Retrieval-Augmented Generation

Grounding LLMs in External Knowledge

NLP Course – Lecture 1

Advanced Topics in Natural Language Processing

Why Do LLMs Hallucinate?

The Problem

- LLMs predict the most likely next token
- They have no access to real-time information
- Knowledge is frozen at training time
- No mechanism to verify facts

The Solution: RAG

- Retrieve relevant documents first
- Augment the prompt with facts
- Generate grounded responses
- Cite sources for verification

This lecture: How to ground LLMs in external knowledge

RAG is the most widely deployed technique for making LLMs factually accurate.

Act I: RAG & AI Agents

Making LLMs Useful in the Real World

The Hallucination Problem

The Problem

LLMs confidently state wrong facts:

- "The current CEO of OpenAI is..." (outdated)
- "The 2024 Olympic gold medalist was..." (unknown)
- "Your company's Q3 revenue was..." (not in training data)

Root Causes

- Knowledge frozen at training time
- No access to private/recent information
- Model "fills in gaps" with plausible text

Why This Matters

For real applications, we need:

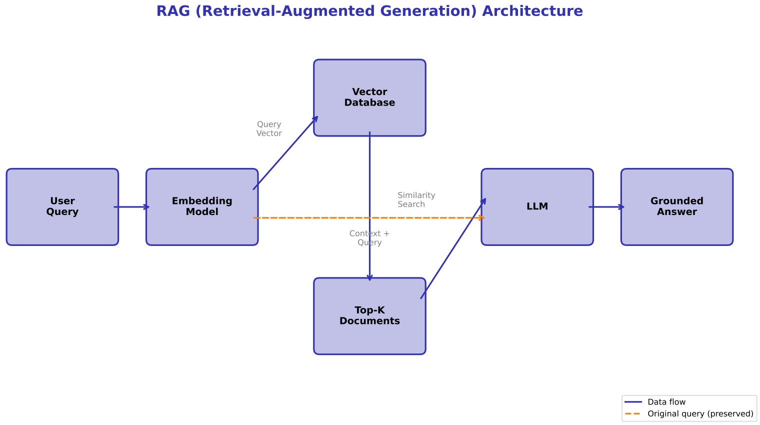
- Access to current information
- Grounding in verifiable sources
- Ability to say "I don't know"

Connection to Ethics Week

Remember: LLMs don't "know" anything – they predict tokens. Without grounding, this is dangerous.

Solution: Don't try to store everything in parameters. Retrieve at inference time.

RAG: The Elegant Solution



Key insight: Separation of concerns – parametric knowledge (the model) vs. retrieved knowledge (the database)

Core Idea
Instead of: $p(y|x)$ (generate from query alone)
RAG marginalizes over retrieved documents:

$$p(y|x) = \sum_{z \in \text{top-}k} p(z|x) \cdot p(y|x, z)$$

Why no z on left? We sum over all z (marginalization) – the result depends only on x.

Where: $x =$ query, $z =$ retrieved doc, $y =$ response

Key Equation: Dense Retrieval

$$\text{sim}(q, d) = \frac{E_q(q)^T \cdot E_d(d)}{||E_q(q)|| \cdot ||E_d(d)||}$$

Retrieval probability (softmax):

$$p(z_i|x) = \frac{\exp(\text{sim}(x, z_i)/\tau)}{\sum_{j=1}^{k} \exp(\text{sim}(x, z_j)/\tau)}$$

You Already Know This!
This is just attention over an external memory.

Lewis et al. (2020): "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"

RAG Formula: A Concrete Example

Query x: "What is the capital of France?"

