Retrieval-Augmented Generation

Bridging Large Language Models and Information Retrieval

Sun Yiqun

Research Scientist @ MTRI

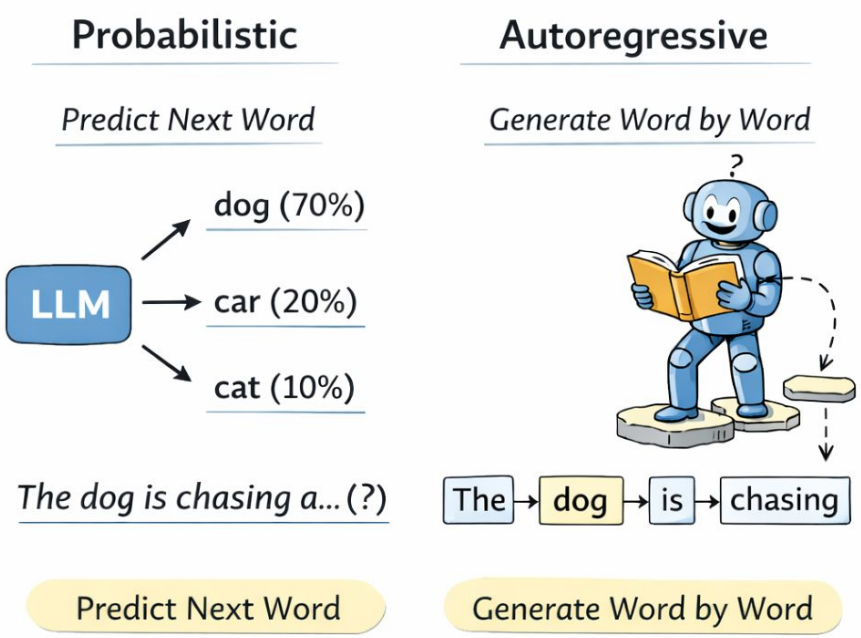
BComp & PhD in CS, NUS

Course Structure & Learning Outcomes

- **LLMs as Black-Box Processors:** Define an LLM operationally, understanding pretraining, instruction tuning, and prompting mechanics.
- **Why LLMs Fail in Practice:** Analyze limitations including parametric staleness, private data gaps, and hallucinations.
- **The Standard RAG Pipeline:** Deconstruct the system architecture (ingest, chunk, index, retrieve, prompt, generate).
- **Retrieval Methods Deep Dive:** Compare sparse (BM25), dense, late interaction, and multi-stage ranking strategies.
- **Open Topics:** Explore evaluation frameworks (RAGAS), agentic architectures, and security.

> **The Core Philosophy:** RAG bridges the advanced language reasoning and generative power of LLMs with the factual reliability, updatability, and strict provenance of classical Information Retrieval systems.

What is a Large Language Model?



Probabilistic Engine: Generates text by mapping sequences of tokens to mathematical probabilities.

Autoregressive Generation: Iteratively predicts the next token based purely on the preceding context sequence.

Generation Control: Temperature controls randomness. Temp=0.0 forces the model to strictly pick the highest probability token.

$$P(w_1, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, \ldots, w_{t-1})$$

Input: "The sky is"
Internal Process: P("blue")=0.85, P("clear")=0.10, P("cloudy")=0.04...
Output: "blue" → Next Input: "The sky is blue"...

The Black-Box Mental Model

Treat the LLM as a stochastic function: **f(prompt) → completion**.

1. The Input

Your prompt is the interface—combining context, system instructions, and formatting rules.

2. The Processor

Performs tasks like summarization or extraction based entirely on statistical pattern continuation.

3. The Output

The generated text. Because reliability isn't guaranteed natively, external retrieval is required.

```
f("Summarize the following report in 2 sentences:
   [Data]")
 = "Here is the summary: [Output text]"
```

Pretraining: How LLMs Become General

Self-Supervised Learning

Models learn language patterns by predicting missing data across massive text corpora.

```
Example: "The quick brown fox jumps over
the [MASK]" → "lazy dog".
```

Produces a foundation model with broad competence, but **no explicit factual database**.

Scaling Laws

Model performance scales predictably and drastically with three main factors.

