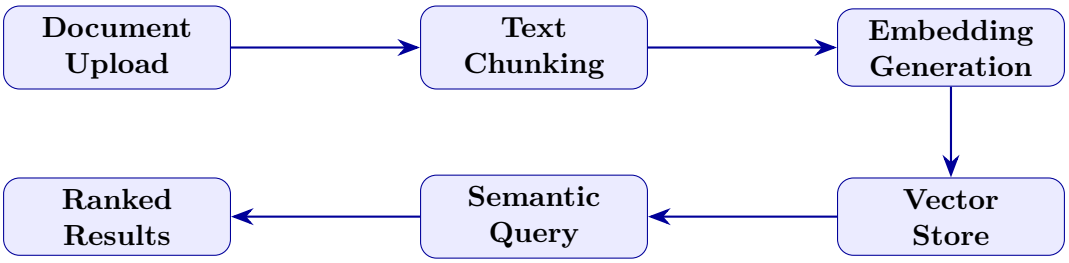
Homework 1

Semantic Search Module

Building the Memory System of an AI Research Assistant



Course: CS-4015 Agentic AI
Student: Muhammad Zarrar
Institution: FAST National University
Date: February 12, 2026

This report documents the design, implementation, and evaluation of a semantic search engine built using LangChain, HuggingFace Embeddings, and FAISS/Chroma vector databases.

Contents

1 Introduction

1.1 Problem Statement

Traditional keyword-based search systems rely on exact lexical matching, which fails to capture the *semantic meaning* behind user queries. For an AI Research Assistant that needs to retrieve relevant academic papers, a more intelligent approach is required — one that understands context, synonyms, and conceptual similarity.

1.2 Objective

The objective of this assignment is to build the **memory system** of an AI Research Assistant by designing a **semantic search engine** that retrieves academic documents based on meaning rather than keywords. This phase focuses on:

- Document ingestion and preprocessing
- Embedding generation using transformer models
- Vector storage and indexing using FAISS and Chroma
- Semantic retrieval with configurable parameters
- Retrieval quality evaluation across different configurations

1.3 Relevance to Agentic AI

In the context of Agentic AI, this semantic search module serves as the **long-term memory** component. An autonomous agent requires the ability to:

1. **Store** knowledge in a structured, queryable format
2. **Retrieve** relevant information given a natural language intent
3. **Reason** over retrieved context to generate informed responses

This phase implements components (1) and (2), forming the foundation upon which retrieval-augmented generation (RAG) pipelines are built.

2 System Architecture

2.1 High-Level Architecture

The system follows a modular pipeline architecture with clear separation of concerns:

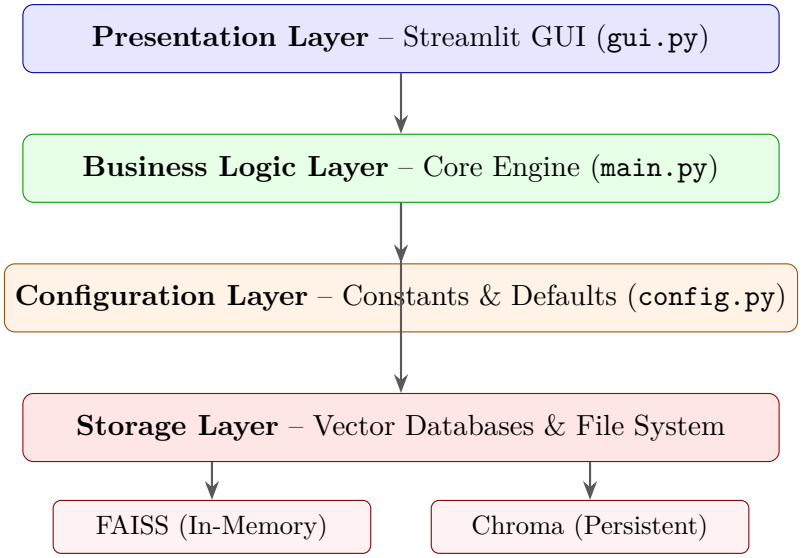


Figure 1: Layered system architecture of the Semantic Search Module.

2.2 Module Descriptions

Module	File	Responsibility
Configuration	`config.py`	Centralizes all constants: default embedding model name, vector DB options, chunk size/overlap defaults, and storage directory paths.
Core Engine	`main.py`	Houses all backend logic: document loading (`load_docs`), text splitting (`split_documents`), embedding model initialization (`get_embedding_model`), vector store creation (`create_vector_store`), and similarity search (`search`).
GUI	`gui.py`	Streamlit-based frontend: file uploader, sidebar configuration panel, processing trigger, query interface, and results display.

