Conversational Chatbot (NLP)

Advanced NLP System for Academic Document Retrieval

Students: Tayma Mokrani , Ilies Mraihi

Email: tayma.mokrani@fsb.ucar.tn , ilies.mraihi@fsb.ucar.tn

Course: CI3 - MLOPS
Instructor: Besma Guesmi
FSB - Computer Science Department

January 10, 2025

GitHub Repository: `https://github.com/TaymaMokrani/Conversational_Chatbot_MLOPS`

Contents

1 Summary

Work overview

This project presents an AI-powered chatbot that helps users easily find information from institutional documents using natural language.

1.1 Key Achievements

- **Document Processing**: Ingestion and preprocessing of various PDF files
- **Intelligent Chunking**: Text segmentation that preserves context
- **Hybrid Retrieval**: Integration of BM25 and vector search with RRF
- **Chat Interface**: Web interface with conversation history
- **Performance Optimization**: Smart caching that reduces processing time
- **Deployment Ready**: Dockerized application

2 System Architecture

Architecture Highlights

The system is designed like a microservices architecture, with separate components that can be developed and modified independently, while still working together .

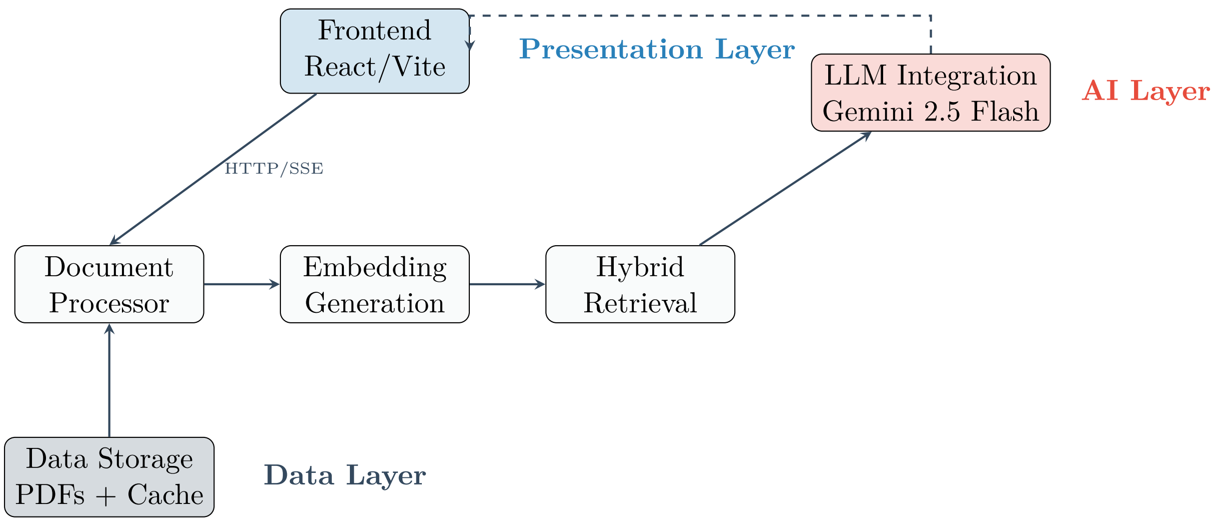


Figure 1: System architecture showing component interactions and data flow

2.1 Directory Structure

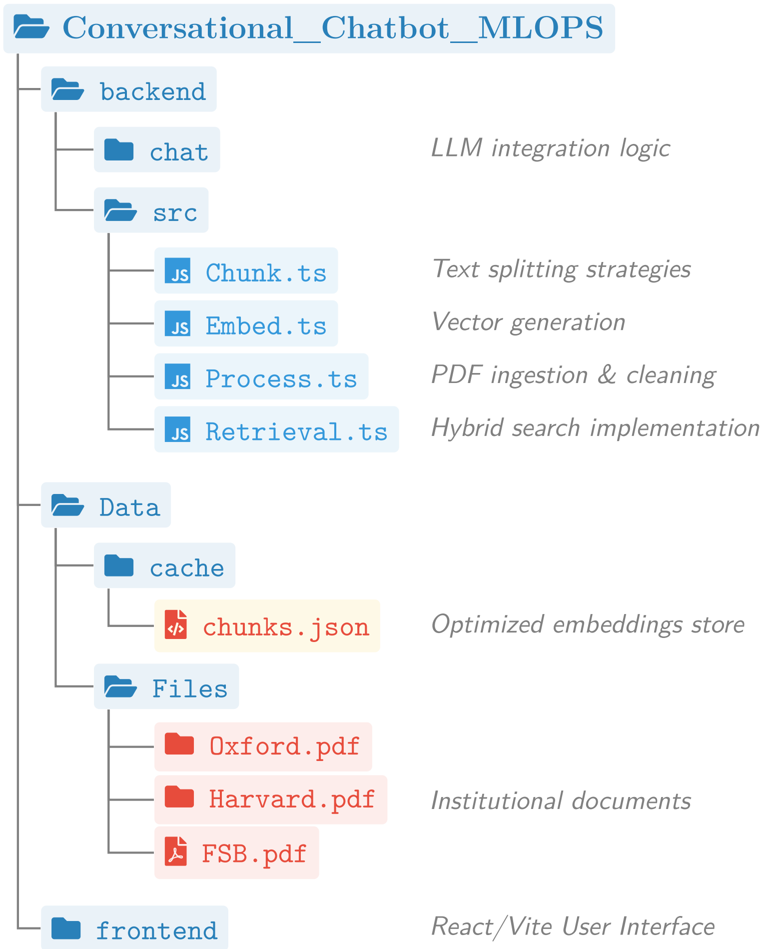


Figure 2: Visual representation of the project directory structure

2.2 Technology Stack

Component	**Technology**	**Purpose**
Backend Runtime	Bun 1.3.5	High-performance JS runtime
Frontend	React 19 + Vite	UI development
LLM	Google Gemini 2.5	Advanced reasoning
Embeddings	all-MiniLM-L6-v2	semantic vectors
Vector Ops	Custom implementation	Cosine similarity
Text Processing	pdf-parse	PDF extraction
API Framework	Elysia.js	Fast API server
Deployment	Docker	Containerization

Table 1: Technology stack used in the project

2.3 Data Flow

The system handles user queries through this pipeline:

1. **Query Reception**: User input submitted via the chat interface
2. **Embedding Generation**: Conversion to semantic vector representations
