

Homer - Technical Report

*A Retrieval-Augmented Generation System for Nuclear Engineers
Documentation*

Florent Bergé
Mathieu de la Barre
IMT Atlantique

Supervised by: Mr. Vincenzo De Florio

SCK CEN
Belgian Nuclear Research Centre
April 1st - July 18th, 2025

Classification: Restricted

Contents

0.1	Abstract	3
0.2	Introduction	3
1	Concepts	4
1.1	Retrieval-Augmented Generation	4
1.2	Core Components	4
1.2.1	Vectorstore	4
1.2.2	Tokens, Embeddings and Embedding Models	5
1.2.3	Chat Model	5
1.3	Document Processing Pipeline	5
1.3.1	Indexing	5
1.3.2	Chunking Techniques	6
1.4	Framework Architecture	6
1.4.1	LangChain Framework	6
1.4.2	LangGraph	6
2	Architecture	8
2.1	Code architecture	9
2.2	Streamlit Client	10
2.2.1	Discussion Page	10
2.2.2	Report Page	11
2.2.3	Document Page	12
2.2.4	Model Page	13
2.3	Configuration Management	13
2.3.1	Configuration Architecture	13
2.3.2	Dynamic Host Resolution	14
2.3.3	Configuration Integration with LangGraph	14
2.3.4	Model Selection Strategy	14
2.4	Agents architecture	15
2.4.1	Graph State Architecture	16
2.4.2	State Reducers	16
2.4.3	Memory and Persistence	16
2.4.4	RetrievalAgent - Conversational RAG Implementation	17
2.4.5	ReportAgent - Structured Document Generation	21
2.4.6	IndexAgent - Document Ingestion Pipeline	26
2.5	Retrieval System	28
2.5.1	Vector Store Management	28
2.5.2	Search Configuration	29
2.5.3	Document Lifecycle Management	29
2.5.4	Experimental Support: Hybrid Retrieval with Milvus	30

2.6	Error Handling and Resilience	32
2.6.1	Comprehensive Error Management	32
2.6.2	Logging Architecture	32
2.6.3	Caching Strategy	32
3	Vision Parser	33
3.1	VisionLoader	33
3.2	Multi-Modal Parsing Architecture	34
3.3	Text Validation System	34
3.3.1	Fallback Mechanism	35
4	Setup	36
4.1	Homer Git	36
4.2	Python	36
4.3	Virtual Environment	36
4.4	Start the Streamlit app	36
4.5	Server execution	37
5	How to use Homer	38
5.1	Home page	38
5.2	Documents page	38
5.3	Configuration Page	39
5.4	Models Page	40
5.5	Discussion Page	41
5.6	Report Page	41
6	Next steps - Future development	43
6.1	.exe file	43
6.2	Client-Server Setup with L6 and SSH Tunneling	44
6.3	Agentic RAG	45
6.4	Future Integration within the SCK CEN Infrastructure	46

0.1 Abstract

This report presents the results of a research and development project conducted by Florent Bergé and Mathieu de la Barre, engineering trainees from IMT Atlantique, between March 1st and July 18th, 2025. The project focused on the design and implementation of Homer, a generative AI assistant tailored for the retrieval and interpretation of technical knowledge within the framework of the SMR-LFR nuclear reactor program at SCK CEN.

The work was carried out under the supervision of Mr. Vincenzo De Florio, and builds on existing specifications, which are assumed as prior knowledge for the reader. The report details both the conceptual foundations and the technical implementations of the assistant, from document analysis and vectorization strategies to evaluation protocols and system architecture. The resulting tool is expected to significantly improve information accessibility and decision-making efficiency for domain experts engaged in advanced nuclear research.

0.2 Introduction

The SCK CEN has progressively accumulated a substantial body of knowledge through its involvement in the MYRRHA program and, more recently, the SMR-LFR nuclear reactor initiative. This expertise has been systematically documented in the Conceptual Design Report (CDR), a dynamic reference work currently exceeding 3,500 pages, as well as in an extensive knowledge base of requirements, design rationales, and engineering constraints. These resources are managed within Polarion environments dedicated to Requirements and Configuration Management. In light of the growing complexity and scale of these assets, SCK CEN has initiated an exploration into the use of generative AI technologies. The goal is to develop a reasoning assistant—akin to an expert system—that can support engineers in efficiently retrieving critical information related to MYRRHA and in navigating toward the most pertinent enterprise knowledge resources.

The result of this initiative is Homer, a multi-user assistant based on Retrieval-Augmented Generation (RAG) architecture. Designed as a secure, cloud-hosted solution within SCK CEN's infrastructure, it is tailored to respect the confidentiality and sensitivity of the internal documentation.

The name Homer pays homage to the ancient Greek poet traditionally regarded as the author of the Iliad and the Odyssey. In conceptual harmony with Alexandria—a symbolic reference to the accumulation and preservation of knowledge—Homer stands for its interpretation and transmission[1].

Chapter 1

Concepts

1.1 Retrieval-Augmented Generation

The Retrieval-Augmented Generation (RAG) pattern is an industry-standard approach to building applications that use language models to process specific or proprietary data that the model doesn't already know. The architecture is straightforward and best suited for large and fast-changing data sets (compared to fine-tuning).

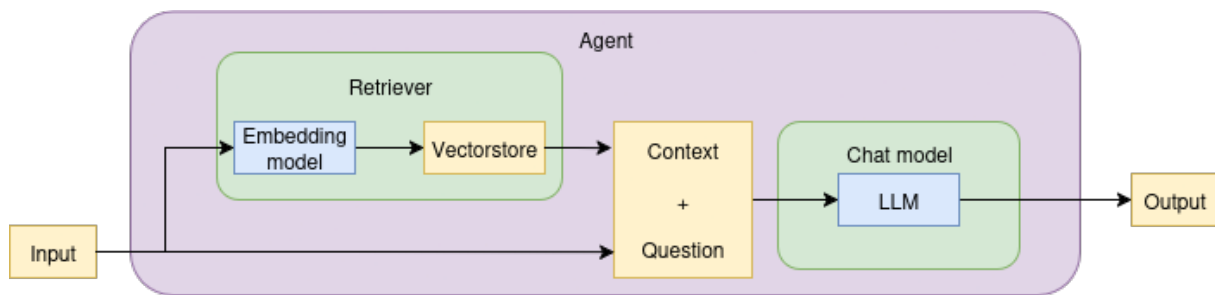


Figure 1.1: RAG application flow

1. The user issues a query
2. The application performs a search on the vectorstore based on the user input
3. The top k results matching the query are extracted. Both the results and the query are then packaged together in a prompt and sent to the Large Language Model (LLM)
4. The LLM outputs a structured answer based on both the input and the retrieved context

1.2 Core Components

1.2.1 Vectorstore

A vectorstore is a specialized database designed to store and retrieve high-dimensional vectors efficiently. In the context of RAG applications, vectorstores serve as the foundation for semantic search capabilities. They enable the system to find relevant documents based on meaning rather than exact keyword matches. The vectorstore indexes document embeddings and supports similarity searches using distance metrics such as cosine similarity, Euclidean distance, or dot product. Popular vectorstore solutions include Pinecone, Weaviate, Chroma, and FAISS, each offering different trade-offs between performance, scalability, and ease of deployment.

1.2.2 Tokens, Embeddings and Embedding Models

Tokens are the fundamental units of text that language models process. Text is broken down into tokens, which can represent words, subwords, or characters depending on the tokenization strategy. Modern models typically use subword tokenization methods like Byte Pair Encoding (BPE) or SentencePiece.

Embeddings are dense vector representations of text that capture semantic meaning in a high-dimensional space. Each embedding is a numerical array where semantically similar concepts are positioned closer together in the vector space. These embeddings enable the system to understand context and meaning beyond simple keyword matching.

Embedding models are specialized neural networks trained to convert text into these vector representations. Popular embedding models include OpenAI’s text-embedding-ada-002, Sentence-BERT, and various models from Hugging Face’s transformers library. The choice of embedding model significantly impacts the quality of document retrieval and overall system performance.

1.2.3 Chat Model

The chat model is the large language model responsible for generating responses in the RAG system. It receives both the user’s query and the retrieved context documents as input, then synthesizes this information to produce coherent, contextually relevant answers. Common chat models include GPT-3.5/4, Claude, Llama, and other instruction-tuned language models. The chat model’s performance depends on its ability to understand context, follow instructions, and generate accurate responses based on the provided information.

1.3 Document Processing Pipeline

1.3.1 Indexing

An indexing pipeline allows transforming files into useable data for the RAG application. The data pipeline processes each document individually by completing the following steps:

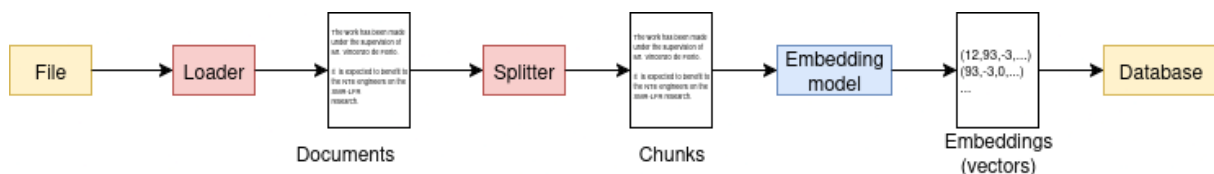


Figure 1.2: Indexing pipeline

1. **Loading:** Data is extracted from the documents
2. **Chunking:** The document is divided into smaller parts—recommended between 256 and 512 chunks—according to a chunking technique
3. **Chunk enrichment:** Adds metadata fields that the pipeline creates based on the content in the chunks, such as the source document, the titles and other relevant information
4. **Indexing:** All the chunks are turned into embeddings and stored into a vector database according to an indexing algorithm[3]

1.3.2 Chunking Techniques

The system supports two chunking approaches:

Recursive Character Splitting: The data is divided into chunks of a given size with specified overlap between adjacent chunks. The overlap prevents sentences from being split in the middle, which could potentially cause loss of meaning. This method provides consistent chunk sizes but may not preserve semantic boundaries.

Semantic Text Chunking: This technique requires an embedding model to divide content based on meaning rather than character count. It preserves most sentences or propositions intact and then combines semantically related chunks together. While more computationally expensive, this approach often results in more coherent and contextually meaningful chunks.

1.4 Framework Architecture

1.4.1 LangChain Framework

LangChain is a popular open-source framework designed to simplify the development of applications powered by large language models. It provides a comprehensive set of tools and abstractions that streamline the implementation of RAG systems by offering pre-built components, standardized interfaces, and seamless integrations with various LLMs, vector databases, and data sources. The framework follows a modular architecture that allows developers to mix and match components while maintaining clean, maintainable code.

Key LangChain Components:

- **The Retriever:** A core abstraction that defines a standard interface for fetching relevant documents based on a query. It acts as the bridge between the user's question and the vectorstore, encapsulating the search logic and returning a list of relevant documents
 - *VectorStoreRetriever*: Performs similarity search on vector databases
 - *MultiQueryRetriever*: Generates multiple variations of the user query to improve retrieval coverage
- **The Document Class:** LangChain's standardized representation of a piece of text content along with its associated metadata. Each Document object contains:
 - *page_content*: The actual text content of the document chunk
 - *metadata*: A dictionary containing additional information such as source file, page number, creation date, document type, and custom fields

1.4.2 LangGraph

LangGraph is a framework built on top of LangChain that enables the creation of stateful, multi-actor applications with large language models. It represents a paradigm shift from simple linear chains to complex, graph-based workflows where different components can interact, make decisions, and maintain state over time. LangGraph is particularly valuable for RAG applications that require sophisticated orchestration, conditional logic, human-in-the-loop workflows, or multi-step reasoning processes.