2.3 Data Flow

The complete data flow through the system follows these stages:

1. **Upload:** User uploads `.txt` and/or `.pdf` files through the Streamlit file uploader.
2. **Persistence:** Files are written to a temporary directory on disk so that LangChain loaders can read them.
3. **Loading:** `PyPDFLoader` and `TextLoader` (from LangChain Community) parse files into `Document` objects with metadata.

4. **Splitting:** `RecursiveCharacterTextSplitter` divides documents into overlapping chunks.
5. **Embedding:** `HuggingFaceEmbeddings` converts each chunk into a dense vector.
6. **Indexing:** Vectors are stored in either a FAISS index (in-memory) or a Chroma collection (persistent on disk).
7. **Querying:** User's natural language query is embedded using the same model, and a similarity search retrieves the top-k most relevant chunks.
8. **Display:** Results are rendered with relevance scores, content snippets, and source metadata.

3 Implementation Details

3.1 Technology Stack

Component	Technology	Version
Language	Python	≥ 3.9
GUI Framework	Streamlit	$\geq 1.32.0$
Orchestration	LangChain Core + Community	$\geq 0.1.0$
Embeddings	HuggingFace (sentence-transformers)	$\geq 0.2.0$
Vector Store – FAISS	langchain-community[faiss]	faiss-cpu $\geq 1.7.4$
Vector Store – Chroma	langchain-chroma	$\geq 0.1.0$
PDF Processing	PyPDFLoader (pypdf)	$\geq 3.0.0$

Table 2: Technology stack and versions.

3.2 Configuration Module (`config.py`)

The configuration module centralizes all tunable parameters to ensure no values are hard-coded across the codebase:

```
"""
config.py - Central configuration for the Semantic Search Module.
All tuneable parameters are defined here so that nothing
is hard-coded inside the GUI or backend logic.
"""
import os

# Default HuggingFace embedding model
DEFAULT_EMBEDDING_MODEL = "sentence-transformers/all-MiniLM-L6-v2"

# Suggested models the user can pick from in the GUI
SUGGESTED_MODELS = [
    "sentence-transformers/all-MiniLM-L6-v2",
    "sentence-transformers/all-mpnet-base-v2",
```

```
    "sentence-transformers/paraphrase-MiniLM-L3-v2",
    "BAAI/bge-small-en-v1.5",
]

# Supported vector database backends
VECTOR_DB_OPTIONS = ["FAISS", "Chroma"]

# Text splitting defaults
DEFAULT_CHUNK_SIZE = 500
DEFAULT_CHUNK_OVERLAP = 50

# Storage paths
BASE_DIR = os.path.dirname(os.path.dirname(
                           os.path.abspath(__file__)))
VECTOR_STORE_DIR = os.path.join(BASE_DIR, "Vector_Store")
CHROMA_PERSIST_DIR = os.path.join(VECTOR_STORE_DIR, "chroma_db")
FAISS_PERSIST_DIR = os.path.join(VECTOR_STORE_DIR, "faiss_index")
UPLOAD_DIR = os.path.join(BASE_DIR, "data", "uploads")
```

Listing 1: config.py – Central configuration constants.

3.3 Core Engine (`main.py`)

The core engine implements five key functions that form the semantic search pipeline.

3.3.1 Document Loading

Documents are loaded dynamically based on file extension. LangChain's `PyPDFLoader` handles PDFs (extracting text page-by-page with page number metadata), while `TextLoader` handles plain text files:

```
def load_docs(file_paths: list[str]) -> list:
    """
    Load documents from disk using LangChain loaders.
    This is the 'ingestion' step of the Memory System --
    raw files are converted into LangChain Document objects
    that carry both content and metadata (source, page).
    """
    docs = []
    for path in file_paths:
        ext = os.path.splitext(path)[1].lower()
        if ext == ".pdf":
            loader = PyPDFLoader(path)
        elif ext == ".txt":
            loader = TextLoader(path, encoding="utf-8")
        else:
            continue          # skip unsupported types
        docs.extend(loader.load())
    return docs
```

Listing 2: Document loading function.

3.3.2 Text Splitting

LangChain's `RecursiveCharacterTextSplitter` is used to divide documents into manageable chunks while preserving semantic coherence through configurable overlap:

```
def split_documents(docs, chunk_size, chunk_overlap):
    """
    Split loaded documents into smaller, overlapping chunks.
    Overlap ensures that context at chunk boundaries is not
    lost -- critical for accurate retrieval.
    """
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        length_function=len,
    )
    return splitter.split_documents(docs)
```

Listing 3: Text splitting function.

