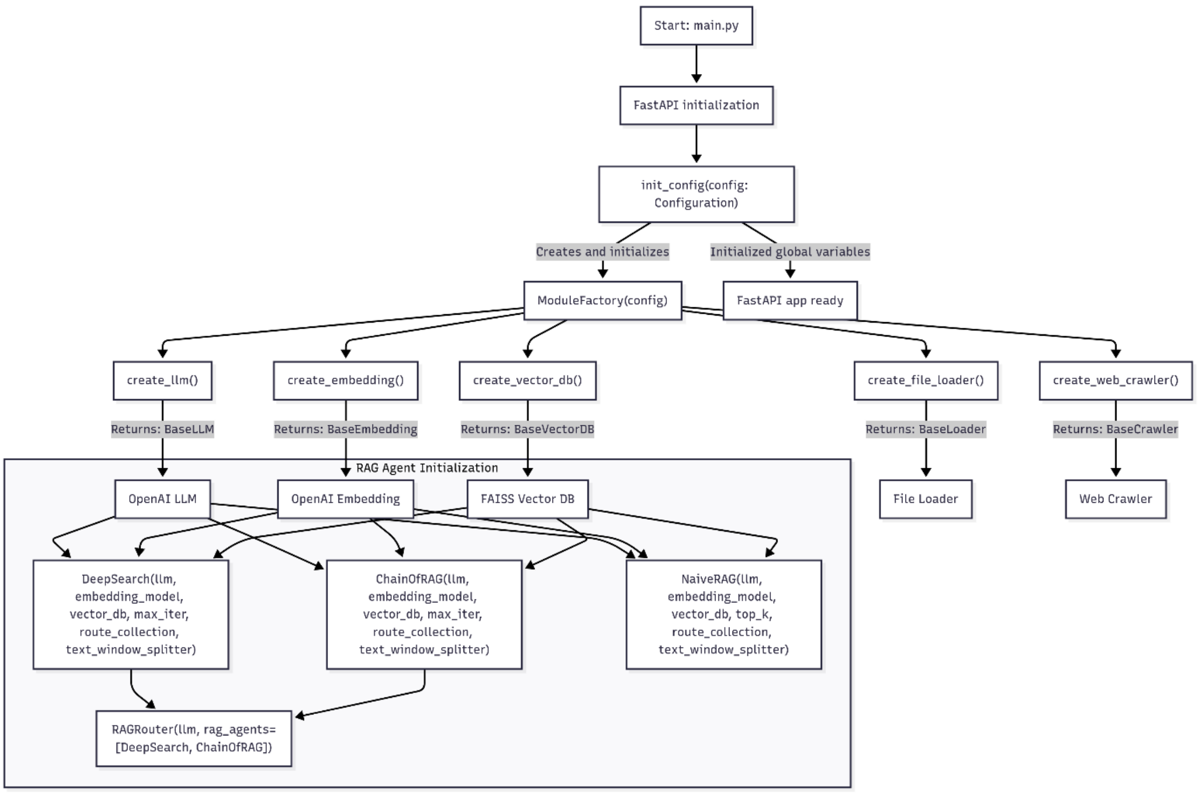
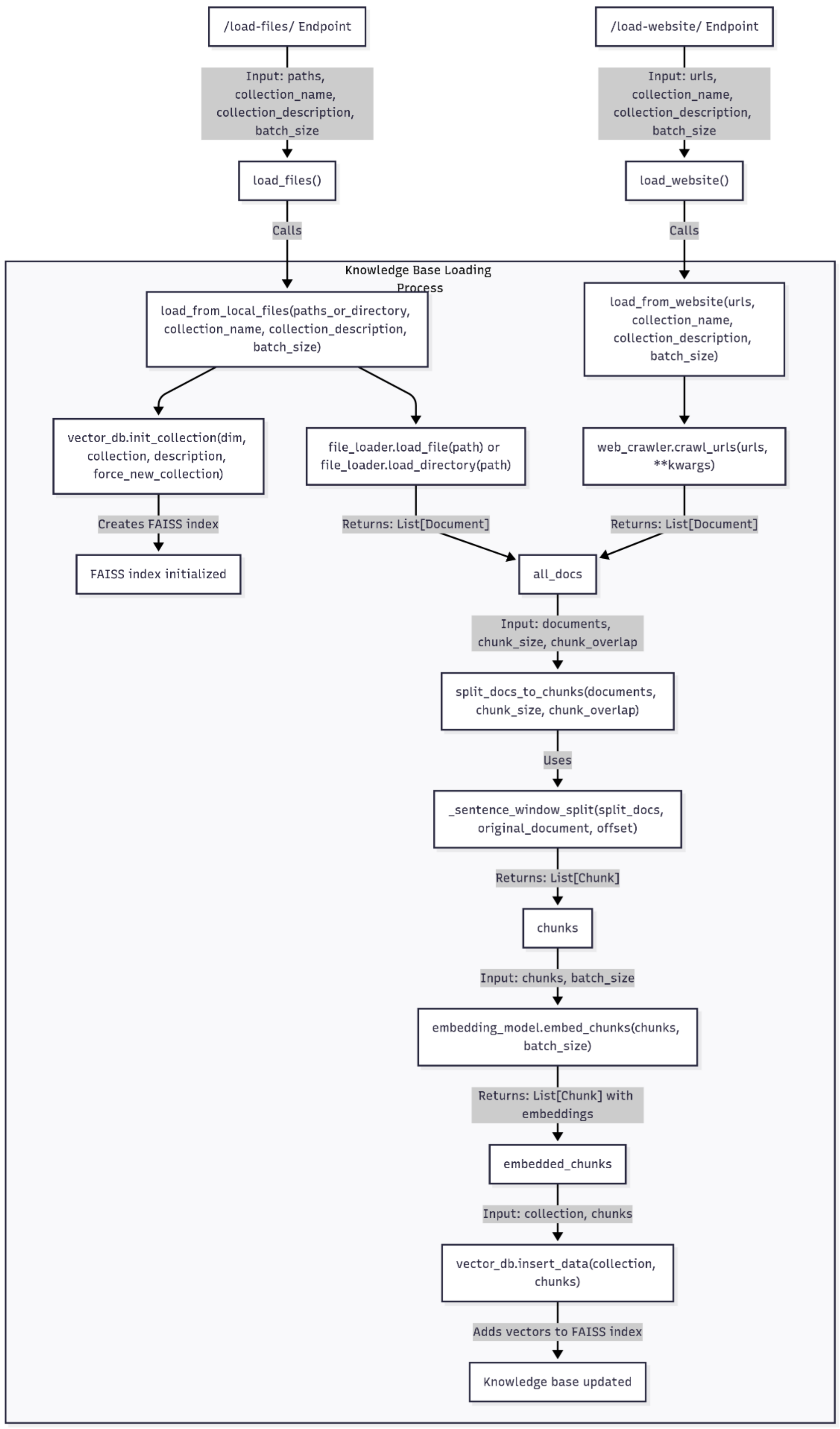
DeepSearcher Flowchart: From `main.py` to RAG Query Resolution