10x Parameters + 10x Data + 100x Compute = Predictable Loss Reduction

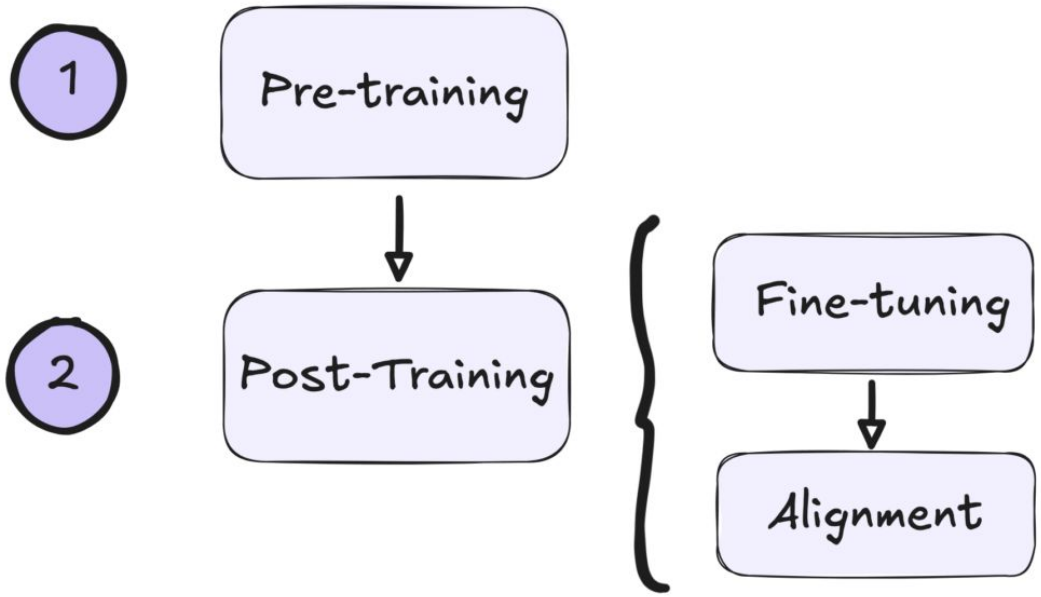
Complex capabilities—like translation and reasoning—often emerge naturally simply by scaling.

The IR Implication: Because pretraining bakes facts amorphously into trillions of opaque matrices, we cannot reliably query an LLM like a SQL database. We must feed it the facts at runtime.

Instruction Tuning & RLHF

Raw foundation models only want to complete sentences. Tuning makes them act as assistants.

- **Instruction Tuning:** Fine-tuning on datasets formatted as explicit instructions. Trains the model to reliably follow strict formats (e.g., "Respond in JSON") and constraints.
- **RLHF: R**einforcement **L**earning from **H**uman **F**eedback aligns outputs based on human ratings. This reduces toxic output and drastically improves conversational helpfulness.



Base Model:
"What is BM25?" → *"What is TF-IDF? What is cosine similarity?" (Continues the pattern)*

Instruction Tuned Model:
"What is BM25?" → *"BM25 is a ranking function used in information retrieval..." (Answers it)*

Image from https://mlops.community/pretraining-breaking-down-the-modern-llm-training-pipeline/

Prompting Styles & Techniques

Zero-Shot

Relying purely on tuned generalization without providing any demonstrations.

```
Prompt: Extract topic.
"ColBERT uses late interaction for
passage retrieval."

Output: Passage Retrieval
```

Few-Shot

Providing input-output demonstrations to establish a strict pattern.

```
Prompt:
Q: "Library hours?" → Schedule
Q: "Where is the lab?" → Location
Q: "Gym closing time?" →

Output: Schedule
```

Chain-of-Thought

Eliciting reasoning steps to solve complex logic before answering.

```
Prompt: Doc: "RTX 4090 trains NNs
fast". Query: "Good deep learning
GPU?" Relevant? Think step-by-step.

Output: NNs means neural networks
(deep learning). RTX 4090 is a GPU.
The doc answers the query. Yes.
```

The Trade-Off: While Few-Shot and Chain-of-Thought drastically improve output accuracy and reasoning logic, they consume significant portions of the model's finite "Context Window" token limits.

Why LLMs Fail

Understanding predictable failure modes to motivate Information Retrieval integration.

Issues with Parametric Knowledge

Temporal Staleness

Factual recall is **frozen at the training cutoff**.

- Unaware of breaking news.
- Misses recent software updates.
- Fails on shifting corporate policies.

Impact: *The model confidently provides advice based on a world that ended in 2023.*

The Private Data Gap

Models train exclusively on **public internet data**.

- Zero knowledge of internal wikis.
- No access to proprietary code.
- Unaware of enterprise records.

