GraphRAG Research Assistant - Technical Deep Dive

A comprehensive explanation of every component, the underlying mathematics, and a complete dry run from search query to final output.

Table of Contents

System Architecture

```
┌──────────────────────────────────────────────────────────────────────
────┐
│                              USER INPUT
│
│                  "transformer attention mechanism"
│
└──────────────────────────────────────────────────────────────────────
────┘
                                    │
                                    ▼
┌──────────────────────────────────────────────────────────────────────
────┐
│                           1. PAPER FETCHING
│
│                       FetcherAgent → arXiv API
│
│          Returns: papers[] with {id, title, abstract, authors, link}
│
└──────────────────────────────────────────────────────────────────────
────┘
                                    │
                     ┌──────────────┴──────────────┐
                     ▼                             ▼
┌───────────────────────────────────┐
┌───────────────────────────────────┐
│     2A. VECTOR INDEXING           │ │      2B. KNOWLEDGE GRAPH
│
│  PDF → Chunks → Embeddings → FAISS│ │  Abstract → LLM →
Entities/Relations  │
│  + BM25 keyword index             │ │  Build traversable graph
│
└───────────────────────────────────┘
└───────────────────────────────────┘
```

```
                         │                              │
                         └──────────────┬───────────────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
────┐
  │                          3. GRAPH VISUALIZATION
  │
  │                Paper similarity graph + Entity knowledge graph
  │
  └────────────────────────────────────────────────────────────────────────
────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
────┐
  │                             4. USER QUERY
  │
  │                     "What is self-attention?"
  │
  └────────────────────────────────────────────────────────────────────────
────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
────┐
  │                          5. HYBRID RETRIEVAL
  │
  │            BM25 (keyword) + FAISS (semantic) → Combined ranking
  │
  └────────────────────────────────────────────────────────────────────────
────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
────┐
  │                            6. RERANKING
  │
  │                LLM scores each chunk for relevance → Re-sort
  │
  └────────────────────────────────────────────────────────────────────────
────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
────┐
  │                         7. CORRECTIVE RAG
  │
  │              Check: Is context relevant? → Warn if not
  │
  └────────────────────────────────────────────────────────────────────────
────┘
                                        │
                                        ▼
  ┌────────────────────────────────────────────────────────────────────────
```

```
────┐
│                        8. LLM GENERATION
│
│             Context + Query → LLM → Final Answer (streaming)
│
└──────────────────────────────────────────────────────────────────
────┘
```

Component Deep Dive

1. Paper Fetching (FetcherAgent)

File: `backend/agents/fetcher_agent.py`

What it does: Searches arXiv API for academic papers matching a query.

Why we use it: arXiv is the largest open-access repository of scientific papers (2M+ papers). It provides structured metadata (title, abstract, authors) and PDF links.

Process:

```
1. User enters: "transformer attention"
2. FetcherAgent constructs query: all:"transformer attention"
3. HTTP GET to: http://export.arxiv.org/api/query?
search_query=all:"transformer attention"
4. Parse XML response → Extract paper metadata
5. Return list of papers with {id, title, summary, authors, published,
link}
```

Rate Limiting: arXiv allows 1 request per 3 seconds. We implement:

- 3-second delay between requests
- Exponential backoff on 429/503 errors (5s → 10s → 15s)
- Maximum 3 retries

---

2. PDF Processing (pdf_processor.py)

File: `backend/core/pdf_processor.py`

What it does: Downloads PDF, extracts text, splits into chunks.

Why we use chunks:

- LLMs have context limits (Llama 3.3: 128K tokens)
- Smaller chunks = more precise retrieval
- Embeddings work better on focused text

Process:

```
1. Convert arXiv link: /abs/2301.12345 → /pdf/2301.12345.pdf
2. Download PDF content via HTTP
3. Extract text using PyMuPDF (fitz) - page by page
4. Split using RecursiveCharacterTextSplitter:
   - chunk_size: 1000 characters
   - chunk_overlap: 100 characters (to preserve context at boundaries)
5. Return list of text chunks
```

Why RecursiveCharacterTextSplitter:

- Tries to split on natural boundaries (paragraphs, sentences, words)
- Falls back to character-level only when necessary
- Preserves semantic coherence within chunks

---

3. Vector Embeddings (faiss_store.py)

File: `backend/core/faiss_store.py`

What it does: Converts text to vectors, stores in FAISS index.