1. Application Entry and Initialization Flow

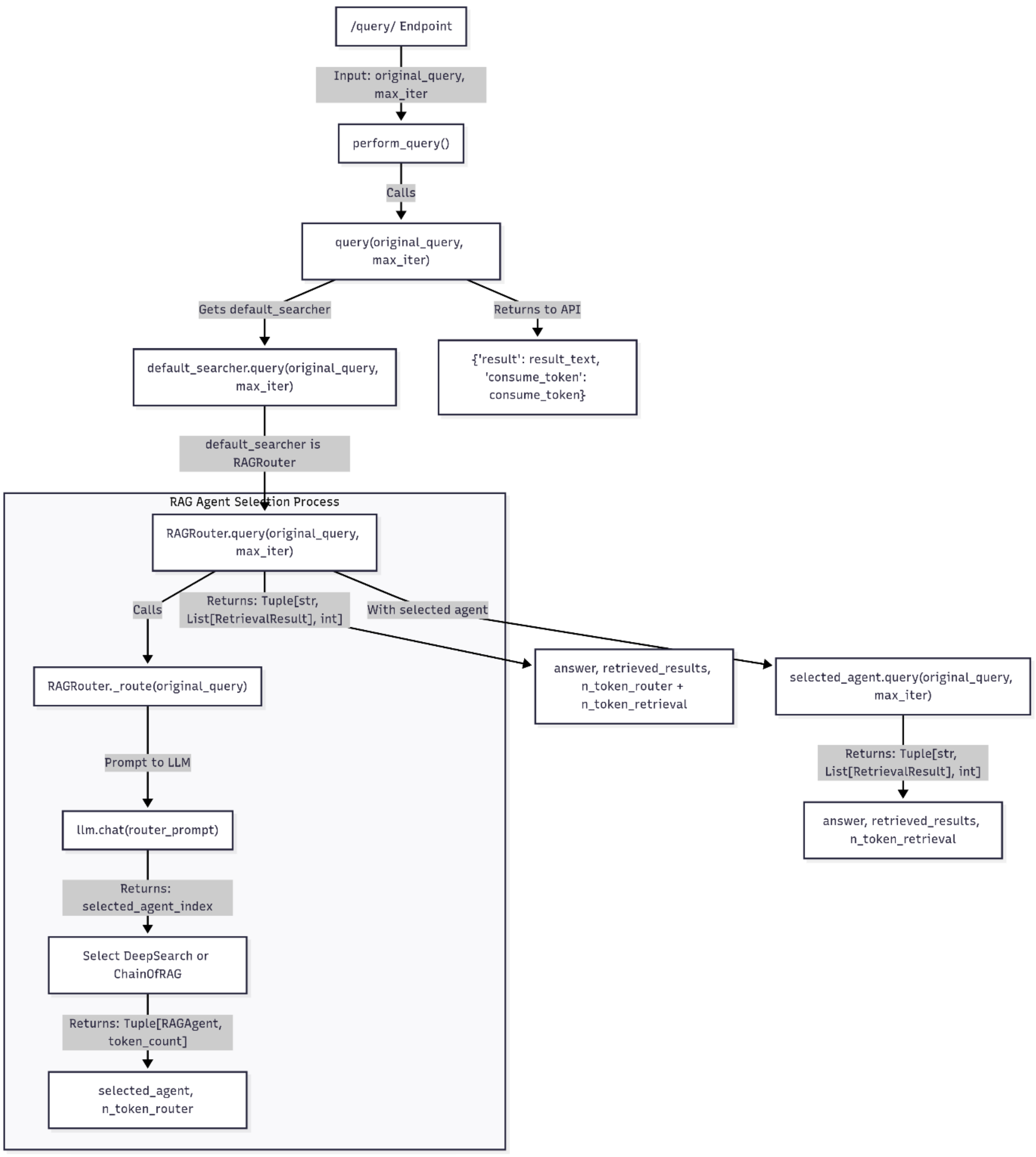


Start: main.py
FastAPI initialization
init_config(config: Configuration)
Creates and initializes
Initialized global variables
ModuleFactory(config)
FastAPI app ready
create_llm()
create_embedding()
create_vector_db()
create_file_loader()
create_web_crawler()
Returns: BaseLLM
Returns: BaseEmbedding
Returns: BaseVectorDB
Returns: BaseLoader
Returns: BaseCrawler
RAG Agent Initialization
OpenAI LLM
OpenAI Embedding
FAISS Vector DB
File Loader
Web Crawler
DeepSearch(llm, embedding_model, vector_db, max_iter, route_collection, text_window_splitter)
ChainOfRAG(llm, embedding_model, vector_db, max_iter, route_collection, text_window_splitter)
NaiveRAG(llm, embedding_model, vector_db, top_k, route_collection, text_window_splitter)
RAGRouter(llm, rag_agents=[DeepSearch, ChainOfRAG])

