# 🔍 Smart Research Assistant

An AI-powered document analysis tool that helps you interact with research papers, reports, and documents through intelligent questioning and comprehension testing.

## ✨ Features

- 📄 **Document Processing**: Upload PDF and TXT files for analysis
- 🤔 **Intelligent Q&A**: Ask questions about your documents with contextual answers
- 🧠 **Challenge Mode**: Test your understanding with AI-generated questions
- 💭 **Conversation Memory**: Maintains context across multiple questions
- 📊 **Answer Evaluation**: Detailed feedback on challenge question responses
- 🎯 **Source References**: Answers include relevant document snippets with page numbers
- 📱 **Modern UI**: Responsive web interface with dark mode support

## 🏗️ Architecture

### System Overview

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Streamlit UI  │───▶│   FastAPI       │───▶│   LangChain     │
│   (Frontend)    │    │   (Backend)     │    │   (Processing)  │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         │                      │                      │
         │                      │                      │
         ▼                      ▼                      ▼
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   User Session  │    │   Document      │    │   Vector Store  │
│   Management    │    │   Processing    │    │   (FAISS)       │
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

### Technology Stack

**Backend Components:**

- **FastAPI**: RESTful API server

- **LangChain**: Document processing and LLM integration
- **LangGraph**: Workflow management and reasoning chains
- **Google Gemini:** AI model for text generation and embeddings
- **FAISS**: Vector database for semantic search
- **PyPDF2**: PDF text extraction

**Frontend Components:**

- **Streamlit**: Web application framework
- **Custom CSS**: Modern UI styling with responsive design
- **Session State Management**: Conversation history and user context

## 📁 Project Structure

```
smart-research-assistant/
├── src/
│   ├── __init__.py        # Marks src as a Python package
│   ├── models.py          # Pydantic models for API requests/responses and internal
data structures
│   ├── workflow.py        # Core AI workflow, document processing logic, and
LangGraph setup
│   └── prompts.py         # Centralized collection of all LLM prompts and templates
├── main.py                # FastAPI backend server application entry point
├── streamlit_app.py       # Streamlit frontend application entry point
├── requirements.txt       # Lists all Python dependencies for pip installation
├── .env.example           # Template for environment variables (e.g., API keys)
├── .env                   # Actual environment variables (create this file from
.env.example)
├── pyproject.toml         # Project configuration for UV, including dependencies
└── README.md              # This comprehensive documentation file
```

### Key Components Explained

#### 1. main.py - FastAPI Backend

This file serves as the main entry point for the backend API.

- **Session Management**: It uses a global dictionary (sessions) to store active user sessions, holding document content, vector stores, conversation history, and challenge questions.
- **API Endpoints**: Defines RESTful endpoints for:
    - /upload: Handles document uploads (PDF/TXT), processes them, generates a

    summary and challenge questions, and initializes a session.
  - /ask: Processes user questions against the uploaded document, leveraging the LangGraph workflow.
  - /challenge/answer: Evaluates user answers to challenge questions.
  - /session/{session_id}/history: Retrieves the full conversation history and document summary for a given session.
  - / and /health: Basic root and health check endpoints.
- **CORS Configuration**: Configures Cross-Origin Resource Sharing to allow the Streamlit frontend to communicate with the FastAPI backend.
- **Error Handling**: Includes try-except blocks and HTTPException to gracefully handle errors during file processing, API calls, and LLM interactions.

**2. streamlit_app.py - Frontend Interface**

This file contains the Streamlit code for the user interface.

- **Page Configuration**: Sets up the Streamlit page title, icon, layout, and sidebar state.
- **Custom CSS**: Applies a modern, responsive design with support for dark mode, enhancing the user experience.
- **Session State**: Utilizes st.session_state to maintain application state across user interactions, including session_id, document_uploaded status, summary, challenge_questions, and conversation_history.
- **Interaction Modes**: Renders different sections based on st.session_state.current_mode:
  - upload: For uploading new documents.
  - summary: Displays the document summary and allows mode selection.
  - ask: Provides an interface for free-form question answering, displaying conversation history and supporting evidence.
  - challenge: Presents AI-generated challenge questions and allows users to submit answers for evaluation.
- **API Integration**: Makes requests calls to the FastAPI backend to perform document upload, ask questions, and submit challenge answers.

