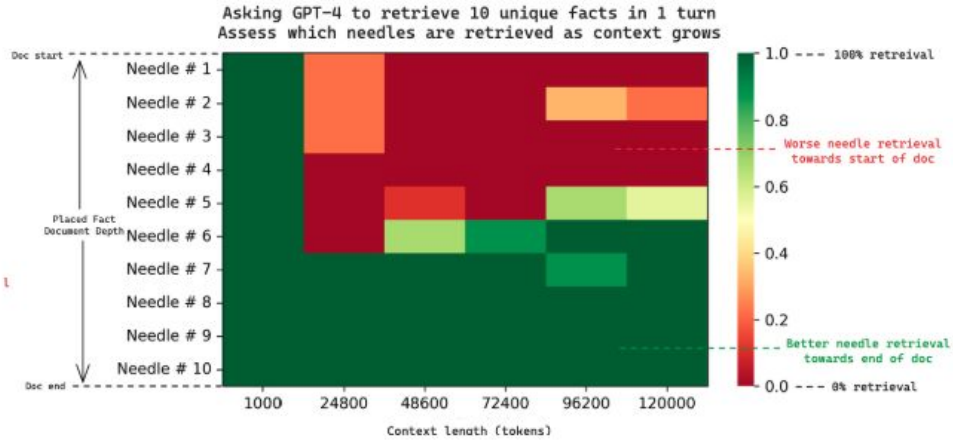
Long Reader:

- Leverages a long-context LLM: context window size upto 30k tokens = Higher semantic density

- Skips complex re-ranking due to high semantic completeness of long docs
- Supports episodic recall across long time-frames (dialogue systems)
- Improved with summarization, hierarchical clustering (RAPTOR)

MULTI-NEEDLE IN A HAYSTACK



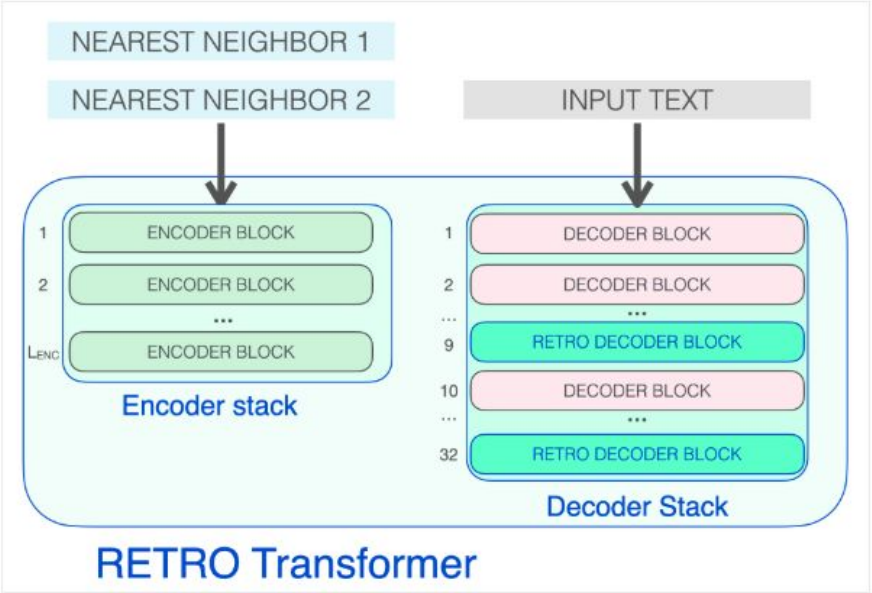


Below we'll walk through benchmark usage and discuss results on GPT-4.

- Performance degrades when you ask LLMs to retrieve more facts and reason about retrieved facts → Recency Bias
- Long-context does not guarantee accurate retrieval and reasoning

Retrieval Transformers: RETRO

- Combines retrieval + transformer layers for real-time, dynamic retrieval
- Retrieval integrated into model architecture itself, not just a pre-step like in standard RAGs
- Nearest neighbor documents retrieved and encoded during generation → injected into specific RETRO decoder blocks via cross-attention
- Allows for fine-grained, token-level integration of external memory, not limited by context window



References

- RAG (https://arxiv.org/abs/2005.11401)
- Indexing optimization using RAPTOR (https://arxiv.org/abs/2302.04761)
- Optimized RAG Flows (CRAGs): https://arxiv.org/abs/2401.15884
- Least to Most Prompting: (https://arxiv.org/pdf/2205.10625)
- Multi-Query: (https://arxiv.org/pdf/2212.10509)
- Step-back prompting: (https://arxiv.org/pdf/2310.06117)
- RAG Fusion: (http://arxiv.org/abs/2005.11401)
- HyDE: (https://arxiv.org/abs/2212.10496)
- Adaptive RAGs: (https://arxiv.org/abs/2403.14403)
- Active RAGs: (https://arxiv.org/abs/2305.06983)
- ReAct: (https://arxiv.org/abs/2210.03629)
- Self RAGs: (https://arxiv.org/abs/2310.11511)
- RAG Agents: (https://arxiv.org/abs/2501.09136)
- Semantic Routing: https://arxiv.org/abs/2407.16833
- Proposition Indexing (https://arxiv.org/abs/2310.03058)
- Retrieval Attention (http://arxiv.org/abs/2409.10516)
- Long RAG (https://arxiv.org/pdf/2406.15319v2)
- Retrieval Transformers (https://arxiv.org/pdf/2112.04426)
- ColBERT (https://arxiv.org/abs/2004.12832)