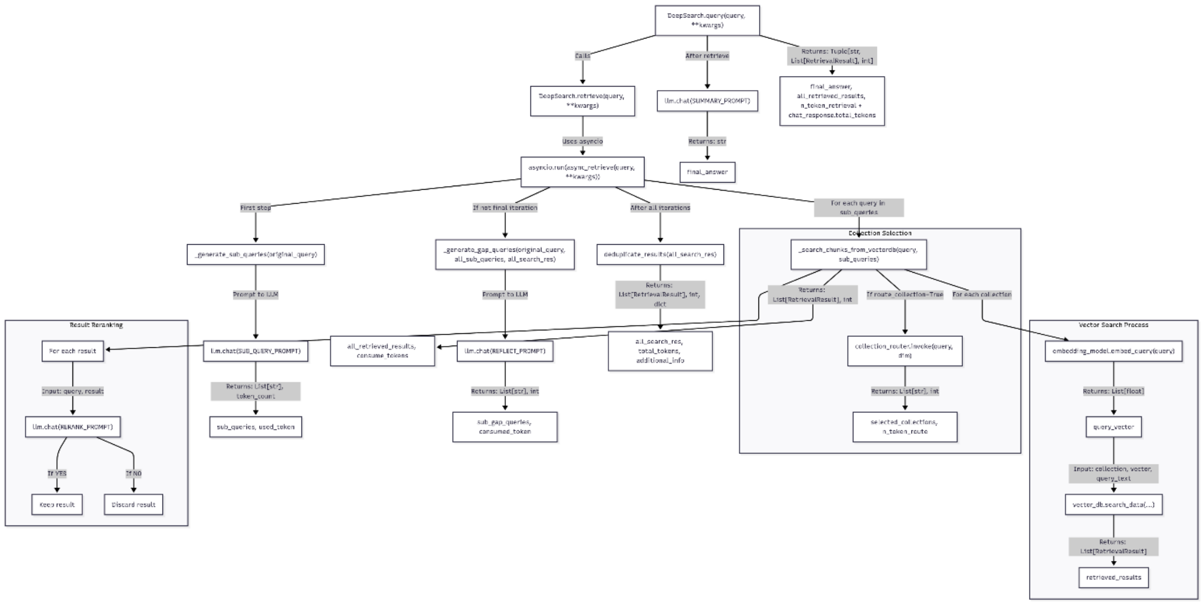
2. Document Loading Flow


/load-files/ Endpoint
Input: paths, collection_name, collection_description, batch_size
load_files()
Calls
/load-website/ Endpoint
Input: urls, collection_name, collection_description, batch_size
load_website()
Calls
Knowledge Base Loading Process
load_from_local_files(paths_or_directory, collection_name, collection_description, batch_size)
load_from_website(urls, collection_name, collection_description, batch_size)
vector_db.init_collection(dim, collection, description, force_new_collection)
Creates FAISS index
FAISS index initialized
file_loader.load_file(path) or file_loader.load_directory(path)
Returns: List[Document]
web_crawler.crawl_urls(urls, **kwargs)
Returns: List[Document]
all_docs
Input: documents, chunk_size, chunk_overlap
split_docs_to_chunks(documents, chunk_size, chunk_overlap)
Uses
_sentence_window_split(split_docs, original_document, offset)
Returns: List[Chunk]
chunks
Input: chunks, batch_size
embedding_model.embed_chunks(chunks, batch_size)
Returns: List[Chunk] with embeddings
embedded_chunks
Input: collection, chunks
vector_db.insert_data(collection, chunks)
Adds vectors to FAISS index
Knowledge base updated