Why embeddings: Text → Numbers allows mathematical similarity comparison. Similar concepts have similar vectors even with different words.

Model: Nomic Embed Text v1.5 (via Gravix Layer API)

- Dimension: 768
- Trained on 235M text pairs
- Optimized for retrieval tasks

Process:

```
1. Send text chunks to Gravix API
2. Receive 768-dimensional vectors
3. Normalize vectors (for cosine similarity)
4. Add to FAISS IndexFlatIP (Inner Product index)
5. Store metadata (paper_id, title, chunk_text) alongside
```

Why FAISS:

- Facebook AI Similarity Search
- Blazing fast: millions of vectors in milliseconds
- Memory-efficient
- IndexFlatIP: Exact nearest neighbor search using inner product

---

4. BM25 Keyword Index (faiss_store.py)

File: `backend/core/faiss_store.py`

What it does: Traditional keyword-based search using BM25 algorithm.

Why we need it: Vector search can miss exact keyword matches. "BERT" might not match "bert" well in embeddings, but BM25 catches it.

BM25 (Best Matching 25):

```
score(D, Q) = Σ IDF(qi) · (f(qi, D) · (k1 + 1)) / (f(qi, D) + k1 · (1 - b
+ b · |D|/avgdl))
```

Where:

- `IDF(qi)`: Inverse Document Frequency - rare terms score higher
- `f(qi, D)`: Term frequency in document
- `k1`: Term saturation parameter (1.2-2.0)
- `b`: Length normalization (0.75)
- `|D|/avgdl`: Document length relative to average

Process:

```
1. Tokenize each chunk: "Self-attention mechanism" → ["self", "attention",
"mechanism"]
2. Build corpus of tokenized documents
3. Create BM25Okapi index
4. For query, compute BM25 scores against all documents
```

---

5. Hybrid Search

File: `backend/core/faiss_store.py` → `hybrid_search()`

What it does: Combines BM25 and vector search results.

Why hybrid: Best of both worlds:

- Vector: Semantic understanding ("car" matches "automobile")
- BM25: Exact keyword matching ("BERT" matches "BERT")

Fusion Formula:

```
combined_score = α · vector_score + (1 - α) · bm25_score_normalized
```

Where:

- `α = 0.5` (default): Equal weight to both methods
- `vector_score`: Cosine similarity from FAISS (0 to 1)
- `bm25_score_normalized`: BM25 score / max(BM25 scores) (0 to 1)

Process:

```
1. Run vector search → Get top 3*k results with scores
2. Run BM25 search → Get scores for all documents
3. Normalize BM25 scores to 0-1 range
4. Compute combined score for each document
5. Sort by combined score, return top k
```

---

6. Knowledge Graph (GraphRAG)

File: `backend/core/knowledge_graph.py`

What it does: Extracts entities and relationships from papers using LLM.

Why GraphRAG: Traditional RAG retrieves similar text. GraphRAG understands structure:

- "Transformer uses self-attention" → relationship
- Navigate from "transformer" to find all related methods

Entity Types:

Type	Examples
CONCEPT	attention mechanism, language model, neural network
METHOD	self-attention, cross-attention, multi-head attention
ALGORITHM	transformer, BERT, GPT
DATASET	ImageNet, GLUE, SQuAD
METRIC	accuracy, F1 score, perplexity

Relationship Types:

Type	Meaning	Example
USES	Method employs concept	"Transformer USES attention"
IMPROVES	Builds upon with changes	"BERT IMPROVES word2vec"
COMPARES_TO	Paper compares methods	"GPT COMPARES_TO BERT"
BASED_ON	Foundational dependency	"ChatGPT BASED_ON GPT-3"
APPLIES_TO	Domain application	"Transformer APPLIES_TO NLP"

Extraction Process:

```
1. Send paper title + abstract to LLM with structured prompt
2. LLM returns JSON:
   {
      "entities": [{"name": "self-attention", "type": "METHOD"}],
      "relationships": [{"source": "transformer", "relation": "USES",
"target": "attention"}]
   }
3. Parse JSON, create Entity and Relationship objects
4. Add to knowledge graph (entities dict, relationships list)
5. Build entity-to-papers index for retrieval
```

Graph Traversal (for retrieval):

```
1. User asks about "attention"
2. Find entity "attention" in graph
3. BFS traversal depth=2:
   - Depth 0: ["attention"]
   - Depth 1: ["self-attention", "multi-head", "transformer"] (direct
relations)
   - Depth 2: ["BERT", "GPT", "encoder"] (relations of relations)
4. Get all papers mentioning any of these entities
5. Use paper IDs to filter vector search
```

---

7. Reranking

File: `backend/core/reranker.py`

What it does: Uses LLM to re-score retrieved chunks for relevance.

