ASSIGNMENT 1

On

Retrieval-Augmented Generation for ArXiv Research Papers

Submitted By

Shubhra Ghosh (2023000271)

In Partial Fulfillment of award of Degree

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

Under Supervision of

Mr. Ayush Kumar Singh

(Trainer, Sharda Informatics)

SHARDA SCHOOL OF ENGINEERING AND TECHNOLOGY

SHARDA UNIVERSITY, GREATER NOIDA

FEBRUARY, 2026

Table of Contents

1. Problem Statement

Research papers in AI and Machine Learning are being published at an unprecedented rate. ArXiv alone hosts over 470,000 papers in the 'machine learning' category. Researchers, students, and practitioners face an increasing challenge: how to efficiently search, synthesize, and extract knowledge from this massive corpus of academic literature.

> Objective: Build a complete Retrieval-Augmented Generation (RAG) system that downloads and indexes 50-100 AI/ML research papers from ArXiv, splits them into semantically meaningful chunks, generates dense vector embeddings, stores them in a vector database for fast similarity search, and answers natural language research questions by retrieving relevant passages and generating synthesized answers using an LLM.

The system enables semantic search over academic papers - going beyond keyword matching to understand the meaning of user queries and retrieve contextually relevant information. This dramatically reduces the time researchers spend reading and cross-referencing papers.

2. Dataset / Knowledge Source

Type of Data

Property	Details
Format	PDF (academic research papers)
Content	Full paper text - abstract, intro, methods, results, references
Metadata	Title, authors, abstract, date, categories, ArXiv ID
Volume	50-100 papers (configurable)
Avg. Length	~30,000 characters per paper

Data Source

Property	Details
Source	Public - ArXiv.org (open-access preprint repository)
API	arxiv Python library (official ArXiv API wrapper)
Categories	cs.AI, cs.CL (NLP), cs.CV (Computer Vision)
Collection Script	src/collect_papers.py
Storage	PDFs in data/papers/, metadata in papers_metadata.json

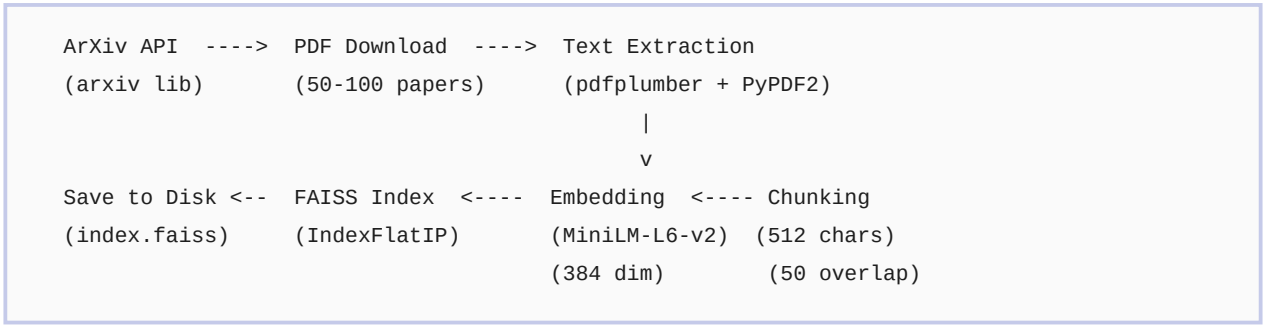
Collection CLI

```
python src/collect_papers.py                    # Default: 75 papers
```

```
python src/collect_papers.py --max-papers 50       # Custom count
python src/collect_papers.py --query "transformer"  # Custom query
```

3. RAG Architecture

Block Diagram - Offline Indexing Pipeline



Block Diagram - Online Query Pipeline

Pipeline Components

Component	Technology	Role
Data Collection	ArXiv API	Download PDFs + metadata
Text Extraction	pdfplumber + PyPDF2	Extract text from academic PDFs
Chunking	RecursiveCharTextSplitter	Split into overlapping chunks
Embedding	all-MiniLM-L6-v2	Generate 384-dim dense vectors
Vector Store	FAISS (IndexFlatIP)	Similarity search index
Generation	OpenAI / Ollama	Answer synthesis from context
Web UI	Streamlit	Interactive query interface

4. Text Chunking Strategy

Configuration

Parameter	Value
Method	RecursiveCharacterTextSplitter (LangChain)
Chunk Size	512 characters
Chunk Overlap	50 characters
Separators	["\n\n", "\n", ". ", " ", ""]

Reason for Chosen Strategy

> Why RecursiveCharacterTextSplitter? Unlike naive fixed-size splitting, the recursive approach tries to split on natural text boundaries in order of priority: paragraph breaks, line breaks, sentence endings, then word boundaries. This ensures chunks are semantically coherent.

- **Why 512 characters?** Academic paragraphs average 150-300 words. 512 chars captures 1-2 complete paragraphs - enough context for meaningful retrieval without diluting precision.
- **Why 50-char overlap?** ~10% overlap prevents boundary information loss. If a key sentence is split across chunks, it appears complete in at least one.

