**RAG Types Implementation**

**Chunker.py**

**Embedings.py**

**Vector\_store.py**

**Vector\_search.py**

**Rag\_types.py**

**Rag\_pipeline.py**

**Main.py**

**Chunker.py**

**1. AgenticTextChunker:**

Implements an intelligent text chunking strategy that combines token-based segmentation with semantic awareness.

**Methods:**

* \_\_init\_\_(max\_tokens=500, token\_overlap=50, model\_name='cl100k\_base')
* Initializes the chunker with specified parameters
* `max\_tokens`: Controls the maximum size of each chunk in tokens
* `token\_overlap`: Determines how many tokens should overlap between chunks
* `model\_name`: Specifies which tiktoken model to use for tokenization
* \_semantic\_split(text: str) -> List[str]
  + Private method for splitting text based on semantic boundaries
  + Uses two strategies:
    - Splits on double newlines (paragraph boundaries)
    - Falls back to sentence splitting if paragraph splitting yields too few chunks
  + Returns a list of semantic chunks
* chunk\_documents(documents: List[Document]) -> List[Document]
* Main chunking method that processes a list of documents
* Steps:
  + Splits each document into semantic chunks
  + Combines chunks while respecting token limits
  + Maintains overlap between chunks
  + Preserves document metadata
  + Adds chunk size information to metadata

**2. RecursiveChunker**

Provides a wrapper around LangChain's RecursiveCharacterTextSplitter for character-based splitting.

**Methods:**

* \_\_init\_\_(chunk\_size=500, chunk\_overlap=50)
* Initializes the RecursiveCharacterTextSplitter with specified parameters
* `chunk\_size`: Maximum size of each chunk in characters
* `chunk\_overlap`: Number of characters to overlap between chunks
* chunk\_documents(documents: List[Document]) -> List[Document]
* Delegates chunking to LangChain's splitter
* Preserves document metadata
* Returns list of chunked documents

**3. SemanticChunker**

Focuses on maintaining semantic boundaries while splitting text into manageable chunks.

**Methods:**

* \_\_init\_\_(max\_chunk\_size=500, overlap=50)
* Initializes chunker with size and overlap parameters
* `max\_chunk\_size`: Maximum size of each chunk
* `overlap`: Amount of overlap between chunks
* chunk\_documents(documents: List[Document]) -> List[Document]
  + Implements semantic chunking logic:
    - Splits documents into paragraphs using double newlines
    - Combines paragraphs while respecting size limits
    - Maintains document metadata
    - Adds chunk size to metadata
  + Returns chunked documents preserving semantic coherence

**4. SentenceChunker**

Specializes in sentence-based segmentation using NLTK with robust fallback mechanisms.

**Methods:**

* \_\_init\_\_(max\_sentences=5, overlap\_sentences=1)
* Initializes chunker with sentence-based parameters
* `max\_sentences`: Maximum number of sentences per chunk
* `overlap\_sentences`: Number of sentences to overlap
* \_ensure\_nltk\_resources()
* Private method to check and download required NLTK resources
* Downloads 'punkt' tokenizer if not available
* \_fallback\_sentence\_split(text: str) -> List[str]
* Backup sentence splitting method when NLTK fails
* Uses rule-based approach with common sentence endings
* Returns list of sentences
* chunk\_documents(documents: List[Document]) -> List[Document]
* Main chunking method that:
  + Attempts NLTK sentence tokenization
  + Falls back to rule-based splitting if needed
  + Creates chunks of specified sentence count
  + Maintains sentence overlap
  + Preserves metadata and adds chunk information

**5. TokenChunker**

Provides precise token-based splitting using tiktoken for accurate token counting.

