Documentation

# System Architecture

A diagram of a process flow

AI-generated content may be incorrect.

## 1. Large Language Model (LLM) - Mistral 7B

* **Choice:** Mistral 7B was selected as the **LLM backend** due to its balance between performance and efficiency. It provides **state-of-the-art language understanding** while being lightweight enough to deploy on limited resources. A self-hosted model instead of an API-based model was chosen as it is more easily scalable with minimal impact on cost.
* **Interaction:** The LLM receives **user queries** from the backend, processes them, and generates relevant responses.

## 2. FAISS Indexing for Knowledge Retrieval

* **Choice**: FAISS (Facebook AI Similarity Search) is an efficient vector database that enables fast retrieval of relevant text passages. It is optimized for high-dimensional vector search. This chatbot uses HNSW (Hierarchical Navigable Small World) which is one of the most efficient algorithms for ANN search in high dimension spaces. It helps the chatbot answer questions based on pre-indexed knowledge sources.
* **Interaction**: The system embeds preprocessed documents into a vector space and stores them in FAISS. When a user query is received, its embedding is compared against stored vectors to retrieve the most relevant passages, which are then used for generating responses.

## 3. Embedding Model - sentence-transformers/all-MiniLM-L6-v2

* **Choice:** This model was chosen for FAISS-based indexing due to its fast and efficient sentence embeddings, which allow for semantic search and retrieval of relevant information.
* **Interaction:** When a user asks a question, the embedding model converts the query into a vector representation. This vector is used to perform approximate nearest-neighbor (ANN) search in the FAISS index to find relevant knowledge base documents.

## 4. Backend - FastAPI

* **Choice:** FastAPI was selected due to its high performance and ease of integration with ML models. It provides asynchronous processing, reducing latency in LLM queries.
* **Interaction**: FastAPI acts as the central hub, handling:
* User requests from the frontend
* Query processing (embedding + FAISS retrieval)
* LLM inference for generating responses
* Returning responses to the frontend

## 5. Frontend - Streamlit

* **Choice**: Streamlit was chosen for its ease of implementation and rapid prototyping capabilities. It allows for quick UI development without extensive web development knowledge.
* **Interaction**: The frontend provides:
* A chat interface for user interaction
* Displays responses from the backend
* Handles user input and forwards queries to FastAPI

**Note:** Frontend and Backend aren’t ideal for scaling the chatbot. I have prioritized ease of implementation as my preferred role is ML/AI.

## 6. Framework

* **Choice**: vLLM+LangChain is used to streamline prompt engineering, retrieval-augmented generation (RAG), model interactions and add memory. LangChain is a high-level API that allows use of these features but it is slow and lacks multi-GPU support(required for scaling). This and other disadvantages are overcome by using a hybrid approach of vllm and langchain.
* **Interaction**: LangChain integrates multiple components:
* Handles retrieval-augmented generation (RAG) by fetching FAISS results and structuring prompts for the LLM.
* Uses memory and conversation history to improve responses.
* Manages API calls to vLLM and FAISS for a modular, maintainable architecture.

Langchain is a high-level API that allows use of features like RAG, chat memory. However it is slower, lacks multi-GPU support and has other disadvantages. These are overcome by using vLLM+Langchain

## 7. Interaction Flow

* The chatbot system operates as follows:

1. The **user enters a query** in the Streamlit interface.
2. The query is **sent to the FastAPI backend**.
3. The backend processes the query:
   * Converts it into an embedding using **MiniLM**.
   * Searches the **FAISS index** for relevant documents.
   * Sends the retrieved context + user query to **Mistral 7B** for response generation.
4. The generated response is **returned to the frontend** for display.