**3. src/workflow.py - AI Processing Engine**

This is the core intelligence hub of the application.

- **Document Processing (DocumentProcessor class)**:
  - extract_text_from_pdf/extract_text_from_txt: Handles text extraction from PDF and TXT files, including robust Unicode cleaning and page number tracking.
  - create_vector_store: Splits document text into chunks, creates vector embeddings using GoogleGenerativeAIEmbeddings, and stores them in a

FAISS vector database for efficient semantic search.

- **Smart Assistant (SmartAssistant class)**:
    - create_workflow: Defines the LangGraph state machine for the question-answering process. This workflow includes steps for:
        - retrieve_context: Fetches relevant document chunks using vector similarity search.
        - grade_context: Uses an LLM to determine if the retrieved context is relevant to the question.
        - generate_answer: Generates a structured answer (including source snippets) if the context is relevant.
        - fallback_answer: Provides a generic response if the context is not relevant.
    - summarize_document: Generates a concise summary of the uploaded document using an LLM.
    - generate_challenge_questions: Creates logic-based challenge questions from the document content using an LLM.
    - evaluate_answer: Evaluates user answers to challenge questions, providing a score, feedback, document reference, and suggestions using an LLM.
- **LLM Integration**: Initializes ChatGoogleGenerativeAI for text generation and GoogleGenerativeAIEmbeddings for creating document embeddings.
- **Utility Functions**: Includes safe_temp_file for secure temporary file handling and clean_text for robust text cleaning.

### 4. src/models.py - Data Models

This file defines the data structures used throughout the application, primarily using Pydantic for data validation and serialization/deserialization.

- **Request/Response Models**: Defines the expected structure for API requests (e.g., QuestionRequest, ChallengeAnswer) and responses (e.g., UploadResponse, QuestionResponse, EvaluationResponse).
- **Session State**: Uses TypedDict for SessionState to define the structure of the data stored per user session in the backend.
- **Data Structures**: Includes models for SourceSnippet (to highlight supporting evidence), ConversationEntry (for chat history), ChallengeQuestion, and AnswerEvaluation.

### 5. src/prompts.py - LLM Prompts

This file centralizes all the prompts used for interacting with the Google Gemini LLM.

- **Structured Prompts**: Contains well-defined prompts for various tasks:
    - SUMMARIZER_PROMPT: For generating document summaries.
    - QA_PROMPT: For answering free-form questions and extracting source

    snippets.
    - CONTEXT_GRADING_PROMPT: For evaluating the relevance of retrieved document context.
    - CHALLENGE_PROMPT: For generating challenge questions.
    - EVALUATION_PROMPT: For evaluating user answers to challenge questions.
- **Fallback Questions**: Provides a set of default challenge questions if the LLM fails to generate them.

## 🚀 Setup Instructions

### Prerequisites

- Python 3.8 or higher
- UV package manager (highly recommended for speed and efficiency) or pip
- A Google Gemini API key

### 1. Clone the Repository

First, clone the project repository to your local machine:

```
git clone <repository-url> # Replace <repository-url> with your actual repository URL
cd smart-research-assistant
```

### 2. Initialize Project with UV (Recommended)

If you don't have uv installed, you can install it via pip:

```
pip install uv
```

Now, initialize the project and install dependencies using uv:

```
# Initialize the project (creates a virtual environment and pyproject.toml if not present)
uv init

# Install dependencies listed in requirements.txt
uv add -r requirements.txt
```

This will create a virtual environment (e.g., .venv) and install all necessary packages into it.

### 3. Environment Configuration

The application requires a Google Gemini API key to interact with the AI models.
Create a .env file in the root directory of your project by copying the example:

```
cp .env.example .env
```

Now, open the newly created .env file and add your Google Gemini API key:

```
GOOGLE_API_KEY=your_gemini_api_key_here
```

**Important:** Replace your_gemini_api_key_here with your actual API key.

### 4. Get Google Gemini API Key

If you don't have a Google Gemini API key, follow these steps:

1. Visit Google AI Studio
2. Sign in with your Google account.
3. Click "Create API Key in new project" or "Get API Key" to generate a new key.
4. Copy the generated API key and paste it into your .env file as shown in step 3.

