Technical Report: Generative AI and RAG Pipeline Implementation
Author: Daniel Muthama
Program: DATA and AI
Date: July 30, 2025

1. Introduction

This report documents the implementation of a Retrieval-Augmented Generation (RAG) pipeline to enhance question-answering systems using generative AI. The pipeline processes a PDF document, retrieves contextually relevant information, and generates precise answers using the **FLAN-T5 model. Key components include:**

- LangChain: For document loading and chunking.
- Sentence-Transformers: For embedding generation.
- FAISS: For efficient vector storage and retrieval.
- Transformers (Hugging Face): For generative text output.

Objective: Demonstrate how RAG improves answer quality over generic generative models by grounding responses in retrieved context.

2. Methodology

2.1 Environment Setup

Dependencies Installed:

```
!pip install numpy==1.26.4 faiss-cpu pydantic==2.9.2
!pip install sentence-transformers==2.2.2 transformers==4.41.1
!pip install langchain==0.1.13 pypdf==3.17.4
```

- Key Libraries:
- `faiss-cpu`: Optimized similarity search.
- `sentence-transformers`: Embeddings using `all-MiniLM-L6-v2`.
- `FLAN-T5`: Text-to-text generation model.

Critical Notes:
- Resolved version conflicts (e.g., `pydantic` and `langsmith`).
- Used retry logic (`tenacity`) to handle model download timeouts.

2.2 Pipeline Implementation

Step 1: PDF Loading and Chunking
- Function: `load_and_chunk_pdf()`
  - Downloads PDF from Google Drive if missing.
  - Splits text into 500-character chunks with 50-character overlap for context continuity.
  - Output: 11 chunks from 3 pages.

Step 2: Embeddings and Vector Store
- Embedding Model: `all-MiniLM-L6-v2` (lightweight, CPU-friendly).
- Vector Database: FAISS for fast nearest-neighbor search.
- Retry Logic: Automated retries for model downloads on failure.

Step 3: FLAN-T5 Initialization
- Model: `google/flan-t5-large` (3.13GB, downloaded with resume capability).
- Pipeline: Configured for CPU with `max_new_tokens=200` and sampling enabled.

Step 4: RAG Query Function

```
def query_rag(question, vectorstore, llm_pipeline, k=3):
    relevant_docs = vectorstore.retrieve(question)[:k]   Top 3 chunks
```

```
    context = "\n".join([doc.page_content for doc in relevant_docs])
    prompt = f"Answer using context:\n{context}\nQuestion: {question}\nAnswer:"
    return llm_pipeline(prompt, max_new_tokens=200)[0]['generated_text']
```

3. Results

3.1 RAG vs. Generic Output Comparison

Query: "Summarize the key points of this document in 100 words."

Approach	**Output**
RAG (Context-Aware)	**-** "Hinton shared critical insights about AI's future in his interview with the Data and AI course."
Generic (No Context)	**-** "There are a number of types of insurance that provide different coverage for a variety of purposes."

Key Insight:

- RAG provides domain-specific answers by leveraging retrieved context.
- Generic FLAN-T5 generates unrelated, template-like responses.

4. Challenges & Solutions

1. Dependency Conflicts:
  - Fixed by pinning versions (e.g., `numpy==1.26.4`).

2. Model Download Failures:
  - Implemented retry logic with exponential backoff.

3. Hardware Limitations:
  - Used CPU-compatible models (e.g., `all-MiniLM-L6-v2`).

5. Conclusion

- The RAG pipeline successfully combines retrieval and generation to improve answer accuracy.
- **Future Work:**
  - Fine-tune FLAN-T5 for domain-specific tasks.
  - Experiment with larger embedding models (e.g., `all-mpnet-base-v2`).

Repository: [GitHub Link] | **Document-Corpus:** [Google Drive Link] | **Colab:** [Colab Link]

Appendix: Code Snippets

Retry Logic for Robust Downloads

```
@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4, max=10))
def initialize_llm(model_name):
    return pipeline("text2text-generation", model=model_name, device=-1)
```

FAISS Vector Store Creation

```
vectorstore = FAISS.from_documents(chunks, embeddings)
```