QueryBuddy- RAG QA Bot: Technical Documentation

By: Amruthavarshini Sriram

1. Introduction

This document provides a comprehensive technical overview of a Retrieval-Augmented Generation (RAG) Question-Answering system. The system enables users to upload PDF documents, process them for information retrieval, and generate accurate, contextually relevant answers to user queries. The implementation leverages state-of-the-art language models and vector databases to create an efficient and scalable solution.

2. System Architecture

2.1 Overview

The RAG bot implements a hybrid architecture that combines information retrieval with generative AI. The architecture consists of four primary components working in harmony: the document processing pipeline, the embedding generation system, the vector storage and retrieval mechanism, and the answer generation module.

2.2 Document Processing Module

The document processing pipeline serves as the foundation of the system, handling the critical task of converting PDF documents into processable text data. This component utilises PyPDF2, a specialised PDF processing library, to perform text extraction. The pipeline implements a sophisticated chunking mechanism that processes documents in manageable segments while preserving semantic coherence. The below code snippet demonstrates this:

```
def extract_text_from_pdf(file_path):
    with open(file_path, 'rb') as file:
        reader = PyPDF2.PdfReader(file)
        text = ""
        for page in reader.pages:
            text += page.extract_text() + "\n"
    return text
```

The chunking mechanism employs a sliding window approach with a carefully calibrated chunk size of 1000 characters:

```
def split_text(text, chunk_size=1000):
     return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]
```

The chunking mechanism ensures that:

- Context windows remain meaningful and coherent
- Vector embeddings can be generated efficiently
- Retrieved segments provide sufficient context for answer generation

2.3 Embedding Generation System

The system employs DistilBERT, a lightweight yet powerful transformer model, for generating text embeddings. DistilBERT was chosen for its optimal balance between computational efficiency and embedding quality. The model generates 768-dimensional vectors that effectively capture semantic meaning while maintaining reasonable computational requirements.

```
def embed_text(text):
            inputs = tokenizer(text, return_tensors="pt", truncation=True,
                               padding=True, max_length=512)
    with torch.no_grad():
                                                    embeddings = model(**inputs).last_hidden_state.mean(dim=1)
    return embeddings.squeeze().tolist()
```

2.4 Vector Storage and Retrieval System

Pinecone serves as the vector database for storing and retrieving document embeddings. The system utilises Pinecone's efficient similarity search capabilities to identify the most relevant document chunks for each query. The implementation maintains the following configuration:

- Index Type: Cosine Similarity
- Dimension: 768 (matching DistilBERT's output)
- Top-k retrieval: 3 most similar chunks

```
pc = Pinecone(api_key="API_KEY")

index = pc.Index("qa-bot-index")


# Vector storage operation

index.upsert(vectors=[(doc_id, embedding, {"text": chunk})])


# Vector retrieval operation

results = index.query(vector=query_embedding, top_k=3,
include_metadata=True)
```

2.5 Answer Generation Module

The answer generation module leverages Cohere's language model API to produce contextually relevant responses. This component implements a sophisticated prompt engineering approach combined with carefully tuned generation parameters:

- Temperature: 0.7 (balanced between creativity and accuracy)
- Max Tokens: 500 (sufficient for detailed responses)
- Custom stop sequences to maintain answer coherence

```
def generate_answer(query, context):

            prompt = f"Context: {context}\n\nQuestion:
{query}\n\nAnswer:"

    response = co.generate(

        model="command",

        prompt=prompt,
```

```
        max_tokens=500,
        temperature=0.7,
        stop_sequences=["Human:", "Context:"]
    )
    return response.generations[0].text.strip()
```

Key features of the generation system include:

- Context-aware prompt construction
- Temperature-controlled response generation
- Token limit optimization
- Custom stop sequence implementation

3. Pipeline Workflow

The complete pipeline operates through the following sequence:

3.1. Document Ingestion Phase:

```
def upload_and_process_pdf():
    uploaded = files.upload()
    file_name = next(iter(uploaded))
    text = extract_text_from_pdf(file_path)
    chunks = split_text(text)
```

