AI Agents for Science

Lecture 4, October 8: Retrieval Augmented Generation

Instructor: Ian Foster
TA: Alok Kamatar

Crescat scientia; vita excolatur

CMSC 35370 -- https://agents4science.github.io
https://canvas.uchicago.edu/courses/67079

Curriculum

1) Why AI agents for science?

AI agents and the sense-plan-act-learn loop. Scientific Discovery Platforms (SDPs): AI-native systems that connect reasoning models with scientific resources.

2) Frontiers of Language Models

Surveys frontier reasoning models: general-purpose LLMs (GPT, Claude), domain-specific foundation models (materials, bio, weather), and hybrids. Covers techniques for eliciting better reasoning: prompting, chain-of-thought, retrieval-augmented generation (RAG), fine-tuning, and tool-augmented reasoning.

3) Systems for Agents

Discusses architectures and frameworks for building multi-agent systems, with emphasis on inter-agent communication, orchestration, and lifecycle management.

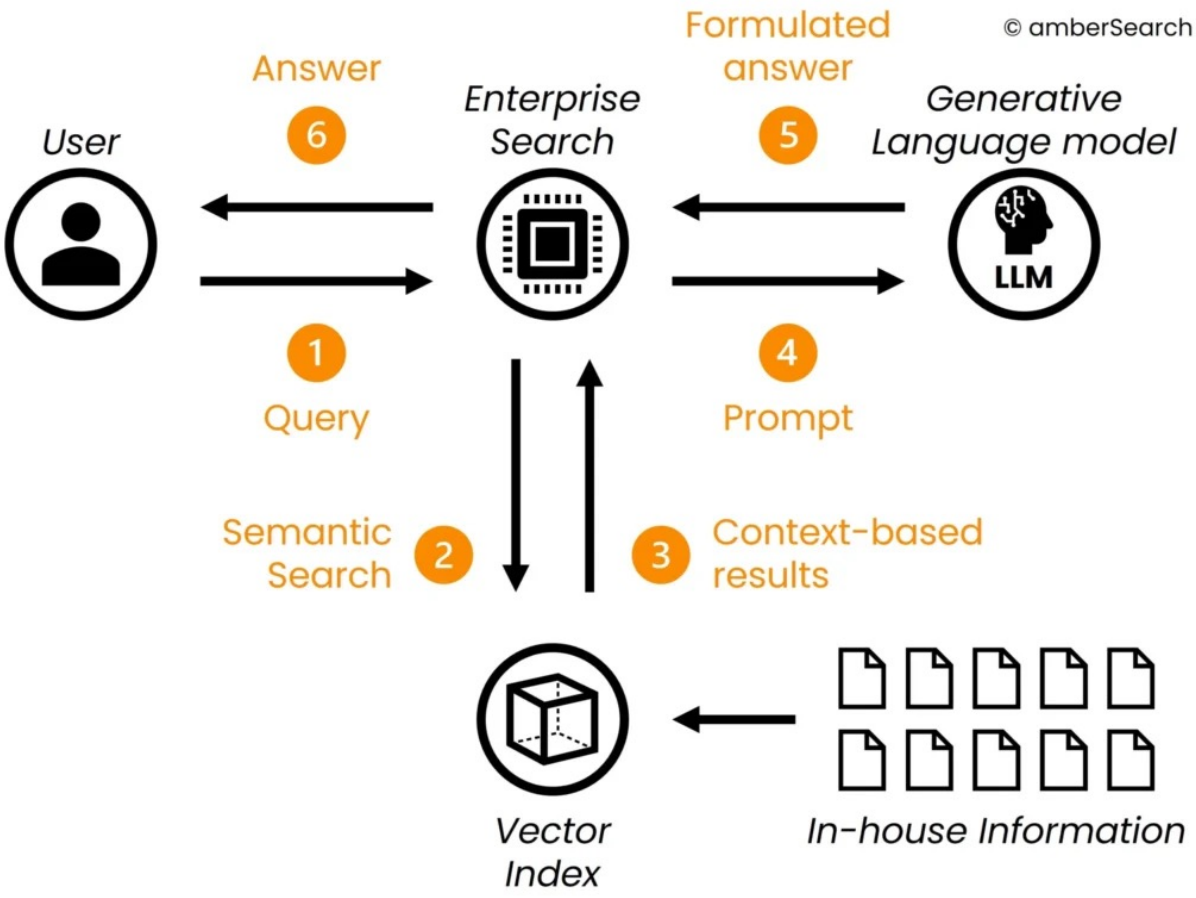
4) Retrieval Augmented Generation (RAG) and Vector Databases

Covers how to augment reasoning models with external knowledge bases, vector search, and hybrid retrieval methods.

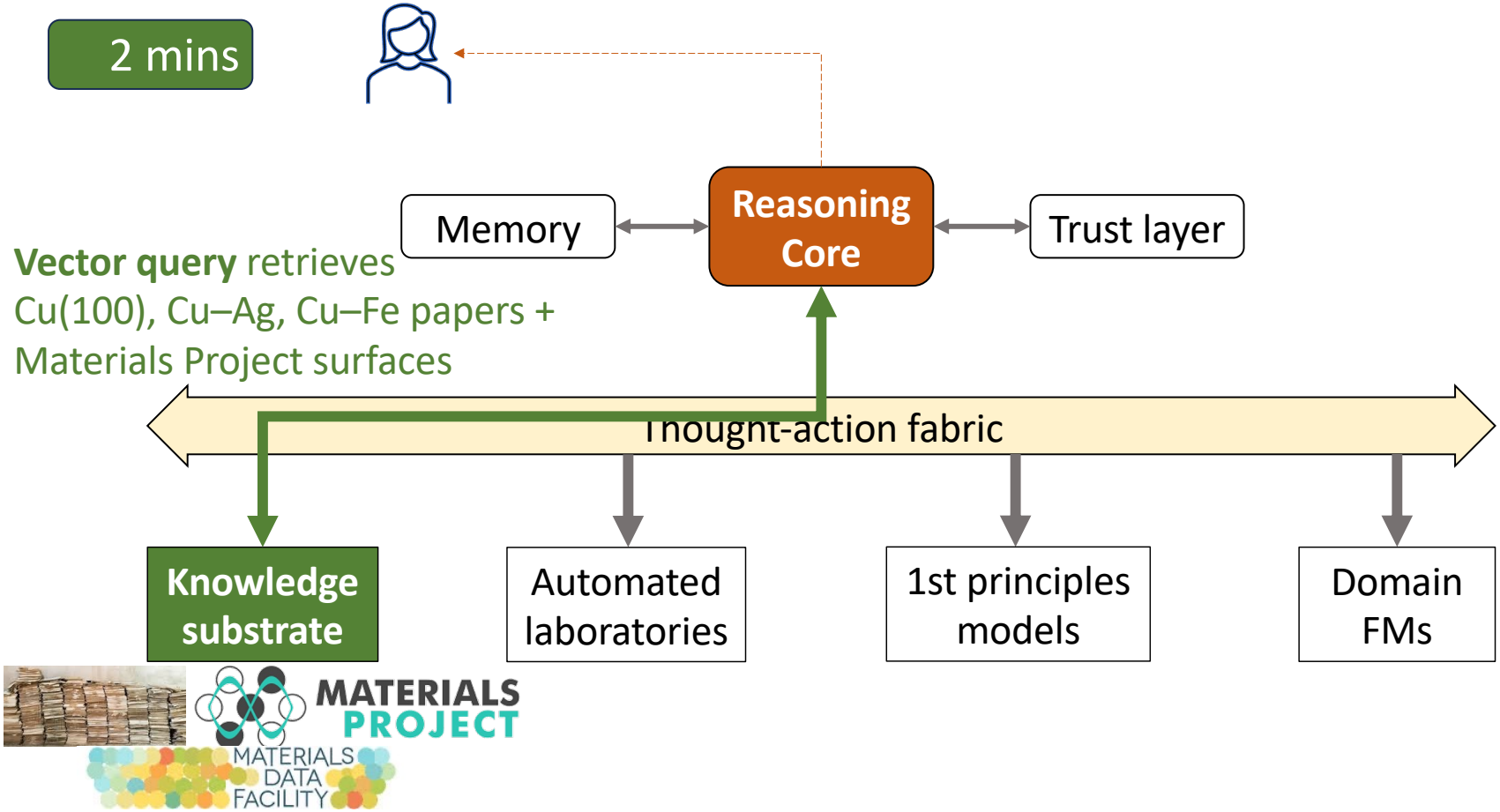
Readings

- Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
- The FAISS library

Retrieval Augmented Generation Architecture model

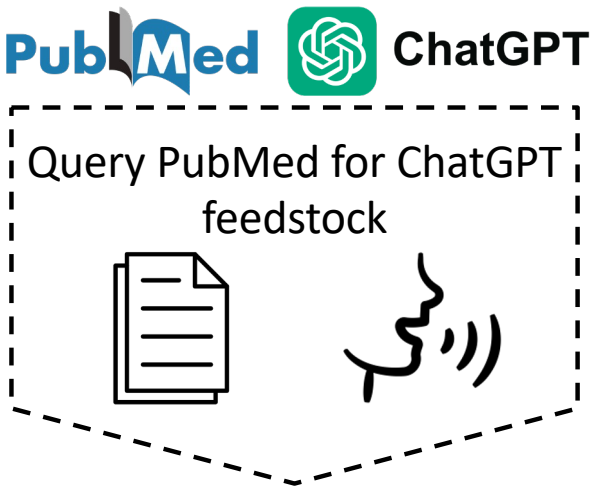


AI-native Scientific Discovery Platform



Recall example: A **peptide expert**

(Prototyped with PubMed and ChatGPT)



Retrieve abstracts **A** from PubMed that reference specified **peptide**

Use ChatGPT to build hypotheses by using retrieval-augmented generation: e.g.:

"Given **A**, on which organism is {**peptide**} acting?"

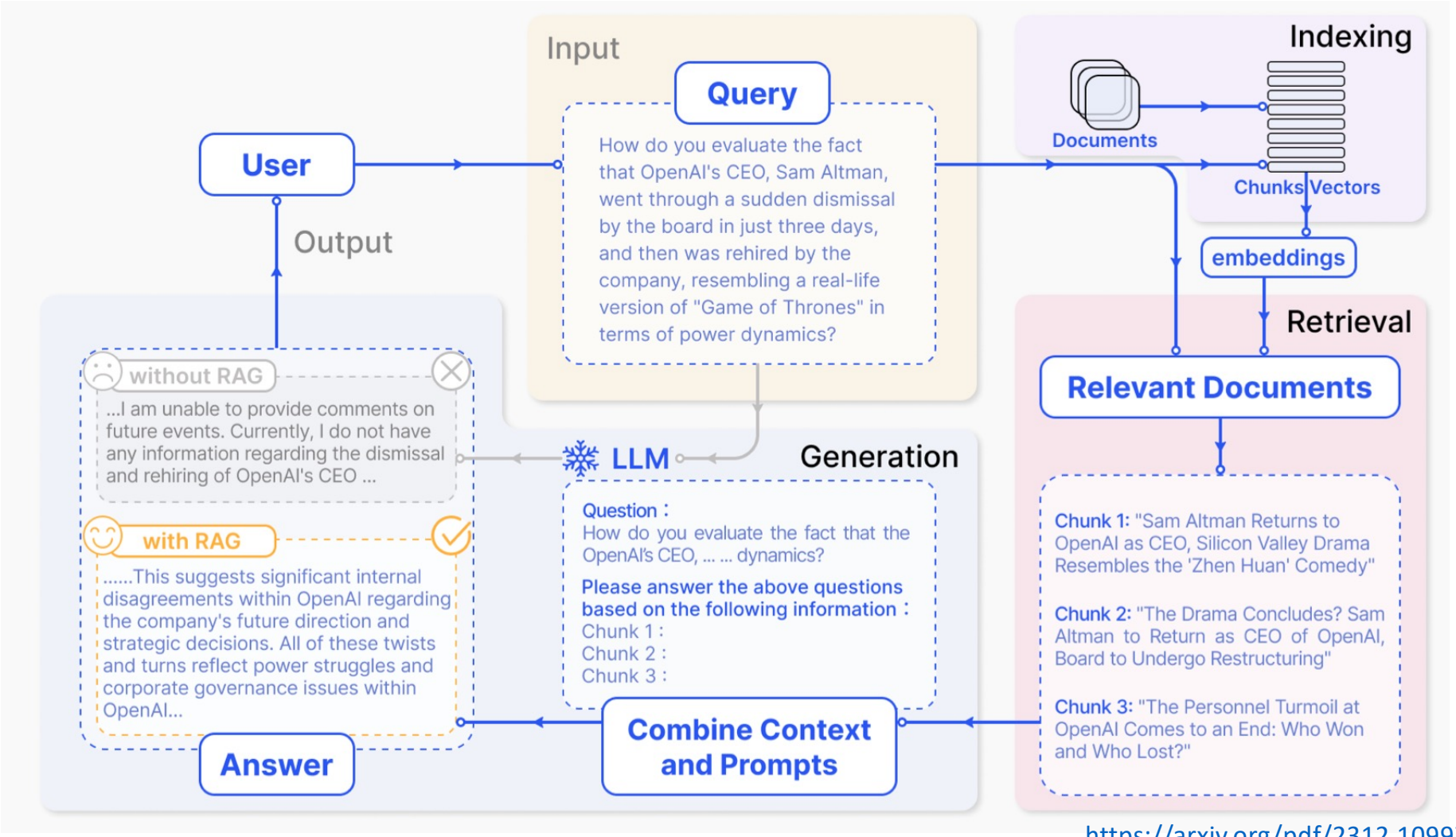
Arvind Ramanathan, Priyanka Setty, et al.