Core LangGraph Components:

- **The Graph:** The core abstraction that defines the structure and flow of your application. It represents a directed graph where nodes perform specific tasks and edges define the transitions between these tasks;

- **Nodes:** Individual processing units within a LangGraph. Each node encapsulates a specific function, receiving the current state as input and returning updated state information;
- **State Management:** Allows information to persist and evolve throughout the execution of a graph. The state serves as a shared data structure that nodes can read from and write to;
- **The Checkpointer:** Responsible for persisting the state of a graph execution at specific points, enabling durability and recovery capabilities. In our implementation, we use an in-memory SQLite database;

```
1 from langgraph.checkpoint.sqlite import SqliteSaver
2
3 builder = StateGraph(...)
4
5 conn = sqlite3.connect(':memory:', check_same_thread=False)
6 memory = SqliteSaver(conn)
7
8 graph = builder.compile(checkpointer=memory)
```

Listing 1.1: Checkpointer usage

- **The Runnable Config:** A configuration object that controls how a graph executes, providing fine-grained control over the execution environment and allowing configuration to be passed inside the nodes. In order to provide a custom configuration to the graph, it needs to be passed in the "configurable" argument of the RunnableConfig and turned into a dictionary. To use a checkpoint, you also have to add a *thread_id* to the configurable dictionary;

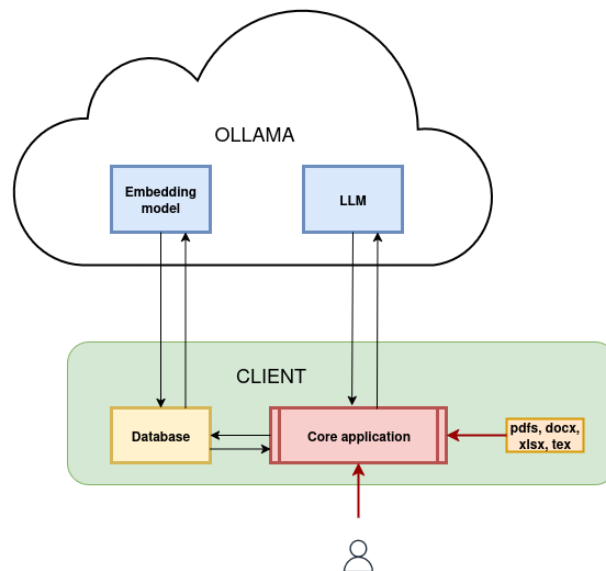
```
1 configuration = {"parameter1":value1, ...}|{"thread_id":1}
2
3 graph.invoke(
4     input=user_input,
5     config={"configurable":configuration}
6 )
```

Listing 1.2: Runnable config usage

Chapter 2

Architecture

Homer is a RAG application that allows users to provide their own documents to augment the agent's knowledge. It is composed of a Streamlit web interface, LangGraph agents, and an Ollama client.



The core Application (interface, agents and vectorstore) runs on the local machine and sends requests to an Ollama client which handles model inferences. The latter can either be running on the local machine as well or on a distant server.

This architecture was chosen because it allows confidential data to stay on the local machine. Indeed, having a shared database between users would have required either managing the access of each user to specific documents or creating user-specific spaces which would have caused redundancy. Hence, the decision was made to perform all small calculations on the local computer and let a server run the models using the Ollama API.

2.1 Code architecture

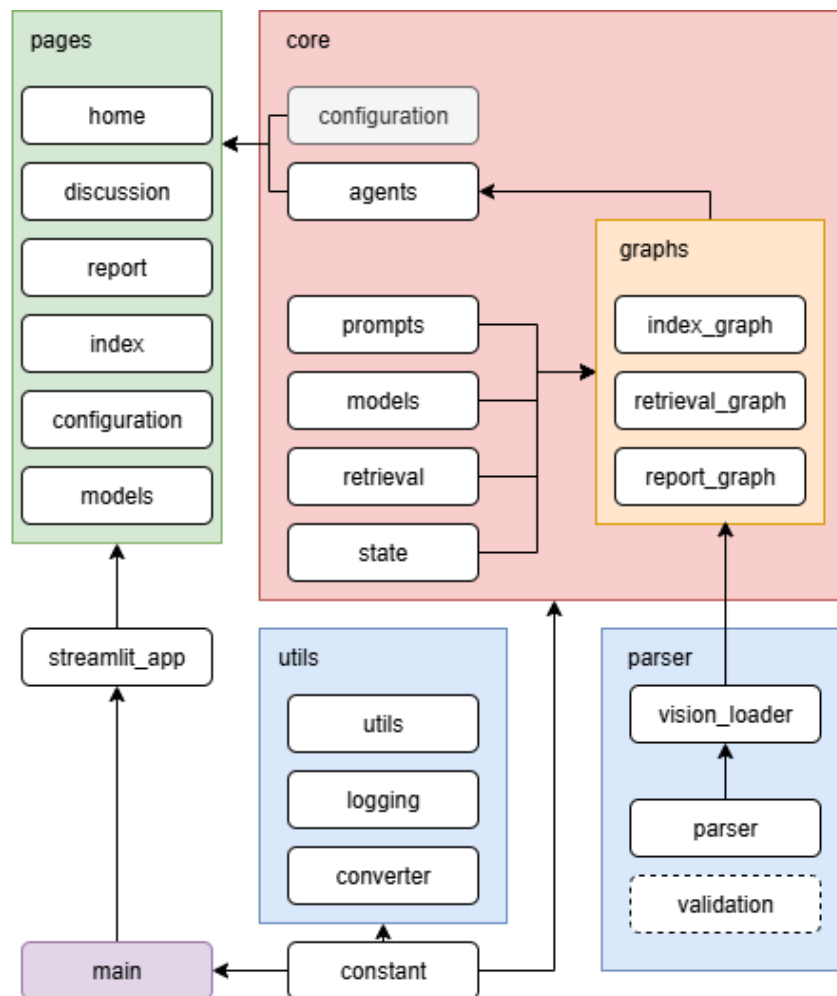


Figure 2.1: Code organization

The application starts by running:

```
1 python main.py
```

This file ensures mandatory directories exist and then runs the line:

```
1 streamlit run streamlit_app.py
```

- **Pages** refers to the streamlit interface pages, which are managed by the file `streamlit_app`
- **core** refers to the core components (configuration and agents related files)
- **utils** are all the utility functions that can be called in any file, such as the log management, functions to load a sqlite3 connection etc
- **parser** refers to the custom pdf parser described in Chapter 3

2.2 Streamlit Client

The Streamlit client is composed of 6 pages:

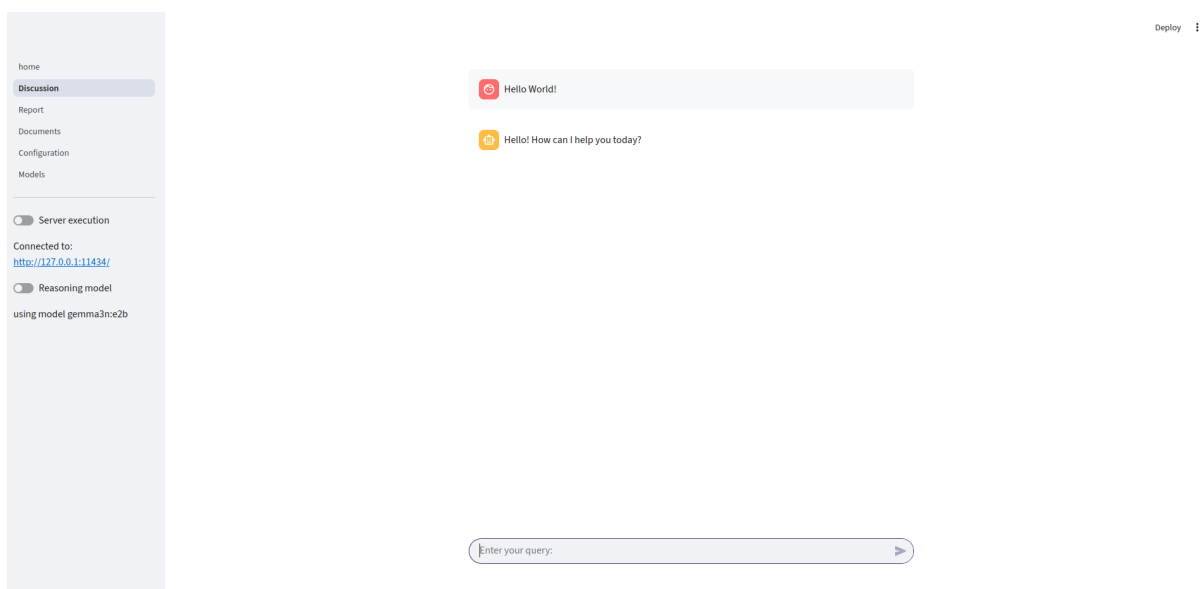
- **Home:** Welcome page with navigation
- **Discussion:** The page where users can chat with the RetrievalAgent
- **Report:** The page where users can send a query to the ReportAgent
- **Documents:** The page to add documents to the knowledge base
- **Configuration:** The page where users can temporarily modify parameters
- **Models:** Allows users to enter an Ollama model to pull

Session state

Streamlit uses a 'session_state' to store session variables such as:

- **baseConfig** (Configuration): Loads the default Configuration
- **ollama_host** (str): The IP address of the distant Ollama client
- **models** (dictionary): The discussion models, reasoning or standard, local or server (server ones are larger and more accurate than local ones)

2.2.1 Discussion Page



This is the main page for asking questions to the RetrievalAgent. The discussion displays in the main page, above the input field at the bottom. It features 2 buttons in the sidebar:

- **Server Execution:** Allows running models on the distant client at the 'ollama_host' address, if available

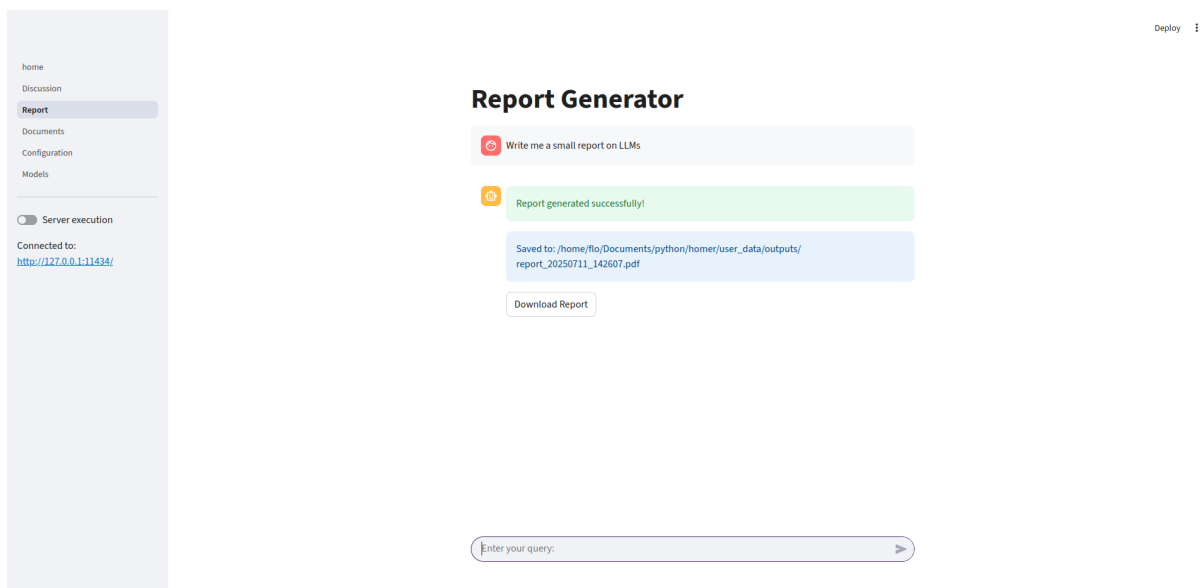
- **Reasoning Model:** Allows switching between traditional and reasoning models. Due to their 'thinking' step, they take more time to output an answer. When Reasoning model is on, the thinking part will be accessible through an expander on top of the answer

The specific session_state values for this page are:

- **retrievalAgent** (RetrievalAgent): The compiled retrieval agent
- **ollama_host**
- **baseConfig**
- **models**

Loading the retrieval agent in the session_state allows compiling the graph only once.