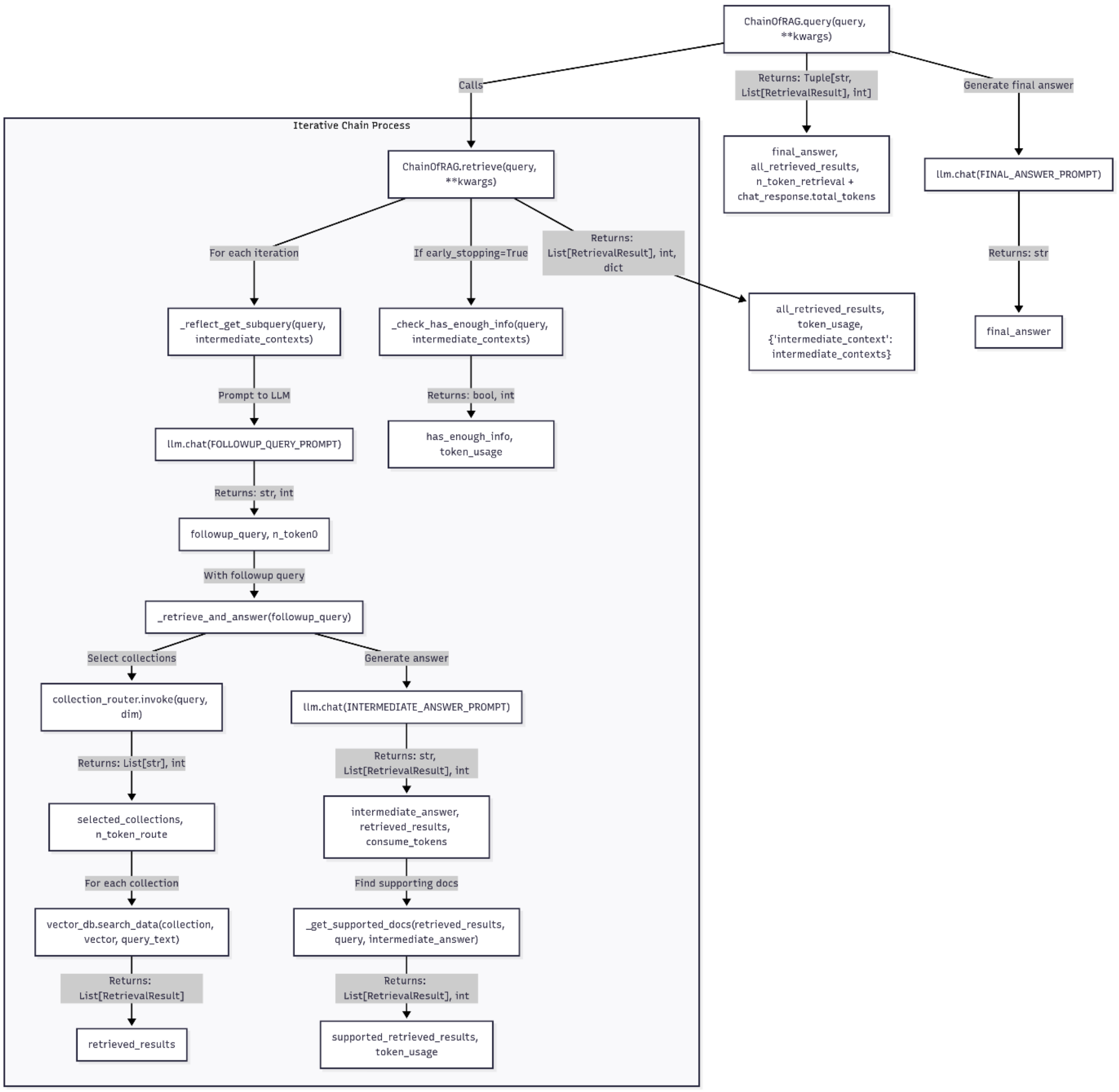
3. Query Execution Flow


/query/ Endpoint
Input: original_query, max_iter
perform_query()
Calls
query(original_query, max_iter)
Gets default_searcher
Returns to API
default_searcher.query(original_query, max_iter)
{'result': result_text, 'consume_token': consume_token}
default_searcher is RAGRouter
RAG Agent Selection Process
RAGRouter.query(original_query, max_iter)
Calls
Returns: Tuple[str, List[RetrievalResult], int]
With selected agent
RAGRouter._route(original_query)
answer, retrieved_results, n_token_router + n_token_retrieval
selected_agent.query(original_query, max_iter)
Prompt to LLM
Returns: Tuple[str, List[RetrievalResult], int]
llm.chat(router_prompt)
answer, retrieved_results, n_token_retrieval
Returns: selected_agent_index
Select DeepSearch or ChainOfRAG
Returns: Tuple[RAGAgent, token_count]
selected_agent, n_token_router