Why reranking: Initial retrieval is fast but approximate. Reranking is slow but precise.

- Embeddings: Bi-encoder (independent text encoding)
- Reranking: Cross-encoder (joint query-document encoding)

Process:

```
1. Receive top 10 retrieved chunks
2. For each chunk, ask LLM:
   "Rate relevance of this text to the query (0-10)"
3. Parse numeric score from response
4. Re-sort by new scores
5. Return top 5
```

Why LLM as reranker:

- Understands nuance and context
- Can reason about relevance
- More accurate than pure embedding similarity
- Trade-off: 10 extra LLM calls per query

---

8. Corrective RAG

File: `backend/agents/chat_agent.py`

What it does: Validates that retrieved context is actually relevant before generating.

Why corrective: Traditional RAG blindly uses retrieved context. Corrective RAG:

1. Checks if context can answer the question
2. Warns user if context seems insufficient
3. Reduces hallucinations

Process:

```
1. Build context from retrieved chunks
2. Ask LLM: "Is this context relevant to answering: {query}? YES/NO"
3. If NO:
   - Add warning note to response
   - (Optional) Reformulate query and re-retrieve
4. Proceed with generation
```

---

9. LLM Generation

File: `backend/core/llm_client.py`

What it does: Generates final response using Groq's Llama 3.3 70B.

Model: Llama 3.3 70B Versatile

- 70 billion parameters
- 128K context window
- Optimized for instruction following
- Hosted on Groq (fast inference)

Prompt Structure:

```
System: You are a research assistant. Answer based ONLY on the provided
context.

User:
Context from Papers:
Paper 1: [Title]
Content: [chunk text]

Paper 2: [Title]
Content: [chunk text]

User's Question: [query]
```

```
Answer:
```

Streaming:

```
stream = client.chat.completions.create(
    messages=messages,
    model="llama-3.3-70b-versatile",
    stream=True  # Enable streaming
)

for chunk in stream:
    yield chunk.choices[0].delta.content  # Yield each token
```

The Mathematics

Cosine Similarity (Vector Search)

Measures angle between two vectors. Used for semantic similarity.

```
cos(θ) = (A · B) / (||A|| × ||B||)

Where:
A · B = Σ(ai × bi)      (dot product)
||A|| = √(Σ ai²)        (magnitude)
```

Range: -1 to 1 (after normalization: 0 to 1)

- 1: Identical direction (same meaning)
- 0: Orthogonal (unrelated)
- -1: Opposite direction

Example:

```
Query embedding: [0.5, 0.8, 0.2]
Doc embedding:   [0.4, 0.7, 0.3]

Dot product: 0.5×0.4 + 0.8×0.7 + 0.2×0.3 = 0.2 + 0.56 + 0.06 = 0.82

Magnitudes:
||Query|| = √(0.25 + 0.64 + 0.04) = √0.93 = 0.964
||Doc||   = √(0.16 + 0.49 + 0.09) = √0.74 = 0.860

Cosine similarity = 0.82 / (0.964 × 0.860) = 0.82 / 0.829 = 0.989
```

High similarity (0.989) → Very relevant!

BM25 Scoring

Probabilistic retrieval model based on term frequency.

IDF (Inverse Document Frequency):

```
IDF(t) = log((N - n(t) + 0.5) / (n(t) + 0.5))

Where:
N = total documents
n(t) = documents containing term t
```

Rare terms get higher IDF → More discriminative

TF Component:

```
TF(t,d) = (f(t,d) × (k1 + 1)) / (f(t,d) + k1 × (1 - b + b × |d|/avgdl))

Where:
f(t,d) = frequency of term t in document d
k1 = 1.5 (saturation parameter)
b = 0.75 (length normalization)
|d| = document length
avgdl = average document length
```

Final Score:

```
BM25(q, d) = Σ IDF(t) × TF(t, d) for each term t in query q
```

---

k-NN Graph Construction

For paper similarity graph, we use k-Nearest Neighbors.

