Pedagogical Report

1. Teaching Philosophy

1.1 Target Audience

The intended learners for this module are graduate students enrolled in INFO 7390: Advanced Data Science and Architecture. These students generally possess:

- Intermediate Python programming experience
- Basic familiarity with machine learning workflows
- High-level awareness of LLMs and API usage
- Limited or no hands-on experience with Retrieval-Augmented Generation (RAG)

I assume learners understand foundational computational concepts—vectors, embeddings, and similarity metrics—but have not yet translated these concepts into a functional system. Many students are comfortable reading research papers and documentation, but have limited exposure to building modular, production-aligned systems from scratch. This module is designed to bridge that gap by providing both technical implementation and architectural reasoning.

The exercises are designed with the assumption that students learn best through **guided construction rather** than passive reading. The target profile values reproducible workflows, explanatory clarity, and progressive exposure to increasing complexity.

1.2 Learning Objectives

By the end of this module, students will be able to:

Conceptual Understanding

1. Explain the motivation behind Retrieval-Augmented Generation and its advantages over pure prompting and model fine-tuning.
2. Describe how embeddings convert text into vector representations and why vector similarity enables retrieval.
3. Identify key components and design tradeoffs in a RAG pipeline, including chunk size, overlap, embedding models, retrieval strategies, and prompt structure.

Technical Skills

4. Implement a complete, minimal RAG system using Python, Sentence Transformers, and the OpenAI API.
5. Load and preprocess unstructured text from PDFs using an extract–chunk–embed pipeline.
6. Construct an in-memory vector store and compute similarity scores using cosine similarity.
7. Retrieve top-k context passages and integrate them into a structured RAG prompt.
8. Execute grounded generation with an LLM and debug common retrieval or hallucination issues.

Analytical & Diagnostic Skills

9. Evaluate retrieval relevance through visualization and similarity analysis.
10. Diagnose system behavior, including empty retrievals, low-quality chunks, and threshold-induced filtering failures.
11. Reason about GIGO principles in the context of RAG (i.e., how upstream data quality influences downstream model performance).

Pedagogical Engagement

12. Complete structured exercises that reveal the impact of changing chunk size, document count, and similarity thresholds.
13. Reflect on how implementation choices influence system accuracy and stability.

1.3 Pedagogical Approach and Rationale

This instructional module adopts a **learning-by-building** philosophy grounded in constructionism and active learning theory. Instead of presenting RAG as an abstract architecture, students internalize each component by implementing it step-by-step in a Jupyter notebook. This creates an experiential link between theory and practice.

Guided Scaffolding

The notebook follows a scaffolded structure:

1. **Explain** – Introduces concepts such as embeddings, vector spaces, and the limitations of LLMs.
2. **Show** – Demonstrates each pipeline component with code that can be inspected and modified.
3. **Try** – Provides progressively open-ended exercises (chunking changes, multi-PDF behavior, thresholding) that encourage exploration and experimentation.

This pattern reduces cognitive load and aligns with the course’s emphasis on Botspeak, GIGO, and computational skepticism.

Modularized Architecture for Transparency

A key decision was to avoid hiding complexity inside external libraries.
Instead of using LangChain, LlamaIndex, or FAISS:

- Chunking is implemented manually
- Embeddings are computed through direct function calls
- Similarity search is performed via an explicit cosine similarity loop
- RAG prompting is assembled as a raw string template

This ensures that students see and reason about *every moving part* of the pipeline. Abstractions are introduced only after foundational understanding is established.

Visualization for Interpretability

Students often treat RAG as a black box. To counter this, the module includes:

- Similarity bar plots
- Threshold and cutoff visualizations
- Multi-document retrieval heatmaps
- Sorted similarity scatter plots

These plots translate abstract numerical concepts into intuitive visual patterns, supporting learners with visual reasoning preferences.

Diagnostic Reasoning and Computational Skepticism

A dedicated debugging section trains students to interrogate the system:

- Are chunks meaningful?
- Are the embeddings high quality?
- Are similarity scores reasonable?
- Is the LLM's answer grounded or hallucinated?