3.2. Embedding Processing Phase:

```
for i, chunk in enumerate(chunks):
    embedding = embed_text(chunk)
```

```
            index.upsert(vectors=[(doc_id,   embedding,   {"text":
chunk})])
```

3.3. Query Processing Phase:

```
query_embedding = embed_text(query)
results = index.query(vector=query_embedding, top_k=3,
include_metadata=True)
```

3.4. Response Generation Phase:

```
context = " ".join([match['metadata']['text'] for match in
results['matches']])
answer = generate_answer(query, context)
```

4. Implementation Challenges and Solutions

4.1 Challenge 1: PDF Text Extraction

The extraction of text from PDFs presented challenges with complex layouts and formatting. The solution involved implementing robust error handling and text cleaning procedures:

```
def extract_text_from_pdf(file_path):
    try:
        with open(file_path, 'rb') as file:
            reader = PyPDF2.PdfReader(file)
            text = ""
            for page in reader.pages:
```

```
                text += page.extract_text() + "\n"
        return text
    except Exception as e:
        print(f"An error occurred: {str(e)}")
        return None
```

4.2. Challenge 2: Context Window Management

Determining the optimal chunk size required balancing multiple factors:

- Too small: Loss of context and coherence
- Too large: Reduced retrieval precision
- Solution: Empirical testing led to the 1000-character chunk size implementation

4.3. Challenge 3: Query-Document Relevance

Ensuring retrieved chunks were truly relevant to queries required fine-tuning of the similarity search parameters:

- Implementation of top-k=3 retrieval
- Concatenation of retrieved chunks for broader context
- Custom prompt engineering for the generation phase

