# Depi: The AI-Powered Study Assistant

## A Technical Deep Dive into Retrieval-Augmented Generation, Summarization, and Automated Q&A

Graduation Project by yossufyasser1

# The Modern Student's Challenge: Information Overload

Students today face an avalanche of digital information—dense PDFs, lengthy textbooks, and complex lecture notes.

The core challenge is no longer access, but **assimilation**.

How can we transform passive reading into active, active, efficient learning? How do we find the precise answer buried in a 300-page document?

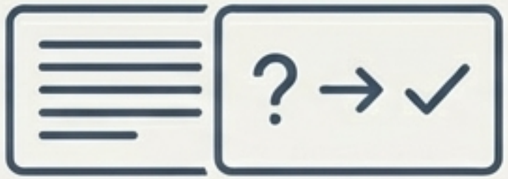
This is the problem we set out to solve.

# Our Solution: A Local-First AI Study Assistant

We engineered a system that empowers students by turning their study materials into interactive knowledge bases.

### 1. Intelligent Summarization

Distill the key insights from long documents into concise, abstractive summaries.



### 2. Automated Q&A Generation

Instantly create flashcards and study questions from any text to reinforce learning.

### 3. Conversational RAG

Chat directly with your documents to get precise, fact-based answers, complete with context.

Our guiding principle: **local-first and private**. All core processing happens on your machine.

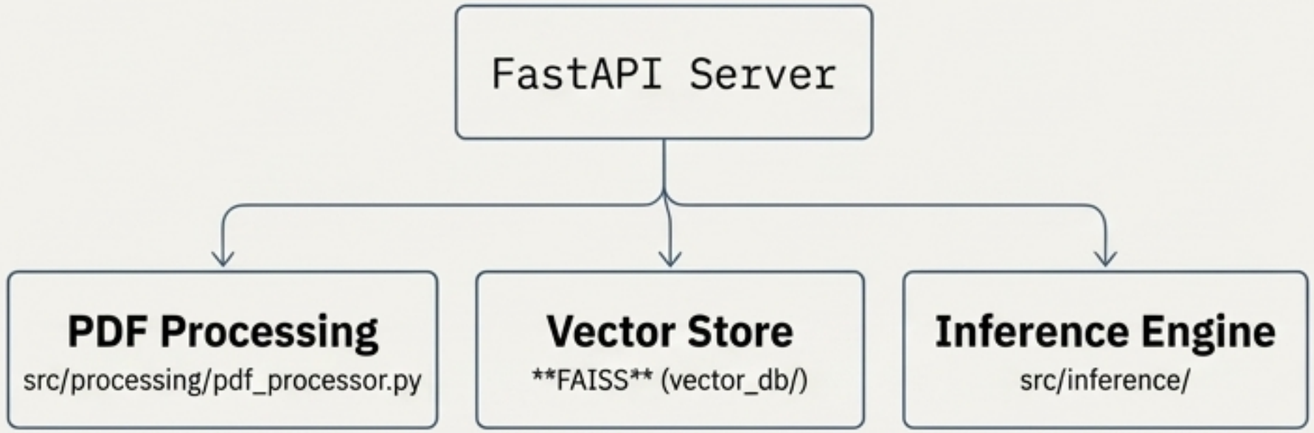
# System Architecture: The Technical Blueprint

The system is built on a robust, modern stack designed for performance and modularity. A FastAPI backend orchestrates all AI tasks.