Retrieved Documents (with similarity scores):

z_1: "Paris is the capital and largest city of France..."	0.92
z_2: "France is a country in Western Europe..."	0.71
z_3: "The Eiffel Tower is located in Paris..."	0.65

Generation Probabilities $p(y|x, z_i)$:

For answer y = "Paris":

- $p(y|x, z_1) = \mathbf{0.95}$ – directly states "Paris is capital"
- $p(y|x, z_2) = \mathbf{0.40}$ – mentions France, not Paris
- $p(y|x, z_3) = \mathbf{0.70}$ – mentions Paris, not as capital

Step 1: Retrieval Probabilities

Softmax: $p(z_i|x) = \frac{e^{\text{sim}_i}}{\sum_j e^{\text{sim}_j}}$

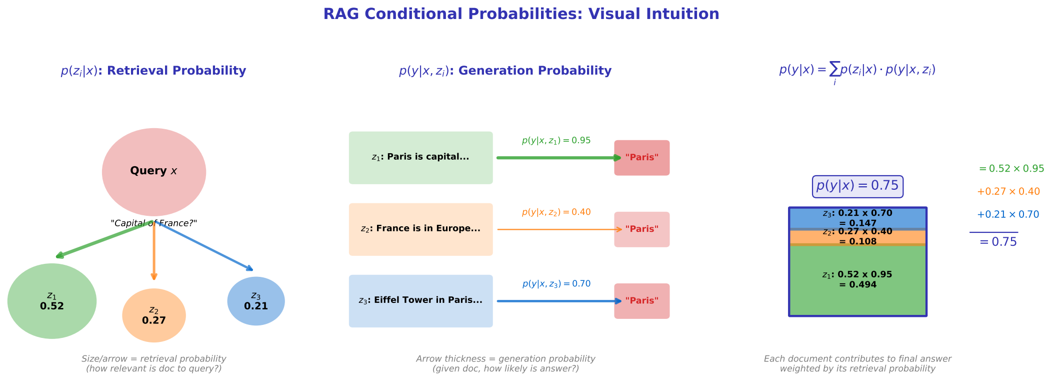
$$\begin{aligned} p(z_1|x) &= 0.52 \quad \text{(most relevant)} \\ p(z_2|x) &= 0.27 \\ p(z_3|x) &= 0.21 \end{aligned}$$

Step 2: Marginalization

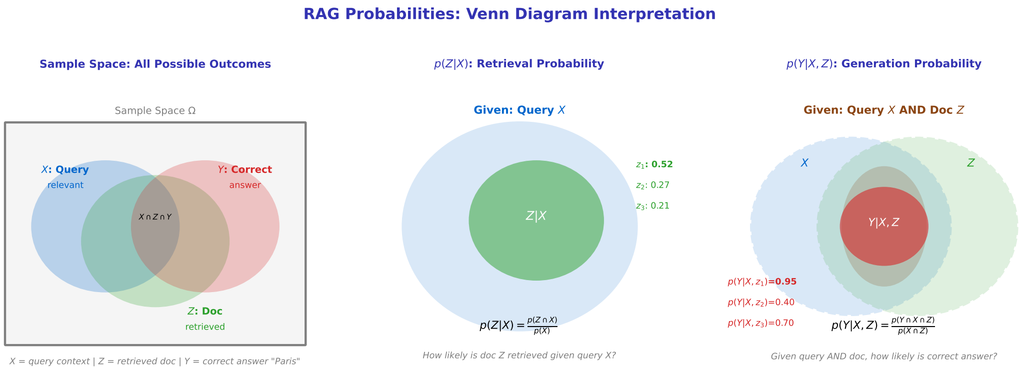
$$p(y|x) = \sum_{i=1}^{3} p(z_i|x) \cdot p(y|x, z_i)$$

$$\begin{aligned} &= 0.52 \times 0.95 \quad \text{(from } z_1\text{)} \\ &+ 0.27 \times 0.40 \quad \text{(from } z_2\text{)} \\ &+ 0.21 \times 0.70 \quad \text{(from } z_3\text{)} \\ &= 0.494 + 0.108 + 0.147 \\ &= \mathbf{0.75} \end{aligned}$$

Key: $p(z|x)$ = how relevant is doc? $p(y|x, z)$ = given this doc, how likely is answer?



