Contents

WEEK 12: RETRIEVAL AUGMENTED GENERATION (RAG)

Introduction to Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) represents a paradigm shift in how large language models (LLMs) access and utilize information. Traditional LLMs rely solely on knowledge encoded in their parameters during training, which creates several fundamental limitations: knowledge cutoffs prevent them from accessing information after their training date, they cannot access private or proprietary data, and they are prone to hallucinating facts that sound plausible but are incorrect. RAG addresses these limitations by augmenting the generation process with relevant information retrieved from external knowledge sources in real-time.

The RAG architecture consists of two main components working in concert: a retriever and a generator. The retriever searches through a knowledge base to find relevant documents or passages based on the input query, while the generator (typically an LLM) uses both the original query and the retrieved context to produce a response. This separation of concerns allows the system to maintain up-to-date information in the knowledge base without retraining the entire model, access arbitrarily large knowledge bases that wouldn't fit in model parameters, and provide attributable, verifiable responses by citing the retrieved sources.

The basic RAG workflow proceeds as follows: First, the user query is embedded into a vector representation using an embedding model. Second, this query embedding is used to search a vector database containing pre-computed embeddings of documents in the knowledge base. Third, the most relevant documents are retrieved based on similarity metrics like cosine similarity. Fourth, these retrieved documents are concatenated with the original query to form an augmented prompt. Finally, this augmented prompt is fed to the LLM, which generates a response grounded in the retrieved information.

Vector Databases and Semantic Search

At the heart of RAG systems lies the concept of semantic search enabled by vector databases. Unlike traditional keyword-based search that matches exact terms, semantic search understands the meaning and context of queries, returning results that are conceptually similar even if they don't share exact keywords. This is achieved through dense vector embeddings that represent text in a high-dimensional semantic space where similar meanings are mapped to nearby points.

Vector databases are specialized data stores optimized for storing, indexing, and querying high-dimensional vectors efficiently. Popular vector databases include FAISS (Facebook AI Similarity Search), Pinecone, Weaviate, Qdrant, Milvus, and Chroma. Each offers different trade-offs in terms of scalability, query speed, accuracy, and ease of deployment. FAISS, for instance, is a highly optimized library for similarity search that can handle billions of vectors, while cloud-based solutions like Pinecone offer managed services with built-in scaling and monitoring.

The efficiency of vector search is crucial for RAG applications, especially at scale. Exact nearest neighbor search using brute-force comparison becomes computationally prohibitive for large databases, requiring $O(n \cdot d)$ operations where n is the number of vectors and d is the dimensionality. To address this, vector databases employ approximate nearest neighbor (ANN) algorithms that trade a small amount of accuracy for dramatic improvements in speed. Common ANN approaches include:

Hierarchical Navigable Small World (HNSW) graphs build a multi-layer graph structure where each layer contains connections between vectors. Search starts at the top layer with long-range connections and progressively moves to lower layers with finer granularity, enabling logarithmic search complexity. HNSW offers excellent recall and query speed, making it one of the most popular indexing methods.

Product Quantization (PQ) compresses vectors by decomposing them into subvectors and quantizing each subvector separately using learned codebooks. This reduces memory requirements and enables fast approximate distance computations, though at the cost of some accuracy. Variants like Optimized Product Quantization (OPQ) improve accuracy by learning a rotation of the vector space before quantization.

Locality-Sensitive Hashing (LSH) uses hash functions designed so that similar vectors are likely to hash to the same buckets. Search then only needs to examine vectors in the same or nearby buckets. While LSH works well for certain metrics like cosine similarity, it typically requires more memory than other methods to achieve comparable accuracy.

The similarity metric used for retrieval is another important consideration. Cosine similarity, which measures the cosine of the angle between vectors, is most common for text embeddings as it is invariant to vector magnitude. Euclidean distance (L2 norm) is sensitive to magnitude and may be more appropriate for certain applications. Dot product similarity combines aspects of both magnitude and direction and is particularly efficient to compute. The choice of metric should align with how the embedding model was trained.

Document Processing and Chunking

Before documents can be retrieved, they must be processed and indexed in the vector database. Document processing involves several steps that significantly impact RAG system performance. The quality of this preprocessing directly affects the relevance and coherence of retrieved information.

Chunking is the process of dividing documents into smaller passages that will be embedded and retrieved independently. This is necessary because embedding models have maximum sequence lengths (typically 512 tokens for older models, up to 8192 for newer ones), but more fundamentally because retrieval is more effective when operating on focused, coherent passages rather than entire documents. Smaller chunks increase retrieval precision by reducing the inclusion of irrelevant information, but may lose context and require retrieving more chunks to provide sufficient information.

Several chunking strategies exist, each with trade-offs:

Fixed-size chunking divides text into chunks of a specified number of tokens or characters, often with some overlap between consecutive chunks. The overlap ensures that information near chunk boundaries isn't lost. For example, chunks of 512 tokens with 128-token overlap. This approach is simple and predictable but may split semantic units awkwardly.