3. **Hybrid Retrieval**: Parallel BM25 and vector-based search
4. **Result Fusion**: Combining rankings using RRF
5. **Context Assembly**: Selecting top-k results
6. **LLM Synthesis**: Generating a natural language response
7. **Response Delivery**: Returning the formatted answer to the user

3 Implementation Details

3.1 Document Ingestion Pipeline

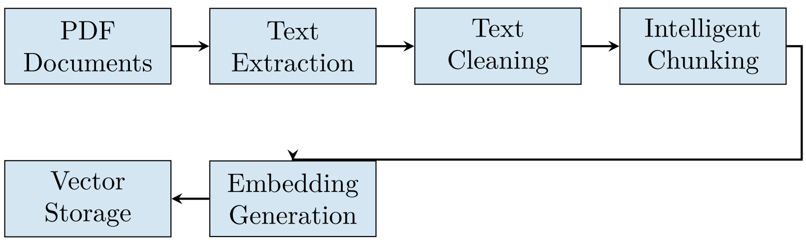


Figure 3: Document processing pipeline

3.1.1 PDF Processing

The process includes:

- **File Ingestion**: PDFs are loaded and converted into a buffer for parsing.
- **Text Extraction**: Raw text is extracted using a PDF parser.
- **Text Cleaning**:
  - Removal of URLs.
  - Elimination of control and non-printable characters.
  - Normalization of whitespace to standardize spacing.
  - Preservation of paragraph breaks.

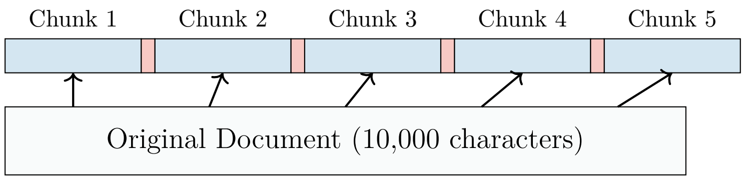
This pipeline ensures that text is clean, structured, and ready for tasks such as chunking, embedding generation, and retrieval.

3.2 Intelligent Chunking Strategy

The system implements an adaptive text chunking mechanism to efficiently segment documents :

- **Adaptive Segmentation**: Text is split into chunks based on sizes and overlaps.
- **Language**: Markdown-specific splitting that preserves structur.
- **Embedding**: Each chunk is converted into a vector and assigned a unique ID.
- **Metadata Tracking**: Optional source information is attached to each chunk.

The following figure demonstrates the chunking and embedding process:



Chunking strategy with 1000-character chunks and 200-character overlap

Figure 4: Visualization of the chunking strategy showing overlaps

3.3 Embedding Generation

- **Pre-trained Model**: Use the MiniLM model (‘all-MiniLM-L6-v2‘) for efficient and accurate sentence embeddings.
- **Feature Extraction**: transformer-based pipeline to generate embeddings from input text.
- **Pooling and Normalization**: Applies mean pooling and normalization .
- **Chunking**: Each text chunk produced in the previous step is transformed into a fixed-size embedding .