**Methods:**

* \_\_init\_\_(max\_tokens=500, overlap\_tokens=50, model\_name='cl100k\_base')
* Initializes token-based chunker
* `max\_tokens`: Maximum tokens per chunk
* `overlap\_tokens`: Number of tokens to overlap
* `model\_name`: Tiktoken model to use
* chunk\_documents(documents: List[Document]) -> List[Document]
  + Implements token-based chunking:
    - Tokenizes document using tiktoken
    - Creates chunks based on token counts
    - Maintains specified token overlap
    - Preserves metadata and adds token count
  + Returns documents split by token count

**6. FixedSizeChunker**

Implements simple character-based chunking with fixed-size chunks.

**Methods:**

* \_\_init\_\_(chunk\_size=1000, overlap=100)
* Initializes fixed-size chunker
* `chunk\_size`: Size of chunks in characters
* `overlap`: Character overlap between chunks
* chunk\_documents(documents: List[Document]) -> List[Document]
  + Implements fixed-size chunking:
    - Splits text into fixed-size pieces
    - Maintains character overlap
    - Preserves metadata
    - Adds chunk size information
  + Returns documents split into fixed-size chunks

**7. ChunkerFactory**

Factory class that provides a single interface for creating different types of chunkers.

**Methods:**

* create\_chunker(method: str, max\_tokens: int, token\_overlap: int) -> object
  + Static method that creates chunker instances
  + Parameters:
    - `method`: Chunking strategy to use ("agentic", "recursive", "semantic", "sentence", "token", "fixed")
    - `max\_tokens`: Maximum tokens/size for chunks
    - `token\_overlap`: Overlap between chunks
  + Returns appropriate chunker instance
  + Raises ValueError for unknown methods

**Embedings.py**

**1. BaseEmbeddingGenerator**

Base class that provides core embedding functionality using SentenceTransformers.

**Methods:**

* \_\_init\_\_(model\_name: str, trust\_remote\_code: bool = False)
  + Initializes the embedding model using SentenceTransformer
  + Parameters:
    - `model\_name`: Name of the embedding model to use
    - `trust\_remote\_code`: Safety flag for loading remote code
* \_preprocess\_text(text: str) -> str
* Basic text preprocessing method
* Strips whitespace from input text
* Can be overridden by subclasses for model-specific preprocessing
* generate\_embeddings(chunks: List[Document]) -> List[List[float]]
* Generates embeddings for a list of document chunks
* Preprocesses each chunk's text content
* Returns list of embeddings as float arrays

**2. E5EmbeddingGenerator**

Specialized implementation for the E5 embedding model.

**Methods:**

* \_\_init\_\_(trust\_remote\_code: bool = False)
* Initializes with E5 specific model ("intfloat/e5-large-v2")
* Inherits from BaseEmbeddingGenerator
* \_preprocess\_text(text: str) -> str
* E5-specific text preprocessing
* Adds "passage:" prefix to input text
* Required for optimal E5 model performance

**3. BGEEmbeddingGenerator**

Specialized implementation for the BGE embedding model.

**Methods:**

* \_\_init\_\_(trust\_remote\_code: bool = False)
* Initializes with BGE specific model ("BAAI/bge-large-en-v1.5")
* Inherits from BaseEmbeddingGenerator
* \_preprocess\_text(text: str) -> str
* BGE-specific text preprocessing
* Adds "Represent this document for retrieval:" prefix
* Required for optimal BGE model performance

**4. GTEEmbeddingGenerator**

Implementation for the GTE embedding model.

**Methods:**

* \_\_init\_\_(trust\_remote\_code: bool = False)
* Initializes with GTE specific model ("Alibaba-NLP/gte-large-en-v1.5")
* Uses default preprocessing from base class

**5. MinilmEmbeddingGenerator**

Implementation for the MiniLM embedding model.

**Methods:**

* \_\_init\_\_(trust\_remote\_code: bool = False)
* Initializes with MiniLM specific model ("sentence-transformers/all-MiniLM-L6-v2")
* Uses default preprocessing from base class

**6. EmbeddingGeneratorFactory**

Factory class for creating appropriate embedding generator instances.