This aligns with the course’s philosophy of computational skepticism—students are taught to **verify rather than trust** model outputs.

Authentic, Real-World Context

RAG is not taught as a toy classroom example.
Students build a PDF-based retrieval system similar to real applications:

- Internal knowledge base assistants
- Customer support chatbots
- Research literature question-answering
- Document summarization and agent systems

This real-world relevance improves motivation and demonstrates practical relevance beyond the classroom.

2. <u>Concept Deep Dive</u>

2.1 Technical and Mathematical Foundations of Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is an architectural pattern that supplements a Large Language Model (LLM) with external, user-controlled knowledge sources to improve accuracy, reduce hallucinations, and enable domain-specific reasoning. RAG operates through two interconnected mathematical processes: (1) transforming text into vector embeddings and (2) performing similarity search to retrieve the most relevant contextual passages before LLM inference.

2.1.1 Embeddings and Vector Space Representations

The foundation of RAG lies in **text embeddings**, which are dense numerical vectors encoding semantic meaning. Formally, an embedding model is a function:

$$f : \mathscr{T} \rightarrow (R)\hat{}d$$

where

- $\mathscr{T}$ is the space of text sequences
- **d** is the embedding dimension (e.g., 384, 768, 1024)

Texts that are semantically similar map to vectors that are geometrically closer. In this project, embeddings are produced using a Sentence Transformers model (`all-MiniLM-L6-v2`), which performs mean pooling over transformer token embeddings to generate sentence-level semantic vectors.

2.1.2 Cosine Similarity as a Retrieval Metric

To determine which document chunks are most relevant to a user query, RAG uses **cosine similarity**, defined as:

$$\mathbf{sim(u,v) = u*v / ||u|| * ||v||}$$

This measure captures the *direction* of two vectors rather than their magnitude, making it robust for high-dimensional semantic spaces.

Given:

- **q** = embedding of the user query
- $\mathbf{c_1, c_2, \ldots, c_{□}}$ = embeddings of document chunks

RAG retrieves the top-k chunks that maximize:

$$\mathbf{sim(q,ci)}$$

Mathematically, this constitutes a **nearest-neighbor search** problem in a vector space.

Because this project uses an in-memory implementation, similarity is computed explicitly for each chunk. In production systems, highly optimized ANN (Approximate Nearest Neighbor) indices such as FAISS, ScaNN, or vector databases are used for scalable retrieval.

2.1.3 Chunking and Context Windows

LLMs cannot ingest unbounded text. Chunking converts lengthy documents into manageable segments:

$$\text{Text} = (w1,w2,\ldots,wn)$$

Chunks are produced by sliding a window of **chunk size** *s* words with **overlap** *o*:

$$\text{Chunkk} = (wk(s-o),\ldots,wk(s-o)+s)$$

This overlapping window helps preserve semantic continuity, preventing sentence breaks that hinder embedding quality. Chunk size introduces a tradeoff:

- Larger chunks → richer context, but less precise retrieval
- Smaller chunks → more precise retrieval, but may lose coherence

Experimentation in this project demonstrates how chunk size directly affects RAG answer quality.

2.1.4 Prompt Construction and Grounded Generation

Once relevant chunks are retrieved, they are injected into the LLM via a carefully structured RAG prompt. The LLM then generates an answer conditioned on both:

1. User question
2. Retrieved context

The project uses a deterministic instruction:

"Answer using only the information in the context. If not present, respond 'I don't know based on the provided documents.'"

This encourages **grounding**, reducing hallucinations by restricting the model's inference to retrieved evidence.

2.1.5 Threshold-Based Filtering

Similarity thresholds introduce another mathematical constraint:

Given threshold $\boldsymbol{\tau}$, retrieved chunks must satisfy:

$$\text{sim}(q,ci) \geq \tau$$

This forces the system to filter low-relevance chunks and supports debugging retrieval quality. Visualizations in the project show how thresholds alter retrieval distribution and can sometimes eliminate all chunks—a valuable teaching moment about retrieval failure modes.

