# Agentic RAG Chatbot: Project Report & Implementation Guide

**Date:** October 1, 2025

## 1. Introduction

This document provides a complete technical walkthrough of the Agentic RAG (Retrieval-Augmented Generation) Chatbot project. The primary objective was to build a sophisticated, agent-based chatbot capable of answering user questions based on the content of multiple uploaded documents in various formats (PDF, DOCX, TXT, CSV, PPTX).

The architecture is designed to be robust and modular, centered around three distinct agents that communicate using a Model Context Protocol (MCP). The system is split into a powerful FastAPI backend that handles all the core logic and a user-friendly Streamlit frontend for interaction.

## 2. Architecture Overview

The system employs a decoupled, two-part architecture: a backend API and a frontend UI.

- **Backend (FastAPI):** This is the brain of the operation. It houses the three core agents and exposes API endpoints for the frontend to communicate with.
  - **IngestionAgent:** Responsible for processing uploaded documents. It parses various file formats, splits them into text chunks, generates embeddings using Google's Gemini models, and stores them in a persistent ChromaDB vector store.
  - **RetrievalAgent:** This agent is responsible for finding the most relevant information. It uses an advanced multi-query and re-ranking strategy to ensure the highest quality context is retrieved from the vector store in response to a user's query.
  - **LLMResponseAgent:** The final agent in the pipeline. It takes the retrieved context and the conversation history, and uses a powerful Large Language Model (Google's Gemini 2.5 Pro) to generate a coherent, context-aware, and conversational answer.
- **Frontend (Streamlit):** A web-based user interface that allows users to upload documents and interact with the chatbot in a conversational manner. It is completely decoupled from the backend, communicating with it via simple HTTP requests.

Model Context Protocol (MCP):
All communication between the agents in the backend is structured using MCP messages. The main.py file acts as a CoordinatorAgent, creating and routing these messages to ensure a clean, organized, and debuggable workflow.

## 3. Setup and Installation

Follow these steps to set up and run the project locally.

### 3.1. Prerequisites

- Python 3.8 or newer.
- pip for package management.

### 3.2. Project Structure

Organize your project in the following folder structure:

```
/agentic-rag-chatbot/
├── agentic_rag_backend/
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── ingestion_agent.py
│   │   ├── retrieval_agent.py
│   │   └── llm_response_agent.py
│   ├── core/
│   │   ├── __init__.py
│   │   └── models.py
│   ├── chroma_db/
│   ├── uploaded_files/
│   ├── .env
│   ├── main.py
│   └── requirements.txt
└── streamlit_ui/
    └── app.py
```

### 3.3. Environment Setup

1. **Create a Virtual Environment:**
   python -m venv venv
   source venv/bin/activate  # On Windows, use `venv\Scripts\activate`

2. Create requirements.txt:
   Create this file in agentic_rag_backend/ with the following content:
   fastapi
   uvicorn[standard]
   python-multipart
   pydantic
   langchain
   langchain-community
   langchain-google-genai
   python-dotenv
   chromadb

```
pypdf
python-docx
python-pptx
unstructured
pandas
streamlit
requests
google-generativeai
typing-extensions
```

3. **Install Libraries:**
   ```
   pip install -r agentic_rag_backend/requirements.txt
   ```

### 3.4. API Key Configuration

1. **Get a Google API Key:** Visit [Google AI Studio](#) to generate a free API key.
2. **Create a .env file** inside the agentic_rag_backend folder.
3. **Add your key** to the file:
   ```
   GOOGLE_API_KEY="YOUR_GOOGLE_API_KEY_HERE"
   ```

## 4. Backend Code Deep Dive (FastAPI)

This section details the code for each component of the FastAPI server.

### 4.1. Core Models (core/models.py)

This file defines the data structures for our API and the MCP messages using Pydantic for validation.

```
from pydantic import BaseModel, Field
from typing import List, Optional, Dict, Any
import uuid

# A single message in the chat history.
class HistoryMessage(BaseModel):
    role: str # 'user' or 'assistant'
    content: str

# Defines the structure for an incoming chat request from the UI.
class ChatRequest(BaseModel):
    query: str
    session_id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    chat_history: List[HistoryMessage] = []
```

```
# Defines the structure for the final chat response sent to the UI.
class ChatResponse(BaseModel):
    answer: str
    sources: List[str]
    session_id: str

# Defines the response for a successful file upload.
class UploadResponse(BaseModel):
    message: str
    filenames: List[str]
    session_id: str

# The flexible payload for our internal agent communication (MCP).
class MCPPayload(BaseModel):
    data: Dict[str, Any]
    query: Optional[str] = None
    context: Optional[List[str]] = None
    answer: Optional[str] = None
    sources: Optional[List[str]] = None
    chat_history: Optional[List[HistoryMessage]] = None

# The standard MCP message structure for inter-agent communication.
class MCPMessage(BaseModel):
    sender: str
    receiver: str
    type: str
    trace_id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    payload: MCPPayload
```