Sentence-based chunking uses natural language processing to identify sentence boundaries and groups sentences together up to a maximum chunk size. This preserves semantic units better than fixed-size chunking but may produce variably sized chunks. Some implementations also use overlapping sentences or include surrounding context for each chunk.

Paragraph or section-based chunking respects the document's structure, using paragraphs, sections, or other structural elements as chunk boundaries. This works well for well-structured documents like technical documentation or academic papers but requires parsing document structure.

Semantic chunking uses embeddings to identify topic boundaries, creating chunks that maintain semantic coherence. This can be done by computing embeddings for sentences and identifying points where similarity drops, indicating topic transitions. While potentially most effective, this approach is more computationally expensive.

Metadata enrichment involves augmenting chunks with additional context that aids retrieval or generation. This might include document title, section headers, creation date, author, or source. Some implementations create a hierarchy where chunk embeddings reference parent document or section embeddings, enabling both fine-grained and coarse-grained retrieval.

Hybrid representations combine dense embeddings with sparse representations like BM25 (a traditional information retrieval algorithm based on term frequency). Hybrid search can improve robustness by leveraging both semantic similarity and keyword matching, catching cases where one approach might fail.

RAG Pipeline Architecture

A production RAG system involves multiple components working together in a carefully orchestrated pipeline. Understanding this architecture is crucial for building effective RAG applications.

Embedding Generation: The first stage involves generating embeddings for both the knowledge base documents and user queries. The choice of embedding model significantly impacts retrieval quality. Modern embedding models like OpenAI's text-embedding-3-large, Cohere's embed-v3, or open-source alternatives like all-MiniLM-L6-v2 or E5-large have been trained specifically for retrieval tasks using contrastive learning on large datasets of query-document pairs.

A critical consideration is ensuring the embedding model is the same (or at least compatible) for both indexing and retrieval. Some advanced approaches use asymmetric models where queries and documents are embedded differently, optimizing for the typical query-document relationship rather than general similarity.

Retrieval Strategy: Beyond basic top-k retrieval, several advanced strategies can improve RAG performance:

Reranking involves first retrieving a larger set of candidates (e.g., top-100) using fast vector search, then applying a more sophisticated but computationally expensive reranker to select the final top-k documents. Rerankers often use cross-encoder models that process query-document pairs jointly, capturing fine-grained interactions that bi-encoder embeddings miss.

Multi-query retrieval generates multiple variations or reformulations of the user's query, retrieves documents for each, and combines the results. This can improve robustness to query formulation and increase the diversity of retrieved information.

Hypothetical document embeddings (HyDE) generate a hypothetical answer to the query using the LLM, embed this answer, and use it for retrieval. The intuition is that an answer is more similar to relevant documents than the query itself.

Recursive retrieval performs multiple rounds of retrieval, where initial results inform subsequent queries. This can be useful for complex multi-step questions.

Prompt Engineering: How retrieved documents are incorporated into the prompt significantly affects generation quality. Best practices include:

- Clearly delineating retrieved context from the user query
- Instructing the model to cite sources from the context
- Providing explicit instructions to say "I don't know" if the context doesn't contain the answer
- Ordering retrieved documents by relevance or using the model to summarize/filter them before generation

Response Generation: The final stage uses an LLM to generate a response based on the augmented prompt. Key considerations include:

- **Context window management**: Ensuring retrieved documents fit within the model's context limit while leaving room for the query and response
- **Citation and attribution**: Instructing the model to cite which parts of the retrieved context support different claims in its response
- **Hallucination mitigation**: Explicitly prompting the model to stick to the provided context and acknowledge uncertainty

Advanced RAG Techniques

As RAG has matured, several advanced techniques have emerged to address specific challenges and improve performance.

Contextual Compression addresses the problem of limited context windows and irrelevant information in retrieved documents. Instead of using entire retrieved chunks, a compression model (which could be the same LLM or a specialized model) extracts only the portions relevant to the query. This allows fitting more documents in the context or providing more focused information to the generator.

Self-RAG introduces self-reflection into the RAG process. The model generates reflection tokens to assess whether it needs to retrieve information, whether retrieved information is relevant, and whether its response is supported by the context. This allows the system to adaptively decide when to retrieve and to self-correct when appropriate.

Active RAG frames retrieval as a multi-step problem-solving process. Rather than a single retrieval step, the system can iteratively retrieve information, assess what additional information is needed, formulate new queries, and retrieve again. This is particularly powerful for complex questions requiring synthesis of information from multiple sources.

Graph RAG combines vector retrieval with knowledge graph traversal. Documents are not only embedded as vectors but also linked in a graph structure representing relationships. Retrieval can follow graph edges to find related information, and the graph structure itself can provide additional context for generation.

Temporal RAG addresses the challenge of time-sensitive information by incorporating temporal metadata into the retrieval process. This might involve weighting more recent documents higher, filtering based on date ranges, or explicitly tracking when information was current.