The following TypeScript implementation demonstrates the embedding generation:

```typescript
import { pipeline } from "@xenova/transformers";

const Embedder = await pipeline(
    "feature-extraction",
    "Xenova/all-MiniLM-L6-v2"
);

export const Embed = async (value: string) => {
  const output = await Embedder(value, {
    pooling: "mean",
    normalize: true
  });
  return Array.from(output.data as Float32Array);
};
```

Listing 1: Embedding Generation with MiniLM (Embed.ts)

3.4 Hybrid Retrieval System

To maximize retrieval accuracy, the system combines traditional and vector-based search using a hybrid approach :

- **Parallel Retrieval**: Queries are executed simultaneously on a BM25 index and a vector store.
- **Reciprocal Rank Fusion (RRF)**: Results from both methods are combined using RRF to produce a single ranked list.
- **Top-K Selection**: The system returns the top-k documents along with their relevance scores.

3.5 Smart Caching Mechanism

To improve performance the system implements an intelligent caching mechanism :

- **File State Tracking**: The system generates a hash based on file size and modification time for all PDF documents in the directory.
- **Change Detection**: By comparing the generated hash with previous runs, the system can detect updated or new files.
- **Cache Reuse**: Only new or modified files are reprocessed, while unchanged data is retrieved from the cache, significantly reducing processing time.

```
function generateFilesHash(filesDir: string, pdfFiles: string[]): string {
    const fileInfos = pdfFiles.map(file => {
        const filePath = path.join(filesDir, file);
        const stats = fs.statSync(filePath);
        return `${file}:${stats.size}:${stats.mtimeMs}`;
    }).sort().join("|");

    let hash = 0;
    for (let i = 0; i < fileInfos.length; i++) {
        const char = fileInfos.charCodeAt(i);
        hash = ((hash << 5) - hash) + char;
        hash = hash & hash;
    }
    return hash.toString(16);
}
```

Listing 2: Intelligent Cache Implementation

3.6 Architectural Decisions Summary

Bun Runtime	**Hybrid Retrieval**	**Google Gemini 2.5 Flash**
Fast, built-in TypeScript , native tools and hot reload	Combines keyword + semantic search , robust RRF fusion	Multilingual, fast, cost-effective , supports tool calling

Table 2: Compact summary of key architectural decisions.

3.7 Configuration Options

Parameter	**Type**	**Default**	**Description**
chunkSize	number	1000	Characters per text chunk
chunkOverlap	number	200	Overlap between chunks
topK	number	5	Number of documents to retrieve
rrfK	number	60	RRF smoothing constant
maxTokens	number	4096	Maximum LLM response tokens
temperature	number	0.7	LLM creativity/temperature

4 MLOps-Oriented Configuration Management

To integrate MLOps principles into our system, we adopted a **configuration versioning strategy** that separates model-related settings from data processing parameters.

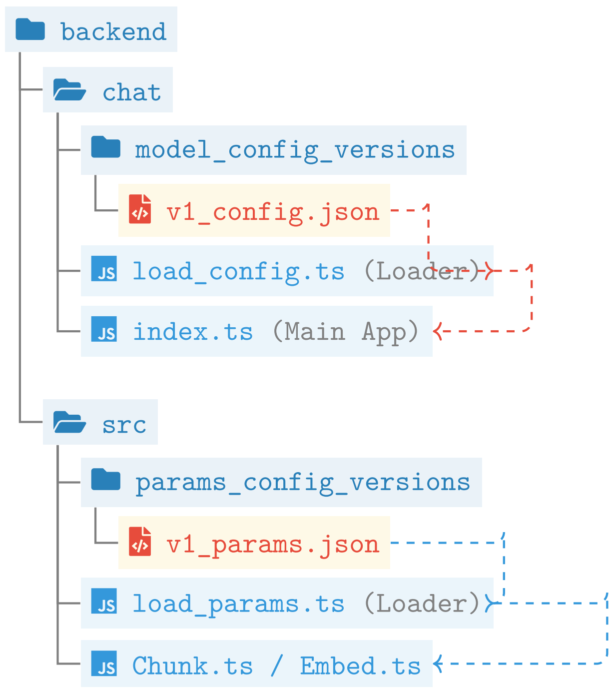


Figure 5: Project directory structure highlighting the separation of configuration versions (JSON) from application logic (TypeScript).