### 4.2. Ingestion Agent (agents/ingestion_agent.py)

This agent handles the processing of uploaded documents.

```
import os
from langchain_community.document_loaders import (
    PyPDFLoader, TextLoader, UnstructuredWordDocumentLoader,
    CSVLoader, UnstructuredPowerPointLoader
)
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_Genai import GoogleGenerativeAIEmbeddings
import chromadb

# Directory to store the persistent vector database.
```

```python
CHROMA_PERSIST_DIRECTORY = "chroma_db"

# Maps file extensions to their corresponding LangChain loader class.
LOADER_MAPPING = {
    ".pdf": PyPDFLoader, ".txt": TextLoader, ".md": TextLoader,
    ".docx": UnstructuredWordDocumentLoader, ".csv": CSVLoader,
    ".pptx": UnstructuredPowerPointLoader
}

def process_documents(file_paths: list[str], session_id: str):
    """Loads, splits, embeds, and stores documents in ChromaDB."""
    all_chunks = []
    # Loop through each uploaded file path.
    for file_path in file_paths:
        # Determine the file extension to select the correct loader.
        ext = "." + file_path.rsplit(".", 1)[-1].lower()
        if ext in LOADER_MAPPING:
            try:
                # Load the document using the appropriate loader.
                loader = LOADER_MAPPING[ext](file_path)
                documents = loader.load()

                # Split the document into smaller, manageable chunks.
                text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
                chunks = text_splitter.split_documents(documents)
                all_chunks.extend(chunks)

            except Exception as e:
                print(f"Error processing file {file_path}: {e}")

    if not all_chunks:
        return

    # Initialize the Google Gemini embedding model.
    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001")

    # Initialize a persistent ChromaDB client.
    client = chromadb.PersistentClient(path=CHROMA_PERSIST_DIRECTORY)

    # Create or get a collection named after the session_id to isolate data.
    collection = client.get_or_create_collection(name=session_id)
```

```
    # Add each chunk to the collection. ChromaDB handles the embedding process.
    for i, chunk in enumerate(all_chunks):
        collection.add(
            ids=[f"chunk_{i}"],
            documents=[chunk.page_content],
            metadatas=[chunk.metadata]
        )
```

### 4.3. Retrieval Agent (agents/retrieval_agent.py)

This agent uses a multi-query and re-ranking strategy to find the best context.

```
from langchain_Genai import GoogleGenerativeAIEmbeddings, ChatGoogleGenerativeAI
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import CommaSeparatedListOutputParser
import chromadb
from core.models import MCPMessage, MCPPayload
import os
from typing import List

CHROMA_PERSIST_DIRECTORY = "chroma_db"

# Prompt to generate alternative queries.
MULTI_QUERY_PROMPT_TEMPLATE = """...""" # Included in previous messages

# Prompt to re-rank retrieved documents.
RERANK_PROMPT_TEMPLATE = """...""" # Included in previous messages

def _generate_search_queries(query: str) -> List[str]:
    """Uses an LLM to generate alternative search queries."""
    # This function creates multiple versions of the user's question.
    # ... (Implementation from previous message) ...

def _rerank_documents(query: str, documents: List[str]) -> List[str]:
    """Uses an LLM to re-rank documents for relevance."""
    # This function takes a large set of documents and finds the top 5
    # most relevant ones for the specific question.
    # ... (Implementation from previous message) ...

def retrieve_context(message: MCPMessage) -> MCPMessage:
    """Handles retrieval by generating queries, searching, and re-ranking."""
    session_id = message.payload.data.get("session_id")
    query = message.payload.query
```

```
    client = chromadb.PersistentClient(path=CHROMA_PERSIST_DIRECTORY)
    collection = client.get_collection(name=session_id)

    # Step 1: Generate multiple queries to cast a wider net.
    search_queries = _generate_search_queries(query)

    # Step 2: Retrieve documents for all generated queries.
    all_results_docs = []
    # ... (Logic to query ChromaDB for each search term and collect unique docs) ...

    # Step 3: Re-rank the combined set of documents to find the best ones.
    reranked_chunks = _rerank_documents(query=query,
documents=initial_chunks_with_metadata)

    # Prepare the MCP response payload with the high-quality context.
    response_payload = MCPPayload(
        data=message.payload.data,
        query=query,
        context=reranked_chunks
    )

    return MCPMessage(
        sender="RetrievalAgent", receiver=message.sender,
        type="CONTEXT_RESPONSE", trace_id=message.trace_id,
        payload=response_payload
    )
```