**Methods:**

* get\_supported\_models() -> List[str]
* Class method that returns list of supported model names
* Used for validation and user information
* create\_generator(model\_name: str, trust\_remote\_code: bool = False) -> BaseEmbeddingGenerator
* Creates appropriate embedding generator based on model name
* Validates model name against supported models
* Returns initialized generator instance
* Raises ValueError for unsupported models

**7. EnhancedEmbeddingGenerator**

Advanced embedding generator with batching and caching capabilities.

**Methods:**

* \_\_init\_\_(model\_name: str, batch\_size: int = 32, trust\_remote\_code: bool = False)
* Initializes enhanced generator with specified model
* Sets up batching and caching systems
* Parameters:
  + `model\_name`: Name of embedding model
  + `batch\_size`: Number of documents to process at once
  + `trust\_remote\_code`: Safety flag for loading remote code
* generate\_embeddings(chunks: List[Document]) -> List[List[float]]
* Generates embeddings with optimized batching and caching
* Features:
  + Checks cache for existing embeddings
  + Processes uncached documents in batches
  + Updates cache with new embeddings
  + Returns combined list of embeddings
* compute\_similarity(embedding1: List[float], embedding2: List[float]) -> float
* Computes cosine similarity between two embeddings
* Uses numpy for efficient computation
* Returns similarity score between -1 and 1
* clear\_cache()
* Clears the embedding cache
* Useful for memory management

**Vector\_search.py**

**1. FaissSearch**

Implements efficient similarity search using Facebook AI Similarity Search (FAISS) library, optimized for large-scale nearest neighbor search.

**Methods:**

* \_\_init\_\_(metric: str = "cosine")
  + Initializes FAISS search with specified distance metric
  + Parameters:
    - `metric`: Either "cosine" or "l2" distance metric
  + Stores metric choice for index building and searching
* build\_index(embeddings: List[List[float]])
* Builds a FAISS index for fast similarity search
* Steps:

1. Converts embeddings to numpy array in float32 format

2. Determines embedding dimension

3. For cosine similarity:

- Normalizes vectors

- Creates IndexFlatIP (Inner Product) index

4. For L2 distance:

- Creates IndexFlatL2 index

5. Adds embeddings to index

* + Returns: Built FAISS index
* search(index, query\_embedding: List[float], top\_k: int = 5)
  + Searches the FAISS index for similar vectors
  + Parameters:
* `index`: Built FAISS index
* `query\_embedding`: Vector to search for
* `top\_k`: Number of results to return
  + Process:
* Reshapes query embedding
* Normalizes if using cosine similarity
* Performs search
* Returns: Tuple of (indices, distances) for top matches

**2. HNSWSearch**

Implements Hierarchical Navigable Small World (HNSW) algorithm for approximate nearest neighbor search, optimized for speed-accuracy trade-off.

**Methods:**

* \_\_init\_\_(M: int = 32, ef\_construction: int = 200, ef\_search: int = 128)
  + Initializes HNSW search with configuration parameters
  + Parameters:
    - `M`: Number of connections per layer in the graph
    - `ef\_construction`: Size of dynamic candidate list during construction
    - `ef\_search`: Size of dynamic candidate list during search
  + These parameters control the trade-off between speed, memory, and accuracy
* build\_index(embeddings: List[List[float]])
  + Builds HNSW index for efficient approximate search
  + Process:
    - Converts embeddings to numpy array
    - Creates HNSW index with specified parameters
    - Configures construction and search parameters
    - Adds embeddings to index
  + Returns: Built HNSW index
* search(index, query\_embedding: List[float], top\_k: int = 5)
* Searches HNSW index for similar vectors
* Parameters similar to FAISS search
* Returns: Tuple of (indices, distances) for nearest neighbors

**3. BruteForceSearch**

Implements exact nearest neighbor search by comparing query vector with all vectors in the dataset. Useful as baseline and for small datasets.