Algorithm:

```
for each paper i:
    for each paper j ≠ i:
        similarity[i][j] = cosine(avg_embedding_i, avg_embedding_j)

    # Keep only top K=3 most similar
    neighbors[i] = top_k(similarity[i], k=3)
```

```
# Build edges (deduplicated)
for each (i, neighbors):
    for each neighbor j with weight w:
        if w > 0.3:  # Minimum threshold
            add_edge(i, j, weight=w)
```

Why k-NN instead of full graph:

- Full graph: n² edges (15 papers → 105 edges) → cluttered
- k-NN: ~k×n edges (15 papers → ~45 edges) → meaningful connections

---

Graph Traversal (BFS)

Breadth-First Search to find related entities.

```
function query_related(start_entities, depth):
    visited = set(start_entities)
    frontier = set(start_entities)

    for d = 1 to depth:
        new_frontier = {}
        for node in frontier:
            for edge in edges where edge.source == node or edge.target ==
node:
                neighbor = other_endpoint(edge)
                if neighbor not in visited:
                    new_frontier.add(neighbor)
                    visited.add(neighbor)
        frontier = new_frontier

    return visited
```

Complexity: O(V + E) where V=entities, E=relationships

Complete Dry Run

Let's trace through a complete example from user input to final output.

Input

- **Search Topic**: "transformer attention mechanism"
- **Papers to fetch**: 5
- **User Query**: "How does self-attention work?"

Step 1: Build Graph API Call

Request:

```
POST /api/build_graph
{
    "topic": "transformer attention mechanism",
    "max_results": 5
}
```

Step 2: Fetch Papers from arXiv

FetcherAgent.fetch_papers():

```
→ HTTP GET: http://export.arxiv.org/api/query?
search_query=all:"transformer attention mechanism"&max_results=5
← XML Response (parsed):
```

Papers Retrieved:

```
[
  {
    "id": "http://arxiv.org/abs/1706.03762",
    "title": "Attention Is All You Need",
    "summary": "The dominant sequence transduction models are based on
complex recurrent or convolutional neural networks...",
    "authors": ["Ashish Vaswani", "Noam Shazeer", ...],
    "published": "2017-06-12",
    "link": "http://arxiv.org/abs/1706.03762"
  },
  {
    "id": "http://arxiv.org/abs/1810.04805",
    "title": "BERT: Pre-training of Deep Bidirectional Transformers",
    "summary": "We introduce a new language representation model called
BERT...",
    "authors": ["Jacob Devlin", ...],
    "published": "2018-10-11",
    "link": "http://arxiv.org/abs/1810.04805"
  },
  // ... 3 more papers
]
```

Step 3: Process PDFs and Create Embeddings

For each paper:

```
Paper: "Attention Is All You Need"
→ Download PDF: http://arxiv.org/pdf/1706.03762.pdf
→ Extract text (15 pages → ~30,000 characters)
→ Split into chunks:
   - Chunk 1: "We propose a new simple network architecture, the
Transformer, based solely on attention mechanisms..."
   - Chunk 2: "An attention function can be described as mapping a query
and a set of key-value pairs to an output..."
   - Chunk 3: "Multi-head attention allows the model to jointly attend to
information from different representation..."
   ... (30 chunks total)

→ Create embeddings:
   - Send chunks to Gravix API
   - Receive 768-dim vectors for each chunk
   - Normalize vectors

→ Add to FAISS index:
   - index.add([chunk_1_vector, chunk_2_vector, ...])
   - Store metadata: [(paper_id, title, chunk_text), ...]

→ Add to BM25:
   - Tokenize: ["transformer", "attention", "mechanism", ...]
   - Add to BM25 corpus
```

After all 5 papers:

- FAISS index: 150 vectors (30 chunks × 5 papers)
- BM25 corpus: 150 tokenized documents
- Documents list: 150 metadata entries

Step 4: Extract Entities (GraphRAG)

For each paper, LLM extraction:

```
Paper: "Attention Is All You Need"

Prompt to LLM:
"Extract entities and relationships from:
Title: Attention Is All You Need
Abstract: The dominant sequence transduction models are based on..."

LLM Response:
{
  "entities": [
    {"name": "Transformer", "type": "ALGORITHM"},
    {"name": "self-attention", "type": "METHOD"},
    {"name": "multi-head attention", "type": "METHOD"},
    {"name": "encoder-decoder", "type": "CONCEPT"},
    {"name": "sequence transduction", "type": "CONCEPT"},
    {"name": "BLEU score", "type": "METRIC"},
    {"name": "WMT 2014", "type": "DATASET"}
  ],
  "relationships": [
    {"source": "Transformer", "relation": "USES", "target": "self-
attention"},
    {"source": "Transformer", "relation": "USES", "target": "multi-head
attention"},
    {"source": "self-attention", "relation": "IMPROVES", "target":
"recurrent attention"},
    {"source": "Transformer", "relation": "APPLIES_TO", "target":
"sequence transduction"}
  ]
}
```