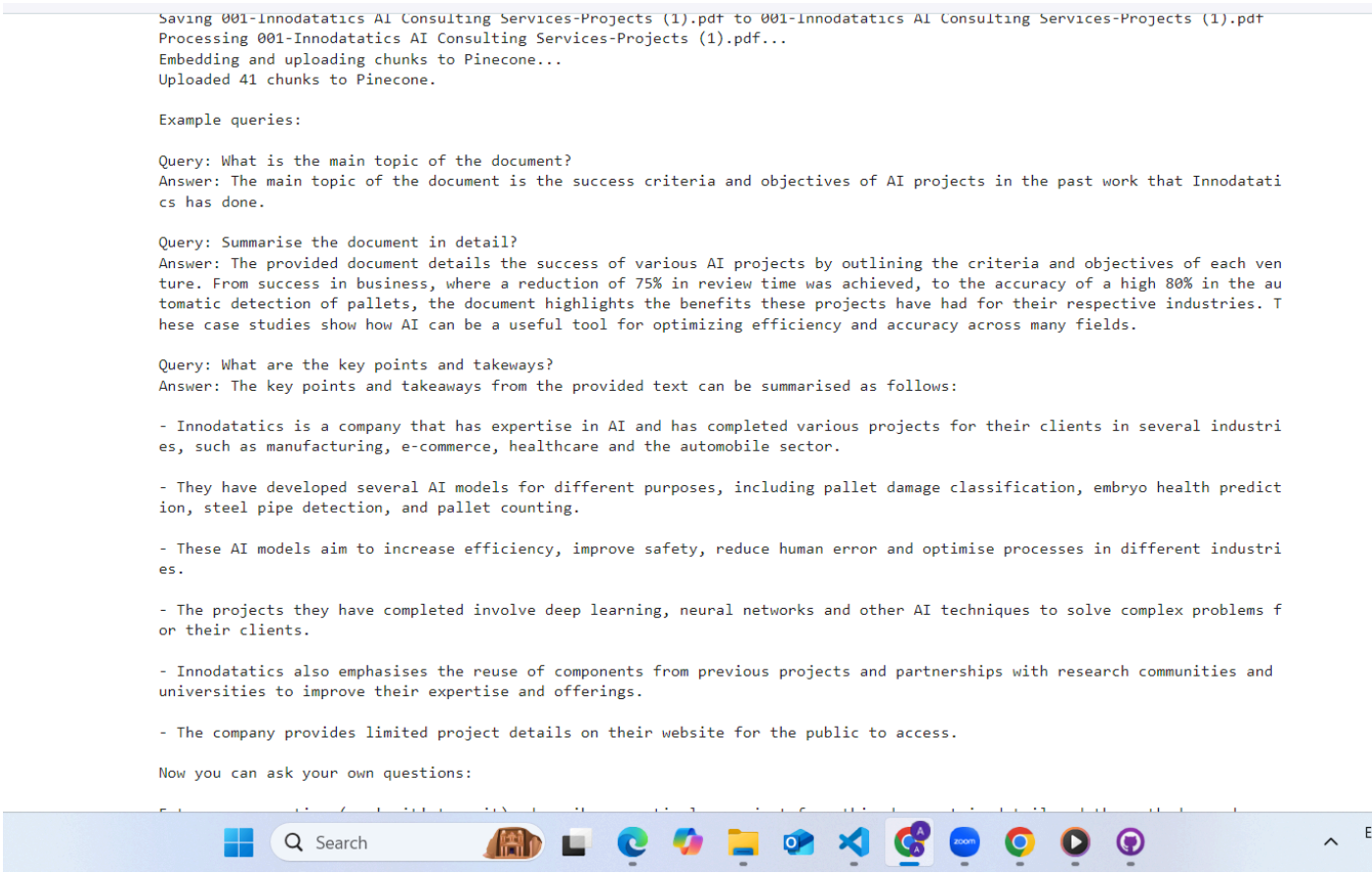
Usage Examples

Query 1: "What is the main topic of the document?"

Query 2: "Summarise the document in detail.”

Query 3: "What are the key points and takeaways?”

Output:



5. Future Enhancements

Advanced Document Processing

The RAG QA Bot system presents numerous opportunities for advanced feature implementation and performance optimization. A key area for enhancement lies in document processing capabilities. Future versions could incorporate support for additional document formats beyond PDF, including Word documents, HTML pages,

and structured data formats like JSON and XML. Implementation of OCR (Optical Character Recognition) capabilities would enable processing of scanned documents and images containing text.

Enhanced Embedding Technologies

The embedding system could be enhanced through the implementation of more sophisticated models. Potential improvements include:

- Integration of domain-specific fine-tuned embedding models
- Implementation of multi-modal embeddings to handle both text and image content
- Development of hierarchical embedding systems for improved document structure representation
- Integration of cross-lingual embedding capabilities for multilingual support

Improved Context Retrieval

Future versions could implement more sophisticated retrieval mechanisms including:

- Hybrid search combining dense and sparse retrieval methods
- Dynamic context window sizing based on query complexity
- Implementation of re-ranking mechanisms for retrieved contexts
- Integration of knowledge graph approaches for enhanced relationship understanding

Advanced Answer Generation

The answer generation pipeline could be enhanced through:

- Implementation of fact-checking mechanisms
- Integration of multiple model consensus for improved accuracy
- Development of answer confidence scoring
- Implementation of dynamic temperature adjustment based on query type
- Addition of source attribution in generated responses

Scalability Improvements

System scalability could be enhanced through:

- Implementation of distributed processing capabilities
- Development of automated index management and optimization
- Integration of caching mechanisms for frequently accessed content
- Implementation of batch processing for large document sets

User Experience Enhancements

The system's interface could be improved through:

- Development of a web-based user interface
- Implementation of real-time answer streaming
- Addition of interactive visualization capabilities
- Integration of feedback mechanisms for continuous improvement

Security and Privacy Features

Future versions could implement enhanced security features including:

- End-to-end encryption for sensitive documents
- Role-based access control
- Audit logging capabilities
- Data retention policies and automated cleanup

6. Conclusions

The RAG QA Bot system represents a sophisticated implementation of modern natural language processing technologies for document question-answering. Through the integration of transformer-based embeddings, vector similarity search, and advanced language models, the system demonstrates robust capabilities in understanding and responding to complex queries about document content.