### 4.4. LLM Response Agent (agents/llm_response_agent.py)

This agent generates the final conversational answer.

```
from langchain_core.prompts import PromptTemplate
from langchain_Genai import ChatGoogleGenerativeAI
from core.models import MCPMessage, MCPPayload, HistoryMessage
from typing import List

# Prompt to make a follow-up question understandable on its own.
CONDENSE_QUESTION_PROMPT = """...""" # Included in previous messages

# Main prompt for generating the final answer.
RAG_PROMPT_TEMPLATE = """...""" # Included in previous messages
```

```
def _format_chat_history(chat_history: List[HistoryMessage]) -> str:
    # Helper to format history for the LLM.
    # ...

def condense_question(message: MCPMessage) -> str:
    """Creates a standalone question from the conversation history."""
    # This ensures that retrieval works well even for follow-up questions
    # like "what about the second one?".
    # ... (Implementation from previous message) ...

def generate_response(message: MCPMessage) -> MCPMessage:
    """Generates the final answer using the LLM."""
    query = message.payload.query
    context = message.payload.context
    chat_history = message.payload.chat_history

    # Format the context and history into strings.
    context_str = "\n---\n".join(context)
    history_str = _format_chat_history(chat_history)

    # Initialize the powerful Gemini 2.5 Pro model.
    prompt = PromptTemplate.from_template(RAG_PROMPT_TEMPLATE)
    llm = ChatGoogleGenerativeAI(model="models/gemini-2.5-pro", temperature=0.3)

    rag_chain = prompt | llm

    # Invoke the LLM to get the final answer.
    answer = rag_chain.invoke({
        "context": context_str, "chat_history": history_str, "question": query
    }).content

    # Prepare the final MCP response.
    response_payload = MCPPayload(
        # ...
        answer=answer,
        sources=[chunk[:100] + "..." for chunk in context]
    )

    return MCPMessage(
        sender="LLMResponseAgent", receiver=message.sender,
        type="FINAL_RESPONSE", trace_id=message.trace_id,
        payload=response_payload
    )
```

### 4.5. Main App / Coordinator (main.py)

This file sets up the FastAPI server and acts as the CoordinatorAgent, managing the MCP message flow.

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from typing import List
from core.models import ChatRequest, ChatResponse, UploadResponse, MCPMessage,
MCPPayload
from agents.ingestion_agent import process_documents
from agents.retrieval_agent import retrieve_context
from agents.llm_response_agent import generate_response, condense_question
import os
from dotenv import load_dotenv

# Load the GOOGLE_API_KEY from the .env file.
load_dotenv()

# Initialize the FastAPI application.
app = FastAPI(title="Agentic RAG Chatbot API", version="1.0.0")

UPLOAD_DIRECTORY = "./uploaded_files"
os.makedirs(UPLOAD_DIRECTORY, exist_ok=True)

# Endpoint for uploading and processing documents.
@app.post("/upload", response_model=UploadResponse)
async def upload_documents_endpoint(session_id: str, files: List[UploadFile] = File(...)):
    # ... (Saves files locally) ...
    # Triggers the IngestionAgent to process the saved files.
    process_documents(file_paths=saved_files, session_id=session_id)
    # ...

# Endpoint for the chat functionality.
@app.post("/chat", response_model=ChatResponse)
async def chat_with_documents_endpoint(request: ChatRequest):
    # This endpoint acts as the CoordinatorAgent.