The basic idea behind RAG

Q: “Which known drugs might modulate the FXN gene pathway?”

SYSTEM: Use only the following context to propose candidate drugs.
CONTEXT:
[Document 1...]
[Document 2...]
QUESTION: Which known drugs might modulate the FXN gene pathway?

The LLM provides a more accurate answer because it has access to accurate and pertinent information



https://arxiv.org/pdf/2312.10997

Questions

Q: “Which known drugs might modulate the FXN gene pathway?”

We have many documents (e.g., PubMed papers, internal documents, medical records). How can we use this corpus to help the model answer our query?

- What information is relevant?
- How do I find that information?
- How much information should I provide?
- What is the best way to present that information?
- How do I evaluate performance?
- How do I handle information of different types?

“Which known drugs might modulate the FXN gene pathway?” -- Which documents are relevant?

- We can search on metadata or on text matches: e.g., articles for which the title or the body text includes the word “FXN”
- But we want to find papers that discuss “related topics”: e.g., drugs that modulate related gene pathways, or that describe “modulation of the FXN gene pathway” but using different words
- Computers don’t understand meaning: they store text as strings
  - They can check equality (“cat” vs. “dog”), but not that both are animals
- We want a way to represent words and sentences so that *similar meanings* are *close together*

BM25 (“Best Matching 25”)

- A widely used lexical retrieval algorithm that scores documents based on the frequency of query terms in each document, adjusted for term importance (IDF) and document length:

$$\text{score}(q,d) = \sum_{t \in q} IDF(t) \cdot \frac{f(t,d) \cdot (k_1 + 1)}{f(t,d) + k_1 \cdot (1 - b + b \cdot | d | / avgdl)}$$

- Where:
  - $f(t,d)$: frequency of term *t* in document *d*
  - $| d |$: document length
  - *avgdl*: average document length in corpus
  - k_1: term-frequency scaling factor (≈ 1.2–2.0)
  - b: length normalization factor (≈ 0.75)
  - IDF(t): inverse document frequency = $\log \frac{N - n_t + 0.5}{n_t + 0.5}$

Boosts relevance

Prevents long docs dominating

Downweights common words

Distributional Hypothesis

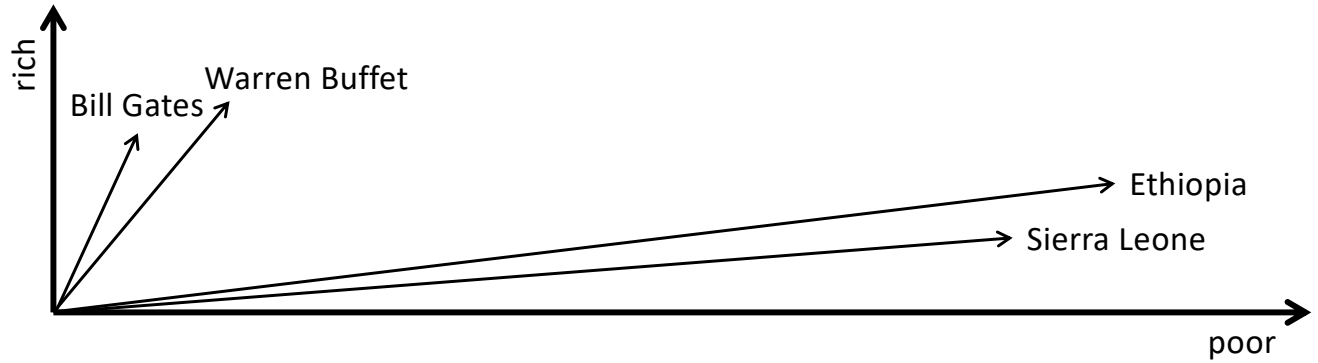
- *Words that occur in the same contexts tend to have similar meanings* (Harris, 1954*)
- One of the most successful ideas in modern NLP
- Greatly boosts performance if used correctly
- Idea: Count the co-occurrence of tokens, e.g., within sentences

	word1	word2	word3	word4	word5	word6	word7
word1	0	2	0	3	5	0	1
word2		0	1	5	2	0	3
word3			0	1	0	0	1
word4				0	6	0	1

* https://www.tandfonline.com/doi/pdf/10.1080/00437956.1954.11659520

Distributional Hypothesis

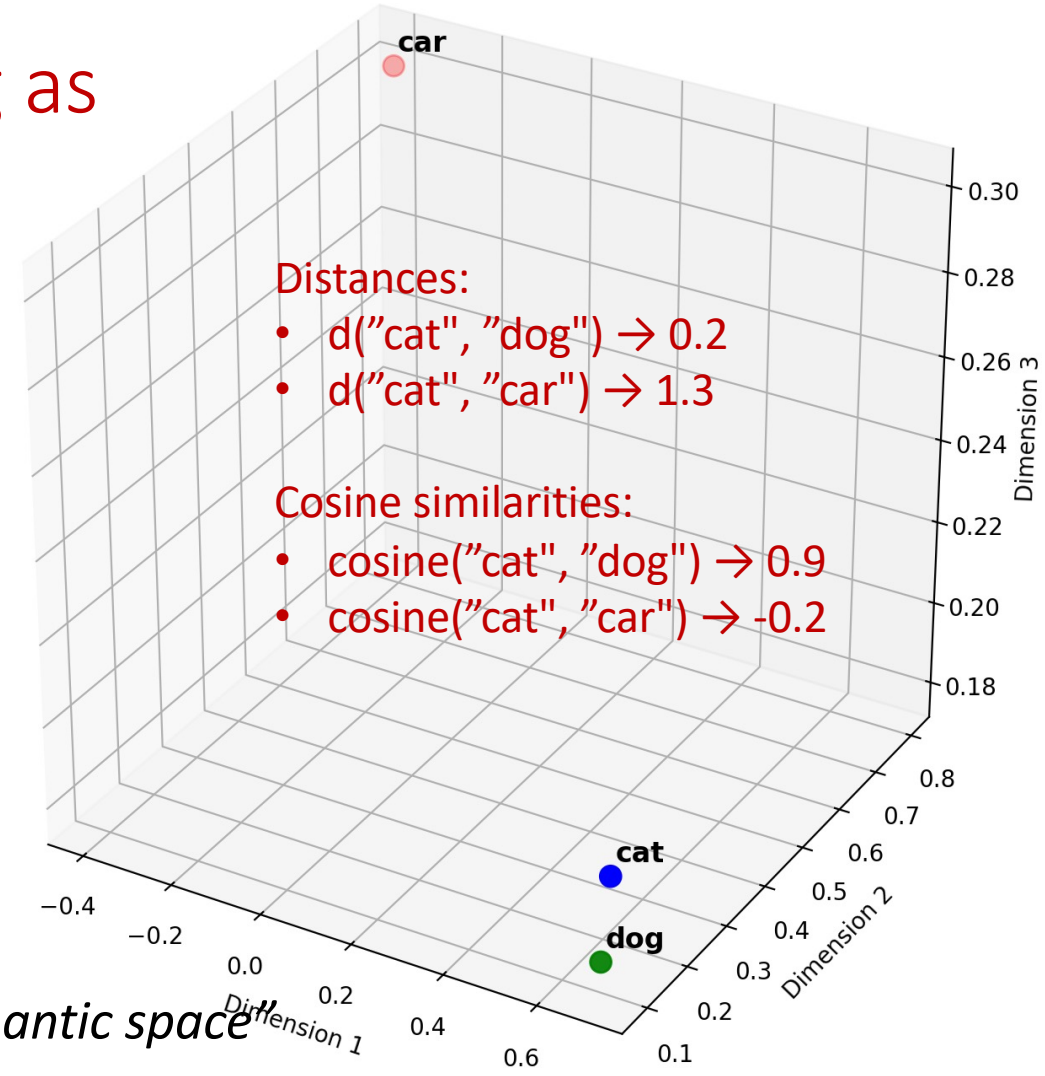
- E.g., we might find upon scanning a large corpus that:
  - *Bill Gates* often appears next to “Microsoft,” “philanthropy,” “billionaire.”
  - *Warren Buffett* co-occurs with “investor,” “billionaire,” “fortune”
  - *Ethiopia* and *Sierra Leone* co-occur with “developing,” “poverty,” “Africa”
- The *frequency and context* of those pairings can be used to infer relationships: who or what tends to be associated with *wealth* or *poverty*, *people* or *countries*, etc.



* https://www.tandfonline.com/doi/pdf/10.1080/00437956.1954.11659520

Representing meaning as points in space

- Turn each piece of text (word, sentence, paragraph) into a list of numbers: can be interpreted as a **vector**
- Similar meanings → similar vectors. E.g.:
  - cat → [0.7, 0.1, 0.2]
  - dog → [0.69, 0.09, 0.18]
  - car → [-0.4, 0.8, 0.3]



Text → vector → position in a "semantic space"

Learning the geometry of language

- Nowadays we don’t count co-occurences. Instead, we train an embedding model (a small neural network) to place words or sentences in this space

- During training, the model learns that:
  - Words appearing in similar contexts (“doctor” and “nurse”) should be close
  - Words with different contexts (“doctor” and “banana”) should be far apart

- The result: a map where **distance ≈ semantic difference**

Training embedding models (in brief!)

Embedding models are trained by showing them examples of texts that belong together and those that don't, and teaching them, through loss functions, to make the first pair close and the second pair distant in vector space.

1) Word-level embeddings (e.g., Word2Vec Skip-gram):
   - Input: a target word
   - Predict: its neighboring words (context)
   - The network adjusts word vectors so that words appearing in similar contexts (e.g., "doctor", "nurse") have similar embeddings

2) Sentence-level embeddings (e.g., Sentence-BERT, OpenAI Embeddings)

- Use a **contrastive loss**

```
L = max(0, d(anchor, positive) - d(anchor, negative) + margin)
```

- Given pairs (anchor, positive, negative):

  anchor = "What is photosynthesis?"

  positive = "The process by which plants make food using sunlight."

  negative = "The capital of France is Paris."