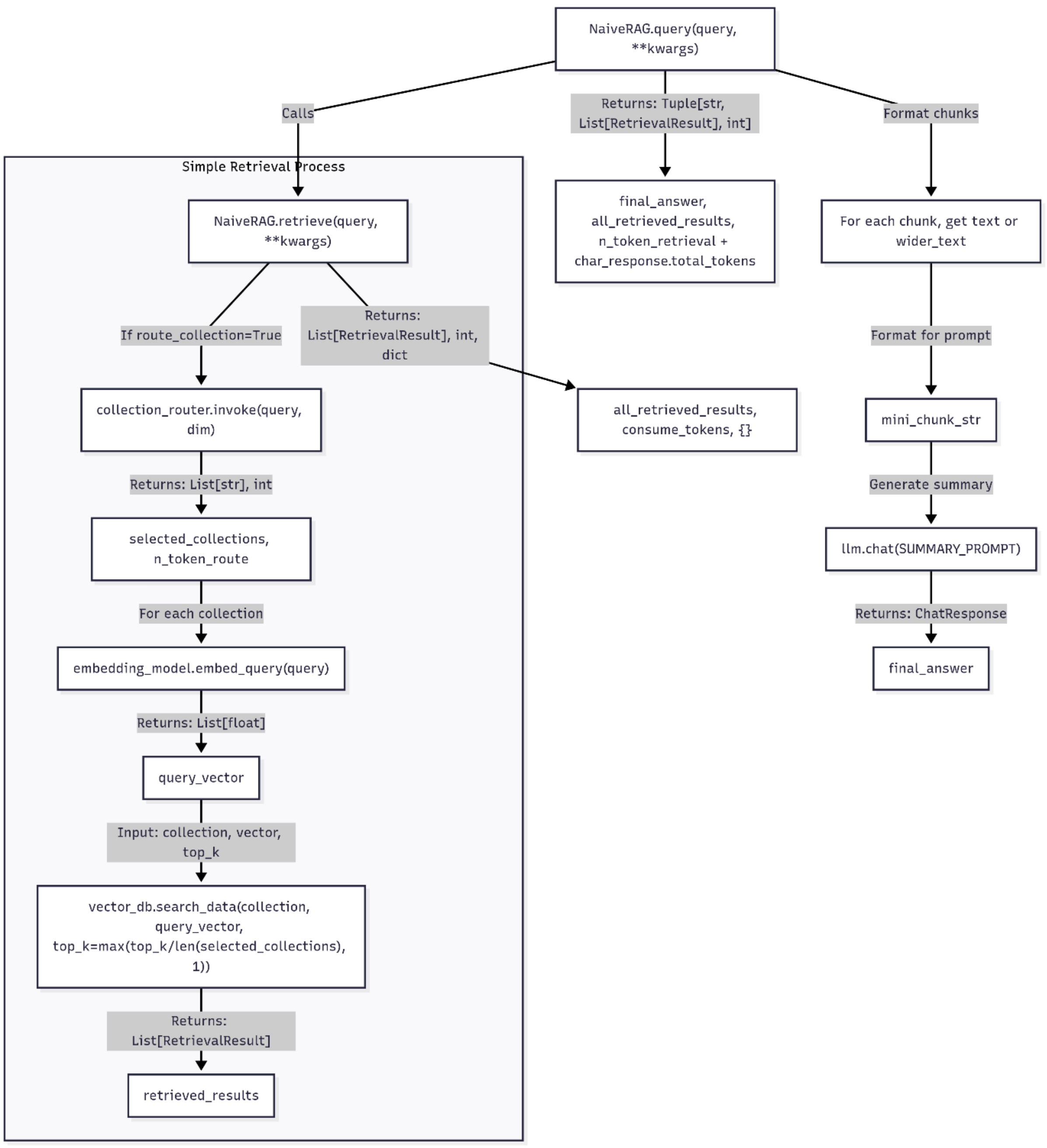
4. DeepSearch Agent workflow


DeepSearch.query(query, **kwargs)
Calls
After retrieve
Returns: Tuple[str, List[RetrievalResult], int]
DeepSearch.retrieve(query, **kwargs)
llm.chat(SUMMARY_PROMPT)
final_answer, all_retrieved_results, n_token_retrieval + chat_response.total_tokens
Uses asyncio
Returns: str
asyncio.run(async_retrieve(query, **kwargs))
final_answer
First step
If not final iteration
After all iterations
For each query in sub_queries
Collection Selection
_generate_sub_queries(original_query)
_generate_gap_queries(original_query, all_sub_queries, all_search_res)
deduplicate_results(all_search_res)
_search_chunks_from_vectordb(query, sub_queries)
Prompt to LLM
Prompt to LLM
Returns: List[RetrievalResult], int, dict
Returns: List[RetrievalResult], int
If route_collection=True
For each collection
Result Reranking
For each result
llm.chat(SUB_QUERY_PROMPT)
all_retrieved_results, consume_tokens
llm.chat(REFLECT_PROMPT)
all_search_res, total_tokens, additional_info
collection_router.invoke(query, dim)
Vector Search Process
embedding_model.embed_query(query)
Input: query, result
Returns: List[str], token_count
Returns: List[str], int
Returns: List[str], int
Returns: List[float]
llm.chat(RERANK_PROMPT)
sub_queries, used_token
sub_gap_queries, consumed_token
selected_collections, n_token_route
query_vector
If YES
If NO
Keep result
Discard result
Input: collection, vector, query_text
vector_db.search_data(...)
Returns: List[RetrievalResult]
retrieved_results

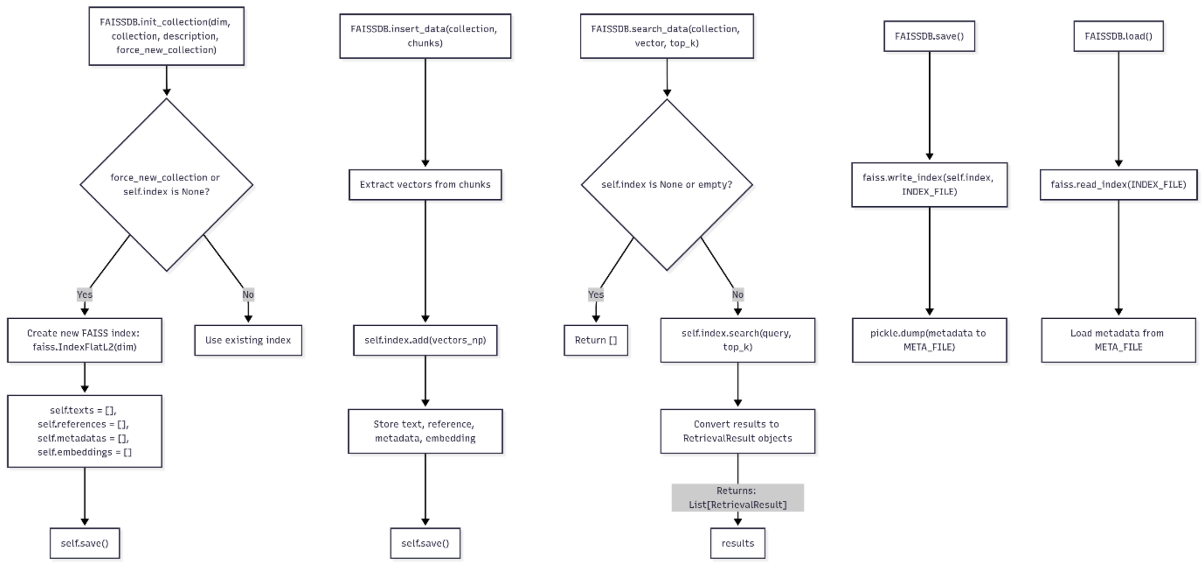
5. Chain of RAG


ChainOfRAG.query(query, **kwargs)
Calls
Returns: Tuple[str, List[RetrievalResult], int]
Generate final answer
Iterative Chain Process
final_answer, all_retrieved_results, n_token_retrieval + chat_response.total_tokens
llm.chat(FINAL_ANSWER_PROMPT)
ChainOfRAG.retrieve(query, **kwargs)
For each iteration
If early_stopping=True
Returns: List[RetrievalResult], int, dict
Returns: str
all_retrieved_results, token_usage, {'intermediate_context': intermediate_contexts}
final_answer
_reflect_get_subquery(query, intermediate_contexts)
_check_has_enough_info(query, intermediate_contexts)
Prompt to LLM
Returns: bool, int
llm.chat(FOLLOWUP_QUERY_PROMPT)
has_enough_info, token_usage
Returns: str, int
followup_query, n_token0
With followup query
_retrieve_and_answer(followup_query)
Select collections
Generate answer
collection_router.invoke(query, dim)
llm.chat(INTERMEDIATE_ANSWER_PROMPT)
Returns: List[str], int
Returns: str, List[RetrievalResult], int
selected_collections, n_token_route
intermediate_answer, retrieved_results, consume_tokens
For each collection
Find supporting docs
vector_db.search_data(collection, vector, query_text)
_get_supported_docs(retrieved_results, query, intermediate_answer)
Returns: List[RetrievalResult]
Returns: List[RetrievalResult], int
retrieved_results
supported_retrieved_results, token_usage