## High-Level Flow

`Client (UI/API) <--> FastAPI Server (main.py)`

## Core Backend Services




## Technology Stack

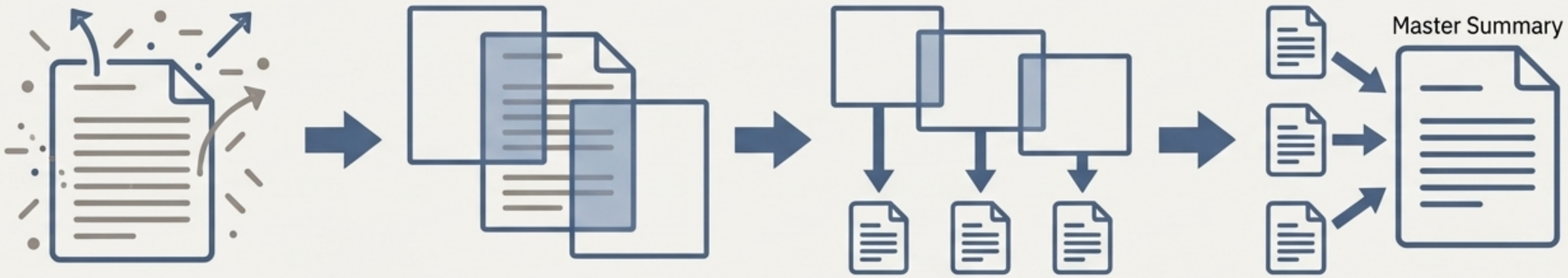
| | |
|---|---|
| **Framework** | `FastAPI` |
| **Orchestration** | LangChain |
| **AI Models** | Transformers, `sentence-transformers` |
| **LLM Generation** | Ollama (e.g., `llama3.2`) |
| **Vector DB** | FAISS (Facebook AI Similarity Search) |

# Deep Dive: Advanced Text Summarization

To summarize long documents effectively, we overcome the token limits of standard models using a hierarchical approach.

- Model: `facebook/bart-large-cnn`
- Rationale: Chosen for its proven ability to generate high-quality, human-like abstractive summaries, not just extracting sentences.

## Hierarchical Strategy




### Step 1: Pre-processing

The input text is aggressively cleaned to remove artifacts like table of contents dot leaders, repeated headers/footers, and de-hyphenate lines.

### Step 2: Chunking

The cleaned text is divided into large, overlapping chunks (e.g., ~4000 characters with a 400-character overlap).

### Step 3: Map & Reduce

Each individual chunk is summarized by the BART model.

### Step 4: Synthesize

The summaries of all chunks are concatenated and fed back into the model one last time to create a final, cohesive master summary.

# Summarization in Practice: The Code

Our implementation in `src/inference/summarizer.py` encapsulates this hierarchical logic. Key parameters are fine-tuned for quality.

```python
# src/inference/summarizer.py (Conceptual)
from transformers import pipeline

# Initialize the pipeline with our chosen model
summarizer = pipeline(
    "summarization",
    model="facebook/bart-large-cnn"
)

def create_hierarchical_summary(long_text, max_len, min_len):
    """
    Cleans, chunks, and synthesizes summaries for long documents.
    """
    # 1. Clean the text to remove noise
    cleaned_text = clean_document_text(long_text)

    # 2. Split text into manageable chunks
    chunks = chunk_text(cleaned_text, chunk_size=4000, overlap=400)

    # 3. Summarize each chunk individually (Map step)
    chunk_summaries = summarizer(chunks, ...)

    # 4. Combine and create a final summary (Reduce step)
    final_summary_input = " ".join(chunk_summaries)
    final_summary = summarizer(
        final_summary_input,
        max_length=max_len, # default: 1000
        min_length=min_len, # default: 350
        num_beams=5,
        repetition_penalty=1.2
    )
    return final_summary
```

# Deep Dive: Automated Q&A Pair Generation

To facilitate active recall—a proven study technique—we automate the creation of question-answer pairs directly from the source material.

**Model:** `google/flan-t5-base`
**Rationale:** FLAN-T5 is an instruction-tuned model. It excels at following structured prompts, making it ideal for generating well-formed Q&A pairs from a given context.

### 1. Input


### 2. Prompting

```
Generate questions and answers based on this text: {text}
```

### 3. Generation


### 4. Output

```
[
  {
    "question": "What is FLAN-T5's primary strength?",
    "answer": "FLAN-T5 excels at following structured prompts, making it ideal for generating well-formed Q&A pairs."
  },
  {
    "question": "How is the model used?",
    "answer": "It automates the creation of Q&A pairs directly from source material."
  }
]
```

# Q&A Generation: The `QA_model.ipynb` Implementation

The logic for this feature was developed and tested in `QA_model.ipynb`. It focuses on a straightforward inference pipeline using the pre-trained FLAN-T5 model.

## Process Steps

### 1. Data Preprocessing

The source document text is loaded and cleaned (similar to the summarizer) to ensure the model receives high-quality context, free of formatting noise.