Marginalization: Sum over all docs, each weighted by retrieval probability times generation probability



Conditional probability: We restrict the sample space to the given event, then measure probability within it

RAG Notation Reference

Query and Response

- x – User query (input question)
- y – Generated response (output)
- q – Query after embedding

Documents and Retrieval

- z – Retrieved document(s)
- z_i – The i-th retrieved document
- $\mathcal{Z}$ – Full document corpus
- d – Single document in corpus
- k – Number of documents retrieved (top-k)

Embedding Functions

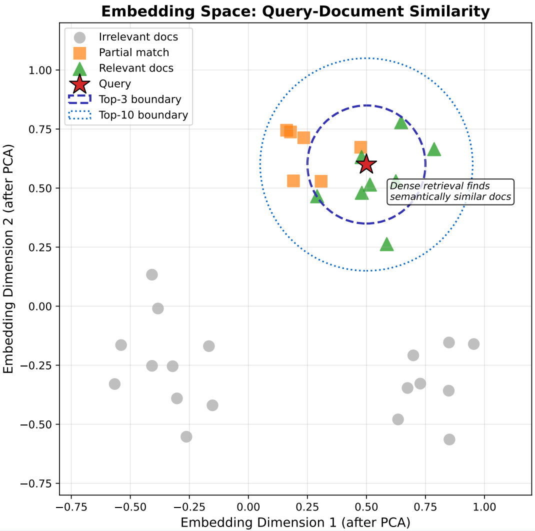
- $E_q(\cdot)$ – Query encoder (embeds queries)
- $E_d(\cdot)$ – Document encoder (embeds documents)
- Often $E_q = E_d$ (same encoder for both)

Similarity and Probability

- $\text{sim}(q, d)$ – Cosine similarity between query and document vectors
- τ – Temperature parameter (controls softmax sharpness)
- $p(z|x)$ – Probability of retrieving document z given query x
- $p(y|x, z)$ – Generation probability given query and retrieved docs

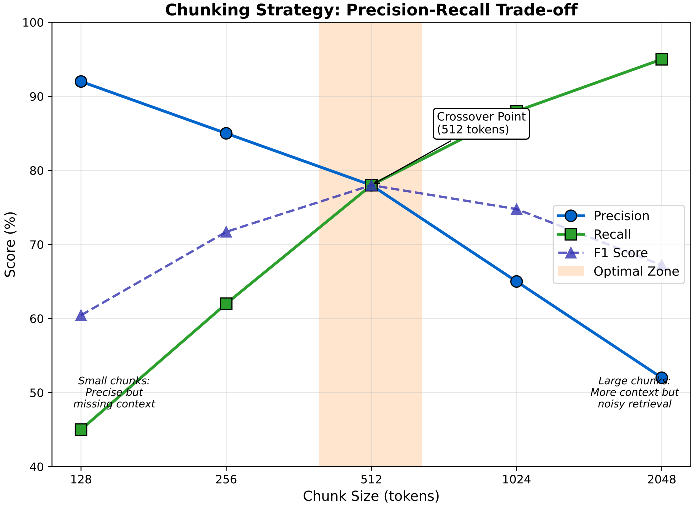
Understanding notation: Embedding similarity drives retrieval, retrieval augments generation

Visualizing the Embedding Space



Dense retrieval works by finding documents whose embeddings are closest to the query embedding

Chunking Trade-offs: Precision vs Recall



Rule of thumb: Start with 512 tokens, adjust based on your retrieval quality metrics

RAG Components in Practice

Embedding Models

- Sentence transformers
- OpenAI embeddings
- Cohere, Voyage, etc.