2.2.2 Report Page



This page allows the user to request a report to the ReportAgent. The sidebar features 2 options:

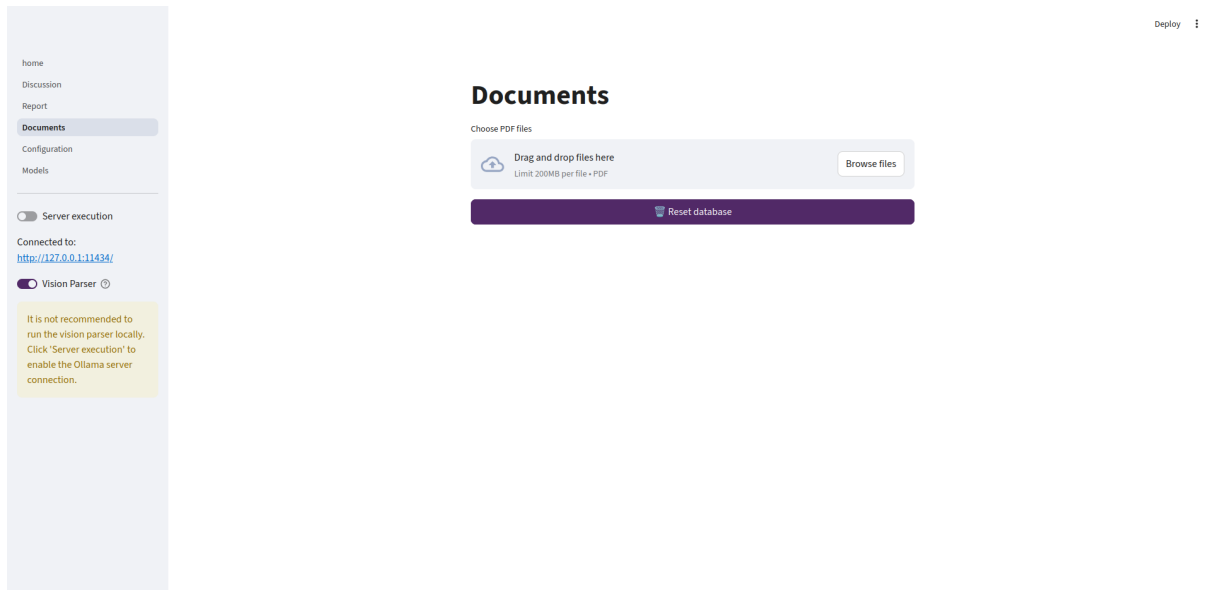
- **Server Execution**
- **Writing:** Allows switching between 'general' and 'technical' writing style

Once a report is completed, a button to download it.

The specific session_state values for this page are:

- **reportAgent** (ReportAgent): The compiled report agent
- **report_history** (list): A list of previous reports from the session
- **ollama_host**
- **baseConfig**

2.2.3 Document Page



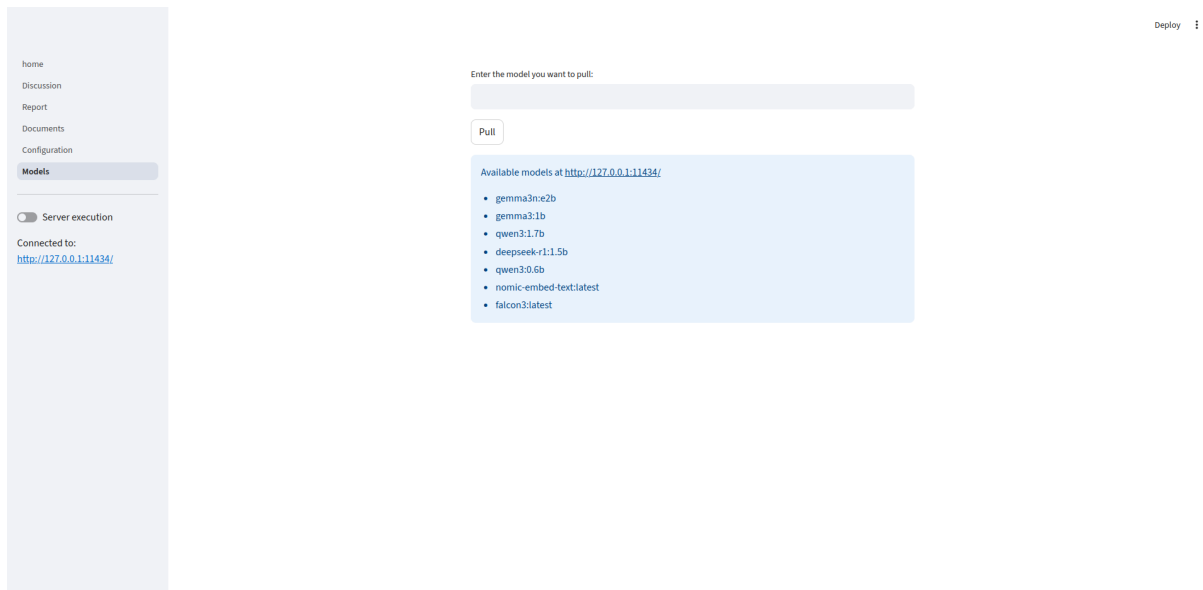
This page allows the user to send documents to the documents pipeline (IndexAgent). The sidebar features 2 options:

- **Server Execution**
- **Vision parser:** Allows switching between PyMuPDF parser and a custom Vision parser (using a LLVM)

The specific session_state values for this page are:

- **indexAgent** (IndexAgent): The compiled indexing pipeline
- **ollama_host**
- **baseConfig**

2.2.4 Model Page



This page allows the user to pull a model to either the local or server ollama client.

2.3 Configuration Management

Configuration
<ul style="list-style-type: none"> + embedding_model: str + number_of_parts: int + ocr: bool + ollama_host: str + query_model: str + response_model: str + report_model: str + vision_model: str + writing_style: Literal["technical" , "general"]
<ul style="list-style-type: none"> + asdict(): dict[str, Any] + from_runnable_config(RunnableConfig): Configuration

2.3.1 Configuration Architecture

Homer implements a centralized configuration system through the **Configuration** dataclass located in `src/core/configuration.py`. This approach ensures type safety, validation, and consistent parameter management across all system components.

The configuration system utilizes Python's `dataclass` decorator with the `kw_only=True` parameter, enforcing explicit parameter naming and improving code readability. The implementation supports both static default values and dynamic configuration loading based on Ollama client availability.

```

1 @dataclass(kw_only=True)
2 class Configuration:
3     # Report configuration
4     number_of_parts: int = field(default=5)
5
6     # Ollama configuration
7     ollama_host: str = field(default=OLLAMA_LOCALHOST)
8
9     # Model specifications
10    embedding_model: str = field(default="nomic-embed-text")
11    response_model: str = field(default="gemma3:1b")
12    # ... additional model configurations

```

Listing 2.1: Configuration dataclass structure

2.3.2 Dynamic Host Resolution

The system implements intelligent host resolution through the `_is_ollama_client_available()` function, which performs connectivity checks with a 2-second timeout. When the remote Ollama client at `OLLAMA_CLIENT` is available, the configuration automatically switches to server execution mode. Otherwise, it falls back to local execution using `OLLAMA_LOCALHOST` (<http://127.0.0.1:11434/>). This dual-mode operation enables:

- **Local development** with reduced model capabilities but full functionality
- **Production deployment** with access to larger, more capable models on dedicated hardware
- **Automatic failover** ensuring system availability regardless of server status

2.3.3 Configuration Integration with LangGraph

The configuration system integrates seamlessly with LangGraph's `RunnableConfig` through the `from_runnable_config()` class method. This enables:

```

1 configuration = Configuration.from_runnable_config(config)
2 configurable = config.get("configurable") or {}

```

Listing 2.2: Configuration integration with LangGraph

Each graph node receives configuration parameters through the standardized LangGraph configuration mechanism, ensuring consistent behavior across all agents and maintaining separation of concerns between business logic and configuration management.

2.3.4 Model Selection Strategy

Homer implements a sophisticated model selection strategy supporting four distinct model categories. These are the defaults:

Category	Local Model	Server Model	Use Case
Standard	<code>gemma3n:e2b</code>	<code>gemma3:4b-it-qat</code>	General responses
Reasoning	<code>qwen3:0.6b</code>	<code>qwen3:30b-a3b</code>	Complex analysis
Embedding	<code>nomic-embed-text</code>	<code>nomic-embed-text</code>	Document vectorization
Vision	N/A	<code>qwen2.5vl:3b-q4_K_M</code>	PDF parsing with OCR

Table 2.1: Model selection strategy across different categories

These values can be modified in the `src/pages/discussion.py` file.

The reasoning models incorporate a `<think>...</think>` mechanism, enabling explicit reasoning steps that are displayed separately from the final response in the user interface.