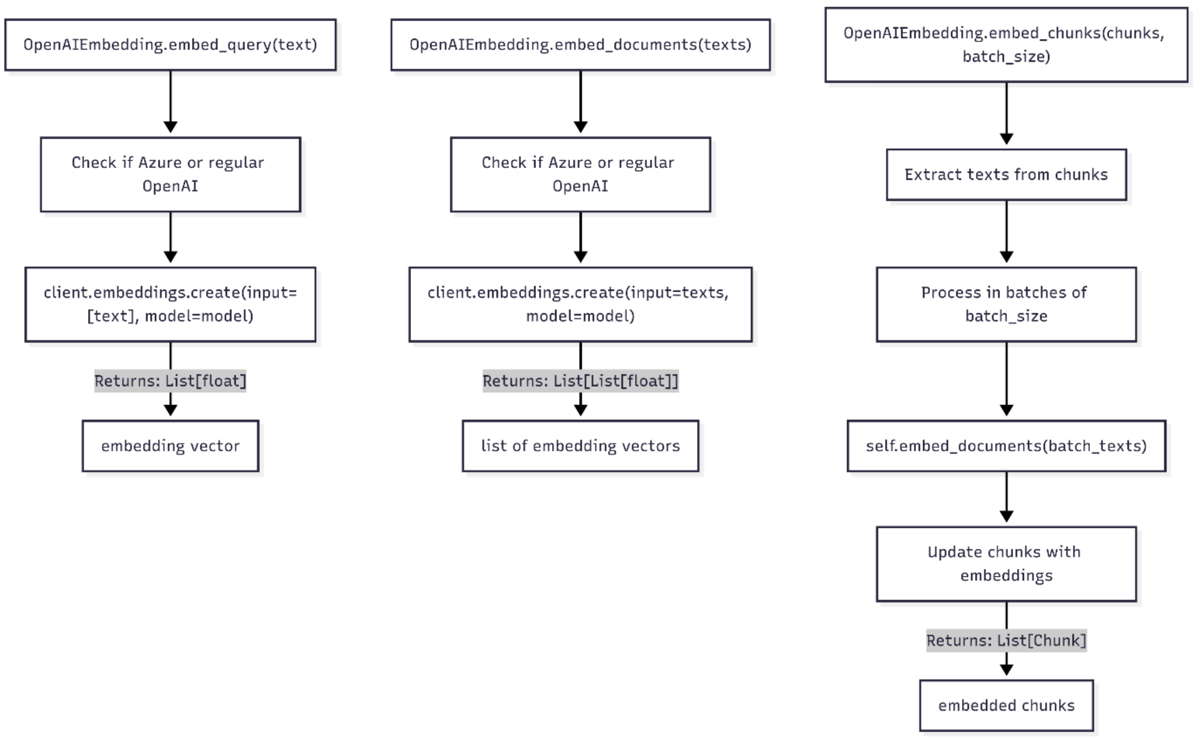
6. Naïve RAG


NaiveRAG.query(query, **kwargs)
Calls
Returns: Tuple[str, List[RetrievalResult], int]
Format chunks
Simple Retrieval Process
NaiveRAG.retrieve(query, **kwargs)
final_answer, all_retrieved_results, n_token_retrieval + char_response.total_tokens
For each chunk, get text or wider_text
If route_collection=True
Returns: List[RetrievalResult], int, dict
Format for prompt
collection_router.invoke(query, dim)
all_retrieved_results, consume_tokens, {}
mini_chunk_str
Returns: List[str], int
Generate summary
selected_collections, n_token_route
llm.chat(SUMMARY_PROMPT)
For each collection
Returns: ChatResponse
embedding_model.embed_query(query)
final_answer
Returns: List[float]
query_vector
Input: collection, vector, top_k
vector_db.search_data(collection, query_vector, top_k=max(top_k/len(selected_collections), 1))
Returns: List[RetrievalResult]
retrieved_results