Output

Dense vectors (e.g., 1536-dim)

Vector Databases

- FAISS (Facebook)
- Pinecone (managed)
- ChromaDB (local)
- Weaviate, Milvus

Key Operation

Approximate nearest neighbor search

Chunking Strategies

- Fixed-size (512 tokens)
- Semantic (by paragraph)
- Hierarchical (nested)
- Sliding window

Trade-off

Small chunks = precise retrieval
Large chunks = more context

The choice of chunking strategy significantly impacts retrieval quality

Vector Databases Explained

What Is a Vector Database?
Specialized database for storing and querying high-dimensional vectors (embeddings).

Key Operation: ANN Search
Approximate Nearest Neighbor (ANN):

- Exact search is $O(n)$ – too slow
- ANN trades accuracy for speed
- Typical: 95%+ recall at 10-100x speedup

Index Structures

- HNSW (Hierarchical Navigable Small World)
- IVF (Inverted File Index)
- LSH (Locality Sensitive Hashing)

Popular Vector Databases
Open Source:

- FAISS (Meta) – In-memory, very fast
- ChromaDB – Simple, Python-native
- Milvus – Distributed, scalable
- Weaviate – GraphQL interface

Managed Services:

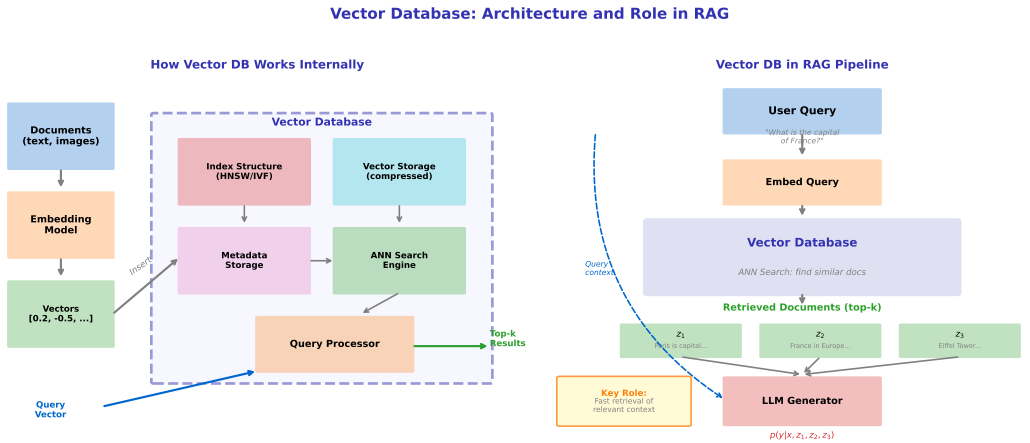
- Pinecone – Fully managed
- Qdrant – Self-hosted or cloud

Typical Workflow

1. Embed documents $\rightarrow$ vectors
2. Store vectors with metadata
3. Query: embed query $\rightarrow$ find top-k similar

Vector databases are the "memory" that makes RAG possible at scale

Vector Database: Architecture and Role in RAG



Vector DBs enable fast retrieval: embed documents once, search in milliseconds at query time

Approximate Nearest Neighbor: Why and How

Approximate Nearest Neighbor (ANN): The Core Idea

The Problem

```
Given: Database of n vectors
       D = {d_1, d_2, ..., d_n}

Query: Find k vectors closest to q

Exact solution requires:
  - Compute distance to ALL n vectors
  - Sort and return top-k
  - Time: O(n) per query
```

n = 1 billion? That is 1 billion distance calculations per query!