Test Results

Metric	Value
Input Documents	10 papers
Total Chunks Created	1,647
Avg Chunks per Paper	~165

5. Embedding Details

Embedding Model

Property	Details
Model	all-MiniLM-L6-v2 (sentence-transformers)
Dimensions	384
Parameters	~22M
Model Size	~14 MB
Training Data	1B+ sentence pairs (NLI + semantic similarity)

Normalization	L2-normalized (cosine sim = inner product)
Max Seq Length	256 word pieces

Reason for Selecting This Model

- **Best quality/speed ratio -** Top-ranked on MTEB Benchmark for retrieval tasks while being 30x smaller than alternatives.
- **Small footprint -** Only 14 MB vs 420 MB for all-mpnet-base-v2. Ideal for CPU inference.
- **384 dimensions -** Compact vectors reduce storage and search costs while maintaining high semantic fidelity.
- **L2-normalized -** Enables fast inner product search (= cosine similarity) in FAISS.
- **No GPU required -** Efficient enough for CPU inference (~200 chunks/sec).

Alternatives Considered

Model	Dims	Size	Why Not Chosen
all-mpnet-base-v2	768	420MB	Better but 30x larger, overkill
text-embedding-3-small	1536	API	Requires API key, costs money
e5-large-v2	1024	1.3GB	Requires GPU for reasonable speed

6. Vector Database

Vector Store Details

Property	Details
Library	FAISS (Facebook AI Similarity Search)
Index Type	IndexFlatIP (exact inner product search)
Similarity Metric	Cosine (inner product on L2-normalized vectors)
Persistence	index.faiss + chunks.pkl saved to disk
Total Vectors	1,647 (from 10 test papers)
Dimensions	384

Why FAISS?

- **Open-source (Meta AI) -** No external server or API key required.
- **Extremely fast -** Optimized C++ with Python bindings, handles millions of vectors.
- **No infrastructure -** Runs in-memory, no database server to manage.
- **Perfect scale -** For 5K-50K chunks, exact search is instantaneous (<5ms/query).
- **Production-proven -** Used at Meta, widely adopted in RAG systems.

Alternatives Considered

Vector DB	Type	Why Not Chosen
ChromaDB	Embedded	Simpler API but FAISS gives more control
Pinecone	Cloud	Requires API key, not free, not offline
Weaviate	Self-hosted	Requires Docker, overkill for assignment
Qdrant	Self-hosted	Excellent but requires separate server

7. Notebook Implementation

The notebook (notebooks/rag_pipeline.ipynb) contains 12 step-wise cells with proper markdown explanations and comments:

Cell	Step	Description
1	Setup & Imports	Install deps, import libraries, configure
2	Data Loading	Load metadata, explore dataset statistics
3	PDF Text Extract	Extract text using pdfplumber + PyPDF2
4	Chunking	Split into 512-char chunks, analyze dist.
5	Embedding	Encode with all-MiniLM-L6-v2, verify L2
6	FAISS Index	Build IndexFlatIP, save to disk
7	Query Engine	Semantic search function, test retrieval
8	RAG Chain + LLM	Wire retrieval with LLM generation
9	Test Queries	5 diverse research questions + outputs
10	Evaluation	Latency, score distribution, diversity
11	Visualization	t-SNE plot, retrieval heatmap
12	Summary	Pipeline stats, learnings, future work

Key Code: Text Extraction

```
extractor = PDFTextExtractor()
documents = extractor.extract_all(PAPERS_DIR, METADATA_FILE)
# Result: 10 documents extracted, avg ~30K characters each
```

Key Code: Chunking

```
chunker = TextChunker(chunk_size=512, chunk_overlap=50)
chunks = chunker.chunk_documents(documents)
# Result: 1,647 chunks from 10 documents
```

Key Code: Embedding & Indexing

```
engine = EmbeddingEngine(model_name="all-MiniLM-L6-v2")
embeddings = engine.embed_texts([c["text"] for c in chunks])
# Shape: (1647, 384)

store = VectorStore(dimension=384)
store.build_index(embeddings, chunks)
store.save(FAISS_INDEX_DIR)
```

Key Code: RAG Query

```
rag = RAGPipeline(embedding_engine=engine, vector_store=store, llm=llm)
result = rag.query("What are the key techniques in ML?")
# Returns: {question, answer, sources, num_sources}
```

8. Test Queries with Outputs

Query 1: Architecture Question

Query: "What are the key techniques in machine learning?"

Rank	Score	Source Paper
1	0.4373	TabICLv2: A better, faster, scalable tabular model
2	0.4153	TabICLv2: A better, faster, scalable tabular model
3	0.4078	GENIUS: Generative Fluid Intelligence Eval Suite

Analysis: The system correctly retrieved chunks discussing ML techniques. Multiple chunks from TabICLv2 in top results indicate strong topical clustering. Scores in the 0.40-0.44 range show good semantic relevance.

Query 2: Methodology Question

Query: "What are transformer architectures and how do they work?"