After all 5 papers:

```
Knowledge Graph:
- Entities: 35
- Relationships: 48
- Entity types: {CONCEPT: 12, METHOD: 10, ALGORITHM: 8, DATASET: 3,
METRIC: 2}
```

Step 5: Build Paper Similarity Graph

Calculate average embedding per paper:

```
paper_1_chunks = vectors[0:30]   # 30 chunks from paper 1
paper_1_avg = mean(paper_1_chunks)  # 768-dim average vector
# Normalize
paper_1_avg = paper_1_avg / norm(paper_1_avg)
```

Compute pairwise similarities:

```
Similarity Matrix:
        Paper1  Paper2  Paper3  Paper4  Paper5
Paper1   1.0     0.85    0.72    0.68    0.55
Paper2  0.85     1.0     0.78    0.71    0.62
Paper3  0.72    0.78     1.0     0.81    0.70
Paper4  0.68    0.71    0.81     1.0     0.75
Paper5  0.55    0.62    0.70    0.75     1.0
```

k-NN with K=3:

```
Paper1 top-3: [Paper2: 0.85, Paper3: 0.72, Paper4: 0.68]
Paper2 top-3: [Paper1: 0.85, Paper3: 0.78, Paper4: 0.71]
```

```
Paper3 top-3: [Paper4: 0.81, Paper2: 0.78, Paper5: 0.70]
Paper4 top-3: [Paper3: 0.81, Paper5: 0.75, Paper2: 0.71]
Paper5 top-3: [Paper4: 0.75, Paper3: 0.70, Paper2: 0.62]
```

Deduplicated edges (weight > 0.3):

```
Edges: [
  {source: Paper1, target: Paper2, weight: 0.85},
  {source: Paper1, target: Paper3, weight: 0.72},
  {source: Paper1, target: Paper4, weight: 0.68},
  {source: Paper2, target: Paper3, weight: 0.78},
  {source: Paper2, target: Paper4, weight: 0.71},
  {source: Paper3, target: Paper4, weight: 0.81},
  {source: Paper3, target: Paper5, weight: 0.70},
  {source: Paper4, target: Paper5, weight: 0.75}
]
Total: 8 edges
```

Step 6: API Response

```
{
  "nodes": [/* 5 papers */],
  "edges": [/* 8 edges */],
  "knowledge_graph": {
    "nodes": [/* 35 entities */],
    "edges": [/* 48 relationships */]
  },
  "session_id": "a1b2c3d4"
}
```

Step 7: User Selects Papers and Asks Question

Selected Papers: Paper 1 ("Attention Is All You Need"), Paper 2 ("BERT")

Query: "How does self-attention work?"

Request:

```
POST /api/agent_action
{
  "action": "chat",
  "selected_papers": [Paper1, Paper2],
  "query": "How does self-attention work?",
  "session_id": "a1b2c3d4",
  "use_hybrid": true,
  "use_reranking": true
}
```

Step 8: Hybrid Search

Vector Search:

```
query_embedding = embed("search_query: How does self-attention work?")
# → 768-dim vector

scores, indices = faiss_index.search(query_embedding, k=30)
# Returns indices of 30 most similar chunks

# Filter to selected papers (Paper 1 and Paper 2)
vector_results = [
  {chunk: "Self-attention...takes three inputs: query, key, value...",
score: 0.89},
  {chunk: "The attention weights are computed by taking the softmax...",
score: 0.86},
  {chunk: "Multi-head attention performs attention in parallel...", score:
0.82},
  ... (10 results total)
]
```

BM25 Search:

```
query_tokens = ["self", "attention", "work"]
bm25_scores = bm25.get_scores(query_tokens)
# Returns score for each of 150 documents

# Top BM25 results (filtered to Paper 1 & 2):
bm25_results = [
  {chunk: "Self-attention, sometimes called intra-attention...", bm25:
12.5},
  {chunk: "...self-attention layer connects all positions...", bm25:
10.2},
  ...
]
```