# Scalability

## Handling Increased Load

The chatbot system is designed to **gracefully scale** to handle increasing user demand. To achieve this, the architecture leverages **asynchronous processing and optimized model inference** to reduce response latency. The key strategies include:

1. **Efficient LLM Inference with vLLM**
   * **Continuous Batching:** vLLM efficiently batches multiple incoming requests, reducing computation overhead.
   * **Tensor Parallelism:** Enables optimized GPU utilization, distributing workload across multiple GPUs.
2. **Retrieval Optimization with FAISS**
   * **Approximate Nearest Neighbor (ANN) search** speeds up retrieval, ensuring fast document lookup even with a growing knowledge base.
   * **Sharding FAISS Indexes** allows for distributed storage and retrieval across multiple nodes.
3. **Asynchronous API Calls with FastAPI**
   * Supports non-blocking request handling, preventing bottlenecks during high-traffic periods.
   * Users don’t have to wait for sequential processing, improving throughput.

## Scaling Infrastructure to Support 10,000 Users

To support **10,000 concurrent users**, the system follows a **microservices-based, scalable deployment strategy**:

1. **Horizontal Scaling (Scaling Out)**
   * **Multiple API instances:** FastAPI can be **deployed with multiple replicas** behind a load balancer (e.g., **NGINX** or **AWS ALB**) to handle increased traffic.
   * **Distributed FAISS Indexing:** FAISS shards can be stored across multiple nodes, enabling parallel retrieval.
   * **Multiple vLLM Inference Servers:** LLM inference can be **distributed across multiple GPU nodes** to handle concurrent queries.
2. **Vertical Scaling (Scaling Up)**
   * **More powerful GPUs (A100, H100, or TPU instances)** can improve LLM inference time.
   * **Using multi-GPU setups with model parallelism** enables handling longer or more complex queries and accelerates response time.
   * **Increasing FAISS memory allocation** allows faster lookup times for large-scale knowledge bases.
   * Using **vLLM’s PagedAttention** to optimize memory usage and allow more concurrent inference requests.

## Potential Bottlenecks & Mitigation Strategies

|  |  |  |
| --- | --- | --- |
| **Bottleneck** | **Cause** | **Strategy** |
| **LLM Inference Latency** | High computation time for large models | Use **vLLM batching** and **GPU parallelism** |
| **Slow FAISS Retrieval** | Large-scale vector search | Use **sharded FAISS** and **ANN optimizations** |
| **High API Request Load** | Too many concurrent user requests | Deploy **multiple FastAPI instances** behind a load balancer |
| **GPU Memory Overload** | Too many simultaneous LLM queries | Use **multi-GPU scaling and offloading** |
| **Slow Response Time** | Query processing and retrieval delay | Implement **Redis caching** and **precompute embeddings** |

# Cost Considerations

## Optimizing Compute Costs

Nvidia A100 is one of the most powerful GPUs and is fully compatible with Mistral7b. According to my research, it costs around 1,00,000 INR monthly (provided by cyfuture cloud).

* + - 1. **Using vLLM for Efficient Inference**
* vLLM enables efficient inference by leveraging **PagedAttention**, which reduces GPU memory overhead and allows **batch processing**.
* This improves token throughput per second, reducing **GPU usage costs**.
  + - 1. **GPU Autoscaling**
* Instead of keeping GPUs running at all times, **auto-scaling** ensures that GPU instances **shut down when idle** and spin up when needed.
  + - 1. **Load Balancing to Prevent Resource Overuse**
* Distributing inference requests across multiple vLLM instances prevents unnecessary GPU bottlenecks and optimizes workload allocation.

## Storage & Database Cost Optimization

One of the disadvantages of using the HNSW index in FAISS is its high memory (RAM) consumption. Based on my calculations, 150 GB of RAM will be required to store round 1000 books of roughly 600 pages each. By AWS, this would cost about 80,000 INR per month (cheaper alternatives can be searched for). Instead, the embeddings can be stored on SSD to reduce cost (not ideal as it is slower).  
Index Sharding decreases cost by reducing memory footprint per instance.

## Efficiency Considerations for an LLM-Based System

* Implementing a **max token limit** for responses ensures that the LLM doesn’t generate unnecessarily long replies, reducing compute costs.
* GPU auto-scaling reduces cost.
* Dimensionality reduction using PCA speeds up similarity searchs.
* Mixed Precision(FP16) reduces memory usage and speeds up inference.