### 2. Model Inference (Not Training)

We use the pre-trained `google/flan-t5-base` model directly. No fine-tuning is necessary for this task due to the model's powerful instruction-following capabilities.

## Inference Code

### 3. Inference Pipeline Code

```
# From QA_model.ipynb (Conceptual)
from transformers import T5ForConditionalGeneration, T5Tokenizer

# Load the pre-trained model and tokenizer
model_name = "google/flan-t5-base"
model = T5ForConditionalGeneration.from_pretrained(model_name)
tokenizer = T5Tokenizer.from_pretrained(model_name)

def generate_qa_from_context(context_text):
    # Craft the instruction-based prompt
    prompt = f"Generate questions and answers based on this text: {conte

    # Tokenize and generate
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids
    outputs = model.generate(input_ids, max_length=512, num_beams=4)

    # Decode and parse the raw output into a clean Q/A list
    qa_string = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return parse_qa_string(qa_string)
```

# Deep Dive: Retrieval-Augmented Generation (RAG)

RAG is the core of our conversational AI. It allows the system to answer questions using facts drawn **directly** from the user's documents, preventing hallucinations and ensuring relevance.

```
`Grounded Answer = Relevant Retrieved Knowledge + LLM Reasoning`
```

## How It Works

### 1. Retrieve

Before answering, the system searches the indexed document database (FAISS) to find the most relevant snippets of text related to the user's question.

### 2. Augment

These retrieved snippets are injected into the prompt as "context" for the Large Language Model.

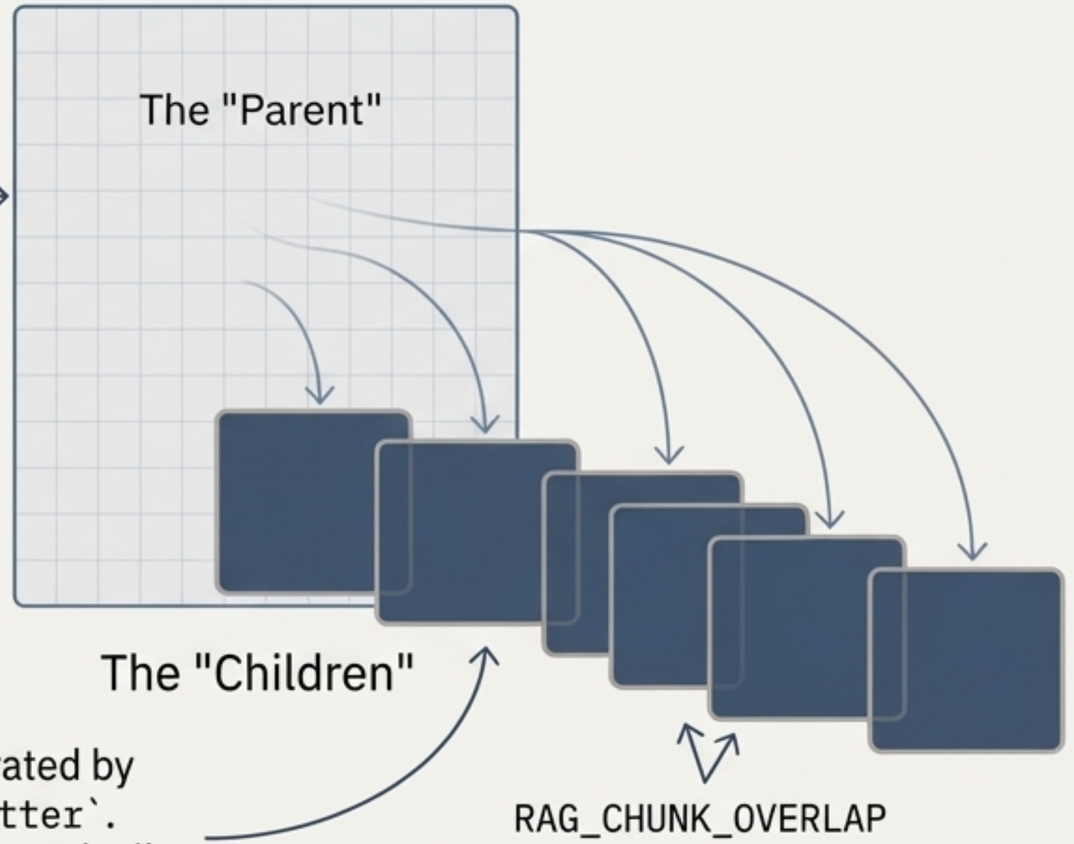

### 3. Generate

The LLM (Ollama) is instructed to formulate an answer *based exclusively on the provided context*.