2.4 Agents architecture

Agents are organised as shown:

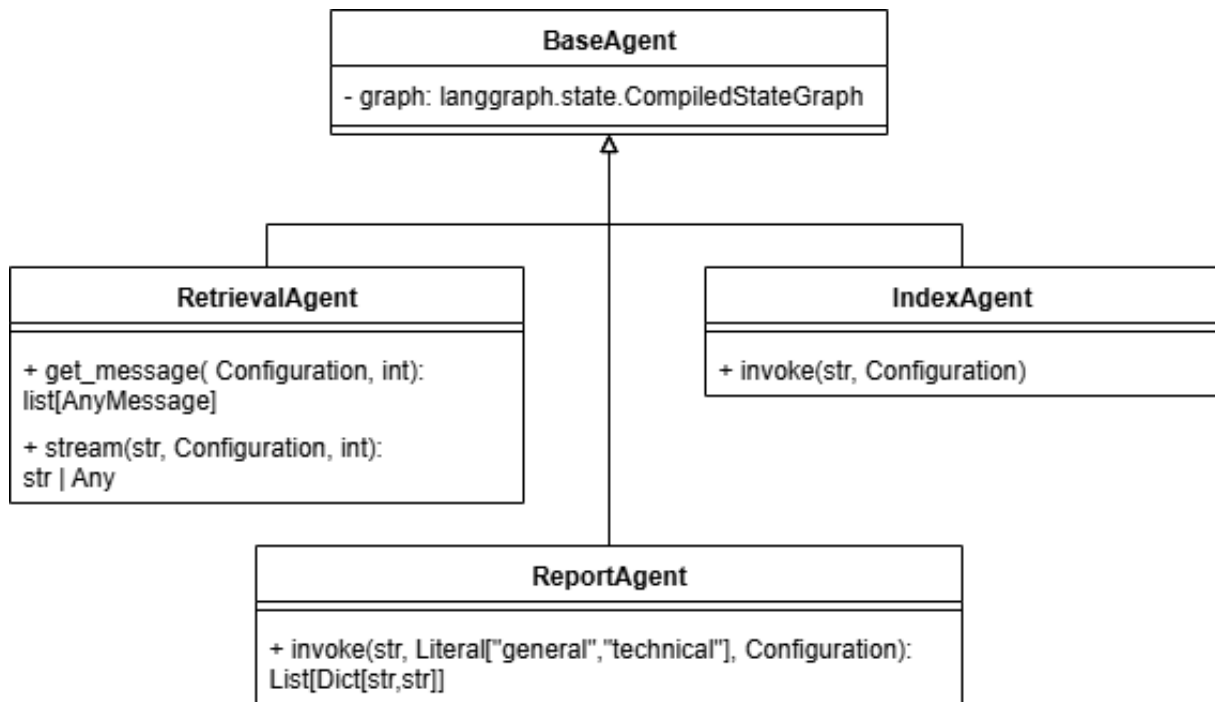
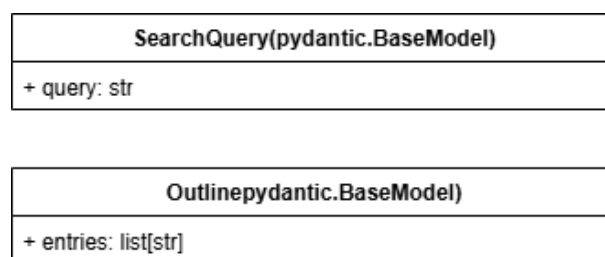


Figure 2.2: Agents class diagram

The `RetrievalAgent` and `ReportAgent` both use custom structured output at some point in their execution. They come in the form of pydantic's `BaseModels`:



2.4.1 Graph State Architecture

Homer implements a sophisticated state management system using LangGraph's state abstractions. Each agent maintains its specific state structure while sharing common interfaces:

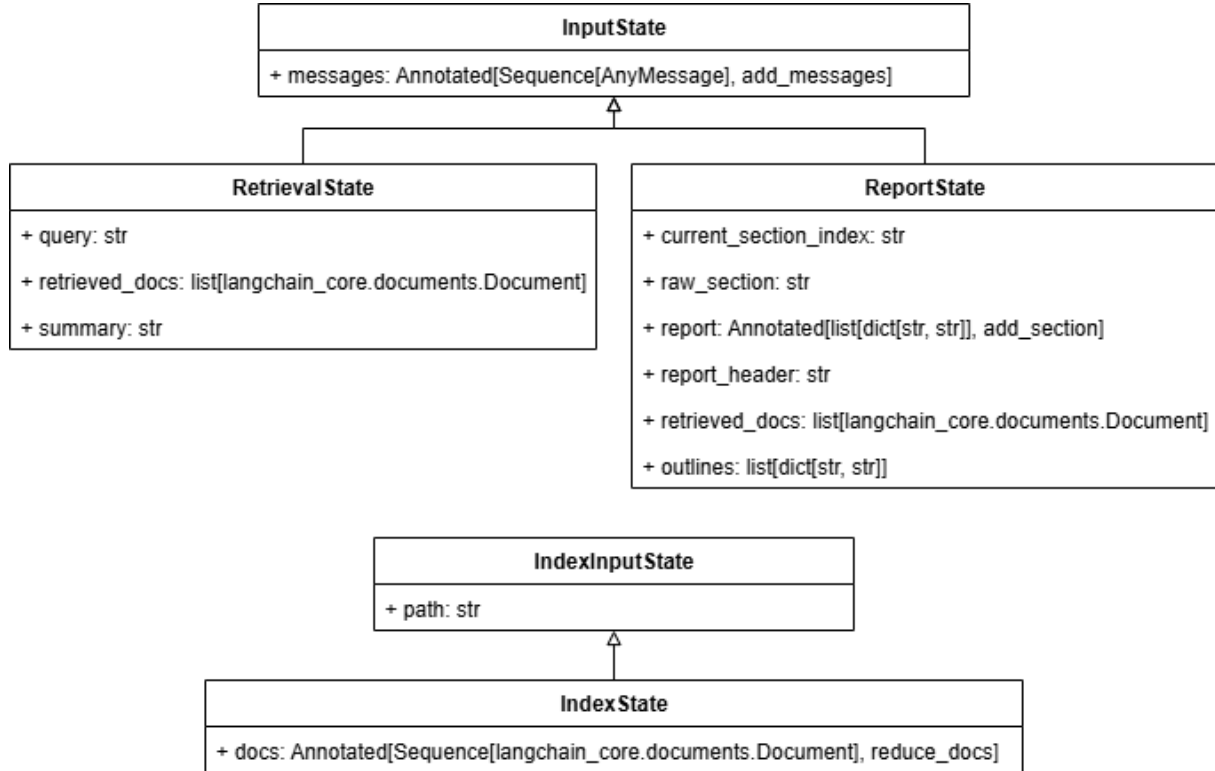


Figure 2.3: Agents states

2.4.2 State Reducers

The system implements custom state reducers for complex state transitions:

```

1 def reduce_docs(existing: Optional[Sequence[Document]],
2                 new: Union[Sequence[Document], str, Literal["delete"]]):
3     if new == "delete":
4         return []
5     # Handle various input types and transformations
  
```

Listing 2.3: Document state reducer

2.4.3 Memory and Persistence

- **Conversation Memory:** RetrievalAgent uses SQLite-based checkpointing for conversation persistence
- **Document Storage:** ChromaDB provides persistent vector storage with metadata
- **Session State:** Streamlit manages user session state for UI consistency

In a previous version, making the discussions persistent has been considered, but the streamlit interface made switching discussions difficult. You can find some files related to this in

`src/schemas` and `src/core/database.py`.

To enable the persistent database, the value `":memory:"` from `src/utils/utils.py`, `get_connection` should be changed to the persistent database directory. It will not work out of the box as some functions related to this have been removed of the current release.

2.4.4 RetrievalAgent - Conversational RAG Implementation

The RetrievalAgent implements a stateful conversational retrieval system using LangGraph's state management capabilities. The agent maintains conversation context through an SQLite checkpointer, enabling multi-turn conversations with persistent memory.

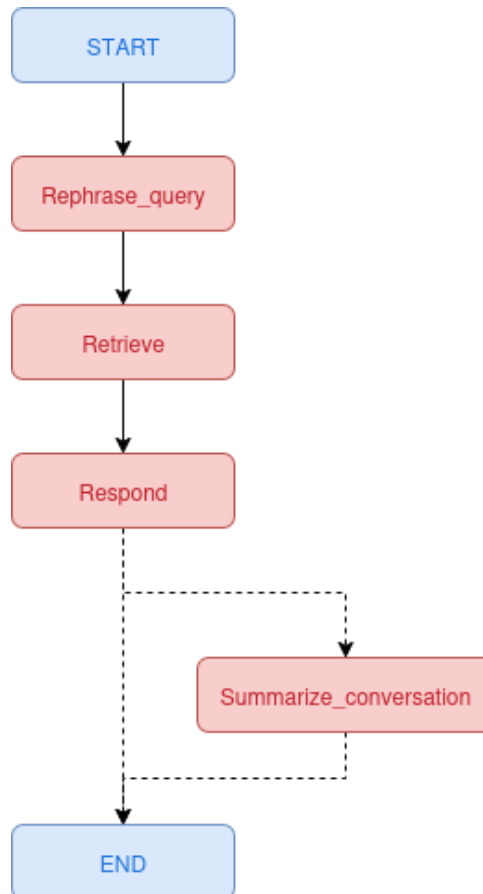


Figure 2.4: Retrieval graph structure

Nodes:

Rephrase query Generate an optimized search query based on conversation context. This function analyzes the current conversation state and generates a search query optimized for document retrieval, based on the last interaction.

```

1 def rephrase_query(
2     state: RetrievalState, *, config: RunnableConfig
3 ) -> Dict[str, Union[str, List]]:
4
5     configuration = Configuration.from_runnable_config(config)
6 
```

```

7      # Load and configure model
8      model = load_chat_model(
9          model=configuration.query_model,
10         host=configuration.ollama_host
11     ).with_structured_output(SearchQuery)
12
13     # Prepare message context
14     previous_messages = (
15         format_messages(state.messages[-3:-1])
16         if len(state.messages) >= 3
17         else "There were no previous messages."
18     )
19
20     # Create prompt
21     system_prompt = prompts.REPHRASE_QUERY_SYSTEM_PROMPT.format(
22         previous_messages= previous_messages,
23     )
24     user_prompt = state.messages[-1].content
25
26     messages = [
27         ("human", combine_prompts(system_prompt, user_prompt)),
28     ]
29
30     # Generate rephrased query
31     generated = cast(SearchQuery, model.invoke(messages, config))
32
33     retrievalAgentLogger.info(f"Generated query: '{generated.query}'")
34
35     return {
36         "query": generated.query,
37         "retrieved_docs": "delete" if state.retrieved_docs else [],
38     }

```

Listing 2.4: Rephrase query node

Retrieve Retrieve relevant documents based on the generated query. This function takes a search query from the state and retrieves the most relevant documents from the indexed document collection using vector similarity search. It uses the configured embedding model to encode the query and find matching documents.

```

1  def retrieve(
2      state: RetrievalState, *, config: RunnableConfig
3  ) -> Dict[str, List[Document]]:
4
5      configuration = Configuration.from_runnable_config(config)
6
7      # Load embedding model
8      embeddings = load_embedding_model(
9          model=configuration.embedding_model,
10         host=configuration.ollama_host
11     )
12
13     # Retrieve documents
14     with retrieval.make_retriever(embedding_model=embeddings) as
15         retriever:

```

```

15     response = retriever.invoke(state.query, config)
16
17     if response:
18         retrievalAgentLogger.info(f"Successfully retrieved {len(
19             response)} documents")
20         for doc in response:
21             retrievalAgentLogger.debug(f"Document: {doc.page_content
22                 } from {doc.metadata.get('source', 'unknown')}\n")
23     else:
24         retrievalAgentLogger.warning("No documents retrieved for the
25             query")
26
27     return {"retrieved_docs": response}