4.1 Model Configuration Versioning

Located in `backend/chat/model_config_versions`, this module controls the LLM behavior. A loader script (`load_config.ts`) dynamically injects these settings, allowing us to swap models or prompt strategies without changing the application code.

```json
{
  "model": {
    "provider": "openrouter",
    "name": "google/gemini-2.5-flash",
    "maxSteps": 5
  },
  "rag": {
    "topK": 5
  },
  "systemPrompt": "You are a helpful assistant... ALWAYS use the RAG
     tool when the user asks about specific details... If information
     is not found, let the user know."
}
```

Listing 3: Example Model Configuration (v1_config.json)

4.2 Pipeline Parameter Versioning

Located in `backend/src/params_config_versions`, this module controls the Retrieval pipeline. This allows for testing of chunking strategy.

```json
{
  "chunking": {
    "chunkSize": 1000,
    "overlapSize": 200
  },
  "embedding": {
    "modelName": "Xenova/all-MiniLM-L6-v2",
    "pooling": "mean",
    "normalize": true
  },
  "processing": {
    "removeUrls": true,
    "normalizeWhitespace": true,
    "preserveNewlines": true
  },
  "retrieval": {
    "topK": 5,
    "bm25": { "k1": 1.5, "b": 0.75 },
    "hybrid": { "rrfK": 60 }
  }
}
```

Listing 4: Example Pipeline Parameters (v1_params.json)

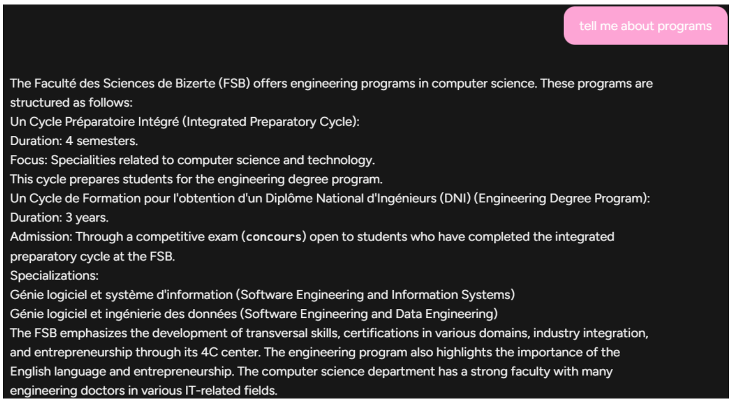
4.3 Implementation Constraints and Limitations

While applying MLOps principles improved modularity, certain industrial MLOps components were adapted or omitted due to specific project constraints:

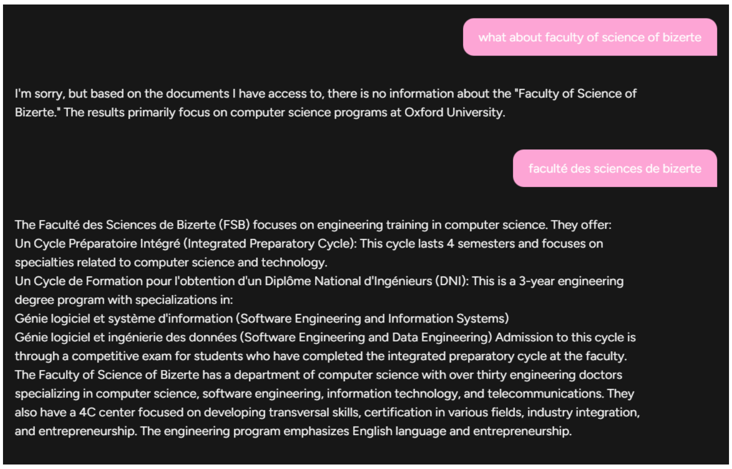
- ⚠ **No Data Versioning (DVC)**: Standard data versioning was not doable because the dataset consists of documents from different universities (FSB, MIT, Harvard, etc.) with no changes applicable. The "data" is treated as a static knowledge base.
- ⚠ **No Training/Test Registry**: As the system relies on pre-trained Large Language Models (Gemini) and pre-trained Embeddings (MiniLM), there is no model training phase. Consequently, we cannot store training metrics (Loss, Accuracy) or maintain a labeled test set for regression testing.

5 Common Failure Modes

1. **Ambiguous Queries**: General questions are too vague. The system often focuses on a single document and ignores others.



2. **Formatting Issues**: Tables or non-standard PDF elem may not be parsed correctly.
3. **Multilingual Institution Names**: The system struggles to recognize institution names written in different languages or if there are small typos.



4. **Incomplete Context for Broad Questions**: Broad questions often result in the system focusing on a single relevant document, missing information from others unless explicitly referenced.