# Our RAG Strategy: The Parent-Child Architecture

To make retrieval both precise and efficient, we structure the document's data in a 'Parent-Child' relationship.

The original, complete document. It represents the ultimate source of truth but is too large to fit into a model's context window.




Small, indexed text chunks generated by `RecursiveCharacterTextSplitter`. Each chunk is a "child" that is semantically linked to the parent document.

**Why this architecture is powerful**

- **Precision**: The system retrieves small, highly relevant child chunks, pinpointing the exact information needed.
- **Efficiency**: Searching over small vector embeddings is orders of magnitude faster than processing a full document.
- **Context Integrity**: Overlapping the chunks (`RAG_CHUNK_OVERLAP`) ensures that semantic meaning isn't lost at the boundaries.

# The RAG Workflow in Action: A User's Query

Here's the step-by-step journey of a question posed to the `/chat` endpoint.


### 1. User Question

`"What is the main argument of chapter 2?"`

### 2. Embed

The question is converted into a vector using `` `sentence-transformers/all-mpnet-base-v2` ``.

### 3. Retrieve

FAISS performs a similarity search on the indexed 'child' chunks, fetching the top-k (e.g., `` `RAG_TOP_K=5` ``) chunks most similar to the question's vector.

### 4. Augment

The text from these retrieved chunks is compiled into a single `context` block, separated by `` `---DOCUMENT SEPARATOR---` ``.

### 5. Prompt

```
Context: [Text from retrieved chunk 1] --- [Text from retrieved chunk 2] ...

Question: What is the main argument of chapter 2?

Answer:
```

### 6. Generate

Ollama's `` `llama3.2` `` model generates a concise answer synthesized purely from the provided context.

# The RAG Codebase: Indexing and Retrieval

The entire RAG process is orchestrated by two key modules, separating the one-time indexing from the per-query retrieval.

## 1. Indexing: Creating the Knowledge Base

`src/processing/improved_rag_retriever.py`

This happens once when a document is uploaded. We create and save the 'child' chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import FAISS

# Define our chunking strategy ("Parent" -> "Children")
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=RAG_CHUNK_SIZE,
    chunk_overlap=RAG_CHUNK_OVERLAP
)
chunks = text_splitter.split_documents(documents)

# Create the vector store from child chunks and save it
db = FAISS.from_documents(chunks, embeddings_model)
db.save_local(FAISS_DB_DIR)
```

## 2. Retrieval & Generation

`src/inference/improved_llm_reader.py`

This runs for every chat message, finding context and generating an answer.

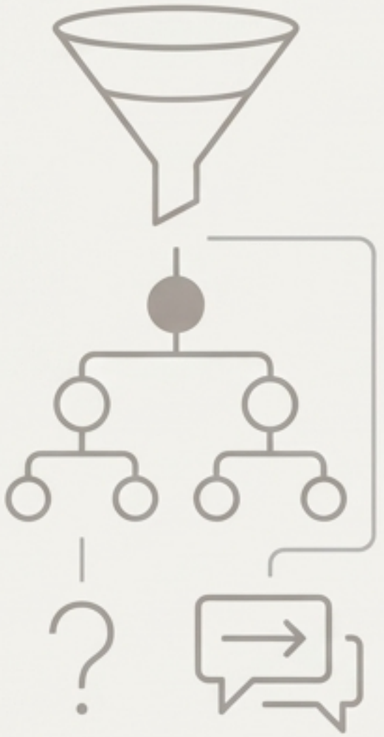
```
# Load the pre-existing FAISS database
db = FAISS.load_local(
    FAISS_DB_DIR,
    embeddings_model,
    allow_dangerous_deserialization=True
)

# Find relevant child chunks for the new question
retrieved_docs = db.similarity_search(question, k=top_k)
context = format_context(retrieved_docs)