The Mathematics

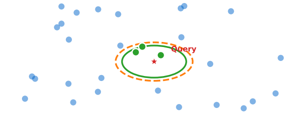
```
Exact k-NN:
  N_k(q) = argmin |S|=k  max  ||q - d||
                       d in S

c-Approximate k-NN:
  For all d in ANN_k(q):
    ||q - d|| <= c * ||q - d*||

  where d* is the true k-th neighbor
  and c >= 1 is the approximation factor
```

c = 1.05 means we accept neighbors at most 5% farther than optimal

Visual Intuition



The Trade-off

Method	Time	Recall	Use Case
Exact (brute)	O(n)	100%	Small datasets
IVF	O(sqrt(n))	~95%	Medium scale
HNSW	O(log n)	~99%	Production
LSH	O(1)*	~90%	Massive scale

Key: Accept 1-5% accuracy loss for 100-1000x speedup

** LSH: O(1) query but O(n) space for hash tables*

ANN is the key enabler for billion-scale vector search: trade small accuracy for massive speedup

The c-Approximate k-NN Guarantee

Exact k-NN Problem
Given query q and database $D = \{d_1, \ldots, d_n\}$, find:

$$N_k(q) = \underset{S \subseteq D, |S|=k}{\arg\min} \max_{d \in S} \|q - d\|$$

c-Approximate k-NN
An algorithm returns $\text{ANN}_k(q)$ such that:

$$\boxed{\forall d \in \text{ANN}_k(q) : \quad \|q - d\| \leq c \cdot \|q - d^*\|}$$

where d^* is the **true k-th nearest neighbor** and $c \geq 1$ is the **approximation factor**.

What This Means

- $c = 1.0$: Exact (no approximation)
- $c = 1.05$: At most 5% farther
- $c = 1.10$: At most 10% farther

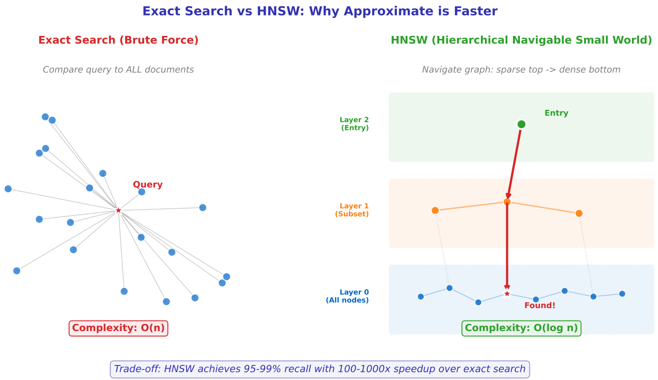
The Trade-off

$c \to 1$	Slower, exact
$c > 1$	Faster, approximate

In Practice
Most systems achieve $c \approx 1.01$ to 1.05 with 100–1000$\times$ speedup.

The c-approximation guarantee means returned neighbors are at most c times farther than the true nearest

HNSW: The Most Popular ANN Algorithm



HNSW builds a navigable graph: start at sparse top layers, greedily descend to find nearest neighbors

HNSW: Hierarchical Graph Structure

The Key Idea
Combine two concepts:

1. **Skip Lists**: Hierarchical layers for $O(\log n)$ traversal
2. **Navigable Small World**: Each node connected to "nearby" nodes

Layer Structure

- Layer 0: All n nodes (dense)
- Layer 1: $\sim n/m_L$ nodes
- Layer 2: $\sim n/m_L^2$ nodes
- Top: Few entry points

How are nodes assigned?
Each node's max layer is **random**:

$$\ell = \lfloor -\ln(\text{uniform}(0,1)) \cdot m_L \rfloor$$

Most nodes: layer 0 only. Few "lucky" nodes reach higher layers (like express stops).

Construction Algorithm
For each new vector v:

1. Sample max layer ℓ (formula on left)
2. Insert v into layers $0, 1, \ldots, \ell$
3. At each layer, connect to M nearest neighbors

Key Parameters

M	Max connections/node
ef	Search beam width
m_L	Level multiplier

Typical: $M = 16$, $ef = 100$, $m_L = 1/\ln(M)$

Intuition: Like a subway system – express lines (top layers) connect major hubs, local lines (layer 0) reach everywhere.