2.2 Connection to INFO 7390 Course Themes

RAG is an ideal teaching topic because it intersects with several major course themes:

2.2.1 GIGO: Garbage In, Garbage Out

RAG strongly illustrates the GIGO principle:

- Poor PDF extraction → poor chunks
- Poor chunking → weak embeddings
- Weak embeddings → irrelevant retrieval

- Irrelevant retrieval → hallucinated answers

Every upstream choice influences downstream output. Students directly see this chain reaction by adjusting chunk size, overlap, similarity thresholds, and document quality.

RAG makes GIGO *visible* and *measurable*.

2.2.2 Botspeak and Prompt Engineering

Botspeak refers to designing systems that clearly communicate task intent to LLMs. RAG provides a perfect platform for this:

- The model must be instructed to use only the retrieved context.
- The structure of the prompt affects hallucination rates.

The separation of “Context:” and “Question:” enforces semantic boundaries.

Students learn how subtle differences in prompt framing impact stability and reliability.

2.2.3 Computational Skepticism

One of the course’s key values is questioning model output rather than accepting it at face value.

RAG reinforces this by requiring:

- Inspection of retrieved chunks
- Checking whether answers are grounded in context
- Identifying when thresholds remove all evidence
- Using visualizations to evaluate similarity distributions

By providing tools that expose the internal mechanics of retrieval, the module encourages students to adopt a skeptical, evidence-based mindset.

2.2.4 Architecture Thinking and Modular System Design

RAG aligns with the course’s emphasis on modular architectures. The project breaks the system into:

- PDF loader
- Text splitter
- Embedder
- Vector store
- Retriever
- Prompt builder

- LLM client

Each component is independently replaceable. This reinforces composability—one of the core architectural skills taught in INFO 7390.

2.3 Relationship to Real-World Data Science Workflows

RAG is no longer an academic curiosity; it is a foundational component of real-world AI systems. This module introduces students to a workflow that mirrors how industry teams build scalable retrieval-enhanced applications.

2.3.1 Enterprise Knowledge Assistants

Companies increasingly build "ChatGPT over Data" tools for:

- Internal documentation
- Research literature
- Policy compliance documents
- Customer support knowledge bases

This project gives students a direct blueprint for such systems.

2.3.2 Data Engineering and Preprocessing Pipelines

Loading PDFs, cleaning text, chunking, embedding, and caching outputs mirrors the ETL workflows common in data engineering roles.

RAG reinforces that **models do not work without pipelines**.

2.3.3 Retrieval Systems as Core Infrastructure

RAG teaches core retrieval engineering concepts similar to:

- Semantic search
- Nearest neighbor indexing
- Recommendation systems
- Metadata-based filtering

Many machine learning roles require fluency with these systems.

2.3.4 Hybrid AI Systems

Modern AI applications often combine:

- Traditional retrieval

- Embedding-based semantic search
- LLM reasoning

This hybrid approach is increasingly becoming the default architectural strategy in production systems such as:

- Copilot
- Google's Search Generative Experience
- Enterprise RAG agents
- Multimodal assistants

Students trained in this module gain exposure to an in-demand design pattern.

2.3.5 Debugging and Monitoring AI Systems

Real-world AI workflows require:

- Observability
- Monitoring retrieval quality
- Ensuring grounding
- Detecting drift in embeddings or data quality

The debugging section of the notebook serves as an introduction to monitoring the internal health of retrieval-driven systems.

3. Implementation Analysis

3.1 Architecture and Design Decisions

The implementation of the RAG Learning Lab follows a modular, transparent architecture designed to highlight the interplay between data preprocessing, retrieval, and LLM-based generation. The goal was not simply to produce a working system, but to craft an architecture that **reveals each decision point** to learners and enables experimentation.

3.1.1 Layered Pipeline Design

The pipeline consists of six distinct layers:

1. **Document Ingestion** – Loads one or multiple PDFs and extracts raw text.
2. **Chunking Layer** – Splits long-form text into manageable overlapping chunks.
3. **Embedding Layer** – Converts text chunks into vector representations.

4. **Vector Store Layer** – Stores embeddings and exposes a similarity-based retrieval interface.
5. **Retrieval Layer** – Performs nearest-neighbor search using cosine similarity.
6. **RAG Orchestration Layer** – Builds prompts and integrates the retrieval output into the LLM.

