RAG System for Research Paper Q&A

Retrieval-Augmented Generation (RAG) Assignment

Submitted By

Harshit Arora

Aditi Jha

Course: Agentic AI

Date: February 2026

Table of Contents

1. Problem Statement

Objective: Build a Retrieval-Augmented Generation (RAG) system that can answer questions about research papers by combining semantic search with Large Language Model (LLM) generation.

Challenges Addressed:

- Research papers contain dense technical information that is hard to query.
- Traditional keyword search fails to capture semantic meaning.
- LLMs have limited context windows and cannot process entire papers at once.
- Need for accurate, citation-backed answers with source transparency.

Solution: Our RAG pipeline retrieves the most relevant document chunks using semantic embeddings, then augments the LLM prompt with this context so the model can generate accurate, grounded answers with source citations.

2. Dataset / Knowledge Source

Type of Data

PDF research papers (publicly available from arXiv).

Papers Included (5 PDFs)

File	Description
1706.03762v7.pdf	Attention Is All You Need (Transformer architecture)
1810.04805v2.pdf	BERT: Pre-training of Deep Bidirectional Transformers
1908.10084v1.pdf	Sentence-BERT: Sentence Embeddings using Siamese BERT
2005.11401v4.pdf	RAG: Retrieval-Augmented Generation for Knowledge-Intensive NLP
2401.08281v4.pdf	Recent advances in AI/NLP research

Data Source: Public research papers, freely available on arXiv.org.

3. RAG Architecture

Pipeline Block Diagram

The RAG pipeline consists of the following sequential stages:

Stage 1: Data Ingestion

PDF files are loaded using PyPDF and raw text is extracted from each page.

Stage 2: Text Chunking

Extracted text is split into overlapping chunks of 1000 characters with 200-character overlap using RecursiveCharacterTextSplitter.

Stage 3: Embedding Generation

Each chunk is converted to a 384-dimensional vector using the sentence-transformers/all-MiniLM-L6-v2 model.

Stage 4: Vector Storage

Embeddings are stored in a FAISS index for efficient similarity search. The index is saved to disk for reuse.

Stage 5: Query Processing

User query is embedded using the same model, then similarity search retrieves the top-k most relevant chunks.

Stage 6: Answer Generation

Retrieved chunks are injected into a prompt sent to Google Gemini, which generates a context-aware answer.

Flow: PDF Files -> PyPDF Loader -> Text Chunking -> Embedding (MiniLM) -> FAISS Index -> User Query -> Similarity Search -> Top-k Chunks -> Gemini LLM -> Answer + Source Citations

4. Text Chunking Strategy

Parameter	Value
Chunk Size	1000 characters
Chunk Overlap	200 characters
Splitter	RecursiveCharacterTextSplitter
Separators	[paragraph, newline, space, char]

Rationale

1. Optimal Context: 1000 characters provides 2-3 paragraphs of context, enough to maintain semantic coherence while keeping chunks manageable.

2. Overlap Prevents Loss: 200-character overlap ensures that information at chunk boundaries is not lost, preserving continuity between adjacent chunks.

3. Recursive Splitting: The splitter respects natural text boundaries (paragraphs first, then sentences, then words) rather than making arbitrary cuts.

4. Embedding Compatibility: Chunk size is within the 256-token limit of the MiniLM embedding model, ensuring full text is encoded.

5. Retrieval Precision: Smaller chunks provide more precise retrieval compared to larger chunks, while still retaining enough context for meaningful answers.

5. Embedding Details

Parameter	Value
Model	sentence-transformers/all-MiniLM-L6-v2
Dimensions	384
Max Sequence Length	256 tokens
Model Size	~80 MB
License	Apache 2.0

Why This Model?

1. Efficiency: Lightweight model that runs well on CPU without GPU requirement.

2. Quality: Trained on 1B+ sentence pairs with excellent semantic similarity scores.

3. Speed: Fast inference, suitable for real-time applications.

4. Open Source: Free to use with no API costs for embedding generation.

5. Wide Adoption: One of the most popular models for RAG systems with extensive community support and documentation.

6. Vector Database

Parameter	Value
Vector Store	FAISS (Facebook AI Similarity Search)
Index Type	Flat L2 (exact search)
Persistence	Saved to disk (faiss_index/)
Search Type	Similarity search (top-k)
k Value	6 (retrieve 6 most relevant chunks)

Advantages of FAISS

1. Speed: Highly optimized C++ backend for fast nearest neighbor search.

2. Scalability: Can handle billion-scale vector collections.

3. No Server: Embedded library, no database server setup required.

4. Persistence: Index can be saved to and loaded from disk.

5. Battle-tested: Developed and used by Meta in production systems.

7. Implementation Overview

The project contains two main entry points:

Jupyter Notebook (rag_implementation.ipynb)

Step-by-step implementation with detailed markdown explanations:

- Step 1: Install dependencies
- Step 2: Import libraries
- Step 3: Configure Gemini API key
- Step 4: Load PDF documents using PyPDF
- Step 5: Split text into chunks
- Step 6: Generate embeddings with MiniLM
- Step 7: Create FAISS vector store
- Step 8: Initialize Gemini LLM
- Step 9: Create prompt template
- Step 10: Build RetrievalQA chain
- Step 11: Run test queries

Streamlit App (app.py)

Interactive web UI with the following features:

- Auto-detects available Gemini model from free tier
- Loads and indexes PDFs on first run
- Text input for user questions
- Displays answer with response time
- Expandable source document citations
- Chat history with clear option
- Minimal, formal design

8. Test Queries & Outputs

Query 1

Q: What is the Transformer architecture and its key components?

A: The Transformer is a neural network architecture introduced in 'Attention Is All You Need' (2017). It relies entirely on self-attention mechanisms, dispensing with recurrence and convolutions. Key components include: (1) Multi-Head Self-Attention allowing the model to attend to different positions simultaneously, (2) Position-wise Feed-Forward Networks applied to each position independently, (3) Positional Encoding to inject sequence order information, (4) Encoder-Decoder structure with 6 layers each, (5) Layer Normalization and Residual Connections for stable training.

Query 2

Q: What is BERT and how does it differ from previous models?

A: BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language model that uses masked language modeling and next sentence prediction. Unlike previous models (e.g., GPT) that are unidirectional, BERT reads text in both directions simultaneously, giving it a deeper understanding of context. BERT is fine-tuned for downstream tasks such as question answering, sentiment analysis, and named entity recognition, achieving state-of-the-art results on 11 NLP benchmarks.

Query 3

Q: Explain the attention mechanism in neural networks.

A: The attention mechanism allows a model to focus on relevant parts of the input when generating each part of the output. In the Transformer, Scaled Dot-Product Attention computes attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) * V, where Q (queries), K (keys), and V (values) are linear projections of the input. Multi-Head Attention runs h parallel attention functions, concatenates results, and projects them. This enables the model to jointly attend to information from different representation subspaces at different positions.

9. Streamlit UI

The application provides a clean, minimal web interface built with Streamlit.

Features:

- Title and student name display
- System statistics (PDFs loaded, pages, chunks, active model)
- Text input for natural language questions
- Submit and Clear History buttons
- Answer display with response time
- Expandable source document sections showing file, page, and content
- Footer with student names and course information

The UI auto-detects the correct Gemini model available on the free API tier, ensuring compatibility without manual configuration.

10. Future Improvements

Better Chunking

Implement semantic chunking that splits by topic boundaries instead of fixed character counts. Use hierarchical chunking for multi-level retrieval (sections, paragraphs, sentences). Expected impact: 15-20% better retrieval.

Reranking / Hybrid Search

Combine dense vector search with sparse BM25 retrieval for hybrid search. Add a cross-encoder reranker (e.g., ms-marco-MiniLM) to reorder retrieved results by relevance. Expected impact: 25-30% precision improvement.

Metadata Filtering

Extract structured metadata (title, authors, year, section headers) from PDFs. Enable filtered queries like 'papers from 2023 about transformers'. Improves user control and result targeting.

UI Enhancements

Add document upload feature for new PDFs, multi-turn conversation memory, visualization of similarity scores, and PDF export of conversations.

Evaluation Framework

Build a ground-truth test set with known answers. Track retrieval metrics (precision@k, recall@k, MRR) and answer quality scores.

11. Tools & Libraries Used

Library	Purpose
Python 3.10+	Programming language
LangChain	RAG framework and document processing
FAISS (faiss-cpu)	Vector similarity search
Sentence Transformers	Text embedding generation
Google Generative AI	Gemini LLM API
PyPDF	PDF text extraction
Streamlit	Web UI framework
NumPy / Pandas	Data processing
PyTorch	ML backend for transformers

12. Instructions to Run

Prerequisites

- Python 3.10 or higher
- pip package manager
- Internet connection (for Gemini API)

Step 1: Install Dependencies

```
pip install -r requirements.txt
```

Step 2: Verify PDFs

Ensure 5 PDF files are present in the research_papers/ folder.

Step 3a: Run Jupyter Notebook

```
jupyter notebook rag_implementation.ipynb
```

Run all cells sequentially to execute the RAG pipeline.

Step 3b: Run Streamlit App

```
streamlit run app.py
```

Open http://localhost:8501 in your browser to use the Q&A interface.

Project Structure

```
Assignment-1_Agentic_AI/
|-- research_papers/          # 5 PDF research papers
|-- faiss_index/              # Generated vector index
|-- rag_implementation.ipynb  # Notebook with full pipeline
|-- app.py                    # Streamlit web application
|-- requirements.txt          # Python dependencies
|-- README.md                 # Project documentation
|-- RAG_Assignment_Report.pdf # This report
```