Impact: *Cannot answer basic workflow questions like "What did our CEO say in yesterday's meeting?"*

The Architectural Solution: Decouple knowledge storage from language generation. Retain the LLM purely for its reasoning and semantic synthesis capabilities, but strictly utilize an external database for providing the facts.

Hallucination: Causes

Fluency Optimization

Models optimize for **token probability**, learning to sound utterly convincing rather than objectively true.

Loss function ignores real-world ground truth.

Data Compression

Compressing terabytes of text **blurs factual boundaries**, forcing the model to mathematically approximate facts.

No distinct "memory slots" for individual records.

Over-generalization

Models map concepts to shared vector spaces, falsely **merging distinct entities** simply because they co-occurred.

Conflates authors who write on similar topics.

Key Insight: LLMs are not broken databases; they are highly advanced pattern-matching "dream machines." Hallucination is not a bug—it is the model's intended default behavior when factual grounding is missing.

Hallucination: Scenarios

🎓 The Academic Mirage

Scenario: Generating plausible paper titles with fake DOIs.

Why it happens: Authors and keywords co-occur in the weights. A structurally perfect fake citation looks probabilistically "correct".

Impact: Users cite non-existent papers, failing integrity checks.

The Phantom API

Scenario: Generating code that calls a library function that never existed.

Why it happens: The model learned the naming conventions of the library and confidently extrapolated a fabricated method.

Impact: Code compilation fails, wasting developer hours.

How to Mitigate This: By instructing the LLM to condition its generation on retrieved text snippets, we anchor its probabilistic sequences to concrete, verifiable facts.

The Standard RAG Pipeline

Combining the power of external knowledge bases with language generation.

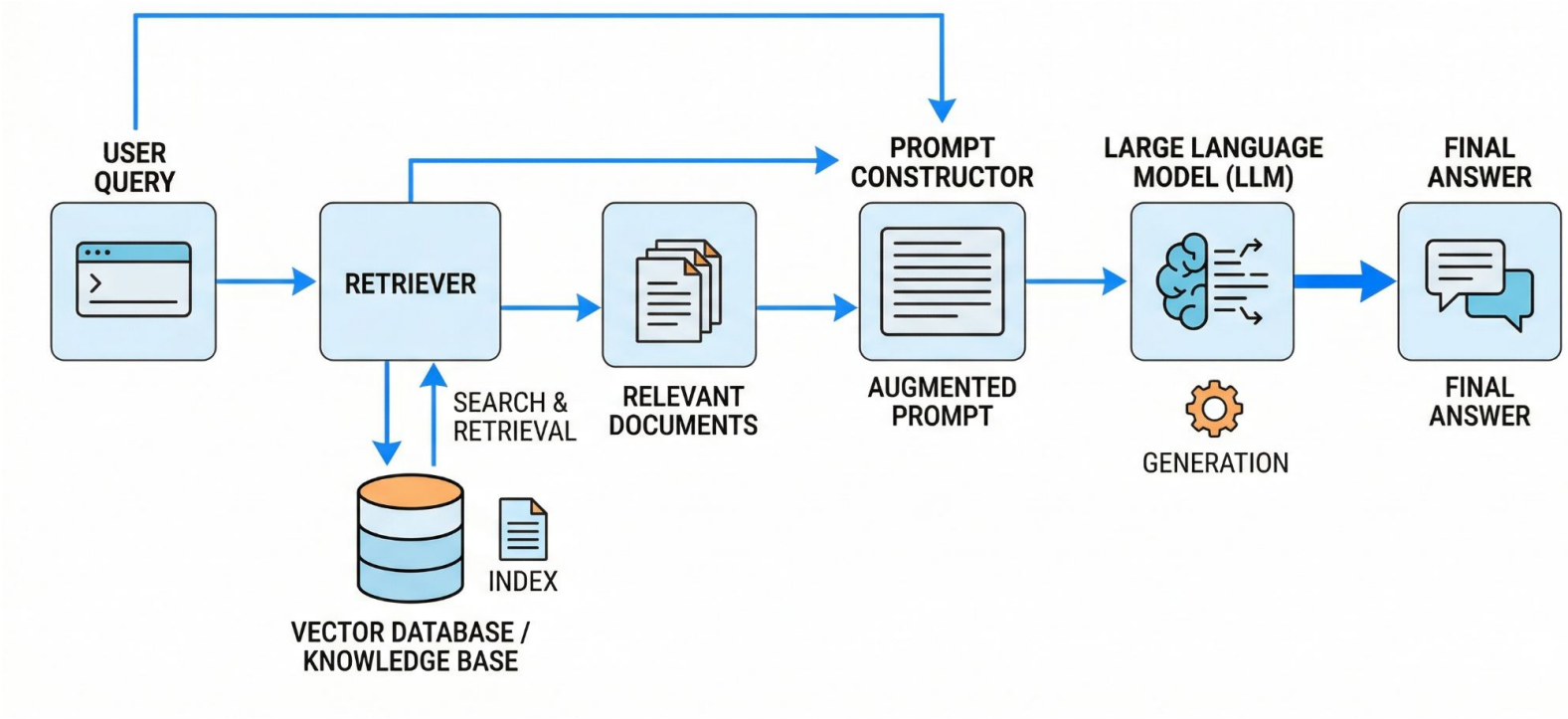
What is RAG?