This layered structure mirrors the architecture of production-grade RAG systems while remaining lightweight enough for students to understand and modify. Each module is small, purpose-driven, and replaceable.

3.1.2 Simplicity Over Abstraction

A deliberate design choice was to **avoid heavy frameworks** such as LangChain, LlamaIndex, and Pinecone in the primary instructional notebook. While these tools are powerful, they hide crucial mechanisms behind layers of abstraction.

To support the course's emphasis on computational transparency and skepticism, core components were implemented manually:

- Chunking uses explicit sliding windows rather than library defaults.
- Cosine similarity is manually computed using NumPy rather than external vector stores.
- The vector store is a simple Python dataclass rather than a black-box service.
- Prompts are raw strings rather than templating engines.

This allows learners to understand:

- Why retrieval might fail,
- How embeddings interact with chunk structure,
- How thresholds influence system robustness, and
- Why prompt structure affect grounding?

Only after building the minimal version can students meaningfully appreciate higher-level abstractions.

3.1.3 Multi-Document Handling Strategy

Rather than concatenating all PDFs into a single long text block and attempting to infer chunk provenance afterward, the architecture treats each document independently during chunking. This produces:

- Cleaner document-label mapping,

- Better debugging visibility,
- More accurate multi-PDF retrieval visualizations.

The choice to assign `doc_labels` at chunk-creation time eliminates complexity and prevents potential infinite loops and misalignment bugs in size-based inference approaches.

3.1.4 Explicit Prompt Structure for Grounding

The RAG prompt was intentionally structured as:

```
Context:

<retrieved chunks>

Question: <user question>

Instructions:

- Answer only using the provided context.

- If the answer is not present, say:

  "I don't know based on the provided documents."
```

This explicit format aligns with Botspeak principles and reduces hallucination likelihood. It also encourages students to reason about prompt alignment—whether the model is conditioned correctly or if ambiguity remains.

3.2 Libraries and Tools Chosen and Why

The implementation uses a minimal toolset selected for:

- Transparency
- Pedagogical value
- Ease of installation
- Industry relevance

3.2.1 Sentence Transformers

Why:

- Strong pedagogical value — embeddings are visible and interpretable
- Industry standard — used widely in semantic search systems

- Lightweight and CPU-friendly
- Deterministic output enables reproducible experiments

The model `all-MiniLM-L6-v2` was chosen because it strikes a balance between performance and speed, allowing embedding of hundreds of chunks without GPU hardware.

3.2.2 pypdf

Why:

- Simple interface for PDF text extraction
- No heavy dependencies
- Allows students to inspect extracted text and detect issues such as ligatures, OCR failure, and formatting irregularities

This directly ties into the GIGO theme—poorly extracted text leads to poor chunks, poor embeddings, and poor retrieval.

3.2.3 OpenAI API (Chat Completions)

Why:

- Stable, predictable results
- Strong alignment with RAG tasks
- Minimal boilerplate
- Students already familiar with the API from earlier assignments
- High-quality grounded outputs

Using small, affordable models (e.g., `gpt-4.1-mini`) also keeps the cost manageable during experimentation.

3.2.4 NumPy for Similarity Calculations

Why:

- Enables manual implementation of cosine similarity
- Reinforces mathematical understanding
- Encourages students to reason about vector operations
- Avoids reliance on specialized ANN libraries before conceptual mastery

Learners can see the full retrieval logic and directly modify scoring behavior.

3.2.5 Matplotlib / Seaborn for Visualization

Why:

- Supports interpretability of retrieval behavior
- Histograms, bar plots, and scatter plots highlight cluster structure and separation
- Visual debugging helps students intuitively understand embedding quality and chunk relevance

This ties directly into computational skepticism.

3.3 Performance Considerations

Although the implementation is small-scale, several performance considerations were intentionally built into the design.

3.3.1 Chunk Size vs. Embedding Cost

Embedding cost grows linearly with:

$$\text{number of chunks} = n / s - o$$

where *n* is text length, *s* is chunk size, and *o* is overlap.