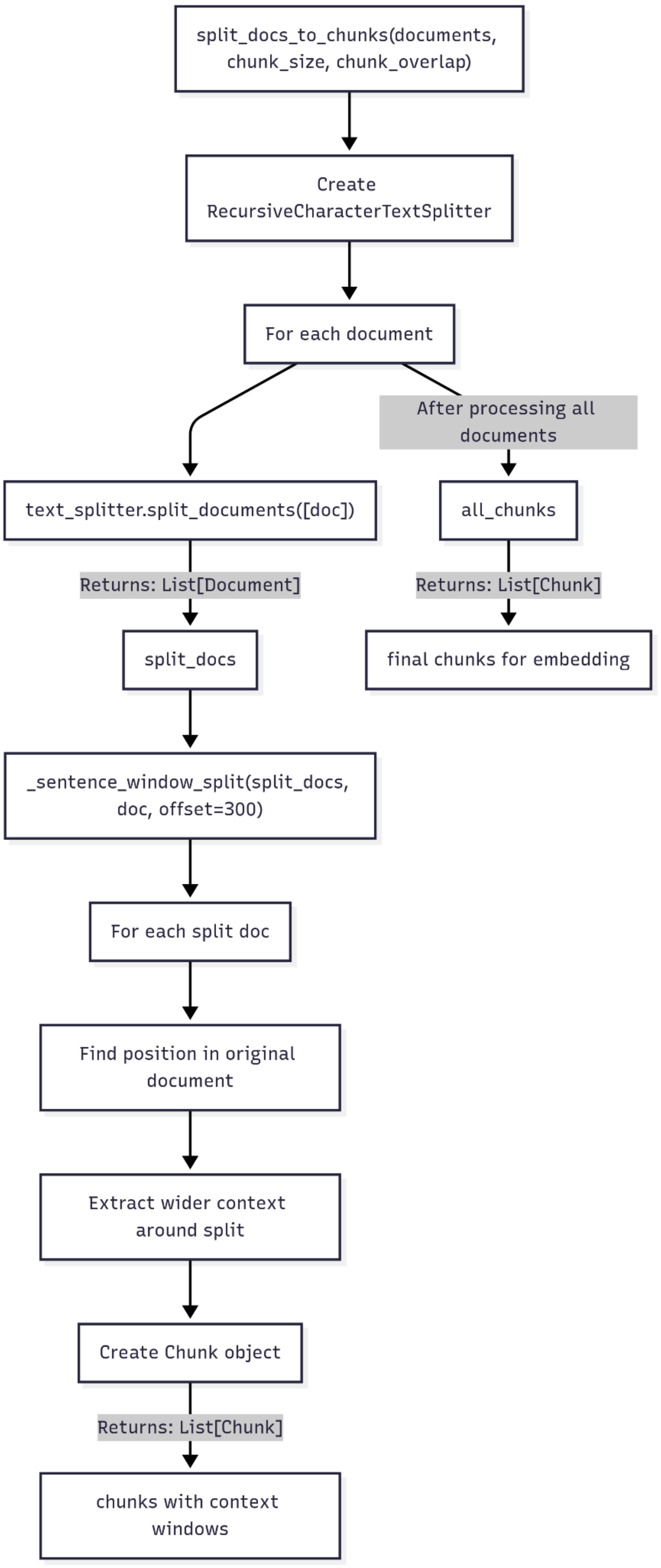
7. FAISS Vector Database Operations


FAISSDB.init_collection(dim, collection, description, force_new_collection)
force_new_collection or self.index is None?
Yes
No
Create new FAISS index: faiss.IndexFlatL2(dim)
Use existing index
self.texts = [], self.references = [], self.metadatas = [], self.embeddings = []
self.save()
FAISSDB.insert_data(collection, chunks)
Extract vectors from chunks
self.index.add(vectors_np)
Store text, reference, metadata, embedding
self.save()
FAISSDB.search_data(collection, vector, top_k)
self.index is None or empty?
Yes
No
Return []
self.index.search(query, top_k)
Convert results to RetrievalResult objects
Returns: List[RetrievalResult]
results
FAISSDB.save()
faiss.write_index(self.index, INDEX_FILE)
pickle.dump(metadata to META_FILE)
FAISSDB.load()
faiss.read_index(INDEX_FILE)
Load metadata from META_FILE