    # 1. Create a standalone question based on chat history.
    condense_request_payload = MCPPayload(
        query=request.query, chat_history=request.chat_history, data={}
    )
    condense_request_msg = MCPMessage(
```

```
    sender="CoordinatorAgent", receiver="LLMResponseAgent",
    type="CONDENSE_REQUEST", payload=condense_request_payload
)
standalone_question = condense_question(condense_request_msg)

# 2. Send MCP message to RetrievalAgent.
retrieval_request_payload = MCPPayload(
    data={"session_id": request.session_id}, query=standalone_question
)
retrieval_request_msg = MCPMessage(
    sender="CoordinatorAgent", receiver="RetrievalAgent",
    type="RETRIEVAL_REQUEST", payload=retrieval_request_payload
)
context_response_msg = retrieve_context(retrieval_request_msg)

# 3. Send MCP message to LLMResponseAgent to get the final answer.
llm_request_payload = MCPPayload(
    data={}, query=standalone_question,
    context=context_response_msg.payload.context,
    chat_history=request.chat_history
)
llm_request_msg = MCPMessage(
    sender="CoordinatorAgent", receiver="LLMResponseAgent",
    type="GENERATION_REQUEST", payload=llm_request_payload
)
final_response_msg = generate_response(llm_request_msg)

# 4. Return the final answer to the UI.
return ChatResponse(
    answer=final_response_msg.payload.answer,
    sources=final_response_msg.payload.sources,
    session_id=request.session_id
)
```

## 5. Frontend Code Deep Dive (Streamlit)

This section explains the user interface code.

### 5.1. Streamlit App (streamlit_ui/app.py)

```
import streamlit as st
import requests
import uuid
```

```python
# URL of our running FastAPI backend.
BACKEND_URL = "[http://127.0.0.1:8000](http://127.0.0.1:8000)"

def initialize_session_state():
    """Initializes session variables to persist across reruns."""
    # A unique ID for the entire chat session.
    if "session_id" not in st.session_state:
        st.session_state.session_id = str(uuid.uuid4())
    # Stores the list of chat messages (user and assistant).
    if "messages" not in st.session_state:
        st.session_state.messages = []
    # Stores the names of successfully uploaded files.
    if "uploaded_files" not in st.session_state:
        st.session_state.uploaded_files = []

# --- Main App Logic ---
st.set_page_config(page_title="Agentic RAG Chatbot", layout="wide")
st.title("📄 Agentic RAG Chatbot")

initialize_session_state()

# Sidebar for file uploading.
with st.sidebar:
    st.header("Upload Documents")
    uploaded_files = st.file_uploader(
        "Choose files", accept_multiple_files=True
    )

    if uploaded_files and st.button("Process Documents"):
        with st.spinner("Processing documents..."):
            # Prepare files for sending to the backend API.
            files_to_upload = [("files", (f.name, f.getvalue(), f.type)) for f in uploaded_files]

            # Make a POST request to the /upload endpoint.
            response = requests.post(
                f"{BACKEND_URL}/upload?session_id={st.session_state.session_id}",
                files=files_to_upload
            )
            # ... (Handles success and error messages)

# Main chat interface.
st.header("Chat with your Documents")
```

```python
# Display all previous messages stored in the session state.
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])
        # ... (Displays sources in an expander)

# Get new user input.
if prompt := st.chat_input("Ask a question..."):
    # Add user message to the UI and session state.
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    # Prepare payload for the backend /chat endpoint.
    chat_payload = {
        "query": prompt,
        "session_id": st.session_state.session_id,
        "chat_history": [
            {"role": msg["role"], "content": msg["content"]}
            for msg in st.session_state.messages[:-1]
        ]
    }

    # Make a POST request to the /chat endpoint.
    response = requests.post(f"{BACKEND_URL}/chat", json=chat_payload)

    # Display the assistant's response.
    # ... (Handles displaying the answer, sources, and errors)
```

## 6. Running the Application

You need to run the backend and frontend in two separate terminals.

**Terminal 1: Run the Backend API**

```bash
# Navigate to the backend folder
cd agentic_rag_backend

# Activate the virtual environment
source ../venv/bin/activate

# Start the server
```

```
uvicorn main:app --reload
```

**Terminal 2: Run the Streamlit UI**

```
# Navigate to the frontend folder
cd streamlit_ui

# Activate the virtual environment
source ../venv/bin/activate

# Start the Streamlit app
streamlit run app.py
```

A new tab will open in your browser at http://localhost:8501 with the running application.

## 7. Conclusion

This project successfully implements a powerful, multi-document Agentic RAG chatbot. It meets all the core requirements of the initial problem statement, including a three-agent architecture, MCP-based communication, support for diverse document formats, and a conversational, multi-turn UI. The advanced retrieval strategy ensures high-quality, relevant answers, making the chatbot a robust and practical tool for document-based question answering.