# Build prompt and invoke the Ollama LLM
prompt = f"Context: {context}\n\nQuestion: {question}\n\nAnswer:"
answer = ollama_llm.invoke(prompt)
```

# Project Summary: A Powerful Learning Toolkit

We have successfully designed and implemented a multi-faceted AI Study Assistant that bridges the gap between static documents and dynamic, active learning.

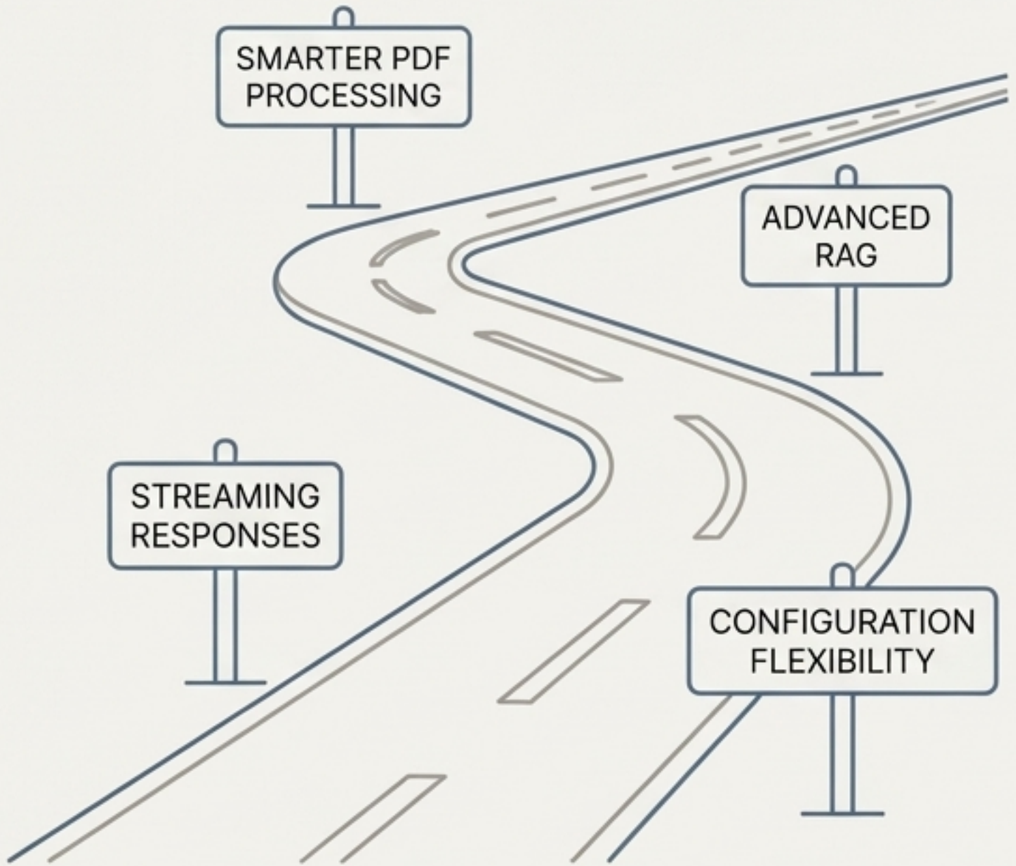


- ✓ **Robust Backend**: A scalable FastAPI application serves all AI functionalities.
- ✓ **Advanced Summarization**: A hierarchical strategy produces high-quality summaries of any length.
- ✓ **Automated Study Aids**: FLAN-T5 generates accurate Q&A pairs to test comprehension.
- ✓ **Fact-Grounded Chat**: An efficient RAG pipeline with a Parent-Child architecture provides precise, trustworthy answers from source materials.
- ✓ **Local & Private**: Core functionality runs locally via Ollama, ensuring user data privacy.

# Future Enhancements and Roadmap

This project provides a strong foundation. The following enhancements are planned to further improve its capabilities.

- **Smarter PDF Processing:** Improve logic for intelligently removing headers, footers, and page numbers to create cleaner context.
- **Advanced RAG:** Implement re-ranking of retrieved documents to place the most relevant context first and add context length caps.
- **Improved User Experience:** Add support for streaming responses in the chat interface for lower perceived latency.
- **Configuration Flexibility:** Develop a model registry within the configuration to allow for easily switching between different LLMs or embedding models.



# Explore the Project

github.com/yossufyasser1/Depi-grad-project-

Questions & Discussion