```

Listing 2.5: Retrieve node

Respond Generate a conversational response based on retrieved documents and chat history. This function creates a contextual response using the retrieved documents as context, considering the conversation history and any existing conversation summary. It handles message history efficiently by using summaries for older conversations.

```

1 def respond(
2     state: RetrievalState, *, config: RunnableConfig
3 ) -> Dict[str, Union[List[BaseMessage], List, str]]:
4
5     configuration = Configuration.from_runnable_config(config)
6
7     # Load model
8     model = load_chat_model(
9         model=configuration.response_model,
10        host=configuration.ollama_host
11    )
12
13    # Prepare context
14    previous_messages = ya_format_messages(state.messages[-3:-1] if len(
15        state.messages)>2 else [])
16    context_docs = format_docs(state.retrieved_docs) if state.
17        retrieved_docs else ""
18
19    # Create prompt
20    system_prompt = prompts.RESPONSE_SYSTEM_PROMPT.format(
21        context = context_docs,
22        summary = state.summary if state.summary else "",
23    )
24
25    messages = [
26        ("system", system_prompt)
27    ] + previous_messages + [
28        ("human", state.messages[-1].content)
29    ]
30
31    # Generate response
32    response = model.invoke(input=messages, config=config)
33
34    return {

```

```

33     "messages": [response],
34     "retrieved_docs": [],
35     "query": "",
36 }

```

Listing 2.6: Respond node

Summarize conversation Create or extend a conversation summary to manage long chat histories. This function generates a concise summary of recent conversation messages, either creating a new summary or extending an existing one. This helps maintain context while keeping prompt sizes manageable for long conversations.

```

1 def summarize_conversation(
2     state: RetrievalState, *, config: RunnableConfig
3 ) -> Dict[str, str]:
4
5     configuration = Configuration.from_runnable_config(config)
6
7     # Get existing summary
8     existing_summary = state.summary if state.summary else ""
9     messages_to_summarize = ya_format_messages(state.messages[-6:]) #
        Last 6 messages
10
11     # Determine prompt based on existing summary
12     if existing_summary:
13         summary_system_prompt = f""This is summary of the conversation
        to date:
14 <summary>
15 {existing_summary}
16 </summary>
17
18 Extend the summary by taking into account the new messages:""
19
20     else:
21         summary_system_prompt = "Create a summary of the conversation:"
22
23     # Load model
24     model = load_chat_model(
25         model=configuration.query_model,
26         host=configuration.ollama_host
27     )
28
29     # Create prompt
30     messages = [
31         ('system', summary_system_prompt)
32     ] + messages_to_summarize
33
34
35     response = model.invoke(messages, config)
36
37     return {"summary": response.content}

```

Listing 2.7: Summarize conversation node

Determine whether conversation summarization should occur. This function implements the logic for deciding when to summarize the conversation based on message count. It triggers sum-

marization every 6 messages to keep conversation context manageable while preserving important information.

```

1 def should_summarize(
2     state: RetrievalState, *, config: RunnableConfig
3 ) -> str:
4     message_count = len(state.messages)
5     should_trigger = message_count % 6 == 0
6
7     if should_trigger:
8         return "summarize_conversation"
9     else:
10        return END

```

Listing 2.8: Summarize conditional edge

2.4.5 ReportAgent - Structured Document Generation

The ReportAgent generates comprehensive technical reports through a multi-stage pipeline that creates structured, well-organized documents from the knowledge base.

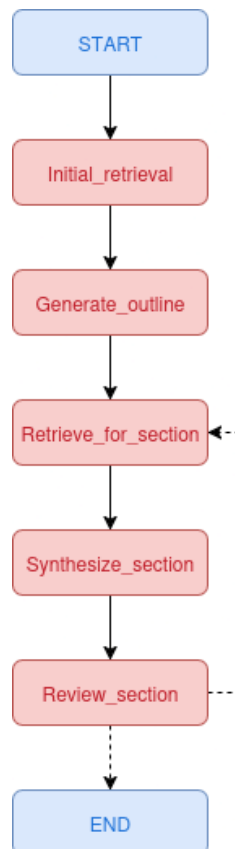


Figure 2.5: Report graph structure

Nodes:

Initial retrieval Retrieve documents based on the input state to create the outline. This function performs the initial document retrieval using the user’s main query to gather relevant

context that will inform the outline generation. It serves as the foundation for understanding what information is available for the report.

```

1 def initial_retrieval(
2     state: ReportState, *, config: RunnableConfig) -> dict[str, list[
3         Document]]:
4
5     configuration = Configuration.from_runnable_config(config)
6
7     # Extract main query from the last message
8     main_query = get_message_text(state.messages[-1])
9
10    # Setup retriever with embedding model
11    with retrieval.make_retriever(
12        embedding_model=load_embedding_model(model=configuration.
13            embedding_model, host=configuration.ollama_host),
14    ) as retriever:
15        # Perform document retrieval
16        response = retriever.invoke(main_query, config)
17
18    return {"retrieved_docs": response}

```

Listing 2.9: Initial retrieval node

Generate outline Generate an outline based on the user query and retrieved context. This function creates a structured outline for the report using the initially retrieved documents as context. It uses a language model with structured output to ensure the outline has the proper format and number of sections specified in the report configuration.

```

1 def generate_outline(
2     state: ReportState, *, config: RunnableConfig) -> dict[str, Any]:
3
4     configuration = Configuration.from_runnable_config(config)
5
6     # Extract query and context information
7     main_query = get_message_text(state.messages[-1])
8     context_text = "\n\n".join([doc.page_content for doc in state.
9         retrieved_docs])
10
11    # Load model with structured output
12    model = load_chat_model(model=configuration.report_model, host=
13        configuration.ollama_host).with_structured_output(Outline)
14
15    # Create prompt with context and requirements
16    system_prompt = prompts.OUTLINE_SYSTEM_PROMPT.format(
17        context = format_docs(state.retrieved_docs),
18        number_of_parts = configuration.number_of_parts
19    )
20    user_prompt = state.messages[-1].content
21    messages = [
22        ("human", combine_prompts(system=system_prompt, user=user_prompt
23    ))
24    ]
25
26    # Generate outline using structured output
27    generated = cast(Outline, model.invoke(messages, config))

```

```

25
26     # Initialize report structure and determine writing style label
27     style_label = "Technical Report" if configuration.writing_style == "
28     technical" else "General Report"
29     report_header = f"{style_label.upper()}\nTITLE: {main_query}\n\n"
30
31     return {
32         "outlines": generated.entries,
33         "current_section_index": 0,
34         "report_header": report_header,
35     }

```

Listing 2.10: Initial retrieval node

Retrieve for section node Retrieve documents for the current section being processed. This function performs targeted document retrieval for the specific section currently being processed. It uses the section title as the query to find the most relevant documents for that particular part of the report.

```

1 def retrieve_for_section(
2     state: ReportState, *, config: RunnableConfig) -> dict[str, list[
3         Document]]:
4
5     # Validate section state
6     if not state.outlines:
7         return {"retrieved_docs": []}
8
9     if state.current_section_index >= len(state.outlines):
10         return {"retrieved_docs": []}
11
12     configuration = Configuration.from_runnable_config(config)
13
14     current_section = state.outlines[state.current_section_index]
15
16     # Setup retriever and perform document retrieval
17     with retrieval.make_retriever(
18         embedding_model=load_embedding_model(model=configuration.
19         embedding_model, host=configuration.ollama_host),
20     ) as retriever:
21         # Retrieve documents using section title as query
22         response = retriever.invoke(current_section, config)
23
24     return {"retrieved_docs": response}

```

Listing 2.11: Retrieve for section node

Synthesize section node Synthesize raw section content from retrieved documents. This function generates the initial content for the current report section using the documents retrieved specifically for that section. It applies the appropriate writing style (technical or general) and creates comprehensive content that addresses the section topic using the available context.

```

1 def synthesize_section(
2     state: ReportState, *, config: RunnableConfig) -> dict[str, Any]:
3
4     # Validate section state

```



```

5     if not state.outlines:
6         return {"raw_section_content": ""}
7
8     if state.current_section_index >= len(state.outlines):
9         return {"raw_section_content": ""}
10
11    configuration = Configuration.from_runnable_config(config)
12
13    current_section = state.outlines[state.current_section_index]
14    main_query = state.messages[-1].content
15
16    # Load content generation model
17    model = load_chat_model(model=configuration.report_model, host=
18        configuration.ollama_host)
19
20    # Select appropriate prompt based on writing style
21    prompt = prompts.TECHNICAL_SECTION_SYSTEM_PROMPT if configuration.
22        writing_style == "technical" else prompts.
23        GENERAL_SECTION_SYSTEM_PROMPT
24
25    # Format prompt with context and section information
26    formatted_prompt = prompt.format(
27        context = format_docs(state.retrieved_docs),
28        current_section = current_section,
29        main_query = main_query
30    )
31    messages = [
32        ("human", formatted_prompt)
33    ]
34
35    # Generate section content
36    response = model.invoke(messages, config)
37    synthesized_content = get_message_text(response).strip()
38
39    return {"raw_section_content": synthesized_content}

```

Listing 2.12: Synthesize section node

Review section node This function takes the raw content generated in `synthesize_section` and applies a review and polishing process to improve quality, coherence, and alignment with the overall report objectives. It creates the final section content and advances the processing to the next section. It creates the final section content and advances the processing to the next section.

```

1 def review_section(
2     state: ReportState, *, config: RunnableConfig) -> dict[str, Any]:
3
4     # Validate section state
5     if not state.outlines:
6         return {}
7
8     if state.current_section_index >= len(state.outlines):
9         return {}
10
11    configuration = Configuration.from_runnable_config(config)

```

```

12     current_section = state.outlines[state.current_section_index]
13     main_query = state.messages[-1].content
14
15
16     # Load review model
17     model = load_chat_model(model=configuration.report_model, host=
18         configuration.ollama_host)
19
20     # Format review prompt with context
21     prompt = prompts.REVIEW_SYSTEM_PROMPT.format(
22         main_query = main_query,
23         current_section = current_section,
24         draft_section = state.raw_section_content,
25     )
26     messages = [
27         ("human", prompt)
28     ]
29
30     # Generate polished content
31     response = model.invoke(messages, config)
32     polished_content = get_message_text(response).strip()
33
34     # Create final section structure
35     final_section = {"title": current_section, "content":
36         polished_content}
37     next_index = state.current_section_index + 1
38
39     return {
40         "report": [final_section],
41         "current_section_index": next_index
42     }

```

Listing 2.13: Review section node

The following function implements the control logic for the iterative section processing workflow. It checks if there are more sections to process based on the current section index and the total number of outlined sections.

```

1 def should_continue(state: ReportState, *, config: RunnableConfig):
2
3     if not state.outlines:
4         return END
5
6     if current_index >= len(state.outlines):
7         return END
8     else:
9         return "retrieve_for_section"

```

Listing 2.14: Should continue conditional edge

Writing Style Support:

- **Technical Style:** Detailed technical analysis with precise values and specifications
- **General Style:** Accessible explanations suitable for broader audiences

2.4.6 IndexAgent - Document Ingestion Pipeline

The IndexAgent manages the document ingestion pipeline, transforming PDF documents into searchable vector representations stored in the ChromaDB vectorstore.