- The model encodes all three and minimizes:
  - distance(anchor, positive) << distance(anchor, negative)

Using the map

The geometry of this space encodes meaning relationships. You can use it to, for example:

- Find **similar meanings** (“synonyms” → nearest neighbors)
- Retrieve **relevant documents** (e.g., RAG systems)
- Solve **analogies**:
  - vector("king") - vector("man") + vector("woman") ≈ vector("queen")

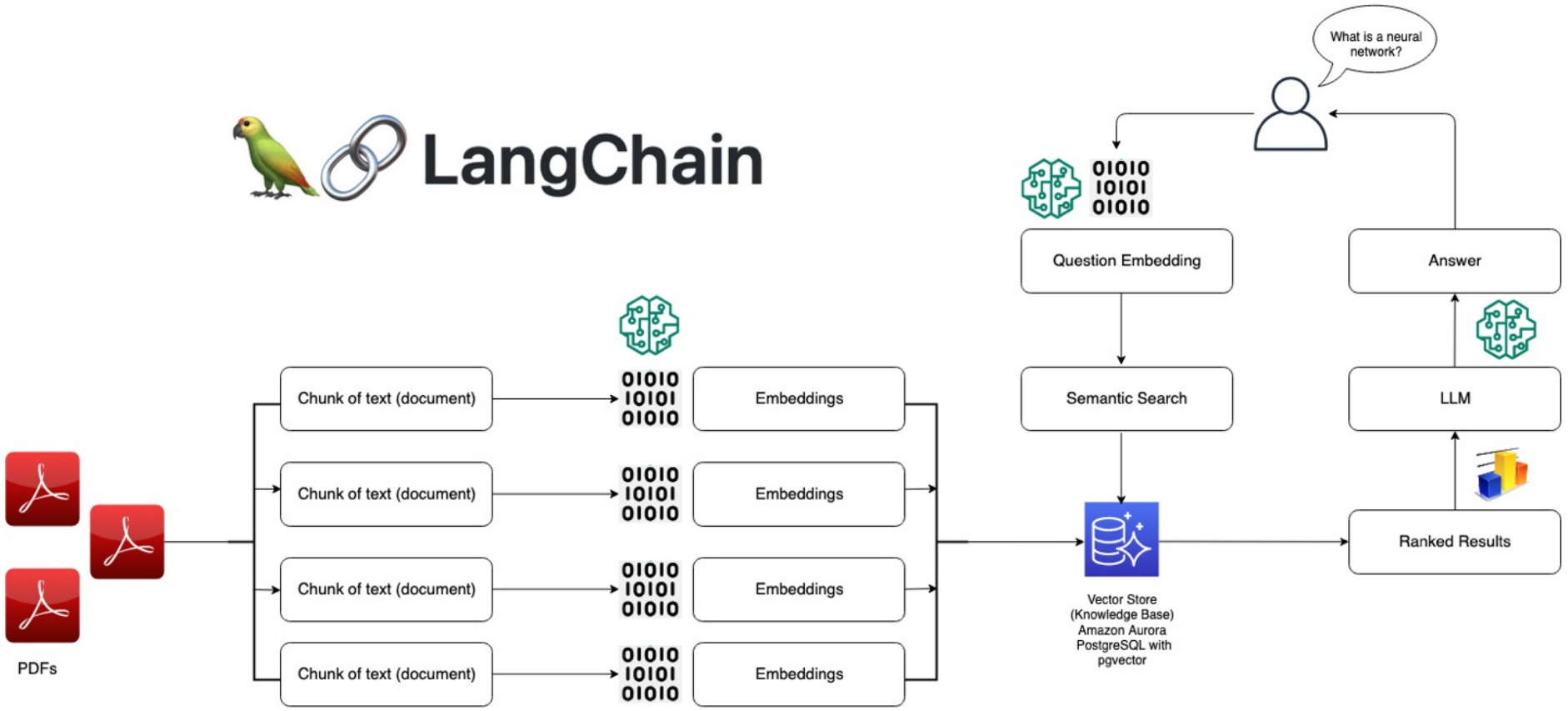
Finding relevant information, revisited

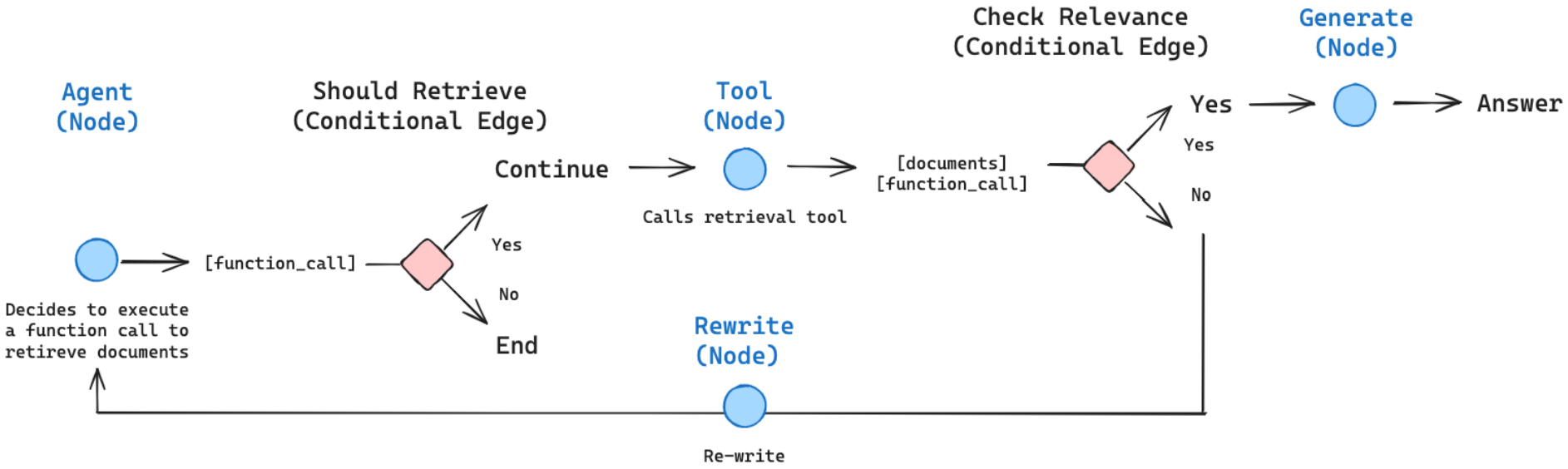
We have a large collection of documents (e.g., PubMed papers, internal documents, medical records). How do we identify what parts of this corpus help answer that query?

1) Partition corpus into smaller chunks, and compute embedding for each

2) Compute an embedding for the query, and then:

- Find chunks that are “near” the query in embedding space
- Rank and return top-k
- Select from top-k to provide to LLM

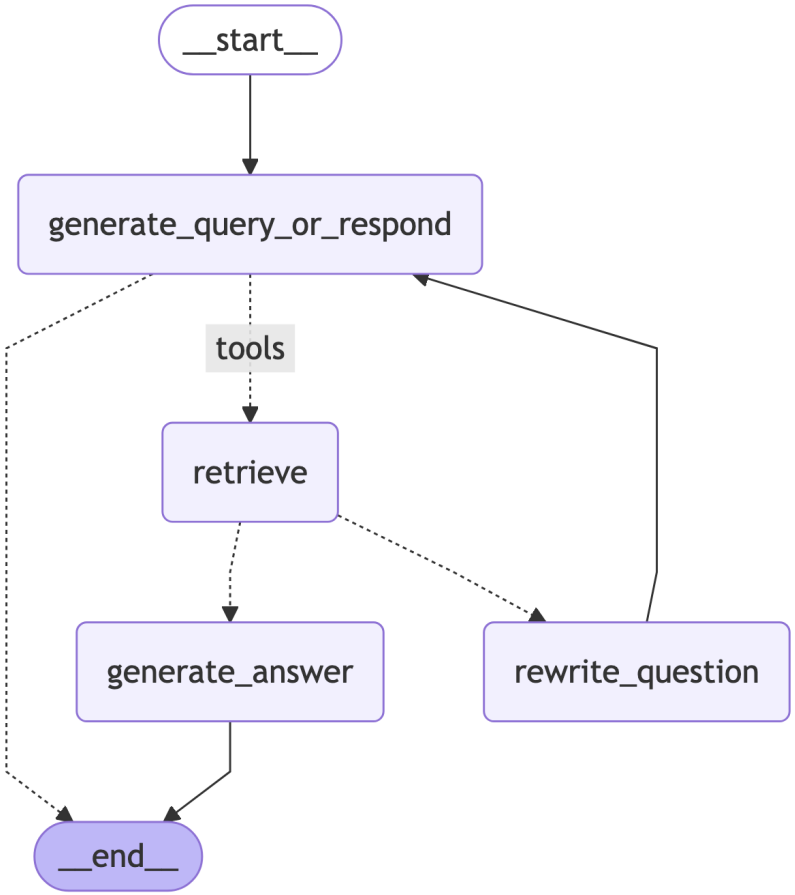

LangChain
What is a neural network?
Question Embedding
Answer
Semantic Search
LLM
Chunk of text (document)
Chunk of text (document)
Chunk of text (document)
Chunk of text (document)
Embeddings
Embeddings
Embeddings
Embeddings
PDFs
Vector Store
(Knowledge Base)
Amazon Aurora
PostgreSQL with
pgvector
Ranked Results


Agent
(Node)
Should Retrieve
(Conditional Edge)
Tool
(Node)
Check Relevance
(Conditional Edge)
Generate
(Node)
Answer
Continue
Yes
Yes
[documents]
[function_call]
Calls retrieval tool
[function_call]
No
No
Decides to execute
a function call to
retireve documents
End
Rewrite
(Node)
Re-write

Retrieval agent

A retrieval agent that decides whether to retrieve context from a vectorstore or respond to the user directly.

- Fetch and preprocess documents that will be used for retrieval
- Index those documents for semantic search and create a retriever tool for the agent
- Build an agentic RAG system that can decide when to use the retriever tool



Agentic RAG with LangGraph

Fetch documents to use in our RAG system. We will use three of the most recent pages from Lilian Weng's excellent blog. We'll start by fetching the content of the pages using `WebBaseLoader` utility:

```
from langchain_community.document_loaders import WebBaseLoader

urls = [
    "https://lilianweng.github.io/posts/2024-11-28-reward-hacking/",
    "https://lilianweng.github.io/posts/2024-07-07-hallucination/",
    "https://lilianweng.github.io/posts/2024-04-12-diffusion-video/",
]

docs = [WebBaseLoader(url).load() for url in urls]
```

```
docs[0][0].page_content.strip()[:1000]
```

https://langchain-ai.github.io/langgraph/tutorials/rag/langgraph_agentic_rag/

Split the fetched documents into smaller chunks for indexing into our vectorstore:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

docs_list = [item for sublist in docs for item in sublist]

text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=100, chunk_overlap=50
)
doc_splits = text_splitter.split_documents(docs_list)
```

```
doc_splits[0].page_content.strip()
```

Index chunks into a vector store to use for semantic search.

1. Use an in-memory vector store and OpenAI embeddings:

```
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEmbeddings

vectorstore = InMemoryVectorStore.from_documents(
    documents=doc_splits, embedding=OpenAIEmbeddings()
)
retriever = vectorstore.as_retriever()
```

2. Create a retriever tool using LangChain's prebuilt `create_retriever_tool`:

```
from langchain.tools.retriever import create_retriever_tool

retriever_tool = create_retriever_tool(
    retriever,
    "retrieve_blog_posts",
    "Search and return information about Lilian Weng blog posts.",
)
```

3. Test the tool:

```
retriever_tool.invoke({"query": "types of reward hacking"})
```

Generate query

Now we will start building components (nodes and edges) for our agentic RAG graph.

Note that the components will operate on the `MessagesState` — graph state that contains a `messages` key with a list of chat messages.

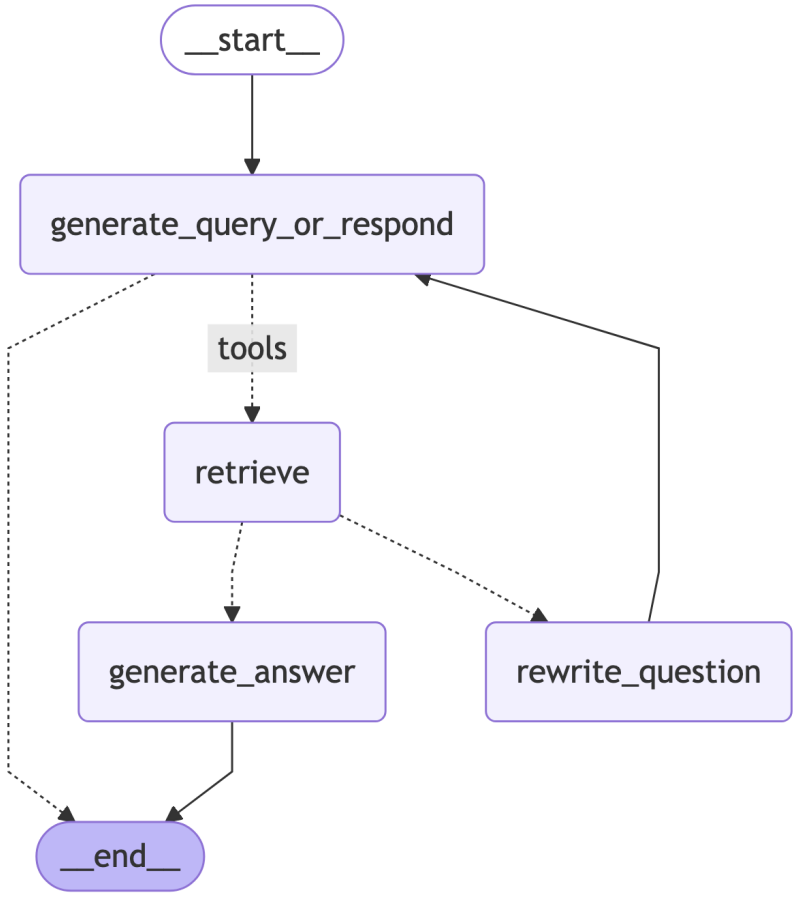
1. Build a `generate_query_or_respond` node. It will call an LLM to generate a response based on the current graph state (list of messages). Given the input messages, it will decide to retrieve using the retriever tool, or respond directly to the user. Note that we're giving the chat model access to the `retriever_tool` we created earlier via `.bind_tools`:

```python
from langgraph.graph import MessagesState
from langchain.chat_models import init_chat_model

response_model = init_chat_model("openai:gpt-4.1", temperature=0)


def generate_query_or_respond(state: MessagesState):
    """Call the model to generate a response based on the current state. Given
    the question, it will decide to retrieve using the retriever tool, or simpl
    """
    response = (
        response_model
        .bind_tools([retriever_tool]).invoke(state["messages"])
    )
    return {"messages": [response]}
```

Ask a question that requires semantic search

```
input = {
    "messages": [
        {
            "role": "user",
            "content": "What does Lilian Weng say about types of reward hacking?
        }
    ]
}
generate_query_or_respond(input)["messages"][-1].pretty_print()
```

```python
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import ToolNode
from langgraph.prebuilt import tools_condition

workflow = StateGraph(MessagesState)

# Define the nodes we will cycle between
workflow.add_node(generate_query_or_respond)
workflow.add_node("retrieve", ToolNode([retriever_tool]))
workflow.add_node(rewrite_question)
workflow.add_node(generate_answer)

workflow.add_edge(START, "generate_query_or_respond")

# Decide whether to retrieve
workflow.add_conditional_edges(
    "generate_query_or_respond",
    # Assess LLM decision (call `retriever_tool` tool or respond to the user)
    tools_condition,
    {
        # Translate the condition outputs to nodes in our graph
        "tools": "retrieve",
        END: END,
    },
)

# Edges taken after the `action` node is called.
workflow.add_conditional_edges(
    "retrieve",
    # Assess agent decision
    grade_documents,
)
workflow.add_edge("generate_answer", END)
workflow.add_edge("rewrite_question", "generate_query_or_respond")

# Compile
graph = workflow.compile()
```



```
REWRITE_PROMPT = (
    "Look at the input and try to reason about the underlying semantic intent / meaning.\n"
    "Here is the initial question:"
    "\n ------- \n"
    "{question}"
    "\n ------- \n"
    "Formulate an improved question:"
)