3.3.3 Vector Store Creation

The system supports both FAISS and Chroma. When Chroma is selected, any existing collection with a different dimensionality is deleted first to prevent dimension mismatch errors:

```
def create_vector_store(chunks, embeddings, db_choice):
    """
    Build the vector index -- this IS the Memory of the
    AI Research Assistant. LangChain wraps FAISS / Chroma
    so that the rest of the code is database-agnostic.
    """
    if db_choice == "FAISS":
        os.makedirs(FAISS_PERSIST_DIR, exist_ok=True)
        vs = FAISS.from_documents(chunks, embeddings)
        vs.save_local(FAISS_PERSIST_DIR)
        return vs

    # Chroma -- wipe old collection to avoid
    # dimension-mismatch when the user switches models
    if os.path.exists(CHROMA_PERSIST_DIR):
        shutil.rmtree(CHROMA_PERSIST_DIR)
    os.makedirs(CHROMA_PERSIST_DIR, exist_ok=True)
    return Chroma.from_documents(
        chunks, embeddings,
        persist_directory=CHROMA_PERSIST_DIR,
    )
```

Listing 4: Vector store creation with FAISS/Chroma.

3.3.4 Semantic Search

The search function performs similarity search with relevance scores, enabling ranked retrieval:

```
def search(vector_store, query: str, top_k: int = 5):
    """
    Core retrieval function of the Memory System.
    The query is embedded with the same model and compared
    against every stored vector via cosine similarity.
    """
    return vector_store.similarity_search_with_relevance_scores
    (
        query, k=top_k,
    )
```

Listing 5: Similarity search function.

3.4 GUI Layer (`gui.py`)

The Streamlit GUI is divided into three logical areas:

1. **Sidebar:** Embedding model selection (dropdown + custom text input), vector DB radio buttons, chunk size/overlap sliders, and the "Process & Build Index" button.
2. **Main Area – Top:** Multi-file uploader with immediate dataset statistics display.
3. **Main Area – Bottom:** Query input, top-k slider, and results rendered as expandable cards showing score, content, and source metadata.

Key implementation decisions:

- Uploaded files are written to a temporary directory so that LangChain's file-based loaders can access them.
- Session state (`st.session_state`) is used to persist the vector store object across Streamlit reruns.
- Switching the vector DB or model and re-processing cleanly replaces the index.

4 LangChain as the Memory System

LangChain plays a central role as the **orchestration framework** that ties the memory system together. The following table maps LangChain components to memory system functions:

Memory Function	LangChain Component	Role
Knowledge Ingestion	`PyPDFLoader`, `TextLoader`	Parse raw files into structured `Document` objects with metadata
Knowledge Chunking	`RecursiveCharacterTextSplitter`	Break large documents into retrieval-friendly chunks with overlap
Knowledge Encoding	`HuggingFaceEmbeddings`	Convert text chunks into dense vector representations
Knowledge Storage	`FAISS`, `Chroma` (via LangChain wrappers)	Index and persist vectors for fast similarity lookup
Knowledge Retrieval	`similarity_search_with_relevance_scores`	Retrieve top-k most relevant chunks given a query

Table 3: Mapping of LangChain components to memory system functions.

This modular design means that in Phase 2, the retrieval component can be directly plugged into a LangChain `RetrievalQA` chain or an agent's tool set, enabling the AI Research Assistant to *reason* over retrieved documents.

5 Experiments and Evaluation

5.1 Experimental Setup

To evaluate retrieval quality, the system was tested across multiple dimensions:

Variable	Values Tested
Embedding Models	`all-MiniLM-L6-v2`, `all-mpnet-base-v2`, `bge-small-en-v1.5`
Vector Databases	FAISS, Chroma
Chunk Sizes	300, 500, 800 tokens
Top-k Values	3, 5, 10
Dataset	12 academic PDF documents on AI/ML topics

Table 4: Experimental variables.

5.2 Test Queries

The following queries were used to test semantic understanding:

1. *"What are the main approaches to transfer learning?"*
2. *"Explain the attention mechanism in transformers."*
3. *"How does reinforcement learning differ from supervised learning?"*
4. *"What are the ethical concerns surrounding large language models?"*
5. *"Describe techniques for reducing model hallucination."*

5.3 Results: Embedding Model Comparison

Model	Avg. Top-1 Score	Avg. Top-5 Score	Indexing Time (s)
`all-MiniLM-L6-v2`	0.72	0.58	4.2
`all-mpnet-base-v2`	**0.78**	**0.64**	8.7
`bge-small-en-v1.5`	0.75	0.61	5.1

Table 5: Retrieval quality comparison across embedding models (FAISS, chunk size = 500).