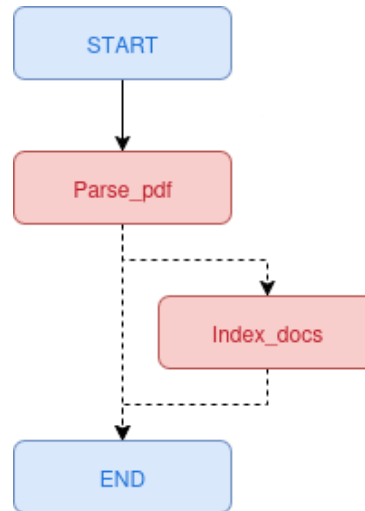


Figure 2.6: Index graph structure

Nodes:

Parse pdf node This function scans a directory for PDF files, loads them using PyMuPDFLoader, splits the content into smaller chunks using RecursiveCharacterTextSplitter, and returns the processed documents. It filters out already processed files to avoid duplicates.

```

1 def parse_pdfs(
2     state: InputIndexState, *, config: Optional[RunnableConfig] = None
3 ) -> dict[str, str]:
4
5     configuration = Configuration.from_runnable_config(config)
6
7     path = Path(state.path)
8     if not path.is_dir():
9         raise FileNotFoundError(f"Directory not found: {state.path}")
10
11     documents = []
12
13     # Get new PDF files (excluding already processed ones)
14     pdf_files = remove_duplicates(
15         base=retrieval.get_existing_documents(),
16         new=[str(p) for p in list(path.glob("*.pdf"))]
17     )
18
19     if not pdf_files:
20         return {"docs": documents}
21
22     #embeddings = load_embedding_model(model=Configuration.
23         embedding_model, host=Configuration.ollama_host)
24
25     text_splitter = RecursiveCharacterTextSplitter(

```

```

25     chunk_size=4000,
26     chunk_overlap=200,
27     length_function=len,
28     is_separator_regex=False
29 )
30
31 # Process each PDF file
32 for pdf_file in tqdm(pdf_files, desc="Loading files..."):
33     # Load the file into a Document object
34     if configuration.ocr:
35         logger.debug("using server parser")
36         loader = VisionLoader(
37             file_path=str(pdf_file),
38             mode = 'single',
39             ollama_base_url= configuration.ollama_host,
40             ollama_model=configuration.vision_model,
41         )
42     else:
43         logger.debug("distant client not found, falling back to
44             local parser")
45         loader = PyMuPDFLoader(
46             file_path=str(pdf_file),
47             extract_tables='markdown',
48             mode= "single"
49         )
50
51     # Split the Document content into smaller chunks
52     document = text_splitter.split_documents(loader.load())
53     #ensure metadata
54     document[0].metadata={"source": pdf_file}
55     # Add them to the list of Documents
56     documents.extend(document)
57
58     return {"docs": documents}

```

Listing 2.15: Parse pdf node

The following function is a conditional edge that skips the `index_docs` node in case there are no documents to index.

```

1  should_index(state: IndexState, *, config: RunnableConfig) -> str:
2
3      if not state.docs:
4          logger.info("No documents to index, ending workflow")
5          return END
6
7      return "index_docs"

```

Listing 2.16: Should index conditional edge

Index documents node This function takes documents from the state, processes them in batches, and adds them to the configured retriever's index using the specified embedding model. After successful indexing, it signals for documents to be removed from the state.

```

1  def index_docs(

```

```

2     state: IndexState, *, config: Optional[RunnableConfig] = None
3 ) -> dict[str, str]:
4
5     configuration = Configuration.from_runnable_config(config)
6
7     # Prepare document batches
8     documents_batch = make_batch(obj=state.docs, size= 20)
9
10    # Index documents using the retriever
11    with retrieval.make_retriever(
12        embedding_model=load_embedding_model(model=configuration.
13            embedding_model)
14    ) as retriever:
15
16        for i, batch in enumerate(tqdm(documents_batch, desc="Adding
17            document batch..."), 1):
18            retriever.add_documents(batch)
19
20    return {"docs": "delete"} # To clear the value of the current thread

```

Listing 2.17: Index doc node

Chunking Strategy

The system supports two chunking approaches:

- **Recursive Character Splitting:** Fixed-size chunks with overlap (4000 characters, 200 overlap)
- **Semantic Chunking:** Meaning-based division using embedding models (commented out for performance)

```

1 text_splitter = RecursiveCharacterTextSplitter(
2     chunk_size=4000,
3     chunk_overlap=200,
4     length_function=len,
5     is_separator_regex=False
6 )

```

Listing 2.18: Text splitting configuration

Batch Processing: Documents are processed in batches of 20 to limit memory usage and provide progress feedback:

```

1 documents_batch = make_batch(obj=state.docs, size=20)
2 for batch in tqdm(documents_batch, desc="Adding document batch..."):
3     retriever.add_documents(batch)

```

Listing 2.19: Batch processing implementation

2.5 Retrieval System

2.5.1 Vector Store Management

Homer utilizes ChromaDB as its primary vector store, providing persistent storage with cosine similarity search capabilities. The retrieval system implements a context manager pattern for resource management:

```

1 @contextmanager
2 def make_retriever(embedding_model: Embeddings, **kwargs):
3     vector_store = Chroma(
4         collection_name=_COLLECTION,
5         collection_metadata={"hnsw:space": "cosine"},
6         embedding_function=embedding_model,
7         persist_directory=VECTORSTORE_DIR,
8     )
9     yield vector_store.as_retriever(
10         search_type="similarity_score_threshold",
11         search_kwargs={"k": 5, "score_threshold": 0.4}
12     )

```

Listing 2.20: Vector store retriever implementation

2.5.2 Search Configuration

The retrieval system supports multiple search strategies:

- **Similarity Search:** Standard cosine similarity with configurable top-k results
- **Similarity Score Threshold:** Filters results below a minimum relevance threshold
- **Maximum Marginal Relevance (MMR):** Balances relevance and diversity in results

Default Parameters:

- **Top-k:** 5 documents per query
- **Score Threshold:** 0.4 minimum similarity
- **Collection:** "HOMER" with cosine similarity space

2.5.3 Document Lifecycle Management

The system provides comprehensive document lifecycle management through utility functions:

```

1 def get_existing_documents() -> list[str]:
2     """Returns list of all indexed document sources"""
3
4 def delete_documents(docs: str | list[str]):
5     """Removes documents by source filename"""

```

Listing 2.21: Document management functions

This enables users to manage their document collection, remove outdated content, and track indexing status through the web interface.

Get existing documents. The function `get_existing_documents` is used to list all the documents currently indexed in the ChromaDB vector store. More specifically, it returns the unique source file names associated with the stored documents. These source names are extracted from the metadata that is attached to each document.

This part of the code connects to the ChromaDB client and retrieves (or create if it does not exist) the "HOMER" collection.

```
1 client = get_chroma_client()
2 collection = client.get_or_create_collection(name=_COLLECTION, metadata=_COLLECTION_METADATA)
```

This line fetches all documents stored in the collection but includes only their metadata, not the actual content or embeddings

```
1 results = collection.get(include=["metadatas"])
```

This block loops through the metadata and collects all values from the "source" field, which usually corresponds to the original file name. A set is used to ensure uniqueness (no duplicates).

```
1 sources = set()
2 for metadata in results["metadatas"]:
3     if metadata and "source" in metadata.keys():
4         sources.add(metadata["source"])
```

Delete documents. The function `delete_documents` is used to remove one or multiple documents from the ChromaDB collection based on their source filename. This is useful for deleting outdated or incorrect files that were previously indexed.

Before deletion, the function ensures that the input is a list, even if the user passed a single string. This provides uniform handling.

```
1 if isinstance(docs, str):
2     docs = [docs]
```

Then, for each source name in the list, the function deletes all documents that have a matching "source" field in their metadata.

```
1 for doc_source in docs:
2     collection.delete(where={"source": doc_source})
```

2.5.4 Experimental Support: Hybrid Retrieval with Milvus

Homer also included an **experimental** backend using **Milvus Standalone**, a high-performance, embedded vector database available on Linux and macOS. Although not yet part of the stable release, this prototype demonstrates how Milvus can support hybrid search capabilities.

Milvus lite does not support hybrid-search and is not available on Windows; hence, ChromaDB was chosen. Milvus would be relevant in the case of a shared database between users, that would be running on a server.

In the example below, we are building a retriever that allows hybrid-search between Dense (HNSW with cosine metric) and BM25 sparse search.

```

1 from langchain_milvus import Milvus, BM25BuiltInFunction
2 from langchain_core.embeddings import Embeddings
3
4 @contextmanager
5 def make_retriever(embedding_model: Embeddings, **kwargs):
6     vector_store = Milvus(
7         embedding_function=embedding_model,
8         builtin_function=BM25BuiltInFunction(), # Enable BM25 for
           sparse vectors
9         vector_field=["dense", "sparse"], # Dense embeddings + sparse
           BM25
10        collection_name=_COLLECTION,
11        connection_args={
12            "uri": VECTORSTORE_DIR, # Local file-based storage like
           ChromaDB
13        },
14        index_params=[
15            {"metric_type": "COSINE", "index_type": "HNSW"}, # Dense
           index
16            {"metric_type": "BM25", "index_type": "AUTOINDEX"} # Sparse
           index
17        ]
18        consistency_level="Strong",
19        drop_old=False, # Don't drop existing collection
20    )
21
22    # Configure retriever with hybrid search and score threshold
23    yield vector_store.as_retriever(
24        search_type="similarity_score_threshold",
25        search_kwargs={
26            "k": 5,
27            "score_threshold": 0.4,
28            # For hybrid search, you can also specify:
29            "ranker_type": "weighted", # Use weighted ranking
30            "ranker_params": {"weights": [0.7, 0.3]}, # Dense: 0.7,
           Sparse: 0.3
31        }
32    )

```

Listing 2.22: Milvus hybrid retriever

Advantages of Milvus.

- **Quick Setup:** Fully embedded—no Docker, no remote server
- **Hybrid Search:** Combines semantic and keyword-based retrieval
- **Scalable:** Designed to scale up to billions of vectors
- **Modular Integration:** Compatible with Homer’s existing architecture

This experimental integration highlights the extensibility of Homer’s architecture. In the future, a parallel `server_retrieve` node could be a possibility, to search for additional context in another knowledge base.

2.6 Error Handling and Resilience

2.6.1 Comprehensive Error Management

Each component implements robust error handling with degradation:

```
1 try:
2     # Main processing logic
3     response = model.invoke(messages, config)
4     logger.info("Operation completed successfully")
5     return {"result": response}
6 except Exception as e:
7     logger.error(f"Error in operation: {str(e)}")
8     # Fallback mechanism
9     return {"result": fallback_response}
```

Listing 2.23: Error handling pattern

2.6.2 Logging Architecture

The system implements structured logging with configurable levels:

```
1 setup_logging("INFO")
2
3 # Usage across components
4 logger = get_logger(__name__)
```

Listing 2.24: Logging implementation

Log Levels and Usage:

- **INFO:** System status, configuration changes, major operations
- **DEBUG:** Detailed operation traces, parameter values, intermediate states
- **WARNING:** Non-fatal issues, fallback activations
- **ERROR:** Failed operations, exceptions with context

2.6.3 Caching Strategy

- **Model Loading:** Models are cached within session state to avoid recompilation
- **Vector Store:** ChromaDB provides built-in caching for embedding lookups
- **Configuration:** Static configuration loading with dynamic host resolution

This architecture ensures optimal performance while maintaining system reliability and user experience quality.