# ML/AI Integration

**Executive Summary:** The chatbot system is designed to provide accurate responses to literature-related queries.

## TechStack used:

Mistral7B model, vllm+Langchainframework, fastAPI, streamlit, FAISS-HNSW Index Vector Database, sentence-transformers/all-MiniLM-L6-v2 embedding model

## Detailed Workflow:

1. **Preprocessing & FAISS Indexing (process\_books.py). Making database for RAG**

* Loads book texts and applies **tokenization and lemmatization**.
* Identifies books that are **not already embedded** in the FAISS database.
* **Dynamically chunks** new book texts.
* Uses **sentence-transformers/all-MiniLM-L6-v2** to generate embeddings batch-wise.
* **Reduces dimensions** via PCA for optimized storage and retrieval.
* **L2 normalizes embeddings** and stores them in FAISS with an **HNSW index** for fast similarity search.
* This process is triggered by the admin whenever the book database is updated.

1. **Retrieval of Relevant Chunks (vector\_db.py).**

* Implements *retrieve\_relevant\_docs*, which retrieves the **top k relevant text chunks** based on query similarity. (Retrieval mechanism for RAG).

1. **Backend API (main.py)**

* Runs a **FastAPI** server that receives user queries from the frontend.
* Calls generate\_response from model.py to generate the chatbot response.

1. **LLM Processing (model.py)**

* Uses **Mistral 7B** as the LLM.
* Maintains **summary memory and buffer memory** to preserve conversational context.
* Cleans the query and applies the same preprocessing as in process\_books.py.
* Calls *retrieve\_relevant\_docs* to obtain **contextually relevant book excerpts**.
* Constructs a **carefully engineered prompt** to minimize hallucination.
* The LLM is invoked with sampling parameters temperature=0.7, top\_p=0.9, max\_tokens=512.
* The LLM generates a response along with a confidence score.
* Memory is **saved** to maintain coherent multi-turn conversations.
* The response is returned to main.py, which sends it to the **Streamlit frontend**.

## Context Management and Prompt Engineering

* Retrieval-Augmented Generation (RAG) ensures factual accuracy by pulling information from indexed book embeddings.
* Summary and buffer memory provide contextual continuity across multiple user interactions.
* Dynamic prompt engineering helps mitigate hallucination while enforcing structured responses.
* Query cleaning ensures that user input is properly formatted and filtered before retrieval.
* System messages guide the LLM to provide responses aligned with the chatbot's purpose and prevent unwanted behavior.
* Confidence scoring helps assess the reliability of generated responses.

## Model Performance Optimization and Latency Considerations

* FAISS HNSW Index ensures fast and efficient similarity search.
* Dimensionality reduction via PCA balances speed and accuracy.
* Token length management optimizes response generation without exceeding token limits.
* Efficient memory management minimizes redundant context storage and improves performance.
* Explained in much more detail across various previous sections

## Handling Model Failures and Fallbacks

* **Edge case handling** ensures the chatbot gracefully declines irrelevant or unsupported queries.
* Low confidence responses are dealt with accordingly.
* Llm.generate() is wrapped in a try-except block.

# Deployment Instructions

Create a new virtual environment and activate it

*python -m venv chatbot-env*

*source vllm-rag-env/bin/activate # On Mac/Linux*

*chatbot-env\Scripts\activate # On Windows*

Install required libraries from requirements.txt

*pip install -r requirements.txt*

To add more books to the database, add them to data/books and run. I have already added “Dune” and “I have no mouth and I must scream” to the FAISS database as an example so this step can be skipped

*python process\_books.py*

Go to <https://huggingface.co/settings/tokens> and create your own token with the Token Type = “Write”. Copy your token. Go to <https://huggingface.co/mistralai/Mistral-7B-v0.1> and request for access to use the Mistral7B model.

Open hf\_login.py and paste your token in the line provided.

**Note:** For now, you can skip creating your own token and use mine provided in email.

Run hf\_login.py

*python hf\_login.py*

Run main.py in one terminal. Open a new terminal and run app.py using streamlit

*python main.py*

*streamlit run app.py*