Multi-modal RAG extends RAG beyond text to incorporate images, audio, video, or other modalities. This requires multi-modal embedding models that can represent different types of content in a shared semantic space, and generators capable of reasoning over multi-modal inputs.

RAG Evaluation

Evaluating RAG systems comprehensively requires assessing multiple dimensions of performance: retrieval quality, generation quality, and end-to-end task performance.

Retrieval Metrics measure how well the retriever finds relevant documents:

- **Recall@k**: The proportion of relevant documents that appear in the top-k retrieved results
- **Precision@k**: The proportion of retrieved documents (top-k) that are relevant
- **Mean Reciprocal Rank (MRR)**: The average of reciprocal ranks of the first relevant document
- **Normalized Discounted Cumulative Gain (NDCG)**: Accounts for both relevance and ranking position with graded relevance judgments

These metrics require ground truth labels indicating which documents are relevant for each query, which can be expensive to obtain. Some approaches use LLM-based evaluation where a strong model judges relevance.

Generation Quality Metrics assess the generated responses:

- **Faithfulness/Groundedness**: Whether the response is supported by the retrieved context. Can be measured by asking an LLM to verify each claim against the context.
- **Answer Relevance**: Whether the response addresses the user's query. Often measured using semantic similarity between query and response.
- **Hallucination Rate**: The frequency of factually incorrect or unsupported statements. Challenging to measure automatically; often requires human evaluation or fact-checking against authoritative sources.

End-to-End Metrics measure overall system performance:

- **Answer Correctness**: For questions with definitive answers, accuracy against ground truth
- **Human Preference**: Side-by-side comparisons by human evaluators

- **Task Success Rate**: For task-oriented applications, whether the system successfully completes the task

Latency and Cost Metrics are crucial for production systems:

- **End-to-end latency**: Time from query to response
- **Retrieval latency**: Time to query the vector database
- **Generation latency**: Time for the LLM to generate the response
- **Cost per query**: Including embedding costs, database queries, and LLM API calls

Effective evaluation often requires a combination of these metrics, balanced against system requirements and constraints.

Challenges and Best Practices

Building production RAG systems involves navigating several challenges:

Context Length Limitations: Even with models supporting long contexts (e.g., 128k tokens), there are practical limits. Retrieving too many documents can introduce noise and increase latency/cost. Best practices include retrieving more candidates initially, then filtering or summarizing to fit within context limits, and using hierarchical retrieval where summaries are retrieved first, then detailed sections.

Retrieval Quality: Poor retrieval undermines the entire system. Improving retrieval involves using high-quality embedding models, implementing hybrid search combining dense and sparse methods, employing reranking for final selection, and continuously evaluating and tuning based on failure cases.

Data Quality and Freshness: RAG is only as good as its knowledge base. Ensuring data quality requires cleaning and preprocessing, removing duplicates, maintaining consistent formatting, and implementing pipelines for regular updates.

Handling Ambiguity: User queries may be ambiguous or under-specified. Techniques include asking clarifying questions, retrieving diverse results covering different interpretations, or using query expansion to explore multiple interpretations.

Privacy and Security: When RAG systems access proprietary or sensitive data, careful implementation is required for access control, ensuring users can only retrieve information they're authorized to access, logging and auditing retrieval requests, and sanitizing retrieved content before including in prompts.

Cost Optimization: Embedding generation, vector database queries, and LLM calls all incur costs. Optimization strategies include caching embeddings and frequent queries, batching operations where possible, using smaller/cheaper models where appropriate (e.g., for reranking), and implementing query routing to use RAG only when necessary.

Practical Implementation Considerations

When implementing RAG systems, several practical considerations ensure robustness and scalability:

Incremental Indexing: Knowledge bases typically grow over time. Implementing incremental indexing allows adding new documents without re-indexing everything. Vector databases typically support this, though some indexing methods (like HNSW) may see degraded performance with many incremental additions, requiring periodic full re-indexing.

Monitoring and Debugging: Production RAG systems require observability including logging queries, retrieved documents, and generated responses, monitoring retrieval recall and precision, tracking latency and error rates, and implementing A/B testing for system improvements.

Failure Handling: Graceful degradation when retrieval fails (database unavailable, no relevant documents found) might involve falling back to the LLM without retrieval, returning cached responses, or clearly communicating to users that information couldn't be retrieved.

User Feedback Loops: Incorporating user feedback improves RAG systems over time through explicit feedback (thumbs up/down, ratings), implicit feedback (click-through rates, dwell time), and using feedback to refine retrieval or update the knowledge base.

Testing Strategies: Comprehensive testing includes unit tests for individual components, integration tests for the full pipeline, regression tests for known good/bad queries, and load testing for performance under scale.

Learning Objectives

- Understand RAG architecture and its advantages over pure LLMs
- Implement vector databases for semantic search
- Master document chunking and preprocessing strategies
- Build and evaluate end-to-end RAG pipelines
- Apply advanced RAG techniques for complex applications