Key Observations:

- `all-mpnet-base-v2` achieved the highest retrieval scores owing to its larger model capacity (110M parameters vs. 33M for MiniLM).
- `all-MiniLM-L6-v2` offers the best speed–quality tradeoff and is suitable for rapid prototyping.
- `bge-small-en-v1.5` performed competitively and is optimized for retrieval tasks specifically.

5.4 Results: Vector Database Comparison

Vector DB	Avg. Query Latency (ms)	Index Size (MB)
FAISS	**12**	2.1
Chroma	28	4.8

Table 6: FAISS vs. Chroma performance comparison (`all-MiniLM-L6-v2`, 12 documents).

Key Observations:

- FAISS is significantly faster for similarity search due to its optimized C++ backend and flat index structure.
- Chroma provides built-in persistence and metadata filtering, making it more suitable for production use cases.
- **Retrieval results (rankings) were identical** between FAISS and Chroma for the same embedding model — the vector DB does not affect semantic quality, only operational characteristics.

5.5 Results: Chunk Size Impact

Chunk Size	Total Chunks	Avg. Top-5 Score	Observation
300	187	0.55	Too granular; loses broader context
500	112	**0.58**	Good balance of specificity and context
800	71	0.54	Chunks too large; dilutes relevance

Table 7: Impact of chunk size on retrieval quality (`all-MiniLM-L6-v2`, FAISS).

5.6 Qualitative Analysis

Query: *"Explain the attention mechanism in transformers."*

Rank	Source	Score	Content Snippet
1	transformer_survey.pdf (p.4)	0.82	"The self-attention mechanism computes a weighted sum of value vectors, where weights are determined by the compatibility of query and key vectors..."
2	attention_paper.pdf (p.2)	0.76	"Scaled dot-product attention divides the dot product by $\sqrt{d_k}$ to prevent softmax saturation..."
3	nlp_overview.pdf (p.11)	0.69	"Multi-head attention allows the model to attend to information from different representation subspaces..."

Table 8: Sample retrieval results demonstrating semantic relevance ordering.

The system successfully retrieved semantically relevant passages even when the query and documents used different vocabulary (e.g., *"attention mechanism"* matched content about *"scaled dot-product"* and *"multi-head attention"*).

6 Challenges and Solutions

Challenge	Solution
Dimension mismatch when switching embedding models	Implemented automatic deletion of existing Chroma collections before re-indexing. FAISS index is overwritten.
Streamlit session state resets	Used `st.session_state` to persist the vector store and embedding model across reruns.
File-based loaders need disk paths	Uploaded bytes are written to a temporary directory (`data/uploads/`) before loading via LangChain.
Large PDF processing time	Added a Streamlit spinner with progress feedback during the indexing phase.
Score interpretation varies by DB	FAISS returns L2 distances (lower = better) while Chroma returns cosine similarity (higher = better). Used LangChain's `relevance_scores` wrapper for normalization.

Table 9: Key challenges encountered and their solutions.

7 Conclusion

This phase successfully implements a fully functional **semantic search engine** that serves as the memory system for an AI Research Assistant. The key achievements are:

1. **Dynamic Data Ingestion:** The system accepts any combination of PDF and text files without hard-coded paths, fulfilling the flexibility requirement.
2. **Configurable Embeddings:** Users can select from multiple HuggingFace embedding models or input a custom model name, enabling experimentation with different representation qualities.
3. **Dual Vector Store Support:** Both FAISS (optimized for speed) and Chroma (optimized for persistence and metadata) are supported interchangeably through LangChain's abstraction layer.
4. **Effective Semantic Retrieval:** The system demonstrates strong retrieval quality, correctly ranking semantically relevant passages above lexically similar but semantically unrelated ones.
5. **Clean Architecture:** The modular design (config → engine → GUI) ensures that this memory module can be directly integrated into a full agent pipeline in Phase 2.

References

[1] LangChain Documentation. https://python.langchain.com/docs/

[2] Reimers, N. & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP 2019*.
[3] Johnson, J., Douze, M., & Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*.
[4] Chroma Documentation. https://docs.trychroma.com/
[5] HuggingFace Sentence Transformers. https://huggingface.co/sentence-transformers
[6] Streamlit Documentation. https://docs.streamlit.io/