Students experiment directly with this tradeoff:

- Smaller chunks → higher retrieval precision but higher embedding cost
- Larger chunks → faster embedding but lower retrieval resolution

The architecture exposes chunk size and overlap as explicit tunable parameters.

3.3.2 Retrieval Scalability

The in-memory retrieval method is: O(n)

cosine similarity operations — perfectly sufficient for tens or hundreds of chunks, but not millions.

This design choice emphasizes pedagogy: students gain intuition before upgrading to scalable vector indices like FAISS, Milvus, or Pinecone.

3.3.3 LLM Latency and Prompt Length

The length of the RAG prompt grows with:

k × average chunk size

To ensure responsiveness:

- A default `k=5` was selected
- Chunk size kept reasonable (200 words)
- No aggressive concatenation of documents

These decisions keep inference latency low enough for interactive teaching contexts.

3.3.4 Threshold Tuning

Applying similarity thresholds introduces new performance dynamics:

- High threshold → possibly zero results → failure mode
- Low threshold → more chunks → increased prompt size & hallucination risk

Students observe these interactions in exercises and visualizations.

3.4 Edge Cases and Limitations

A realistic system must acknowledge where RAG can fail. These limitations are intentionally surfaced so learners understand constraints and develop diagnostic reasoning.

3.4.1 Poor PDF Extraction (OCR Issues)

If PDFs contain scanned images or poorly encoded text:

- `extract_text()` returns garbage or empty output
- Chunks become meaningless
- Retrieval collapses

Students detect this via the debugging suite, which previews extracted text before chunking.

3.4.2 Low-Quality Chunks

Chunking may fail when:

- Chunk size cuts across semantic boundaries
- Overlap is too small
- Documents mix tables, lists, and paragraphs

Low-quality chunks → low-quality embeddings → irrelevant retrieval.

3.4.3 Over-Retrieval and Hallucination

If many nearly-relevant chunks are returned:

- LLM may merge unrelated ideas
- Grounding becomes weaker
- Answers become overly general or subtly incorrect

Thresholding addresses this, but introduces new failure modes.

3.4.4 No Relevant Chunks Retrieved

When similarity scores are uniformly low:

- The model lacks grounding
- Students learn to recognize “I don’t know…” as a correct response
- Reinforces skepticism and model boundaries

3.4.5 Multi-Document Confusion

If documents have overlapping vocabulary but different topics:

- Recall may be high, but precision may suffer
- Students see a retrieval competition between documents
- Visualizations demonstrate cross-document score distribution

3.4.6 Scaling Constraints

This implementation is intentionally simple:

- No approximate nearest-neighbor search
- No metadata filters
- No reranking step
- No context window management beyond chunk concatenation

These limitations are pedagogically useful — students later appreciate why industrial RAG systems incorporate additional components.

4. Assessment & Effectiveness

4.1 Validating Student Understanding

Student understanding in this module is validated through a combination of **implementation-based assessment**, **diagnostic reasoning**, and **guided reflection**, rather than traditional exams or multiple-choice quizzes. This aligns with the course's emphasis on applied mastery and architectural thinking.

4.1.1 Progressive Hands-On Exercises

Understanding is primarily validated through three structured exercises embedded in the notebook:

- **Exercise 1: Chunk Size and Overlap Analysis**
 Students modify chunking parameters and observe how retrieval relevance and answer quality change. Successful completion demonstrates understanding of the relationship between chunk granularity, embedding quality, and retrieval precision.
- **Exercise 2: Multi-Document Retrieval**
 Learners extend the RAG pipeline to handle multiple PDFs and verify whether the system retrieves context from the correct document. This validates understanding of vector-space retrieval and corpus-wide similarity search.
- **Exercise 3: Similarity Thresholding**
 Students introduce a similarity threshold and analyze how overly strict or overly permissive thresholds affect system behavior. This exercise validates understanding of retrieval confidence, false positives, and failure modes.

Each exercise produces observable system behavior that can be directly inspected, making assessment transparent and unambiguous.