The hierarchical structure enables logarithmic search: coarse navigation at top, fine-grained at bottom

HNSW: The Search Algorithm

Greedy Search Procedure

1. Start at entry point (top layer)
2. At each layer:
   - Greedily move to nearest neighbor
   - Repeat until no closer neighbor exists
3. Descend to next layer
4. At layer 0: expand search with beam width *ef*
5. Return top-k from candidates

Complexity

Search:	$O(\log n)$
Insert:	$O(\log n)$
Space:	$O(n \cdot M)$

Why It Works

Small World Property: Any two nodes connected by short path ($\sim \log n$ hops).

Hierarchical Speedup: Top layers skip large distances; bottom layers refine.

Pseudocode

```
search(q, k, ef):
  ep = entry_point
  for layer in top...1:
    ep = greedy(q, ep, layer)
  cands = beam(q, ep, L0, ef)
  return top_k(cands, k)
```

ef controls accuracy/speed trade-off.

HNSW achieves >99% recall with 10–100× speedup; used in FAISS, Pinecone, Weaviate, Qdrant

HNSW: A Simple Example

Setup: 8 cities, find nearest to query "Berlin"

Layer 2 (Top) – 2 nodes

Entry points: Paris, Tokyo

Query: Berlin → Check Paris, Tokyo
→ Paris closer → **go to Paris**

Layer 1 – 4 nodes

Paris, Tokyo, London, Sydney

From Paris → Check neighbors
→ London closer → **go to London**

Layer 0 (Bottom) – all 8 nodes

From London → Check all neighbors
→ **Found: Amsterdam** (nearest!)

What Happened

Layer 2:	2 comparisons
Layer 1:	3 comparisons
Layer 0:	4 comparisons
Total:	**9 comparisons**

Brute Force
8 comparisons (check all)

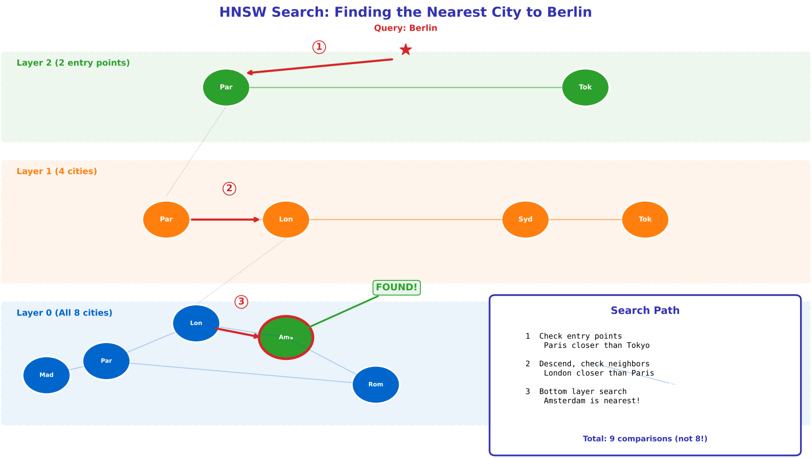
With 1 Billion Nodes

Brute:	1,000,000,000
HNSW:	∼30 (log scale!)

Key Insight
Top layers = "highways"
Bottom layer = "local streets"

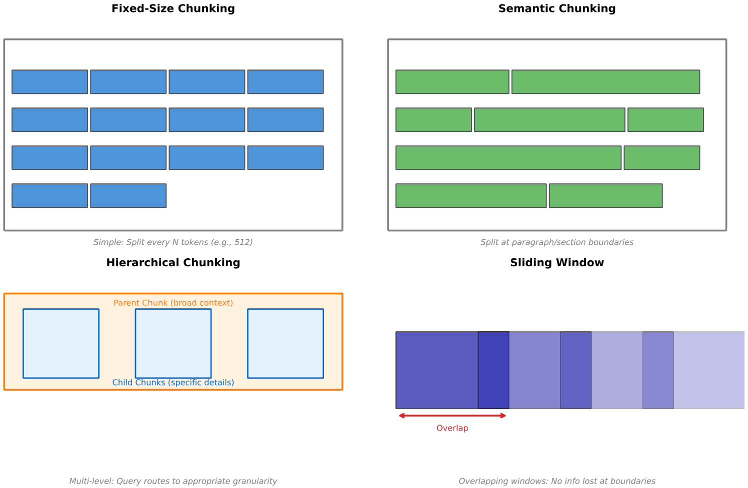
HNSW is like using a map: zoom out to find the region, then zoom in to find the exact location