**Methods:**

* \_\_init\_\_(metric: str = "cosine")
  + Initializes brute force search with specified metric
  + Parameters:
    - `metric`: "cosine" or "l2" distance metric
* build\_index(embeddings: List[List[float]])
* Stores embeddings for brute force comparison
* Simply converts input to numpy array
* Returns: Stored embeddings array
* search(embeddings: np.ndarray, query\_embedding: List[float], top\_k: int = 5)
  + Performs exhaustive search through all embeddings
  + Process for cosine similarity:
    - Calculates cosine similarity with all vectors
    - Sorts to find top matches
  + Process for L2 distance:
    - Calculates L2 norm between query and all vectors
    - Finds k nearest neighbors
  + Returns: Tuple of (indices, distances) for best matches

**4. AnnoySearch**

Implements Approximate Nearest Neighbors Oh Yeah (Annoy) algorithm, optimized for memory efficiency and easy persistence to disk.

**Methods:**

* \_\_init\_\_(metric: str = "cosine", n\_trees: int = 100)
  + Initializes Annoy search configuration
  + Parameters:
    - `metric`: "cosine" (converted to "angular") or "euclidean"
    - `n\_trees`: Number of trees for indexing (affects build time/accuracy trade-off)
* build\_index(embeddings: List[List[float]])
  + Builds Annoy index for approximate search
  + Process:
    - Converts embeddings to numpy array
    - Creates index with specified metric
    - Adds vectors one by one
    - Builds trees for searching
  + Returns: Built Annoy index
* search(index: AnnoyIndex, query\_embedding: List[float], top\_k: int = 5)
* Searches Annoy index for similar vectors
* Uses Annoy's efficient tree-based search
* Returns: Tuple of (indices, distances) for nearest neighbors

**5. SearchFactory**

Factory class that provides a unified interface for creating different search algorithm instances.

**Methods:**

* create\_search\_algorithm(method: str, \*\*kwargs)
  + Static method for creating search algorithm instances
  + Parameters:
    - `method`: One of "faiss", "hnsw", "brute\_force", or "annoy"
    - `\*\*kwargs`: Additional parameters for specific algorithms
  + Returns: Appropriate search algorithm instance
  + Raises: ValueError for unknown methods

**Vector\_store.py**

**1. WeaviateStore**

Implements vector storage and retrieval using Weaviate, a vector search engine designed for scalable similarity search.

**Methods:**

* \_\_init\_\_(embedding\_model: str, weaviate\_url: str = "http://localhost:8080")
  + Initializes Weaviate vector store connection
  + Parameters:
    - `embedding\_model`: Model name for embedding generation
    - `weaviate\_url`: URL for Weaviate server connection
  + Process :
    - Validates and formats URL
    - Sets up Weaviate client
    - Creates unique class name with UUID
    - Initializes schema with properties:
      1. content: text content
      2. source: document source
      3. page\_number: page number in document
      4. chunk\_id: unique identifier for chunk
* store\_embeddings(chunks: List[Document], embeddings: List[List[float]])
  + Stores document chunks and their embeddings in Weaviate
  + Process:
    - Iterates through chunks and embeddings
    - Extracts metadata from each chunk
    - Creates data objects with content and metadata
    - Stores in Weaviate with vector embeddings
* search(query: str, top\_k: int = 5) -> List[str]
  + Performs semantic search in Weaviate
  + Steps:
    - Encodes query using sentence transformer
    - Performs vector similarity search
    - Returns top k matching documents

**2. ChromaStore**

Implements vector storage using ChromaDB, an open-source embedding database.