Retrieval-Augmented Generation grounds AI outputs in authoritative evidence.

The Concept: Instead of relying on frozen internal memory, we give the LLM a **search engine**.

The Mechanism: The system searches a database for relevant facts, forcing the LLM to read them before answering.

The Result: Bridges advanced language skills with your company's private and up-to-date documents.



The Two Operational Phases

Phase 1: Indexing (Offline)

Preparing private data to be searchable.

1. **Ingest:** Extract raw text from PDFs, HTML or documents.
2. **Chunk:** Split text into small, e.g. 512-token segments.
3. **Embed:** Convert to semantic vectors.
4. **Store:** Save to a Vector Database.

Phase 2: Retrieval (Online)

Answering queries dynamically.

1. **Query:** User asks a question.
2. **Search:** Find Top-K chunks via similarity.
3. **Prompt:** Combine chunks + query.
4. **Generate:** Synthesize cited answer.

Raw Data → [Phase 1] → Vector DB → [Phase 2] → LLM → Cited Answer

Component 1: Data Chunking

Chunking

Splitting text strictly by character count breaks sentences. We use a sliding token window.

> "IR obtains resources. It is crucial for web search."

Chunk 1: "IR obtains resources. It is crucial"

Chunk 2: "It is crucial for web search."

* A 50-token overlap preserves sentence context boundaries across chunks, ensuring no keyword is orphaned.

The Coreference Problem

Pronouns lose their subjects when separated by chunk boundaries, destroying retrieval.

Chunk A: "Tim Cook is Apple's CEO."
Chunk B: "He launched the Vision Pro."

If a user asks *"Who launched Vision Pro?"*, the DB fails to retrieve **Chunk B** because the word "Apple" or "Tim Cook" isn't present.

Result:
The vector embedding for Chunk B completely lacks the semantic concept of "Tim Cook".

Advanced solutions preserve context through **semantic chunking**, **pre-resolving entities**, **adding metadata tags**, or **hierarchical retrieval**.

Component 2: Embeddings & Search

1. The Embedding Function

$$E : \text{Text} \to \mathbb{R}^d$$

Transforms **text** into a **vector** in a **semantic space** (e.g. 768-4096 dimensions), represented an array of numbers.

In such space, **geometric proximity** represents **semantic similarity**.

2. Nearest Neighbor Search

$$D^* = \underset{D \in \text{Corpus}}{\arg\max}\, \text{Sim}\left(E(Q), E(D)\right)$$

Relevant documents are retrieved by computing the **geometric proximity** (e.g. cosine similarity) in the semantic space.

Vector DBs e.g. Milvus and FAISS use **Approximate Nearest Neighbor (ANN)** to compute with high efficiency and low latency.

Semantic Power: The query "How to fix a broken screen" and the document "Smartphone display repair guide" share zero meaningful keywords. BM25 fails here, but their dense vectors will be highly similar.

Component 3: Prompting & Synthesis

```
[System]
Answer using ONLY the provided context. If the answer
is not in the context, say "I don't know." Cite the
DocID.

[Context]
Doc-12: The Eiffel Tower was completed in 1889.
Doc-45: It was built for the World's Fair.

[User]
When was the Eiffel Tower built?
```

Prompt Architecture

System Constraints: Forces strict citation and refusal rules.

Dynamic Context: Raw text injected here.

User Intent: Placed at the end to command immediate attention.

LLM Expected Output:
"The Eiffel Tower was built in 1889 [Doc-12]."