def rewrite_question(state: MessagesState):
    """Rewrite the original user question."""
    messages = state["messages"]
    question = messages[0].content
    prompt = REWRITE_PROMPT.format(question=question)
    response = response_model.invoke([{"role": "user", "content": prompt}])
    return {"messages": [{"role": "user", "content": response.content}]}
```

RAG summarized

- **Why RAG?** Ground LLM reasoning in *external* knowledge to improve factuality, recency, provenance
- **How it works:** *Embed → Retrieve → (Re-rank) → Generate → Cite*
  - **Encode documents** → store in vector DB (ANN index)
  - **Encode query** → search top-k nearest vectors via ANN
  - **Return candidates** → re-rank or feed to generator
- **Key levers:** Chunking, embedding choice, index type (HNSW/IVF-PQ), filters, hybrid retrieval, re-ranking (cross-encoders), answer grounding
- **Measure what matters:** retrieval quality (Recall@k, nDCG), answer groundedness & citation quality (RAG-specific evals)

RAG: Core ideas and lineage

- RAG combines parametric LM with non-parametric document index
  - Canonical formulation: retrieve passages (dense retriever); condition generation on them;
- Prior to 2020, open-domain QA was dominated by lexical retrieval methods based on lexical overlap
  - Failed when vocabulary differs: E.g., ”Where did Einstein **study**?” vs. “Einstein **attended** the Polytechnic Institute in Zurich
- In 2020, Dense Passage Retriever (DPR) introduced vector encoding to retrieve semantically similar text even without shared words
  - Common to train two encoders, one for question and one for passage
  - Pre-encode passages

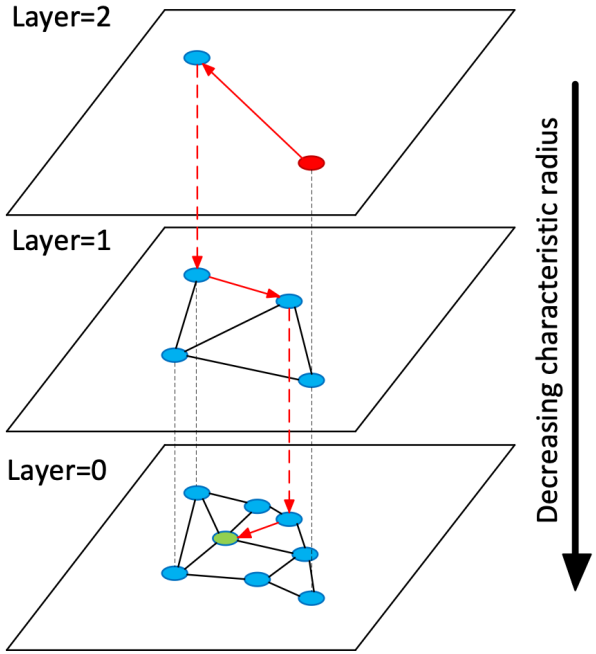
Vector search at scale: ANN indexes & libraries

- When we perform retrieval in RAG systems, we often have millions or billions of document embeddings stored as high-dimensional vectors
  - Naively comparing a query vector to *every* stored vector is O(N): far too slow for real-time applications

- Thus we use **Approximate Nearest Neighbor** (ANN) indexing
  - Data structures that find “good enough” nearest vectors orders of magnitude faster than exhaustive search

HNSW (Hierarchical Navigable Small World Graph)

- Build a **multi-layer graph** where each node is a vector, and edges connect it to its neighbors
- Higher layers form a *small-world network*: sparse connections that allow long "jumps" across space
- At query time:
  - Start from an entry point in the top layer
  - Greedily descend layer by layer, following edges toward vectors closer to the query
  - The process efficiently zeroes in on a local neighborhood of similar vectors



* https://arxiv.org/abs/1603.09320

FAISS: Facebook AI Similarity Search

- An open-source C++/Python library for large-scale vector search
- Supports multiple ANN algorithms (including IVF, HNSW, Product Quantization)
- Runs on CPU or GPU, enabling extremely fast batched searches
- Provides exact search for small datasets and compressed approximate search for billion-scale corpora.

Type	Description	When to use
Flat (brute-force)	Exact search; O(N)	Small datasets (≤100k vectors)
IVF (Inverted File Index)	Clusters vectors into “centroids”; search only nearby clusters	Millions of vectors
IVF-PQ (Product Quantization)	Compress vectors for memory efficiency	Billion-scale datasets
HNSW	Graph-based ANN for high recall	Mid-sized (up to 100M) datasets

Refining RAG

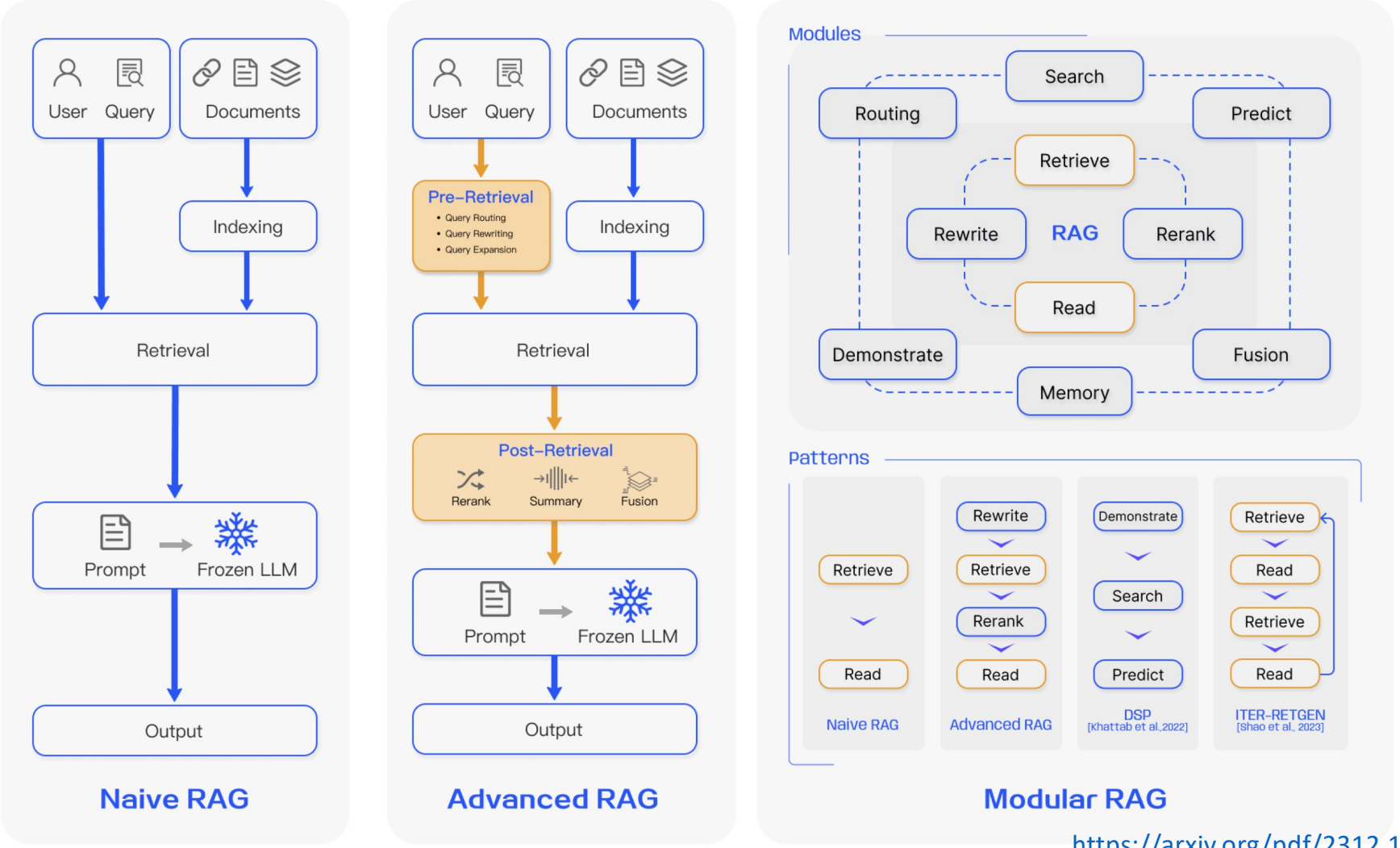
- (We have seen) **Semantic matching**
  - Use dense embeddings to find text semantically close to the query, not just keyword overlaps. E.g., text-embedding-3-large, bge-large-en, E5-mistral
- **Metadata filtering**
  - Tag chunks with source, date, or domain attributes; filter by those (e.g., WHERE doc_type="paper")
- **Hybrid retrieval**: combine semantic and lexical
- (And) **Context-aware retrieval**
  - For multi-hop or compound questions, run *query expansion* or *decomposition*
  - E.g., “Explain the impact of solar flares on aviation” → subqueries: “What are solar flares?” + “How do they affect aviation systems?”

Hybrid retrieval

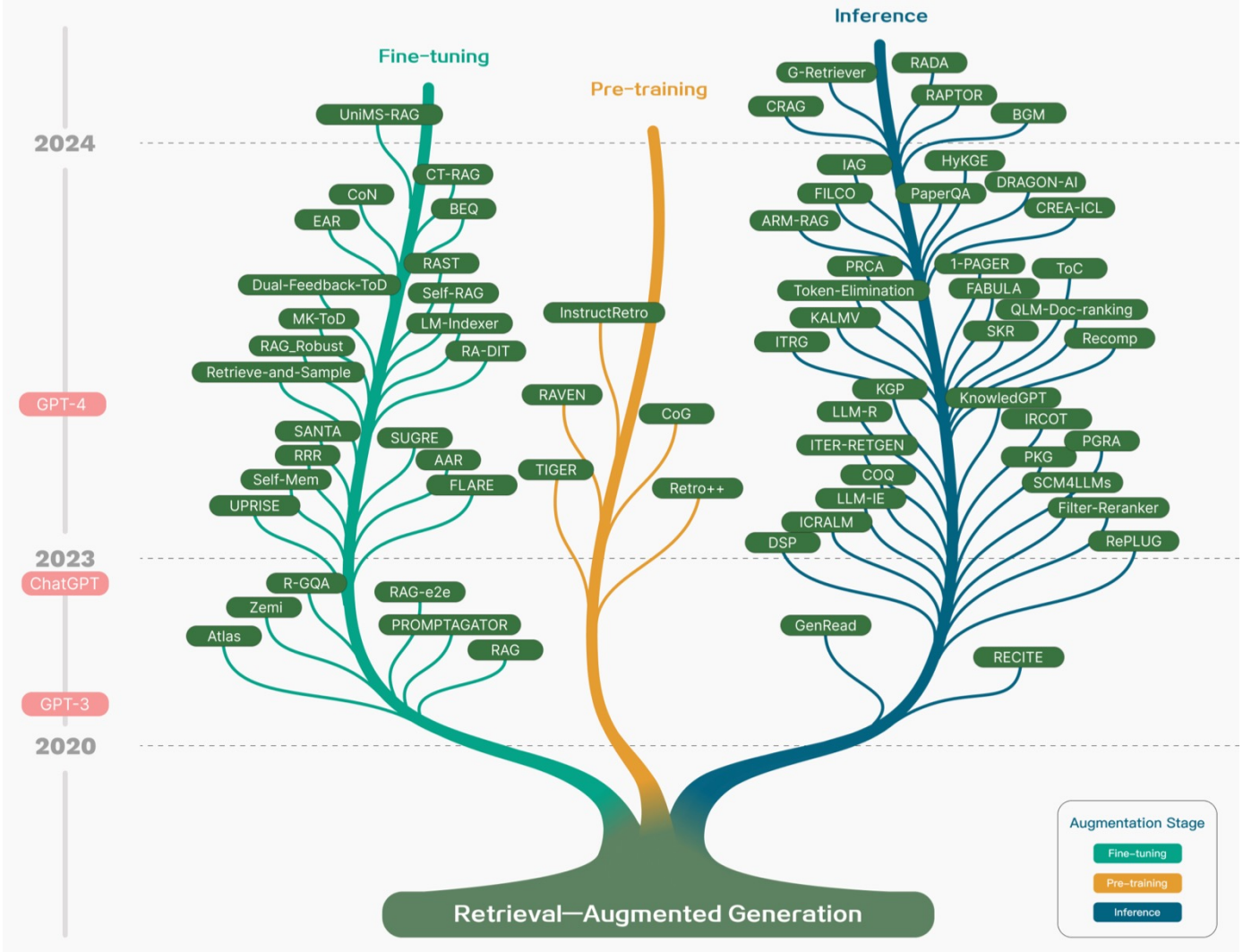
- Combining BM25 + dense retrieval often yields higher recall and better robustness across domains
  - BM25: excels at *exact lexical overlap*: Robust when query terms match document vocabulary
  - Dense retrievers: capture *semantic similarity*: Strong when paraphrases or synonyms appear
- Run both retrievers independently, obtain ranked lists of candidate documents, and merge them into a single unified ranking
  - Use Reciprocal Rank Fusion (RRF) to combine the two scores (rewards consensus)