Chapter 3

Vision Parser

3.1 VisionLoader

Homer implements a dual-path document parsing system supporting both traditional text extraction and advanced vision-based processing. The parser architecture is designed around the `VisionLoader` class, which extends LangChain's `BaseLoader` interface.

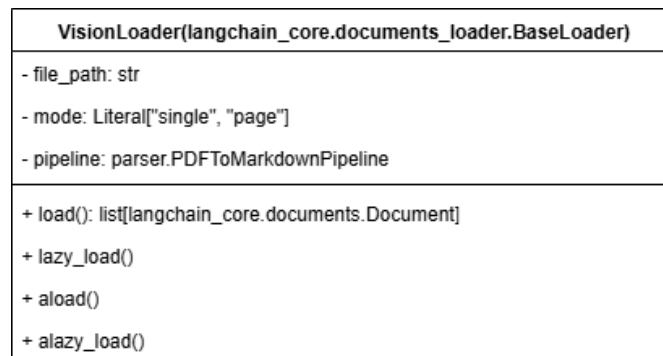


Figure 3.1: Class diagram of the vision loader

The vision parser is a loader based on Langchain's `BaseLoader`. It leverages a custom vision parser pipeline described later.

```
1 class VisionLoader(BaseLoader):
2     def __init__(self, file_path: str, ollama_model: str,
3                   ollama_base_url: str, mode: Literal['single', 'page'] =
4                       'page'):
5         # Configuration and pipeline initialization
6         self.pipeline = parser.PDFToMarkdownPipeline(
7             ollama_model=ollama_model,
8             ollama_base_url=ollama_base_url,
9             enable_validation=False,
10             dpi=400
11         )
```

Listing 3.1: VisionLoader class structure

3.2 Multi-Modal Parsing Architecture

The multi-modal parser uses an Ollama LLVM to extract content from PDF documents, particularly effective for documents containing complex layouts, tables, mathematical formulas, and embedded images. In the current release, we use **Qwen2.5v1**. The implementation uses the following pipeline:

1. **Page Rasterization:** PDF pages are converted to high-resolution PNG images (default 400 DPI)
2. **Vision Model Inference:** Each page image is processed by the configured vision model
3. **Content Extraction:** The model outputs markdown-formatted text preserving structure
4. **Content Validation:** The model output is compared to the **fitz** text extraction to ensure that there are no significant data loss. (Not used in the current release)

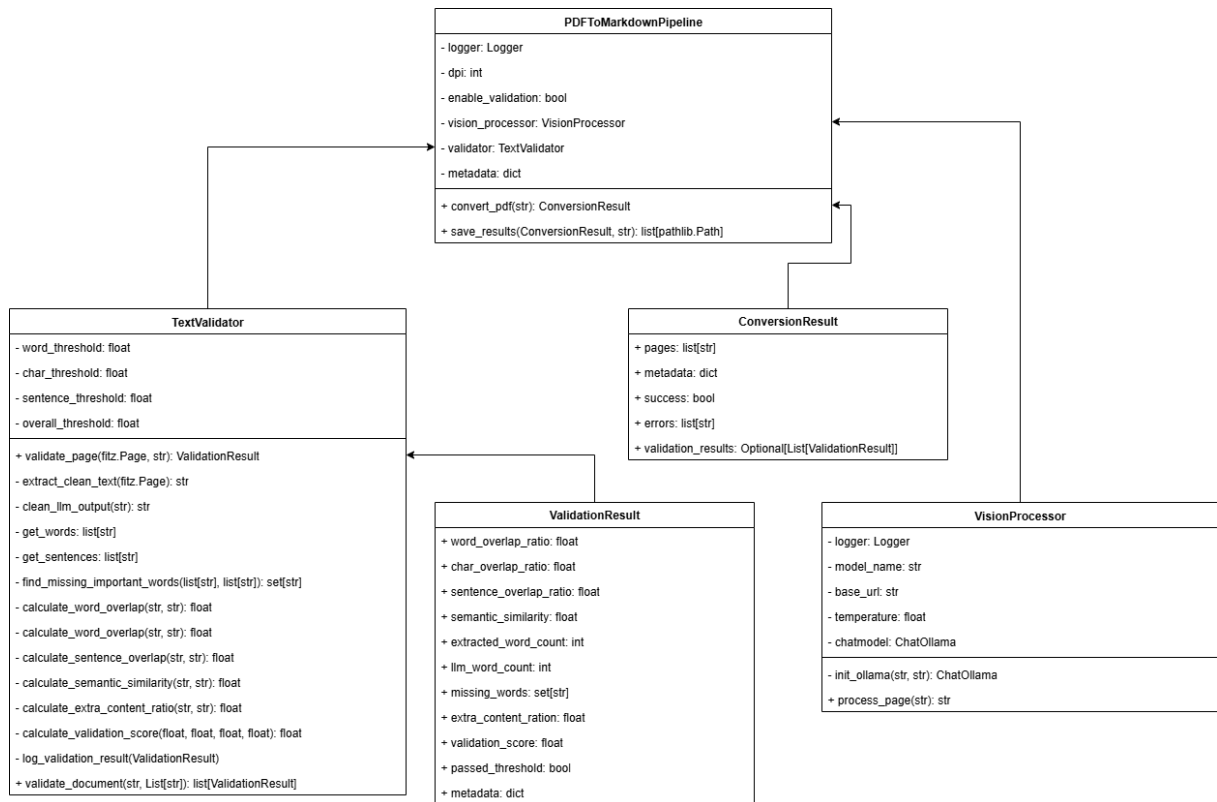


Figure 3.2: Class diagram of the vision parser

The vision parser pipeline has been built using **Claude Sonnet 4** extensively.

3.3 Text Validation System

The parser includes a comprehensive validation system (**src/parser/validation.py**) that compares vision model output against traditional PyMuPDF text extraction. This dual-validation approach ensures accuracy and identifies potential extraction errors.

Validation Metrics:

- **Word Overlap Ratio:** Percentage of words correctly extracted
- **Character Overlap Ratio:** Character-level accuracy using longest common subsequence
- **Sentence Overlap Ratio:** Structural preservation of sentence boundaries
- **Semantic Similarity:** Overall content similarity using normalized edit distance

The validation system calculates a weighted overall score:

$$V_{\text{score}} = 0.4 \times \text{Word Overlap} + 0.2 \times \text{Char Overlap} + 0.3 \times \text{Sentence Overlap} + 0.1 \times \text{Semantic Similarity} \quad (3.1)$$

3.3.1 Fallback Mechanism

When vision parsing is disabled in the runnable config, the system falls back to PyMuPDF text extraction with table markdown support:

```
1 if configuration.ocr:
2     loader = VisionLoader(file_path=str(pdf_file), ...)
3 else:
4     loader = PyMuPDFLoader(
5         file_path=str(pdf_file),
6         extract_tables='markdown',
7         mode="single"
8     )
```

Listing 3.2: Parser fallback mechanism

This ensures consistent functionality across different deployment scenarios while optimizing resource utilization.

Chapter 4

Setup

4.1 Homer Git

To download the project files to your computer, open your terminal or command prompt, and type the following command line that will create a folder with all the files from the project.

```
1 cd your\directory\  
2 git clone https://github.com/flowbrg/homer.git  
3 cd homer
```

4.2 Python

The application requires Python version **3.10** or later. To verify your Python version you can use :

```
1 python --version
```

4.3 Virtual Environment

We highly recommend you to create a **Python virtual environment**. It's an isolated sapce for a project's Python dependencies. It allows us to install and manage packages without affecting other projects or the system-wide Python setup. We use it for **isolation** (keeps project dependencies separate), **consistency** (ensures everyone uses the same packages) and **clean development** (avoids conflicts between different projects). Here is how to setup :

```
1 python -m venv .venv #creates the venv named .venv  
2 .\.venv\Scripts\Activate.ps1 #activates the venv
```

Then you have to install the required packages in the venv using the following line:

```
1 pip install -r requirements.txt
```

4.4 Start the Streamlit app

To launch the HOMER application using Streamlit, simply execute the `main.py` script with the following command:

```
1 python main.py
```

This will start the Streamlit development server and make the application accessible at the default address: `http://localhost:8501` in your web browser

4.5 Server execution

In addition to local execution, Homer also supports running Ollama models on a remote server. This enables faster response times and improves scalability, especially when deploying large language models (LLMs) that require substantial memory and compute resources.

Step 1: Configure the SSH Tunnel (using PuTTY)

- Open PuTTY and enter the IP address or hostname of the remote server.
- In the sidebar, navigate to **Connection** > **SSH** > **Tunnels**.
- In the **Source port**, enter 6780.
- In the **Destination**, enter `localhost:6780`.
- Click **Add**.
- Go back to **Session**, save the session configuration, and click **Open** to start the tunnel.

Step 2 : Server Setup To host an Ollama model on a remote server, follow these steps:

1. **Install Ollama** on the remote machine by following the official installation instructions: <https://ollama.com/download>.
2. **Start the Ollama service** on the remote server using the tunnel port 6780 :

```
1 OLLAMA_HOST=0.0.0.0:6780 ollama serve
```

3. **Pull the desired model** (e.g., llama3) :

```
1 OLLAMA_HOST=0.0.0.0:6780 pull llama3
```

Step 3 : Client Setup On the client side (where the HOMER application runs), you can establish a secure SSH tunnel to forward the remote port to your local machine. This is particularly useful for development environments or when working behind firewalls.

You can change the `OLLAMA_CLIENT` address in the python file `./src/constant.py` where you will find the following line :

```
1 OLLAMA_CLIENT = "http://127.0.0.1:6780/" #change 6780 if you're using an  
other tunnel port
```

This tells the application to send all model-related queries to the remote server rather than to the local instance.

Chapter 5

How to use Homer

5.1 Home page

Once the application is launched (refer to Chapter 4 for details), you will be directed to the **Home** page. A navigation sidebar appears on the left, allowing users to switch between different sections of the application.

To begin, you must upload your documents by navigating to the **Documents** page. This step is essential, as the agent's responses are strictly based on the content of the uploaded documents.

5.2 Documents page

The **Documents** page is where you upload the PDF files you want the application to work with. These documents form the knowledge base that **Homer** will use to answer your questions or generate reports. If you don't upload any documents, the system will have no information to work from.

Step-by-Step Guide

1. **Navigate to the Documents Page**

Use the sidebar on the left to access the **Documents** section.

2. **Enable Server Execution (Recommended)**

In the sidebar, activate the toggle labeled **Server Execution**. This enables the application to process and index documents using a **remote server**, rather than your local machine.

Why this matters: Remote indexing is significantly **faster and more efficient**, especially when working with large documents or using advanced features like vision parsing.

3. **(Optional) Enable Vision Parser**

Below the server execution toggle, you can enable the **Vision Parser**. This feature allows the application to extract content from **scanned documents or images** embedded in PDFs.

Note: This option requires more computing power and works best when server execution is turned on.

4. **Upload Your PDF Files**

Use the upload box to select one or more PDF files from your computer. Once selected, click the **Upload** button to send them to the system. A confirmation message will appear when the process is complete.

5. Manage Your Documents

Uploaded documents are listed below the upload section. Each file can be deleted individually using the **Delete** button next to it.