**Methods:**

* \_\_init\_\_(embedding\_model: str)
* Initializes ChromaDB store
* Process:
  + - Creates ChromaDB client
    - Generates unique collection name
    - Creates collection with cosine similarity space
* store\_embeddings(chunks: List[Document], embeddings: List[List[float]])
  + Stores embeddings in ChromaDB
  + Features:
    - Implements batch processing (100 chunks per batch)
    - Process for each batch:
      * Generates unique IDs
      * Extracts documents and metadata
      * Adds to collection with embeddings
* search(query: str, top\_k: int = 5) -> List[str]
* Performs similarity search in ChromaDB
* Steps:
  + - Encodes query using sentence transformer
    - Queries collection with embeddings
    - Returns matched documents

**3. FaissStore**

Implements vector storage using FAISS (Facebook AI Similarity Search) for efficient similarity search.

**Methods:**

* \_\_init\_\_(embedding\_model: str)
  + Initializes FAISS store
  + Setup:
    - Gets embedding dimension from model
    - Creates FAISS index (FlatL2)
    - Initializes document storage
* store\_embeddings(chunks: List[Document], embeddings: List[List[float]])`
  + Stores embeddings in FAISS
  + Process:
    - Converts embeddings to numpy array
    - Adds to FAISS index
    - Stores document content separately
* search(query: str, top\_k: int = 5) -> List[str]
  + Performs similarity search in FAISS
  + Steps:
  + Encodes query
  + Performs FAISS search
  + Returns matched documents

**4. PineconeStore**

Implements vector storage using Pinecone, a managed vector database service.

**Methods:**

* \_\_init\_\_(embedding\_model: str = "intfloat/e5-large-v2")
  + Initializes Pinecone store
  + Setup:
    - Connects to Pinecone service
    - Uses fixed index name
    - Handles index creation/reuse logic
    - Manages index limits (max 5 indexes)
* store\_embeddings(chunks: List[Document], embeddings: List[List[float]])
  + Stores embeddings in Pinecone
  + Features:
    - Batch processing (100 chunks per batch)
    - Process:
      * Creates vector objects with:
        + Unique IDs
        + Embedding values
        + Metadata (content and chunk metadata)
      * Upserts vectors to index
* search(query: str, top\_k: int = 5) -> List[str]
* Performs similarity search in Pinecone
* Process:
  + - Encodes query
    - Performs vector search
    - Handles different response formats
    - Returns matched content
* list\_indexes()
* Utility method to list available Pinecone indexes
* Returns list of index names

**5. VectorStoreFactory**

Factory class for creating vector store instances.

**Methods:**

* create\_store(store\_type: str, embedding\_model: str, \*\*kwargs) -> object
  + Static method to create vector store instances
  + Supports:
    - "weaviate"
    - "chromadb"
    - "faiss"
    - "pinecone"
  + Parameters :
    - `store\_type`: Type of vector store to create
    - `embedding\_model`: Embedding model to use
    - `\*\*kwargs`: Additional store-specific parameters

**Rag\_types.py**

**1. BaseRAG (Abstract Base Class)**

Serves as the abstract base class for all RAG implementations, defining the core interface and common components.

**Components:**

* `document\_loader`: Handles loading documents from zip files
* `chunker`: Splits documents into manageable chunks
* `embedding\_generator`: Creates vector embeddings from text
* `vector\_store`: Stores and retrieves embeddings
* `search\_algorithm`: Implements search functionality
* `generator`: Handles text generation

**Abstract Methods:**

* `process\_documents(zip\_path: str) -> List[Any]`: Must be implemented by subclasses
* `generate\_response(query: str, max\_length: int) -> str`: Must be implemented by subclasses

**2. StandardRAG**

Implements the basic RAG workflow with straightforward document processing and response generation.

**Methods:**

* process\_documents(zip\_path: str) -> List[Any]
* Implements standard document processing workflow:
  + - Loads documents from zip file
    - Chunks documents into smaller pieces
    - Generates embeddings for chunks
    - Stores embeddings in vector store
    - Builds search index
    - Returns processed documents
* generate\_response(query: str, max\_length: int = 512) -> str
* Generates responses using simple retrieval and generation:
  + - Retrieves relevant documents from vector store
    - Creates context by joining relevant documents
    - Generates response using provided context
    - Returns generated response or error message if no relevant documents found

**3. GraphRAG**

Extends RAG with graph-based knowledge representation for enhanced context understanding.