Fusion (α = 0.5):

```
Combined scores:
Chunk A: 0.5 × 0.89 (vector) + 0.5 × 1.0 (normalized BM25) = 0.945
Chunk B: 0.5 × 0.82 (vector) + 0.5 × 0.95 (normalized BM25) = 0.885
Chunk C: 0.5 × 0.86 (vector) + 0.5 × 0.80 (normalized BM25) = 0.830
...
```

```
Top 10 by combined score: [A, B, C, D, E, F, G, H, I, J]
```

Step 9: Reranking

For each of top 10 chunks, ask LLM:

```
Prompt: "Rate relevance 0-10.
Query: How does self-attention work?
Text: Self-attention, sometimes called intra-attention, relates different
positions of a sequence..."

LLM Response: "9"
```

Reranked scores:

```
Chunk A: rerank_score = 9
Chunk B: rerank_score = 8
Chunk D: rerank_score = 9  (moved up!)
Chunk C: rerank_score = 7
Chunk E: rerank_score = 6
...

Final top 5: [A (9), D (9), B (8), C (7), F (7)]
```

Step 10: Knowledge Graph Context

Find related entities:

```
query = "How does self-attention work?"
# Extract key terms: ["self", "attention"]

# Match entities:
matched = ["self-attention", "attention"]

# BFS traversal (depth=2):
related = [
  "self-attention",
  "attention",
  "multi-head attention",  # depth 1
  "transformer",           # depth 1
  "query-key-value",       # depth 1
  "encoder",               # depth 2
  "BERT"                   # depth 2
]
```

Add to context: "Related concepts from knowledge graph: self-attention, multi-head attention, transformer, query-key-value"

Step 11: Corrective RAG Check

```
context = "Self-attention relates different positions... Multi-head
attention performs..."

Prompt: "Is this context relevant to: How does self-attention work?
(YES/NO)"
LLM Response: "YES"

# Context is relevant, proceed without warning
```

Step 12: Final Generation

Prompt to LLM:

```
System: You are a research assistant. Answer based ONLY on the provided
context.

User:
Context from Papers:

Paper 1: Attention Is All You Need
Content: Self-attention, sometimes called intra-attention, is an attention
mechanism
relating different positions of a single sequence in order to compute a
representation
of the sequence. The attention function can be described as mapping a
query and a set
of key-value pairs to an output, where the query, keys, values, and output
are all vectors...

Paper 2: Attention Is All You Need
Content: We compute the dot products of the query with all keys, divide
each by √dk,
and apply a softmax function to obtain the weights on the values...

(3 more chunks...)

Related concepts from knowledge graph: self-attention, multi-head
attention, transformer, query-key-value

User's Question: How does self-attention work?

Answer:
```

LLM Response (streamed):

```
Self-attention works by computing relationships between different
positions in a sequence.
The mechanism involves three components: queries (Q), keys (K), and values
(V).

Here's how it works step by step:

1. **Query-Key Matching**: For each position, a query vector is compared
against all key
   vectors using dot products. This computes how much each position should
"attend to"
   every other position.

2. **Scaling**: The dot products are divided by √dk (square root of key
dimension) to
   prevent large values that would push softmax into regions with tiny
gradients.

3. **Softmax Normalization**: A softmax function converts the scores into
attention
   weights that sum to 1, creating a probability distribution.

4. **Weighted Sum**: The attention weights are used to create a weighted
sum of the
   value vectors, producing the output for each position.

The mathematical formula is:
Attention(Q, K, V) = softmax(QKᵀ/√dk)V

This allows the model to capture dependencies between any positions in the
sequence,
regardless of their distance, making it particularly effective for long-
range dependencies.

[Based on "Attention Is All You Need" paper]
```

Final Response

```
{
  "response": "Self-attention works by computing relationships between
different positions in a sequence..."
}
```

Summary: Why Each Component Matters

Component	Purpose	Without It
FAISS	Fast similarity search	Slow O(n) linear scan
BM25	Keyword matching	Miss exact term matches
Hybrid	Combine both	Miss either semantic or keyword relevance
Reranking	Precision	Return approximate results
GraphRAG	Structure understanding	Miss relational context
Corrective RAG	Reduce hallucinations	Generate from irrelevant context
Streaming	UX	Wait for full response

This architecture represents state-of-the-art RAG techniques from 2024-2025 research, making your project portfolio-worthy and technically impressive.