If Context is Empty:
"I don't know." (Forced refusal prevents hallucination)

Common Pipeline Failure Modes

Retrieval Failures

Missed Context (Low Recall): Vector search misses the answer; LLM is forced to guess.
Diagnostic: Low Recall@k metric.

Noisy Retrieval (Low Precision): Search brings back distractors, confusing the LLM.
Diagnostic: Low Precision@k metric.

Generation Failures

Ignored Context: LLM relies on its internal memory instead of the provided text.

Hallucination: LLM draws false conclusions from chunks.
Diagnostic: RAGAS Faithfulness score.

Crucial Engineering Practice: Always log the exact retrieved chunk texts alongside the user query. You cannot debug a generative hallucination if you don't know what facts the LLM was given.

Practical Implementation

Building a RAG System with LlamaIndex

Phase 1: Data Ingestion & Indexing

The LlamaIndex Framework

LlamaIndex provides high-level abstractions to connect custom data sources to large language models.

VectorStoreIndex: Automatically handles the chunking (splitting text) and embedding (calling an embedding API) of your documents, storing them in a searchable graph.

Python

```
from llama_index.core import Document,
VectorStoreIndex


documents =
[Document(text=ds["train"]["content"][0])]

index =
VectorStoreIndex.from_documents(documents)
```

Phase 2: The Chat Engine

Querying with Memory

The as_query_engine() method converts the index into a Q&A interface, allowing you to ask questions about your indexed documents.

Internally, the query is encoded into an embedding vector, retrieval is performed on the index, and the returned results populate the LLM's context to generate the final answer.

Python

```
query_engine = index.as_query_engine()

response = query_engine.query(
    "What position does Harry Potter play on his Quidditch team at Hogwarts?"
)

print(response)
```

Try it yourself: https://tinyurl.com/cs3245ragdemo

Retrieval Methods

An Information Retrieval Deep Dive

Sparse Retrieval: TF-IDF vs. BM25

The TF-IDF Limitation

TF (Term Frequency): Unbounded scaling disproportionately favors long, repetitive documents.

IDF: Accurately rewards rare terms.

Term Saturation Example: If "RAG" appears 1 time, BM25 score = 2.0. If it appears 10 times, score = 3.5. If it appears 100 times, score = 3.8. The impact tapers off horizontally to kill keyword stuffing.

The BM25

$$\text{Score}(Q, D) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{\text{TF}(q_i, D) \cdot (k_1 + 1)}{\text{TF}(q_i, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

Saturation (k_1): Caps frequency impact.

Length Penalty (b): Normalizes long documents, preventing massive Wikipedia pages from dominating short, precise FAQ pages.

Dense Retrieval: Models & Training

Contrastive Learning

Pulls queries toward relevant docs, pushes away irrelevant ones.

$$L_q = -\log \frac{e^{\text{sim}(q,d^+)}}{e^{\text{sim}(q,d^+)} + \sum_{i=1}^{k} e^{\text{sim}(q,d_i^-)}}$$

q: The query representation.

d^+: The relevant document.

d_i^-: The i-th irrelevant document (of k).

Pros & Cons

- **Semantic Match:** Excels at synonyms.
- **Cross-Lingual:** Matches across languages.
- **Domain Shift:** Crashes on unseen jargon.
- **Exact IDs:** Struggles with "Error 0x88A".

The Importance of Hard Negatives: A model trained with random negatives learns slowly. The most effective training uses *Hard Negatives*—documents that share exact keywords with the query but don't actually answer it, forcing the encoder to learn deep semantic differences instead of keyword overlap.

Late Interaction (ColBERT)

Token-Level MaxSim

Encodes every token individually. Aggregates maximum similarities:

$$\text{Score} = \sum_{i=1}^{|Q|} \max_{j \in [1,|D|]} (E_{q_i} \cdot E_{d_j})$$

Find the best doc token match for each query token, then sum. This bypasses the bottleneck of a single `[CLS]` token embedding.

Pros & Cons

- **High Precision:** Captures structural nuance missed by single-vector compression.
- **Generalization:** Handles unseen domains.
- **Storage:** Requires ~10-100x more space.
- **Latency:** Query time is slower.

How MaxSim Works: The query token "Apple" matches strongly with the document token "MacBook", while "Fruit" matches with "Orchard". The final score sums these individual peak similarities, preserving deep contextual separation.