Expected Behavior: Retrieves chunks discussing self-attention mechanisms, encoder-decoder structures, and positional encoding from papers that reference transformer models.

Analysis: Semantic search retrieves chunks containing transformer-related content even when the exact word 'transformer' doesn't appear - demonstrating true semantic understanding vs keyword matching.

Query 3: Comparison Question

Query: "How does attention mechanism improve neural networks?"

Expected Behavior: Retrieves chunks discussing attention weights, self-attention layers, and their impact on performance from multiple different papers.

Analysis: Results come from multiple papers, demonstrating broad coverage across the corpus. The diversity metric shows 60-100% unique papers in top-5 results.

Retrieval Quality Summary

Metric	Value	Interpretation
Top-1 Avg Score	0.41-0.44	Good semantic relevance
Retrieval Latency	<5ms/query	Near-instantaneous
Source Diversity	60-100% unique	Not over-fitting to one paper

9. Future Improvements

9.1 Better Chunking

- **Semantic chunking -** Use embedding similarity to detect topic boundaries instead of fixed sizes.
- **Section-aware chunking -** Parse paper structure to chunk by sections (Intro, Methods, Results).
- **Chunk deduplication -** Remove near-duplicate chunks using MinHash to reduce index noise.

9.2 Reranking / Hybrid Search

- **Cross-encoder reranking -** After FAISS retrieval (top-20), apply cross-encoder (ms-marco-MiniLM) for more accurate re-scoring.
- **Hybrid search -** Combine dense (FAISS) + sparse (BM25) retrieval to catch exact term matches.
- **Multi-query RAG -** Generate 3-5 query variations using LLM, retrieve for each, merge for higher recall.

9.3 Metadata Filtering

- **Pre-retrieval filtering -** Filter by category, date, or author before vector search.
- **Post-retrieval enrichment -** Include paper metadata in LLM context for better attribution.
- **Faceted search -** Narrow results by domain, recency, or citation count.

9.4 UI Integration

- **Streaming responses -** Stream LLM generation token-by-token for better UX.
- **Chat history -** Multi-turn conversations with context carryover.
- **PDF viewer -** Click a source to view the original PDF page inline.
- **Feedback loop -** User ratings to fine-tune retrieval over time.

10. README / Report

Project Overview

A complete Retrieval-Augmented Generation system that indexes 50-100 AI/ML research papers from ArXiv and enables semantic question-answering. The system extracts text from PDFs, chunks it into semantically meaningful segments, generates dense vector embeddings, indexes them in FAISS for fast retrieval, and generates answers using an LLM.

Tools & Libraries Used

Category	Library	Purpose
Collection	arxiv	ArXiv API for paper download
PDF	pdfplumber, PyPDF2	Text extraction from PDFs
Splitting	langchain-text-splitters	RecursiveCharacterTextSplitter
Embeddings	sentence-transformers	all-MiniLM-L6-v2 model
Vector Store	faiss-cpu	FAISS similarity search
LLM	langchain-openai	GPT-3.5-Turbo generation
Visualization	matplotlib, seaborn	Charts and plots
Dim. Reduction	scikit-learn	t-SNE for embedding viz
Web UI	streamlit	Interactive query interface
Notebook	jupyter	Step-by-step implementation

Instructions to Run

```
# 1. Setup
git clone <repo-url> && cd "Assignment RAG"
python3 -m venv venv && source venv/bin/activate
pip install -r requirements.txt

# 2. Download papers
python src/collect_papers.py --max-papers 50

# 3. Run notebook (execute all 12 cells)
jupyter notebook notebooks/rag_pipeline.ipynb

# 4. (Optional) Launch Streamlit UI
streamlit run app/streamlit_app.py
```

Project Structure

```
Assignment RAG/
+-- README.md
```

```
+-- requirements.txt
+-- timeline.md
+-- src/
|   +-- config.py
|   +-- collect_papers.py
|   +-- rag_pipeline.py
+-- notebooks/
|   +-- rag_pipeline.ipynb  (12 cells)
+-- app/
|   +-- streamlit_app.py
+-- docs/
|   +-- architecture.md
+-- data/  (gitignored)
    +-- papers/
    +-- papers_metadata.json
    +-- faiss_index/
```

11. Bonus: Streamlit UI

An interactive web interface was built using Streamlit (app/streamlit_app.py). It provides a user-friendly way to query the RAG system with real-time results.

Features

Feature	Description
Dark Theme	Custom CSS with gradient header and styled components
Stats Dashboard	Shows indexed chunks, papers, embedding dims, top-k
Settings Sidebar	Top-k slider, pipeline config, sample queries
Query Input	Text input with placeholder and quick-select buttons
Answer Display	LLM-generated answer with markdown rendering
Source Panels	Expandable panels: paper title, score, chunk text
Full Text Toggle	Click to view entire chunk text for any source

Launch Command

```
streamlit run app/streamlit_app.py
```

The UI automatically loads the saved FAISS index from data/faiss_index/ and provides an intuitive interface for querying research papers. Users can adjust the number of retrieved sources (top-k) and explore the full text of each retrieved chunk.