Additional Components

* `similarity\_threshold`: Controls edge creation in knowledge graph
* `knowledge\_graph`: NetworkX graph for storing document relationships
* `chunk\_to\_node\_map`: Maps document chunks to graph nodes
* `node\_counter`: Tracks node creation

**Methods**

* process\_documents(zip\_path: str) -> List[Any]
* Implements graph-enhanced document processing:
* Follows standard RAG processing
* Additionally builds knowledge graph in background
* Maintains mapping between chunks and graph nodes
* generate\_response(query: str, max\_length: int = 512) -> str
* Matches standard RAG response generation
* Knowledge graph available for potential enhanced retrieval
* \_build\_knowledge\_graph(chunks: List[Any], embeddings: List[List[float]])
  + Private method for knowledge graph construction:
    - Creates nodes for each document chunk
    - Assigns unique IDs to nodes
    - Stores content and embeddings in nodes
    - Maintains chunk-to-node mapping
* \_create\_node\_id() -> str
* Generates unique node identifiers
* \_get\_chunk\_id(chunk: Any) -> str
* Creates hash-based chunk identifiers

**4. AdaptiveRAG**

Implements adaptive retrieval strategies based on confidence thresholds.

**Additional Components**

* `confidence\_threshold`: Controls adaptation of retrieval strategy
* `embeddings`: Stores document embeddings

**Methods**

* process\_documents(zip\_path: str) -> List[Any]
* Implements standard document processing workflow
* Maintains embeddings for adaptive retrieval
* generate\_response(query: str, max\_length: int = 512) -> str
* Generates responses using confidence-based retrieval:
  + - Retrieves relevant documents
    - Creates context from retrieved documents
    - Generates and returns response

**5. RaptorRAG**

Implements RAPTOR (Retrieval-Augmented Prompt Optimization and Reranking) for enhanced response quality.

**Additional Components**

* `reranking\_model`: Model for reranking retrieved passages
* `prompt\_optimizer`: Optimizes prompts for better responses
* `token\_weight\_threshold`: Controls token importance in reranking
* `max\_prompt\_attempts`: Limits prompt optimization attempts

**Methods**

* process\_documents(zip\_path: str) -> List[Any]
* Implements standard document processing workflow
* Maintains embeddings for retrieval
* generate\_response(query: str, max\_length: int = 512) -> str
* Implements RAPTOR response generation:
  + - Retrieves initial set of relevant documents
    - Applies reranking to improve document relevance
    - Attempts prompt optimization
    - Falls back to default prompt if optimization fails
    - Generates and returns response
* Includes error handling for reranking and prompt optimization

1. **CorrectiveRAG**

* Implements CorrectiveRAG for fact-checked response generation.

**Additional Components**

* **fact\_checker**: Verifies the accuracy of generated responses.
* **correction\_threshold**: Confidence level required for response acceptance.
* **max\_correction\_attempts**: Limits the number of correction iterations.
* **embeddings**: Stores document representations for retrieval.

**Methods**

* process\_documents(zip\_path: str) -> List[Any]
  + Implements corrective RAG document processing.
  + Loads and chunks documents, generates embeddings, and stores them in a vector store.
  + Builds an index for efficient retrieval.
* generate\_response(query: str, max\_length: int = 512) -> str
  + Implements response generation with fact-checking:
  + Retrieves relevant documents from the vector store.
  + Generates an initial response based on retrieved context.
  + Verifies the response using fact\_checker.
  + If confidence is below correction\_threshold, refines the response.
  + Reattempts correction up to max\_correction\_attempts.
  + Returns a fact-checked response.
  + Handles cases where no relevant documents are found.

1. **REFEEDRAG**

* Implements RefeedRAG for multi-step response refinement.