$$\text{score}(d) = \sum_{s \in S} \frac{1}{k + \text{rank}_s(d)}$$

S = set of retrieval systems (e.g., {BM25, DenseRetriever})
$\text{rank}_s(d)$ = rank position of document *d* in system *s*'s result list (1 = top)
k = smoothing constant (commonly 60)



https://arxiv.org/pdf/2312.10997



RAG research first focused on leveraging the powerful in- context learning abilities of LLMs in the **inference** stage. Subsequent research delved deeper, gradually integrating more with the **fine-tuning** of LLMs. Researchers have also explored enhancing language models in the **pre-training** stage through retrieval-augmented techniques.

https://arxiv.org/pdf/2312.10997

Fine tuning, in brief

Fine tuning refines a pre-trained model's weights on domain-specific or task-specific examples to improve accuracy, style, or reasoning

- Collect (prompt → ideal response) pairs
- Train with gradient descent on supervised or RL objectives to update model weights
- Validate and deploy new model checkpoint

Type	**Purpose**
Supervised fine-tuning (SFT)	Teach format, reasoning, tone
Instruction tuning	Align with human prompts
Domain tuning	Specialize to specific domains
LoRA / PEFT	Lightweight, adapter-based updates

RAG and fine-tuning

Fine-tuning: Train it on examples like:

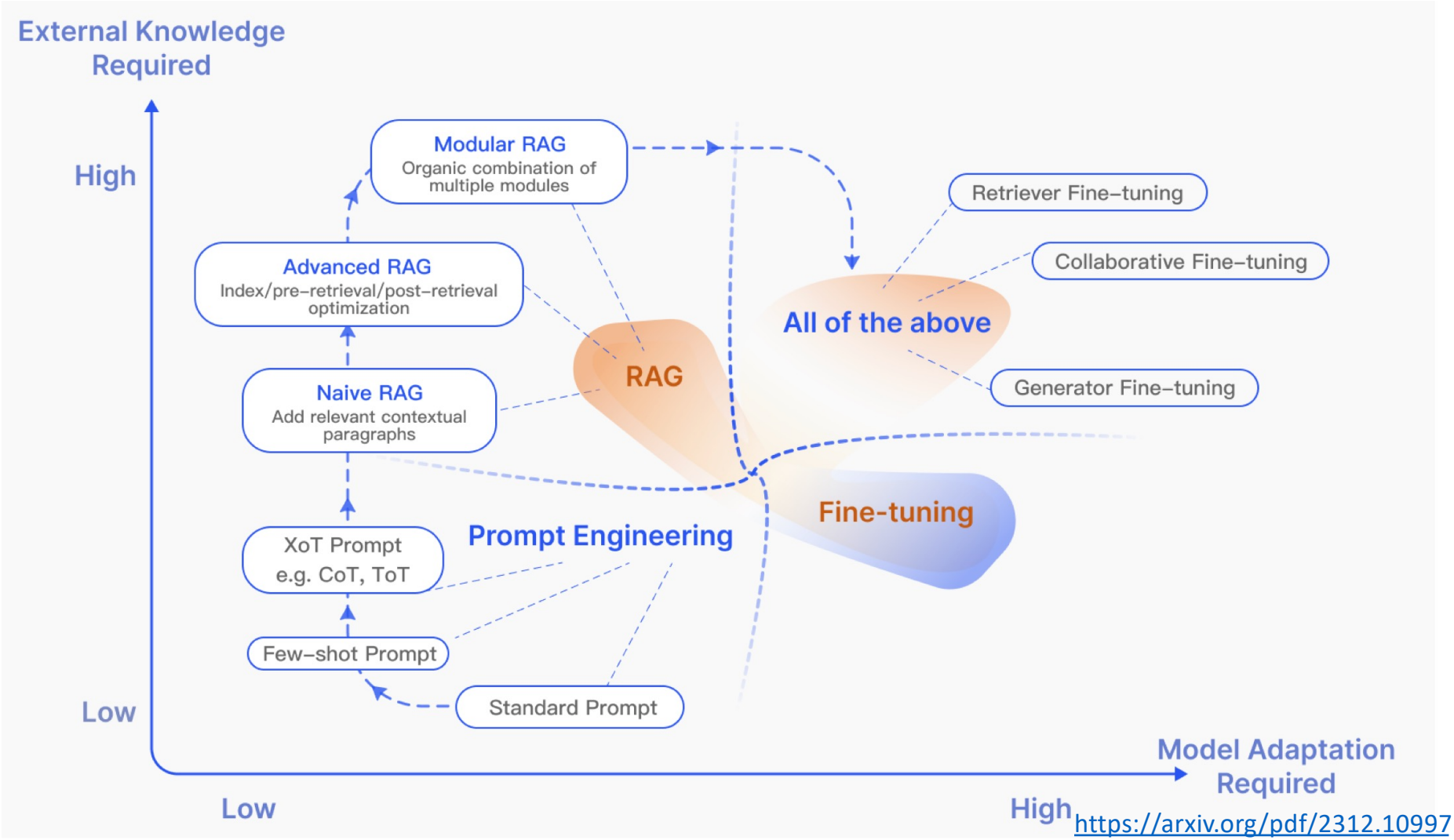
- *Prompt:* "Answer using only the context provided. Always cite sources."
  Input: Retrieved documents + question
  Output: Grounded, cited response.
- Teaches *how to use retrieved evidence*, not *what facts are true*

RAG layer: Supplies up-to-date scientific papers, sensor logs, or lab notebooks at inference time

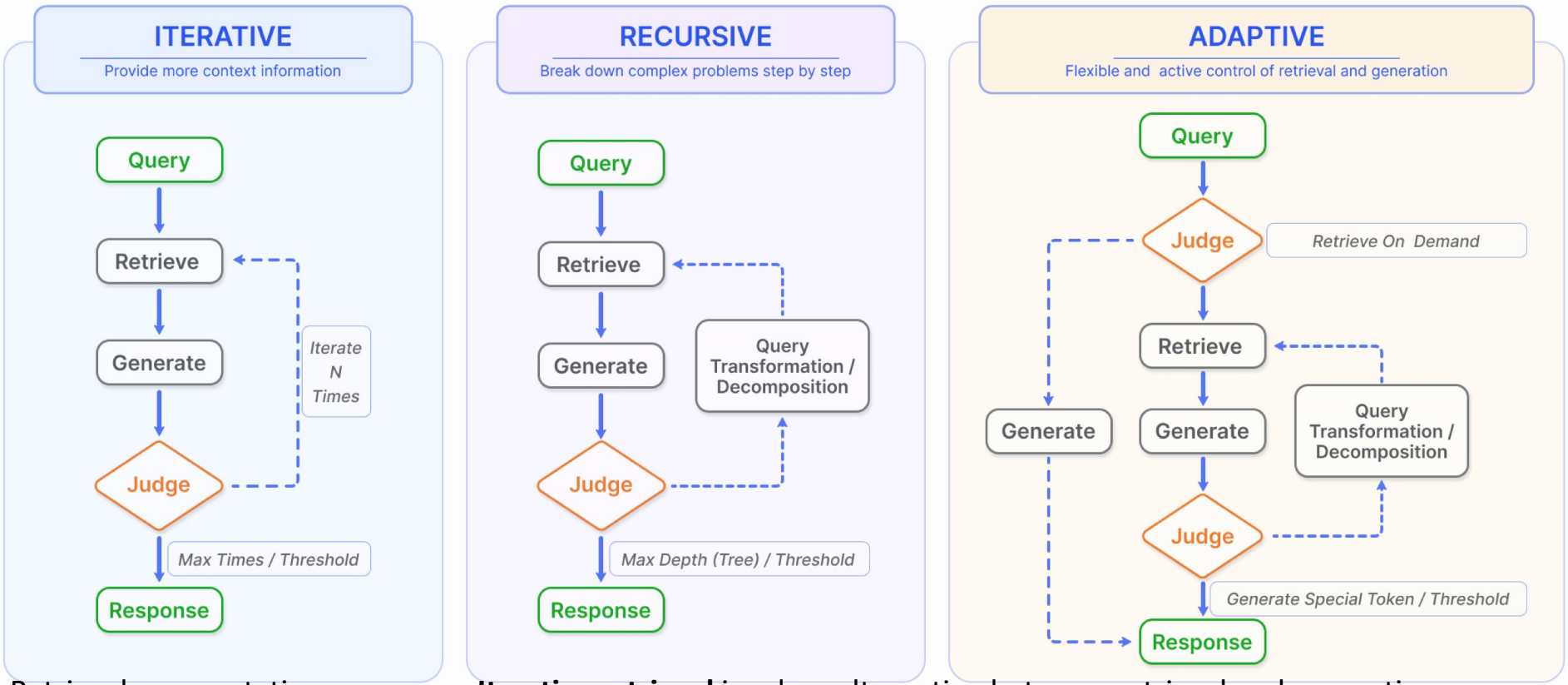
The model has learned *how to integrate evidence* but depends on RAG to fetch the *right* evidence

RAG and pre-training

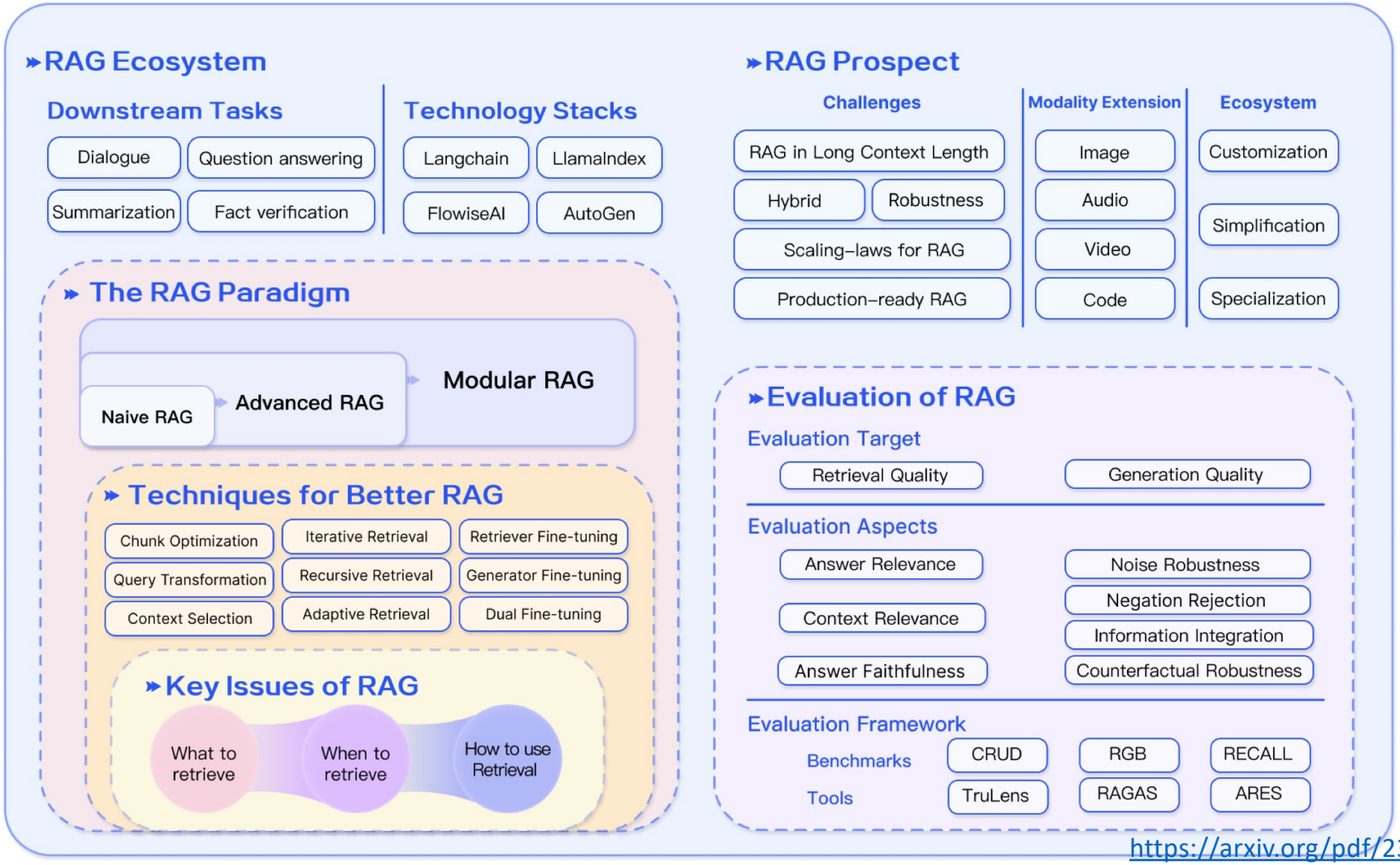
- Most RAG systems add retrieval after pre-training (during inference or fine-tuning).
- Recent research explores injecting retrieval into the **pre-training stage** itself, so the model *learns to read and reason with external memory from the start*
- During pre-training, instead of predicting the next token from a static context, the model is periodically allowed to:
  - Issue a retrieval query based on its current hidden state.
  - Pull in relevant text passages from a large external corpus (Wikipedia, web documents, scientific papers).
  - Incorporate retrieved evidence before continuing token prediction.
  - This makes retrieval part of the model's learning loop, not just a bolt-on module.



https://arxiv.org/pdf/2312.10997



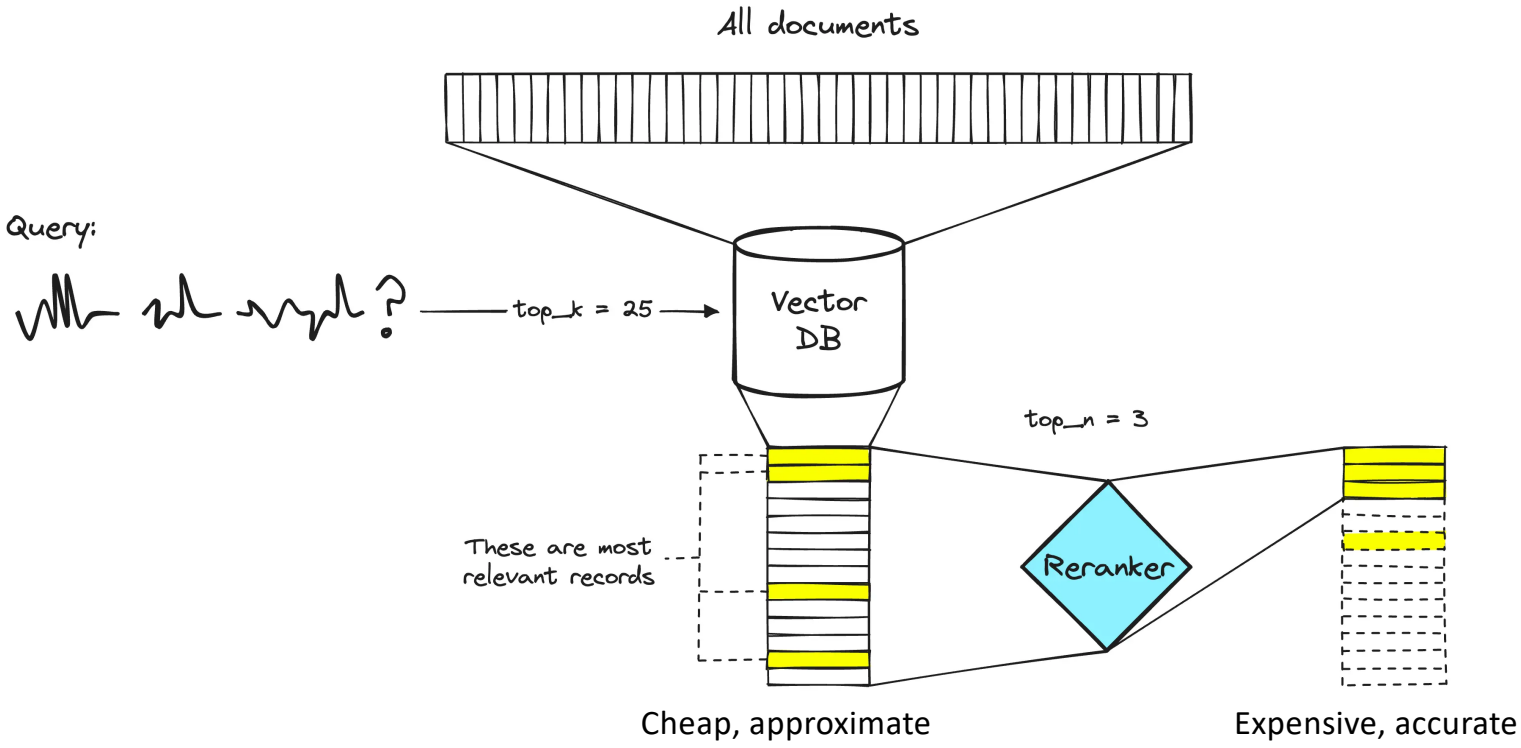
Retrieval augmentation processes. **Iterative retrieval** involves alternating between retrieval and generation, allowing for richer and more targeted context from the knowledge base at each step. **Recursive retrieval** involves gradually refining the user query and breaking down the problem into sub-problems, then continuously solving complex problems through retrieval and generation. **Adaptive retrieval** focuses on enabling the RAG system to autonomously determine whether external knowledge retrieval is necessary and when to stop retrieval and generation, often utilizing LLM-generated special tokens for control. https://arxiv.org/pdf/2312.10997



https://arxiv.org/pdf/2312.10997

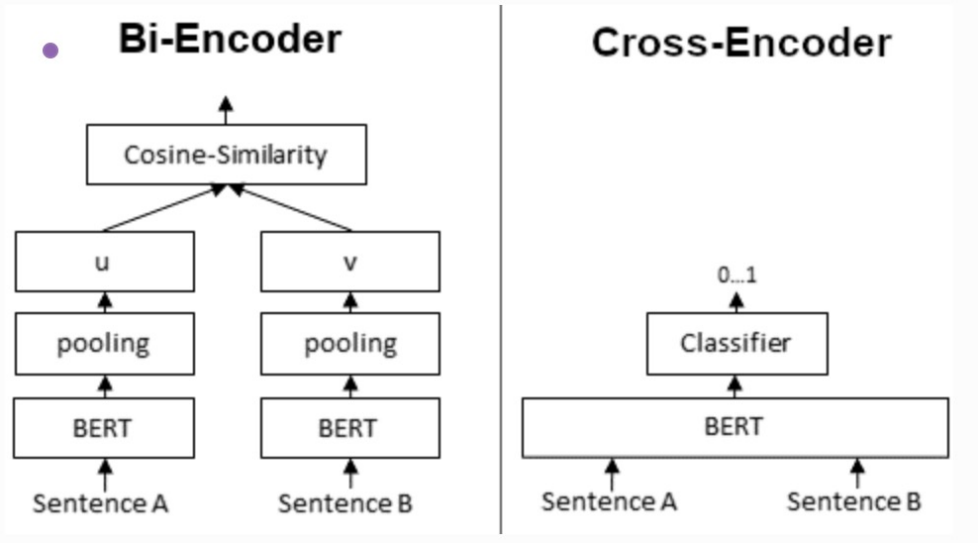
Re-ranking

After a retriever returns the top-k candidate documents for a query, we may want to sort them by how relevant they truly are



Re-ranking approaches: CrossEncoder

- **Bi-Encoders** produce for a given sentence a sentence embedding. We pass to a BERT independently the sentences A and B, which result in the sentence embeddings u and v. These sentence embedding can then be compared using cosine similarity
- For a **Cross-Encoder**, we pass both sentences simultaneously to the Transformer network. It produces then an output value between 0 and 1 indicating the similarity of the input sentence pair.



https://sbert.net/examples/cross_encoder/applications/README.html

query = "Who discovered penicillin?"

top_candidates = [
"Alexander Fleming observed mold inhibiting bacterial growth in 1928 and named the substance penicillin.",
"Marie Curie discovered radioactivity and won two Nobel Prizes for her pioneering work in physics and chemistry.",
"Howard Florey and Ernst Chain later purified penicillin and developed it into a drug used to treat infections.",
"Penicillin is an antibiotic that works by weakening bacterial cell walls, leading to cell rupture.",
"In 1928, a Scottish scientist noticed that Staphylococcus colonies failed to grow near a certain mold contaminant."
]