Multi-Stage Reranking

The Pipeline Pattern

1. **Retrieve (Fast):** Bi-Encoders instantly scan millions of docs → Returns Top 100.
2. **Rerank (Precise):** Cross-Encoder processes query + Top 100 docs *simultaneously* → Returns Top 5.

Allows deep cross-attention mapping between query and document words.

Pros & Cons

- **Maximum Relevance:** State-of-the-art accuracy via cross-attention.
- **Zero-Shot Transfer:** Excellent at generalizing to new domains.
- **Bottleneck:** Unfathomably expensive. Used strictly as a secondary filter.

Latency: An ANNS scanning 1 million documents takes ~20ms. Running a Cross-Encoder over those same 1 million documents would take days. Running a reranker over the Top 100 documents takes seconds.

Hybrid Search

Why Hybrid Search?

Combines Sparse and Dense systems to cover diverse intents.

BM25 Excels At: Exact acronyms, IDs.

Dense Excels At: Conceptual phrasing.

Hybrid retrieval maximizes overall search relevance by combining the **exact lexical matching strengths** of sparse methods with the **deep semantic and contextual understanding** of dense embeddings.

Reciprocal Rank Fusion (RRF)

Merges lists with mathematically incompatible scores.

$$RRF(d) = \sum_{r \in R} \frac{1}{k + r(d)}$$

Example: If k=60, Rank 1 yields 1/61=0.016. Rank 10 yields 1/70=0.014. RRF heavily rewards documents sitting near the very top of both lists.

Evaluating Retrieval Quality

Core IR Metrics

Recall@k: Fraction of relevant docs retrieved. *Did we find the facts?*

MRR: Avg reciprocal rank of the *first* relevant doc.

$$RR = 1 / Rank_{first}$$

NDCG: Evaluates entire ranking order.

Logarithmic discounting heavily rewards finding relevant documents at Rank 1 versus Rank 10.

Recall ensures the LLM receives the facts it needs to avoid hallucinating.
MRR measures if the absolute best fact is at the very top (crucial for strict LLM context limits).
NDCG ensures the overall list is perfectly ordered, minimizing prompt distractors.

Assume **3 total relevant docs**.

Rank	Rel?	Rec@k	RR	DCG
1	0	0.00	0.0	0.00
2	1	0.33	**0.50**	0.63
3	1	0.66	-	0.50
4	0	0.66	-	0.00
5	1	1.00	-	0.38

DCG@5 = 1.51.

Ideal DCG@5= 1 + 0.63 + 0.5 = 2.13.

→ **NDCG@5 = 1.51/ 2.13 = 0.71**.

Open Topics

Evaluation Frameworks, Ongoing Research, and Recommended Reading

Evaluating RAG Architectures

Isolating the Layers

Because bad retrieval can mimic an LLM hallucination (and vice versa), evaluation must isolate the components.

Retrieval Quality: We evaluate the vector engine using classic IR standards like Recall@k, nDCG, and Mean Reciprocal Rank (MRR).

Generation Quality: We evaluate the LLM output for logical correctness and strict faithfulness to the retrieved evidence. Frameworks like RAGAS propose reference-free programmatic metrics.

Image from https://www.analyticsvidhya.com/blog/2024/05/a-beginners-guide-to-evaluating-rag-pipelines-using-ragas/

Ongoing Research & Open Topics

Agentic RAG

Moving beyond single-shot retrieval to multi-hop reasoning with tool usage.

Example: "Compare Q3 and Q4." The agent searches Q3, evaluates the result, realizes it needs Q4, and loops back to issue a second search via an API before formulating an answer.

Graph RAG

Enhancing retrieval with structured entity relationships.

Example: Extracting a knowledge graph with community detection to accurately answer global summarization queries like "What are the main themes across this entire corpus?"

Security & Poisoning

Treating retrieved text as an untrusted attack vector.

Example: "Prompt Injections" hidden in retrieved web pages hijack system instructions. Or "PoisonedRAG" attacks deliberately corrupting the vector space with malicious chunks.

Recommended Reading List

Foundational papers for understanding RAG and modern retrieval techniques:

- **Lewis et al. (2020):** *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. The foundational RAG architecture.
- **Khattab & Zaharia (2020):** *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*.
- **Thakur et al. (2021):** *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of IR Models*. Out-of-domain retrieval limits.
- **Es et al. (2023):** *RAGAS: Automated Evaluation of Retrieval Augmented Generation*.
- **Edge et al. (2024):** *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*.

Questions?

Thank you for your attention.

Contact: duke.sun@mtri.co.jp