Additional Components

* k\_passages: Number of top-ranked passages used in the final response.
* num\_candidates: Number of initial candidate responses generated.
* embeddings: Stores document representations for retrieval.

Methods

* process\_documents(zip\_path: str) -> List[Any]
  + Implements RefeedRAG document processing.
  + Loads, chunks, and generates embeddings for documents.
  + Stores embeddings in a vector store and builds an index for retrieval.
* **generate\_response(query: str, max\_length: int = 512) -> str**
* Implements multi-step response refinement:
* Generates multiple candidate responses based on retrieved documents.
* Extracts additional passages by using candidate answers for retrieval.
* Ranks passages based on semantic similarity.
* Uses top-ranked passages to generate a final comprehensive response.
* Ensures contextual coherence by removing duplicate passages while preserving order.
* Handles cases where relevant documents are unavailable.

1. **Self Reflective RAG**

* Implements SelfReflectiveRAG for iterative query refinement and response generation.

Additional Components

* + relevance\_threshold: Minimum similarity score for document relevance.
  + max\_iterations: Maximum attempts to refine the query.
  + embeddings: Stores document representations for retrieval.

**Methods**

* process\_documents(zip\_path: str) -> List[Any]
  + Implements SelfReflectiveRAG document processing.
  + Loads, chunks, and generates embeddings for documents.
  + Stores embeddings in a vector store and builds an index for retrieval.
* grade\_relevance(query: str, documents: List[str]) -> List[Tuple[str, float]]
  + Computes similarity scores between the query and retrieved documents.
  + Returns a list of documents with corresponding relevance scores.
  + check\_relevance(graded\_docs: List[Tuple[str, float]]) -> bool
  + Checks if at least one retrieved document meets the relevance\_threshold.
* rewrite\_query(original\_query: str, graded\_docs: List[Tuple[str, float]]) -> str
* Generates an improved query if retrieved documents lack relevance.
* Uses retrieved content to refine specificity and clarity.
* generate\_response(query: str, relevant\_docs: List[Tuple[str, float]], max\_length: int = 512) -> str
* Constructs a response based on the most relevant documents.
* Ensures factual alignment with source content.
* process\_query(query: str, max\_length: int = 512) -> str
  + Implements the self-reflective query processing pipeline:
  + Retrieves and grades relevant documents.
  + If relevance is insufficient, rewrites the query and reattempts retrieval.
  + Iterates up to max\_iterations before generating a final response.
  + Ensures the most contextually accurate and relevant response.

1. **FusionRAG**

* Implements FusionRAG for enhanced retrieval using query expansion and Reciprocal Rank Fusion.

Additional Components

* + num\_similar\_queries: Number of alternative queries generated for expanded retrieval.
  + fusion\_k: Controls the impact of rank positions in Reciprocal Rank Fusion (RRF).
  + embeddings: Stores document representations for retrieval.
  + query\_gen\_prompt: Template for generating similar queries to improve search coverage.

Methods

* process\_documents(zip\_path: str) -> List[Any]
  + Implements FusionRAG document processing.
  + Loads, chunks, and generates embeddings for documents.
  + Stores embeddings in a vector store and builds an index for retrieval.
* generate\_similar\_queries(query: str) -> List[str]
  + Generates multiple semantically similar queries using a language model.
  + Enhances retrieval diversity by expanding the original query.
* reciprocal\_rank\_fusion(results\_lists: List[List[Dict]]) -> List[Dict]
  + Merges results from multiple queries using Reciprocal Rank Fusion (RRF).
  + Assigns fusion scores to documents based on their ranking across different queries.
  + Reorders results to prioritize consistently relevant documents.
* generate\_response(query: str, max\_length: int = 512) -> str
* Implements FusionRAG response generation:
* Generates alternative queries to improve retrieval coverage.
* Retrieves documents for each query and merges results using RRF.
* Selects top-ranked passages to form a comprehensive context.
* Generates a final response based on the most relevant retrieved information.
* Ensures robust retrieval by leveraging multiple query perspectives.

