Architecture & Implementation Document for a Complex Query Answering System

# 1. Overview

Objective:  
Build a system that handles complex, non-repetitive queries by students. Instead of simply returning long excerpts, the system will retrieve relevant, detailed information from a corpus and generate a concise, easy-to-understand answer.

Approach:  
Use a Retrieval-Augmented Generation (RAG) pipeline that:  
- Retrieves relevant document sections from a large corpus.  
- Generates a simplified and summarized answer using a fine-tuned generative model.

# 2. Data Storage and Format

## Document Corpus (Detailed Information):

Format: Store your data as detailed documents or paragraphs.  
Structure: Each document should include:  
 - Title: A short, descriptive title.  
 - Content: The full, detailed explanation.  
 - Metadata (Optional): Tags, topics, or section headings that aid in retrieval.

Example JSON Format:

[  
 {  
 "title": "Advanced Quantum Mechanics",  
 "content": "Quantum mechanics is a branch of physics that explains the behavior of matter and energy at the atomic and subatomic levels. It involves complex concepts such as wave-particle duality, uncertainty principles, and quantum entanglement...",  
 "metadata": {"tags": ["physics", "quantum", "advanced"]}  
 },  
 {  
 "title": "Effective Study Techniques for Complex Subjects",  
 "content": "When dealing with complex topics, breaking down information into smaller, manageable segments is crucial. Techniques such as spaced repetition, summarization, and active recall can significantly improve understanding...",  
 "metadata": {"tags": ["education", "study techniques", "complex subjects"]}  
 }  
]

Training Data for Simplification (Optional but Recommended):  
Pair each complex document (or excerpt) with a concise summary.

Example Pair:

{  
 "input": "Quantum mechanics is a branch of physics that explains the behavior of matter and energy at the atomic and subatomic levels, involving complex principles like wave-particle duality and uncertainty.",  
 "output": "Quantum mechanics studies how tiny particles behave in ways that differ from everyday objects, with principles that challenge common intuition."  
}

# 3. System Modules

## A. Data Ingestion & Preprocessing:

Storage: Use JSON files or a database (via Django models) to store your documents.  
Preprocessing: Use spaCy to tokenize and clean the text, and Sentence Transformers to encode the text.  
Indexing: Build a FAISS index for vector similarity search; optionally, use BM25 for keyword-based retrieval.

## B. Retrieval Module:

Function: Accept a student's query and retrieve relevant document sections using BM25 and/or FAISS.  
Outcome: Provide rich context to the generative model.

## C. Generative Model (Simplification & Answering):

Model Options: Pre-trained models like T5, FLAN-T5, or GPT-based models.  
Fine-Tuning: Fine-tune on your paired data (complex input to simplified output) if available.  
Prompt Engineering: Use prompts that instruct the model to provide a clear, concise answer.

## D. Chatbot Interface and Integration:

Telegram Bot: Use the Telegram Bot API (with python-telegram-bot) and store sensitive information (e.g., bot token) in Django settings.  
Django Backend: Manage data and configuration.  
(Optional) Rasa Integration: For additional NLU and conversation management.

# 4. Tools & Technologies

- Programming Language: Python (3.8 or 3.9 recommended)  
- Frameworks: Django for backend; Telegram Bot API and optionally Rasa for the chatbot interface.  
- NLP Libraries: spaCy, Sentence Transformers, Transformers (Hugging Face)  
- Indexing: FAISS for vector search; BM25 for keyword retrieval.  
- Environment: python-dotenv for managing environment variables.

# 5. Implementation Steps

## Step 1: Environment Setup

Create and activate a virtual environment.  
Install dependencies:  
 pip install django rasa transformers sentencepiece spacy faiss-cpu rank\_bm25 python-dotenv  
 python -m spacy download en\_core\_web\_sm

## Step 2: Data Ingestion

Store and load your detailed documents (JSON or Django models).

## Step 3: Indexing and Retrieval

Generate embeddings using Sentence Transformers.  
Build a FAISS index and optionally a BM25 index.

## Step 4: Generative Module

Load and optionally fine-tune a pre-trained model (e.g., FLAN-T5).  
Craft prompts for simplified answer generation.

## Step 5: Integration

Integrate with a Telegram bot (import token from Django settings).  
Combine retrieval and generation to provide answers.

## Step 6: Deployment and Testing

Run the Django server.  
Start the Telegram bot.  
Test the system with sample complex queries.

# 6. System Diagram (Conceptual)

┌─────────────────┐

│ Student Query │

└───────┬─────────┘

│

┌────────▼─────────┐

│ Retrieval Module │ <-- Uses BM25 & FAISS

└────────┬─────────┘

│

┌────────▼─────────┐

│ Generative Model │ <-- Simplifies & summarizes answer

└────────┬─────────┘

│

┌───────▼─────────┐

│ Answer Output │

└─────────────────┘

# 7. Conclusion

By employing a hybrid Retrieval-Augmented Generation approach:  
- Complex information is preserved from detailed documents.  
- Concise, simplified answers are generated for clarity.  
- Modern NLP techniques and a modular design ensure scalability and accuracy.  
  
This design will help answer challenging, non-repetitive student queries in a clear, understandable manner.