6. Reset the Document Database (Optional)

Click the **Reset Database** button to remove all uploaded documents and start fresh.

Important Notes

- Only **PDF files** are supported.
- The AI can only respond based on the documents you provide; no external data is used.
- **Enabling Server Execution** is highly recommended for improved speed and reliability.
- Use the **Vision Parser** for scanned or image-based documents.

5.3 Configuration Page

The **Configuration** page allows you to customize how the application behaves when processing and analyzing your documents. This includes setting which models are used for different tasks and specifying the server connection for remote processing.

Purpose of This Page

This page is intended for adjusting key settings that control how documents are indexed and how responses are generated. While default values are already provided, this page gives you flexibility if you want to update the models or connect to a different server.

Step-by-Step Guide

1. Access the Configuration Page

From the sidebar, click on the **Configuration** section to open the settings editor.

2. Edit the Ollama Host (Server URL)

The first field allows you to change the URL of the server (called the **Ollama Host**) that performs document indexing and answering tasks.

A message will appear to confirm whether the URL format is valid.

3. Configure Models

You can modify the models used by the system. There are different models for different modes:

- **Local Reasoning Model:** Used for **Discussion** if you enable **Reasoning model**.
- **Local Standard Model:** Used for **Discussion** if you disable **Reasoning model**.
- **Server Standard Model:** Used for **Discussion** if you disable **Reasoning model** using a server.
- **Server Reasoning Model:** Used for **Discussion** if you enable **Reasoning model** using a server.
- **Vision Model:** Used to parse PDFs that are converted into images (requires server execution).

4. Save Your Settings

After adjusting the fields as needed, click the **Save Configuration** button. The new settings will be applied immediately, and a success message will appear.

5.4 Models Page

The **Model** page allows you to download and manage models from Ollama. These models are required for the application to perform tasks such as document embedding and question answering.

Purpose of This Page

Before the application can run operations, the required models must be downloaded (also known as "pulled") from the [Ollama model hub](#). This page provides a simple interface to do that, whether you're running the application locally or using a remote server.

Step-by-Step Guide

1. Enable Server Execution (Optional)

In the sidebar, you will find a toggle labeled **Server Execution**. If enabled, the model will be downloaded to the remote server. Otherwise, it will be pulled to your local machine.

2. Enter the Model Name

In the main interface, you'll see a text box labeled **Enter the model you want to pull**. Here, you must type the exact name of the model you want to use.

Note: Model names must match exactly as listed on the Ollama website. For example, `llama3`, `gemma:2b`, etc.

3. Pull the Model

After entering the model name, click the **Pull** button. The system will:

- Check if the model already exists.
- If not, download it and show a progress bar.
- Notify you when the process is complete.

4. View Available Models

At the bottom of the page, you can see a list of all models that are currently available on the selected host (either local or server if enabled).

5.5 Discussion Page

The **Discussion** page provides a chat interface where you can interact with the agent based on your uploaded documents. This is the main interface for asking questions and getting detailed answers, including the agent's reasoning process if enabled.

Step-by-Step Guide

1. **Navigate to the Discussion Page**

Use the sidebar to select the **Discussion** section.

2. **(Optional) Enable Server Execution**

In the sidebar, you will find a toggle labeled **Server Execution**.

- When enabled, responses are generated using a remote server, which is generally faster and more powerful.
- When disabled, everything runs locally on your computer.

3. **(Optional) Enable Reasoning Model**

You can toggle between a standard model and a reasoning model.

- The reasoning model includes a visible "thinking" process before producing the final answer.
- The standard model gives a direct answer without explanation.

4. **Enter Your Query**

At the bottom of the chat, type your query into the input field and press **Enter**.

- Your query will immediately appear in the chat window.
- The agent will begin typing its response in real time.

5. **View the Answer and Thinking (If Enabled)**

If you enabled the reasoning model, you will see an expandable section labeled **Show Thinking** containing the model's intermediate steps or explanation before answering.

Other Features

- **Conversation Threads:** The system keeps track of your conversation history within a session.
- **Model Selection Feedback:** The sidebar shows which model is currently being used.
- **Server Connection Status:** The sidebar also displays the active server used to process queries.

5.6 Report Page

The **Report** page allows users to generate structured reports based on the documents they have uploaded.

Purpose of This Page

This interface enables the generation of a detailed report by interacting with the agent. The report is created from the uploaded documents and can be customized according to the user's needs through several sidebar options.

Step-by-Step Guide

1. Navigate to the Report Page

Use the sidebar to access the **Report** section.

2. Configure Report Settings

- **Model**

If you want to change the model that writes the report you should go to `./src/core/configuration.py` and then find this :

```
1 report_model: str = field(  
2     default = "gemma3:4b",    #change this value,  
3 )
```

We highly recommend you not to use thinking model for this task.

- **Number of Report Sections**

Choose how many sections you want in your report. This allows the report to be structured into multiple logical parts.

- **Documents per Section**

Define how many relevant documents (chunks) should be retrieved and analyzed for each part of the report. Increasing this number may lead to more detailed content but less performance (default is 4), while a lower number can make each section more concise.

- **Server Execution (Optional)**

Toggle the **Server Execution** option if you want the report to be generated using a remote server for better performance.

- **Enter Your Report Topic**

In the main area, you will find a text box where you can describe what your report should be about. Be as specific as possible to get better results.

- **Download the Report**

Once the report is generated, you can download the PDF file on your personal computer.

Chapter 6

Next steps - Future development

6.1 .exe file

To further simplify the deployment process, we plan to provide a standalone executable file (**.exe**) that allows users to run the application without requiring a Python environment or any manual setup.

Current Limitation

At this stage, we do not have the necessary permissions to generate or execute such a file on SCK CEN-managed computers. Specifically, creating executables using tools like PyInstaller requires administrative rights or ICT approval, which we currently lack. An official request must be made to ICT to proceed with this compilation on internal machines.

Proposed Solution

The idea is to use the **PyInstaller** library to convert the Python-based Streamlit application into a self-contained executable. This would eliminate the need for users to manually install Python, create a virtual environment, or install packages.

Benefits of a Standalone Executable

- No Python installation required
- No command-line interaction for the user
- Compatible with environments where dependency installation is restricted
- Simplified deployment for end users or clients

Automation of Environment Setup

A shell script has already been prepared that automates the setup:

- Creates and activates a virtual environment
- Installs all packages from `requirements.txt`
- Launches the Streamlit app

Code to Compile

```
1 import os
2 import subprocess
3
4 # Paths
5 project_dir = r"C:\Users\mdlbarre\Documents\homer-main" #update this
6 venv_dir = os.path.join(project_dir, "venv")
7 requirements_path = os.path.join(project_dir, "requirements.txt")
8 main_script = os.path.join(project_dir, "main.py")
9
10 # Create the venv
11 subprocess.call(f'python -m venv "{venv_dir}"', shell=True)
12
13 # Activate the venv and run commands
14 activate_cmd = os.path.join(venv_dir, "Scripts", "activate.bat")
15
16 # Combine all into a single command string
17 commands = f'''
18 call "{activate_cmd}"
19 pip install -r "{requirements_path}"
20 python "{main_script}"
21 '''
22
23 # Run the above in CMD
24 subprocess.call(commands, shell=True)
```

Listing 6.1: setup.py file in the main root to compile

Then you have to run the following command line in the terminal :

```
1 pyinstaller setup.py
```

The .exe file will be in the dist folder.

6.2 Client–Server Setup with L6 and SSH Tunneling

A potential enhancement under consideration is to extend the architecture so that the Streamlit application runs locally on the user’s L6 machine while offloading model inference to a remote server through an SSH tunnel. This setup would allow users to benefit from the performance of the server-side models without deploying or running heavy computations locally.

Proposed Design

- **L6 as a Client:** The application interface would execute on L6, acting as a lightweight frontend.
- **Remote Server with Ollama:** The Ollama service would run on a remote compute server (e.g., a cluster), which hosts large language models.
- **SSH Tunnel:** A secure tunnel would forward API calls from the local machine to the server, making the remote service accessible as if it were local.

Advantages

- Allows users to run the application without requiring local installation of large models or GPUs.
- Offers improved scalability and model performance by using dedicated hardware.
- You can switch off your personal computer while it's running in the background.

6.3 Agentic RAG

Models like DeepSeek R1 and Qwen3 demonstrate sophisticated tool-use capabilities that enable more autonomous operation compared to traditional pipeline-based approaches like the Retrieval Graph currently implemented in our system. These agentic models can dynamically determine when and how to retrieve information based on query context and their reasoning process.

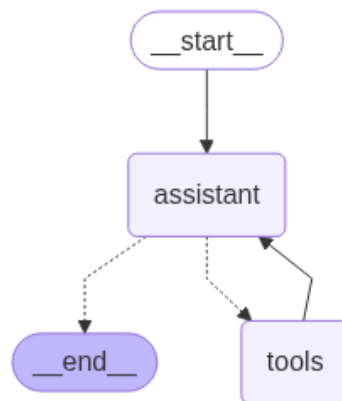


Figure 6.1: Agentic RAG graph

Tool integration

This tool abstraction allows the agentic model to determine when retrieval is necessary, formulate appropriate search queries autonomously, and integrate retrieved information into its reasoning chain.

```

1 @tool
2 def search_for_context(query:str, source: Optional[str], config:
3     RunnableConfig):
4     configuration = Configuration.from_runnable_config(config)
5     embeddings = load_embedding_model(model=configuration.
6         embedding_model)
7     with make_retriever(embedding_model=embeddings) as retriever:
8         results = retriever.invoke(query)
9     if not results:
10         return "No relevant documents found."
11     return format_docs(results)

```

6.4 Future Integration within the SCK CEN Infrastructure

Homer is currently a functional prototype developed to demonstrate AI-assisted document analysis using large language models (LLMs). The next logical step is to integrate this tool within the internal infrastructure of SCK CEN.

Integration Objectives

Integrating Homer into the SCK CEN network will enable:

- Secure and compliant document processing within the protected internal environment.
- Streamlined access to internal resources such as document repositories, identity management systems, and network storage.

Current Limitations

At this stage, Homer has been developed and tested outside the SCK CEN production environment. Integration requires:

- Approval and support from ICT services.
- Validation of security protocols, especially for components involving LLMs and SSH tunnels.
- Adaptation of the current prototype to comply with internal software deployment standards and access control policies.

Continuous Evolution of A.I.

Artificial Intelligence is evolving at a rapid pace, with new models, tools, and capabilities emerging frequently. To ensure that SCK CEN remains at the forefront of this technological shift and can fully leverage AI advancements in a secure and efficient manner, it is essential to establish a dedicated AI team. This team would be responsible for monitoring AI trends, updating internal tools such as HOMER, evaluating new model capabilities, and ensuring safe and ethical deployment aligned with the organization's objectives.

Bibliography

- [1] Florent Bergé & Mathieu de la Barre, *Homer specifications*, SCK CEN, 05/2025
- [2] Raouf Aliouat, Rob Baby, Prabal Deb, Chad Kittel, Ritesh Modi, Ryan Pfalz & Randy Thurman, *Design and develop a RAG solution*, Microsoft Azure, 01/09/2025
- [3] Florent Bergé & Mathieu de la Barre, *Hybrid search for milvus*, SCK CEN, 09/05/2025