```python
from sentence_transformers import CrossEncoder

model = CrossEncoder("cross-encoder/ms-marco-MiniLM-L6-v2")

pairs = [(query, doc) for doc in top_candidates]
scores = model.predict(pairs)

ranked = sorted(zip(top_candidates, scores), key=lambda x: x[1], reverse=True)
for doc, score in ranked:
    print(f"{score:.3f}  |  {doc[:80]}...")
```

7.497 | Howard Florey and Ernst Chain later purified penicillin and developed it into a ...
7.367 | Alexander Fleming observed mold inhibiting bacterial growth in 1928 and named th...
-0.186 | Penicillin is an antibiotic that works by weakening bacterial cell walls, leadin...
-6.691 | Marie Curie discovered radioactivity and won two Nobel Prizes for her pioneering...
-9.254 | In 1928, a Scottish scientist noticed that Staphylococcus colonies failed to gro...

Evaluation: How well did we do? (1) Retrieval

How often and how soon is “correct” doc identified?

- Recall@k: Fraction of time in top k results
- Mean Reciprocal Rank (MRR): Fraction of time is #1

$$\mathrm{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\mathrm{rank}i}$$

- Normalized Discounted Cumulative Gain (nDCG): *relevance* and *rank position* across *multiple relevant* results, where rel_i is graded relevance of item *i* and *IDCG* is ideal DCG score

$$\mathrm{nDCG@}k = \frac{1}{\mathrm{IDCG@}k}\sum_{i=1}^{k}\frac{2^{reli}-1}{\log_2(i+1)}$$

All computed over multiple datasets: e.g., Benchmark for Information Retrieval (BEIR)

Evaluation: How well did we do? (2) Generation

After retrieval and re-ranking, the **generation** step decides whether the proposed answer is *faithful*, *grounded*, and *properly attributed* to evidence

A) Faithfulness / Groundedness

- **Definition:** How well does the generated answer stick to facts found in the retrieved documents?
- **Goal:** Prevent *hallucination*, i.e., the model inventing unsupported claims
- **Checks:**
  - Does every factual statement appear (or logically follow) from retrieved passages?
  - Are external claims absent unless supported by context?
- **Metrics:**
  - *Precision of grounded facts* (fraction of statements supported by retrieved docs).
  - *Factual consistency* via automatic entailment models or LLM-based judges.
- **Example:** If context says, *"Alexander Fleming discovered penicillin in 1928"* and the model answers, *"Fleming discovered penicillin in 1929"* → **unfaithful**

Evaluation: How well did we do? (2) Generation

B) Answer-context overlap

- **Definition:** Measures how much of the generated answer's content directly overlaps or semantically aligns with retrieved evidence.
- **Purpose:** Quantifies how much the model *actually uses* the retrieved material rather than free-associating.
- **Methods:**
  - Compute **token overlap**, **semantic similarity**, or **sentence-level entailment** between generated answer and concatenated context
  - Higher overlap ≈ better grounding, but too high may mean *copy-paste* rather than synthesis

Evaluation: How well did we do?
(2) Generation

C) Citation Attribution

- **Definition:** Evaluates whether the model's citations (e.g., numbered or inline references) correctly point to passages that support the claim
- **Why it matters:** Critical for scientific, medical, or legal RAG—users must trace every fact to its source
- **Checks:**
  - Each citation corresponds to a retrieved document
  - The cited text actually supports the associated statement
  - No "phantom" citations (referencing non-existent or irrelevant docs)

Toolkits for Generation Evaluation

RAGAS – *Retrieval Augmented Generation Assessment Suite*

- Developed by Hugging Face & Intel Labs.
- Automated scoring of RAG outputs across:
  - *Faithfulness* (are answers grounded?)
  - *Answer relevance*
  - *Context recall* (did the retriever fetch needed info?)
  - *Context precision* (was retrieved info actually used?)
- Uses a smaller LLM or NLI (natural-language inference) model for evaluation.
- Produces interpretable scores per query.
- Integrates easily into LangChain, LlamaIndex, or Hugging Face pipelines:

```
from ragas import evaluate
scores = evaluate(answers, contexts, questions)
```

Toolkits for Generation Evaluation

TruLens – *Evaluation & Monitoring Framework for LLM Apps*

- Evaluates RAG systems along three axes:
  - **Relevance** – retrieved documents match the query.
  - **Groundedness** – generated output is supported by evidence.
  - **Correctness** – final answer addresses the question accurately.
- Plugs into LangChain and LlamaIndex pipelines.
- Logs each query–context–generation triplet.
- Provides interactive dashboards to visualize scores and detect hallucinations

Example evaluation loop*

```
for query in dataset:
    retrieved_docs = retriever(query)
    answer = llm.generate(query, context=retrieved_docs)
    scores = {
        "faithfulness": ragas.faithfulness(answer, retrieved_docs),
        "overlap": ragas.overlap(answer, retrieved_docs),
        "citations": trulens.check_citations(answer, retrieved_docs)
    }
    log(scores)
```

* I have not tried this

Evaluation: How well did we do?
(3) End-to-end

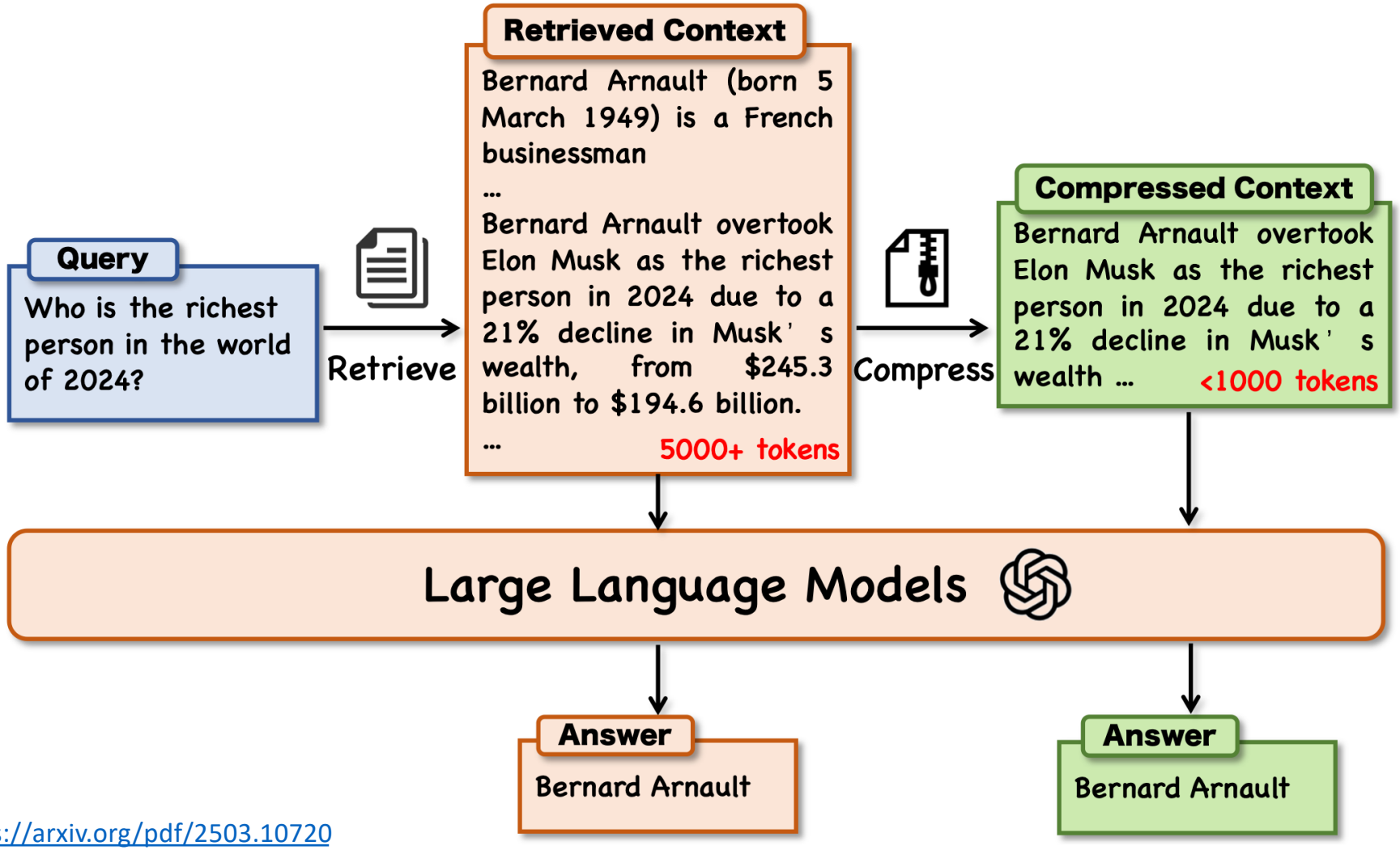
Retrieval and generation metrics assess **intermediate quality**. End-to-end (E2E) evaluation measures **final task success**: i.e., does the system deliver correct, useful, efficient answers for real users*?*

- Overall effectiveness: Accuracy, F1, precision/recall
- User-perceived quality: faithfulness, helpfulness, fluency
- Operational metrics: latency, cost, reliability

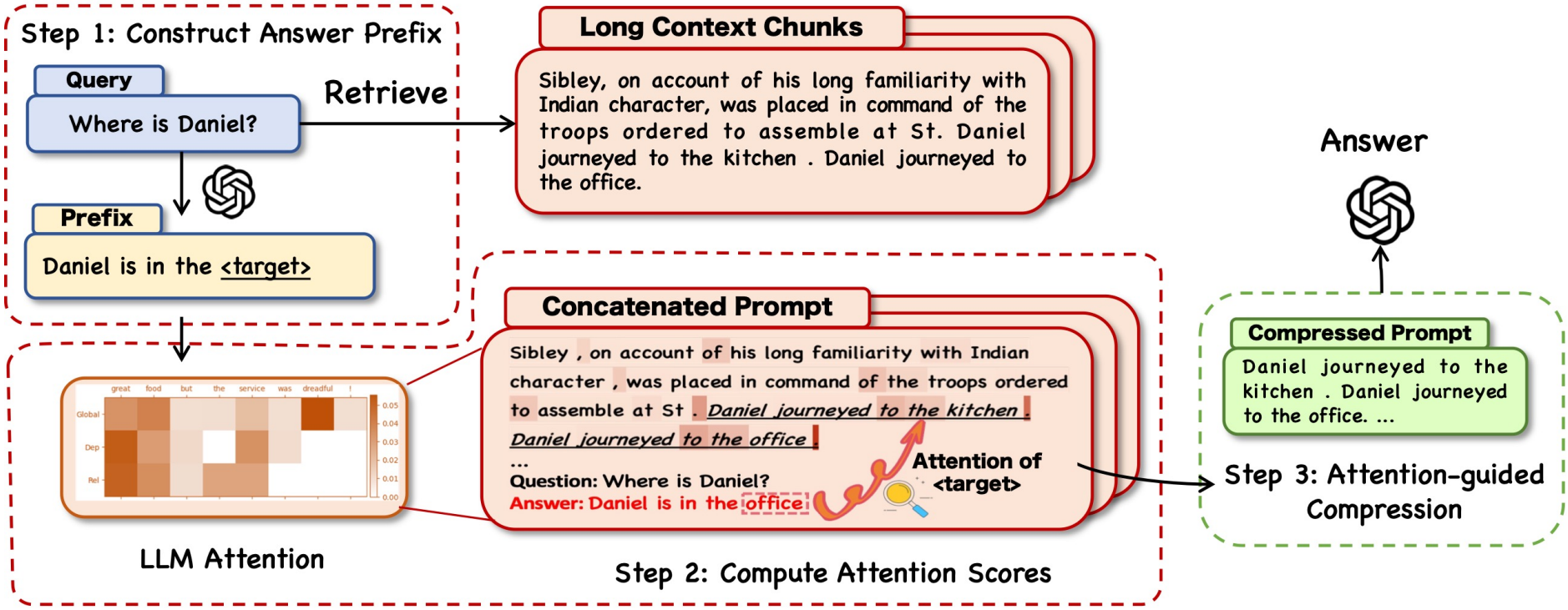
Task Type	Metric	Description
QA / factoid	Accuracy	% answers exactly match gold standard
Multi-fact reasoning	F1, EM (Exact Match)	Combine precision & recall on answer tokens
Summarization / synthesis	ROUGE, BLEU, BERTScore	Compare to human summaries
Information extraction	Precision / Recall / F1	Correct entities or relations extracted
Scientific / analytical RAG	Task-specific metric	e.g., % hypotheses supported, % correct citations

How to Present Retrieved Context

Dimension	Choices / Variants	Why it matters
Number of passages (k)	small (e.g. 3–5) vs large (10–50)	Too many distracts; too few misses evidence.
Passage ordering	by relevance, recency, diversity, random	Position bias in the LLM's attention.
Pruning / compression	full text vs selective sentences vs attention-pruned snippets	Reduces noise and context window usage (AttentionRAG).
Partitioning / batch context	chunk into blocks, rotate, interleave	Helps avoid "lost in the middle" (BriefContext).
Highlighting / markup	highlight query-aligned spans, bold keywords	Draws LLM attention to important parts.
Citation tags or labels	prepend "Doc #1: …", insert inline "(see Doc 3)" markers	Encourages traceability and provenance.
Hybrid context forms	mix text with table, graph, image contexts	Multi-modal retrieval + presentation.

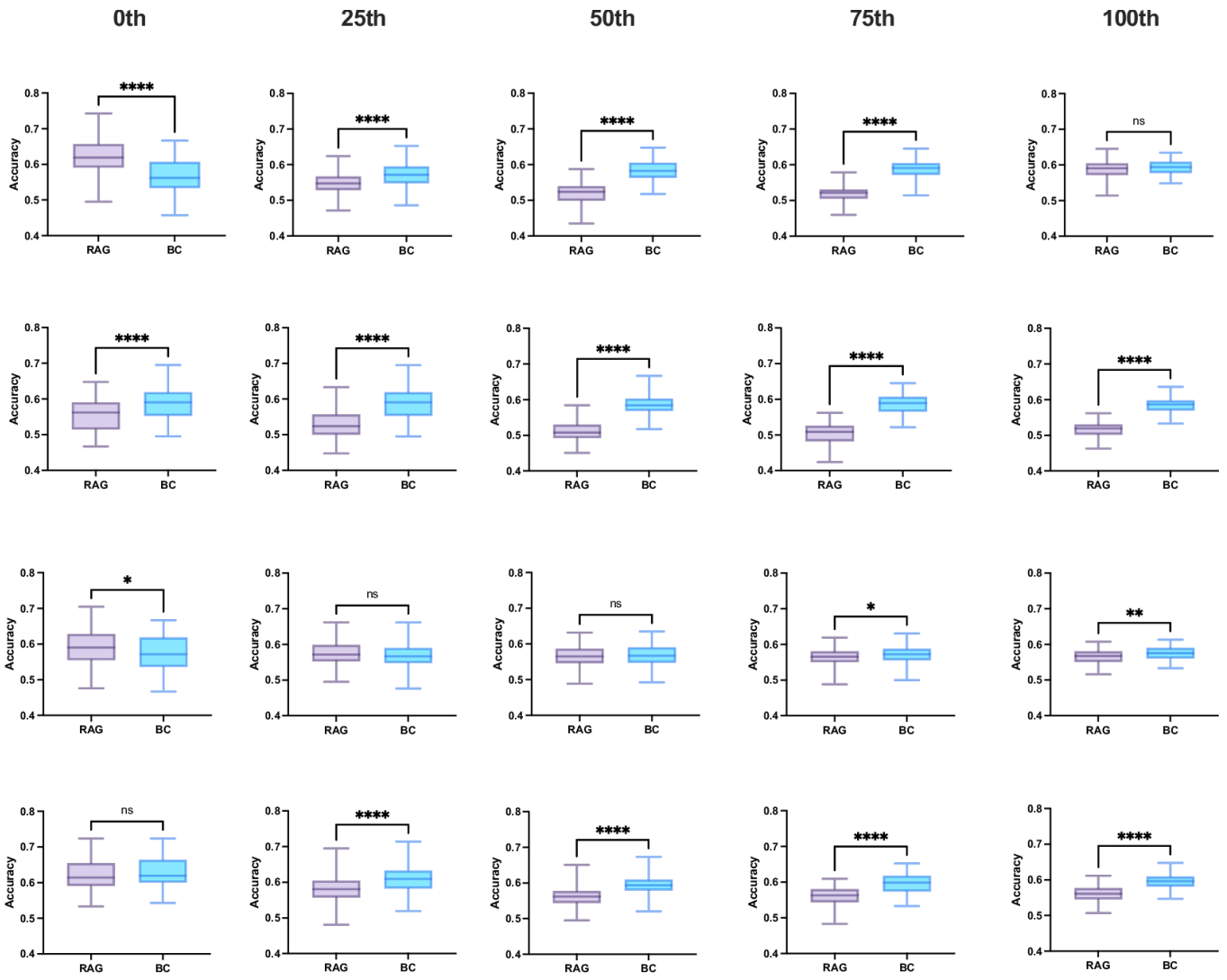


https://arxiv.org/pdf/2503.10720



https://arxiv.org/pdf/2503.10720

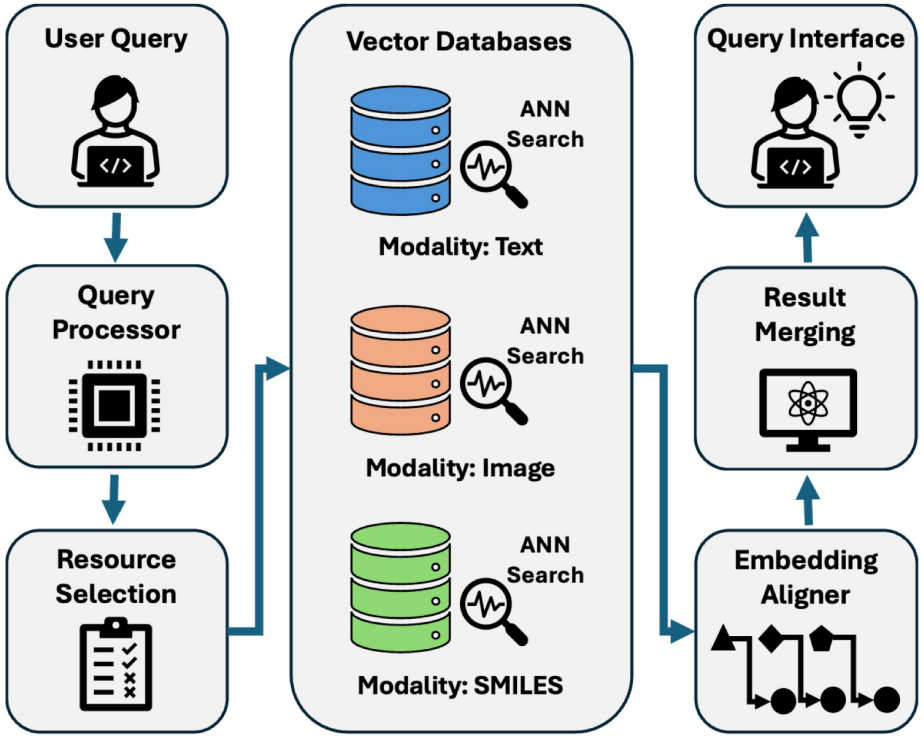
Zhang et al. (2025) on *"BriefContext"* explore how the **order and mix** of retrieved passages in context influence performance. They show that distributing "key information" across shorter, well-ordered context batches outperforms naive concatenation in many biomedical QA tasks

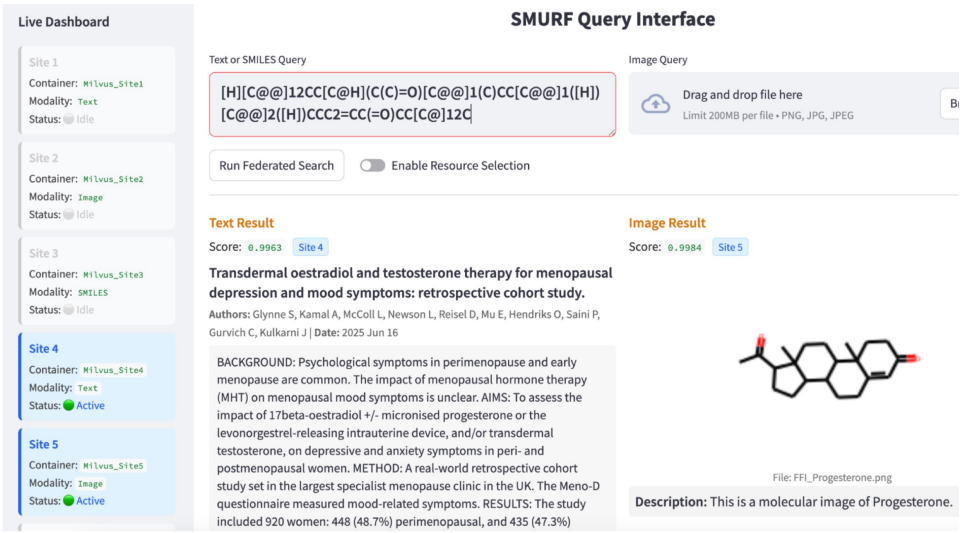


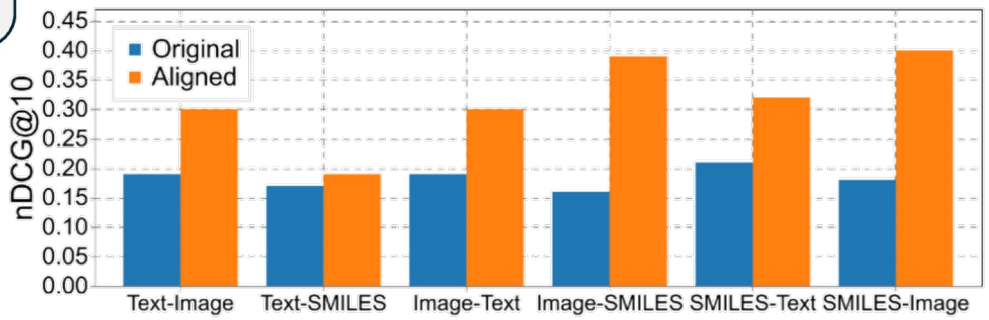
Beyond textual data

Data type	Examples	What the LLM must learn/do
Structured	Databases, CSVs, SQL engines	Generate and interpret queries, aggregate results
Temporal / time-series	Sensors, logs, climate records	Learn temporal reasoning; perform trend analysis
Spatial / geospatial	Maps, satellite imagery	Call GIS APIs; reason with coordinates and regions
Scientific / numeric	Simulation codes, lab data, HPC runs	Request parameters, launch computations, interpret results
Multimodal	Images, spectra, molecules, graphs	Use embeddings or model adapters per modality
Private / dynamic	Company or lab databases, instruments	Handle authentication, provenance, and freshness

Multimodal RAG







Song Young Oh et al.

SMURF: Scientific MUltimodal Retrieval in Federations

Multi-modal RAG vs. database adapters

Say we want an AI assistant to answer: “Show me examples of catalysts that perform well at 200 °C for CO_2 conversion, and include any microscopy images showing the catalyst surface.”

Approach 1: Multi-modal RAG

- The system encodes both **text** (papers, lab notes) and **images** (SEM/TEM micrographs) into a **shared embedding space** (e.g., with CLIP or BLIP-2)
- At query time, the LLM embeds the question and retrieves *semantically related text+image pairs* from a **vector database;** LLM reads and reasons over those retrieved items to generate a natural-language answer, perhaps with thumbnails and captions
- Pros
  - **Semantic flexibility:** Handles vague or descriptive queries ("roughly spherical nanoparticles") even if keywords differ
  - **Cross-modal grounding:** Can reason jointly about visuals and text ("the rough surface in the micrograph corresponds to higher activity")
  - **Unifies unstructured knowledge:** Works when data live in PDFs, figures, reports, not in a structured schema
- Cons
  - **Imprecise for numeric or filtering queries:** "Temperature > 200 °C" is hard to enforce via embeddings
  - **Heavy retrieval & model cost:** Storing and comparing high-dimensional embeddings for many images is expensive
  - **Weak provenance:** Citations and data lineage are harder to guarantee because similarity search is probabilistic

Approach 2: Database adapter (structured query)

- The LLM converts the natural-language query into a **structured query** (e.g., SQL or GraphQL) against a curated experimental database
- The database returns **precise numeric results**: reaction conditions, yields, temperatures. The LLM may then call another retrieval or visualization tool to fetch related images or metadata for the selected entries.
- Pros
  - **High precision, deterministic** filtering and aggregation ("temperature > 200 °C AND efficiency > 85%")
  - **Strong provenance and reproducibility:** Every number ties back to a row in a controlled dataset
  - **Lower latency / smaller footprint** for large numeric datasets
- Cons
  - **Rigid schema:** Cannot easily handle free-form reports or unlabeled images.
  - **Limited semantic generalization:** Fails if user asks in novel phrasing or wants reasoning across multiple modalities
  - **Requires data engineering:** Tables, schema, and API integration must exist

In practice, may want to combine both

- Use **database adapters** for quantitative filtering and verified facts.
- Use **multi-modal RAG** to bring in contextual evidence (plots, figures, methods) and generate an integrated explanation
- For example:
  - Query DB for catalysts > 200 °C → retrieve 5 records
  - RAG retrieves micrographs + paper excerpts
  - LLM synthesizes report with citations and visuals

RAG landscape: Variants and evolution

System / Year	Retrieval Innovation	Generation / Integration Strategy
RAG (Lewis et al., 2020)	Dense Passage Retriever (DPR) encodes queries & documents into vector space for semantic similarity	One-shot retrieval → LLM conditions generation directly on top-k passages
FiD – Fusion-in-Decoder (Izacard & Grave 2021)	Retrieve many passages independently (often > 50)	Decoder jointly attends to all retrieved passages, fusing their evidence during generation
HyDE – Hypothetical Document Expansion (Gao et al. 2023)	Generate a synthetic "ideal answer" → embed → retrieve nearest real docs	Generator guided by retrieved real docs but seeded by synthetic passage
Self-RAG (Asai et al. 2023)	Model autonomously decides when / whether / what to retrieve using special control tokens	Alternates between retrieval ↔ generation within one reasoning loop (self-critique and revision)
GraphRAG (Edge et al. 2024)	Retrieves graph nodes & edges instead of text chunks (knowledge-graph retrieval).	Generator reasons over structured entities and relations to compose graph-grounded answers.
LongRAG (Jiang et al. 2024)	Retrieves larger, semantically grouped or hierarchical chunks optimized for long-context models	Uses long-context generation (4k–32k tokens) to synthesize broader evidence windows

RAG pitfalls

- **Context window overflow / truncation**
  - When too many retrieved tokens exceed the LLM's context limit, older or less-recent chunks get silently dropped, causing loss of critical information or incoherent answers
- **Irrelevant retrieval hurting generation**
  - Poorly matched passages can mislead the model's reasoning, pulling the answer toward unrelated topics or introducing factual noise
- **Citation hallucination**
  - The model fabricates references or associates statements with the wrong source, creating false confidence and undermining provenance
- **Embedding drift when corpora evolve**
  - Updating or re-embedding documents with a new model or altered data distribution changes their vector geometry, breaking previous similarity relationships and degrading retrieval accuracy