**Rag\_pipeline.py**

**RAGPipeline**

The RAGPipeline class implements a comprehensive Retrieval-Augmented Generation (RAG) system that allows users to select and configure various components for document processing and response generation.

**Methods**

* \_\_init\_\_(self)`
* Initializes the RAG pipeline with user interaction and component setup.
* User Interface Components:
  + RAG Type Selection
    - Options: standard, graph, adaptive, raptor
  + Language Model Selection
    - Options: google/flan-t5-large, google/flan-t5-base, t5-small
  + Embedding Model Selection
    - Options: sentence-transformers/all-MiniLM-L6-v2, Alibaba-NLP/gte-large-en-v1.5, intfloat/e5-large-v2, BAAI/bge-large-en-v1.5
  + Chunking Method Selection
    - Options: recursive, semantic, sentence, token, fixed, agentic
  + Vector Database Selection
    - Options: weaviate, chromadb, faiss, pinecone
  + Search Algorithm Selection
    - Options: hnsw, faiss, brute\_force, annoy
* Component Initialization:
  + DocumentLoader
  + Chunker (via ChunkerFactory)
  + EnhancedEmbeddingGenerator
  + VectorStore (via VectorStoreFactory)
  + SearchAlgorithm (via SearchFactory)
  + Language Model (using AutoTokenizer and AutoModelForSeq2SeqLM)
  + RAG Implementation (based on selected type)
* Error Handling:
  + Comprehensive exception handling with traceback
  + Cleanup mechanism setup
* process\_documents(self, zip\_path: str)
* Processes input documents through the selected RAG implementation.
* Features:
  + Takes a zip file path containing documents
  + Delegates processing to RAG implementation
  + Sets cleanup flag
  + Includes error handling with traceback
  + Returns processed documents or empty list on error
* Parameters:
  + `zip\_path`: Path to zip file containing documents
* Returns:
  + List of processed documents or empty list on error
* generate\_response(self, query: str, max\_length: int = 512) -> str
* Generates responses to queries using the configured RAG system.
* Features:
  + Uses selected RAG implementation for response generation
  + Includes error handling
  + Configurable maximum response length
* Parameters:
  + `query`: Input query string
  + `max\_length`: Maximum length of generated response (default: 512)
* Returns:
  + Generated response string or empty string on error
* cleanup(self)
* Handles resource cleanup and management.
* Features:
  + Checks cleanup\_required flag
  + Calls document loader cleanup
  + Includes error handling with logging
* \_\_del\_\_(self)
* Destructor method ensuring proper cleanup.
* Features:
  + Automatically calls cleanup method
  + Ensures resources are properly released

**Main.py**

**Function:main()**

This serves as the entry point for running the RAG (Retrieval-Augmented Generation) pipeline, providing a command-line interface for document processing and query handling.

The main function that orchestrates the entire RAG pipeline execution flow.

**Process Flow:**

* Pipeline Initialization
* Document Processing
* Information Display
* Interactive Query Handling
* Cleanup Management
* Pipeline Initialization
* Creates a new instance of the RAG pipeline
* Handles any initialization errors through try-except block
* Document Processing
* Gets the zip file path from user input
* Processes the documents using the RAG pipeline
* Stores processed documents for later use
* Information Display
* Extracts metadata from processed documents
* Creates a formatted list for display
* Uses tabulate for organized presentation
* Shows:
* File names
* File types
* File sizes
* Interactive Query Loop
* Implements continuous query-response cycle
* Accepts user queries
* Generates responses using the RAG pipeline
* Provides exit option through 'quit' command
* Error Handling
* Handles keyboard interruptions gracefully
* Catches and displays any exceptions
* Prints full traceback for debugging
* Ensures proper error reporting
* Cleanup Management
* Ensures proper resource cleanup
* Runs even if errors occur
* Checks if pipeline was initialized before cleanup