HNSW: Visual Walkthrough



Each layer narrows the search: start broad at the top, refine at the bottom

Chunking Strategies Deep Dive



Chunking is often the difference between RAG that works and RAG that fails – start with 512 tokens, 10% overlap

RAG Evolution: From Naive to Agentic

Naive RAG

- Simple retrieve-then-generate
- Fixed number of chunks
- No query preprocessing

Advanced RAG

- Query rewriting
- Re-ranking retrieved documents
- Iterative retrieval
- Multi-stage retrieval

Modular RAG

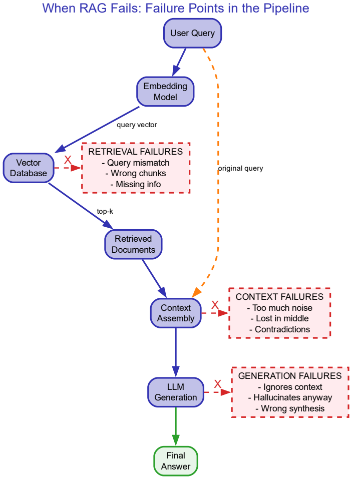
- Self-RAG: decide *when* to retrieve
- CRAG: correct retrieval errors
- Adaptive: retrieve more if needed

Agentic RAG (2024+)

- Agent decides retrieval strategy
- Multiple retrieval sources
- Tool use for specialized queries

Trend: More intelligence in the retrieval process, not just generation

When RAG Fails: Failure Points in the Pipeline



RAG requires careful engineering at every pipeline stage

RAG Failure Mitigations

Retrieval Fixes

- Query expansion/rewriting
- Multi-stage retrieval
- Better chunking strategies
- Cross-encoder re-ranking
- Multiple retrieval passes

Context Fixes

- Smart chunk ordering
- Compression/summarization
- Relevance filtering
- Hierarchical retrieval
- Attention to chunk boundaries

Generation Fixes

- Instruction tuning for RAG
- Citation requirements
- Self-consistency checks
- Confidence calibration
- Fallback to "I don't know"

"Lost in the middle" problem: LLMs often ignore content in the middle of long contexts.
Solution: Place most relevant chunks at beginning and end.

Each failure mode has specific mitigations – production RAG requires all of them

Key Takeaways: RAG

1. **RAG solves hallucination** by grounding LLMs in external documents
2. **Vector search** enables millisecond retrieval from billions of documents
3. **HNSW** provides O(log n) approximate nearest neighbor search
4. **Chunking strategy** critically affects retrieval quality
5. **RAG can fail** at retrieval, ranking, or generation stages

Key Equations:

- Dense retrieval: $\text{sim}(q, d) = \cos(E_q(q), E_d(d))$
- RAG probability: $p(y|x) = \sum_z p(z|x) \cdot p(y|x, z)$

RAG is the foundation of most production LLM applications today.

Further Reading: RAG

Foundational Papers:

- Lewis et al. (2020) - "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"
- Karpukhin et al. (2020) - "Dense Passage Retrieval"
- Malkov & Yashunin (2018) - "HNSW: Hierarchical Navigable Small World Graphs"

Tools & Frameworks:

- Vector DBs: Pinecone, Weaviate, ChromaDB, FAISS
- Frameworks: LangChain, LlamaIndex

Repository: github.com/Digital-AI-Finance/Natural-Language-Processing