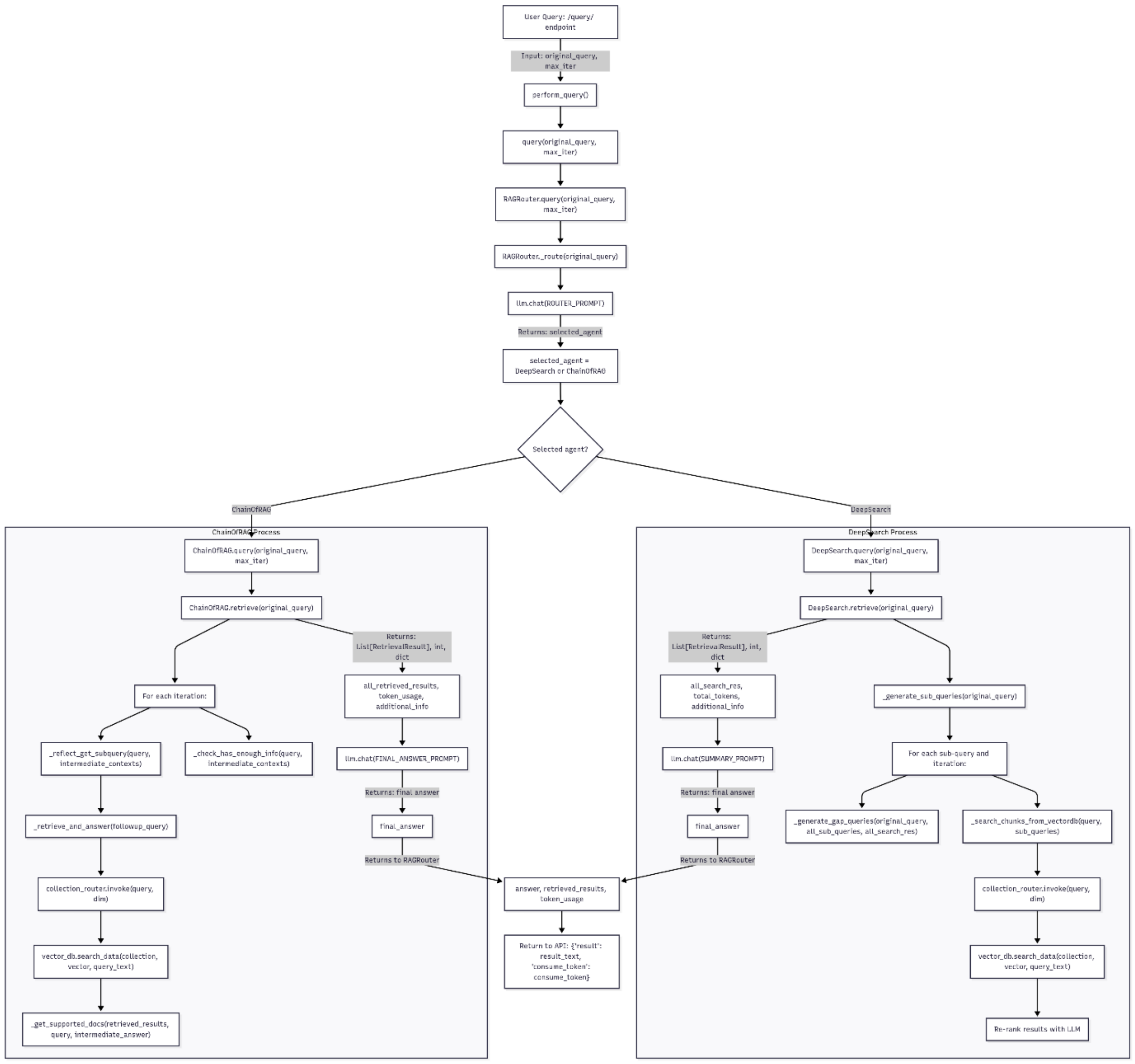
8. OpenAI Embedding Operations


OpenAIEmbedding.embed_query(text)
Check if Azure or regular OpenAI
client.embeddings.create(input=[text], model=model)
Returns: List[float]
embedding vector
OpenAIEmbedding.embed_documents(texts)
Check if Azure or regular OpenAI
client.embeddings.create(input=texts, model=model)
Returns: List[List[float]]
list of embedding vectors
OpenAIEmbedding.embed_chunks(chunks, batch_size)
Extract texts from chunks
Process in batches of batch_size
self.embed_documents(batch_texts)
Update chunks with embeddings
Returns: List[Chunk]
embedded chunks

9. Document Splitting Operations


split_docs_to_chunks(documents, chunk_size, chunk_overlap)
Create RecursiveCharacterTextSplitter
For each document
After processing all documents
text_splitter.split_documents([doc])
all_chunks
Returns: List[Document]
Returns: List[Chunk]
split_docs
final chunks for embedding
_sentence_window_split(split_docs, doc, offset=300)
For each split doc
Find position in original document
Extract wider context around split
Create Chunk object
Returns: List[Chunk]
chunks with context windows

10. **Complete End-to-End Query Flow: From User Request to Response**


User Query: /query/ endpoint
Input: original_query, max_iter
perform_query()
query(original_query, max_iter)
RAGRouter.query(original_query, max_iter)
RAGRouter._route(original_query)
llm.chat(ROUTER_PROMPT)
Returns: selected_agent
selected_agent = DeepSearch or ChainOfRAG
Selected agent?
ChainOfRAG
ChainOfRAG Process
ChainOfRAG.query(original_query, max_iter)
ChainOfRAG.retrieve(original_query)
Returns: List[RetrievalResult], int, dict
all_retrieved_results, token_usage, additional_info
llm.chat(FINAL_ANSWER_PROMPT)
Returns: final answer
final_answer
Returns to RAGRouter
For each iteration:
_reflect_get_subquery(query, intermediate_contexts)
_check_has_enough_info(query, intermediate_contexts)
_retrieve_and_answer(followup_query)
collection_router.invoke(query, dim)
vector_db.search_data(collection, vector, query_text)
_get_supported_docs(retrieved_results, query, intermediate_answer)
DeepSearch
DeepSearch Process
DeepSearch.query(original_query, max_iter)
DeepSearch.retrieve(original_query)
Returns: List[RetrievalResult], int, dict
all_search_res, total_tokens, additional_info
llm.chat(SUMMARY_PROMPT)
Returns: final answer
final_answer
Returns to RAGRouter
_generate_sub_queries(original_query)
For each sub-query and iteration:
_generate_gap_queries(original_query, all_sub_queries, all_search_res)
_search_chunks_from_vectordb(query, sub_queries)
collection_router.invoke(query, dim)
vector_db.search_data(collection, vector, query_text)
Re-rank results with LLM
answer, retrieved_results, token_usage
Return to API: {'result': result_text, 'consume_token': consume_token}

Key Components

1. **FastAPI**: Web framework that handles HTTP requests
2. **Configuration**: Manages settings and component initialization
3. **ModuleFactory**: Creates instances of LLMs, embeddings, etc.
4. **RAGRouter**: Routes queries to appropriate RAG agents
5. **DeepSearch**: Complex RAG agent for comprehensive information retrieval
6. **ChainOfRAG**: RAG agent that decomposes queries into iterative steps
7. **NaiveRAG**: Simple RAG agent for basic retrieval operations
8. **FAISSDB**: Vector database for storing and searching embeddings
9. **OpenAIEmbedding**: Generates embeddings using OpenAI's API

Key Function Inputs/Outputs

Function	Inputs	Outputs	Description
`init_config()`	`config: Configuration`	None (sets globals)	Initializes all system components
`load_from_local_files()`	`paths_or_directory, collection_name, collection_description, batch_size`	None	Loads documents from files into vector DB
`load_from_website()`	`urls, collection_name, collection_description, batch_size`	None	Loads documents from websites into vector DB
`query()`	`original_query: str, max_iter: int`	`Tuple[str, List[RetrievalResult], int]`	Main query function that routes to RAG agents
`RAGRouter._route()`	`query: str`	`Tuple[RAGAgent, int]`	Selects best RAG agent for query
`DeepSearch.retrieve()`	`original_query: str, **kwargs`	`Tuple[List[RetrievalResult], int, dict]`	Retrieves documents using sub-queries and reflection
`ChainOfRAG.retrieve()`	`query: str, **kwargs`	`Tuple[List[RetrievalResult], int, dict]`	Retrieves documents using iterative queries
`split_docs_to_chunks()`	`documents: List[Document], chunk_size: int, chunk_overlap: int`	`List[Chunk]`	Splits documents into smaller chunks
`embed_chunks()`	`chunks: List[Chunk], batch_size: int`	`List[Chunk]`	Adds embeddings to document chunks
`vector_db.search_data()`	`collection: str, vector: List[float], top_k: int`	`List[RetrievalResult]`	Searches for similar vectors in database