4.1.2 Retrieval Visualization as Evidence of Understanding

Students are required to generate and interpret similarity plots and retrieval distributions. Rather than simply producing an answer, learners must explain:

- Why certain chunks were retrieved
- Why do similarity scores cluster or separate
- Why retrieval fails under certain thresholds

The ability to interpret these plots demonstrates conceptual understanding beyond surface-level implementation.

4.1.3 Debugging-Based Assessment

A dedicated debugging cell encourages students to validate each pipeline component:

- PDF extraction

- Chunking quality
- Embedding dimensions
- Retrieval relevance
- Prompt construction
- LLM output grounding

Students demonstrate understanding by identifying and correcting errors rather than merely achieving correct outputs. This form of assessment closely mirrors real-world engineering workflows.

4.2 Common Challenges Students May Face

Despite careful scaffolding, students may encounter several predictable challenges. These challenges are intentionally surfaced to reinforce diagnostic thinking.

4.2.1 Confusion Between RAG and Fine-Tuning

A common misconception is that RAG modifies the model's internal knowledge. Students may initially believe the retrieval updates model weights.

The materials address this by:

- Explicitly separating embedding and generation steps
- Demonstrating that retrieval changes output without altering the model
- Reinforcing that RAG operates entirely at inference time

4.2.2 Poor Retrieval Due to Chunking Errors

Students may struggle with:

- Chunk sizes that are too large or too small
- Overlap values that fragment semantic meaning

These issues are surfaced through retrieval plots and chunk previews, encouraging experimentation rather than silent failure.

4.2.3 Hallucinated Answers Despite Retrieval

Students often assume retrieval guarantees correctness. However, hallucinations may still occur if:

- Chunks are weakly relevant
- Prompts are insufficiently constrained

The notebook directly addresses this through prompt inspection and grounding instructions, reinforcing computational skepticism.

4.2.4 Multi-Document Ambiguity

When documents share vocabulary but differ in meaning, retrieval may pull context from unintended sources. This challenge highlights the importance of metadata, document labeling, and retrieval evaluation—concepts intentionally discussed but not fully abstracted away.

4.2.5 Threshold Overfitting

Students may overuse similarity thresholds, resulting in zero retrieved chunks. This failure mode reinforces the idea that model silence ("I don't know") can be a correct outcome.

4.3 Addressing Different Learning Styles

The instructional design intentionally supports multiple learning styles to maximize effectiveness.

4.3.1 Visual Learners

Visual learners benefit from:

- Similarity bar plots
- Threshold cutoff lines
- Retrieval scatter plots
- Multi-document similarity distributions

These visualizations translate abstract vector-space concepts into intuitive patterns.

4.3.2 Hands-On Learners

Kinesthetic learners engage through:

- Code modification
- Parameter tuning
- Rebuilding pipelines
- Debugging exercises

The notebook encourages experimentation and iteration rather than static execution.

4.3.3 Analytical Learners

Students who prefer formal reasoning benefit from:

- Explicit cosine similarity equations
- Clear mapping from theory to implementation
- Step-by-step architectural breakdowns

The implementation avoids hidden abstractions, allowing analytical learners to reason about system internals.

4.3.4 Reflective Learners

Reflection is encouraged through:

- Debugging commentary
- Comparison of retrieval outcomes
- Written explanation prompts in the notebook

These learners benefit from articulating *why* the system behaves as it does.

4.4 Future Improvements and Extensions

While intentionally minimal, the module is designed to be extensible.

4.4.1 Technical Extensions

Future iterations could include:

- FAISS or ANN-based vector indices
- Metadata-aware retrieval
- Cross-encoder reranking
- Token-based chunking using model context windows
- Automatic evaluation metrics (faithfulness, relevance)

These extensions build naturally on the foundational architecture.

4.4.2 Pedagogical Enhancements

The teaching experience could be further improved by:

- Adding a Streamlit-based interface for interactive querying
- Including a comparative notebook using LangChain or LlamaIndex
- Introducing small benchmark datasets for formal evaluation

- Providing optional challenge assignments for advanced learners

4.4.3 Industry Alignment

Additional modules could explore:

- Enterprise-scale RAG
- Monitoring and observability
- Data drift and embedding updates
- Cost-aware retrieval strategies

These would further align the module with real-world AI engineering practices.