### 5. Run the Application

You need to run both the backend FastAPI server and the frontend Streamlit application.

#### Backend Server

Open your terminal in the project's root directory and run:

```
uv run fastapi dev main.py
```

This command starts the FastAPI server in development mode, which means it will automatically reload if you make changes to the code. The API will be available at http://localhost:8000.

#### Frontend Application

Open a **new terminal window** in the project's root directory and run:

```
uv run streamlit run streamlit_app.py
```

This command starts the Streamlit web application. The web interface will typically open in your default browser at http://localhost:8501.

### Alternative Setup (without UV)

If you prefer not to use uv, you can use pip and venv directly:

```
# Create a virtual environment
python -m venv venv

# Activate the virtual environment
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies from requirements.txt
pip install -r requirements.txt

# Run the backend server
python -m uvicorn main:app --reload --port 8000

# In a new terminal, activate venv and run the frontend application
source venv/bin/activate  # On Windows: venv\Scripts\activate
streamlit run streamlit_app.py
```

## 🔄 How It Works

This section provides a deeper dive into the internal workings of the Smart Research Assistant.

### Document Processing Flow

When a user uploads a document, the following sequence of operations occurs:

1. **Upload**: The Streamlit frontend sends the PDF or TXT file to the FastAPI backend via the /upload endpoint.
2. **Text Extraction**: Depending on the file type, PyPDF2 (for PDFs) or standard file reading (for TXT) is used to extract the raw text content from the document. Page numbers are also extracted and associated with the text.
3. **Text Cleaning**: The extracted text undergoes a cleaning process (clean_text function in workflow.py) to remove problematic Unicode characters and normalize whitespace, ensuring better quality input for the LLM and embeddings.
4. **Chunking**: The cleaned document text is split into smaller, manageable chunks using RecursiveCharacterTextSplitter. This is crucial for handling large documents

and fitting content within LLM context windows. Page number metadata is preserved during this process.

5. **Vectorization**: Each text chunk is then converted into a numerical vector (an "embedding") using GoogleGenerativeAIEmbeddings. These embeddings capture the semantic meaning of the text.
6. **Storage**: The generated embeddings, along with their corresponding text chunks and metadata (like page numbers), are stored in a FAISS (Facebook AI Similarity Search) vector store. This allows for rapid and efficient semantic similarity searches.

**Question Answering Workflow (LangGraph)**

The core reasoning for answering user questions is orchestrated by a LangGraph state machine defined in src/workflow.py. This ensures a structured and robust process:

```
┌─────────────────────┐
│ User Question    │
└──────────┬──────────┘
       │
       ▼
┌─────────────────────┐
│ Retrieve Context│ ← Vector similarity search in FAISS to find relevant document
chunks.
└──────────┬──────────┘
       │
       ▼
┌─────────────────────┐
│ Grade Relevance │ ← A dedicated LLM call evaluates if the retrieved context is
sufficient to answer the question.
└──────────┬──────────┘
       │
   ┌────▼────┐
   │ Relevant? │ (Conditional Edge: "yes" or "no")
   └────┬────┘
  Yes  │   No
       │
  ┌──────▼──────┐  ┌───────────────┐
  │ Generate  │   │ Fallback   │
  │ Answer    │   │ Response    │ (If context is not relevant, a generic apology is
returned.)
 └───────┬───────┘   └──────────────┘
         │
         ▼
┌──────────────────┐
│ Structured    │ (The LLM generates a JSON response with the answer and
supporting source snippets.)
│ Response      │
└──────────────────┘
```

This workflow ensures that answers are always grounded in the document and that the system can gracefully handle cases where relevant information is not found.

**Challenge Mode Process**

The "Challenge Me" mode provides an interactive learning experience:

1. **Question Generation**: After document upload, the SmartAssistant uses a specialized LLM prompt (CHALLENGE_PROMPT) to analyze the document and generate exactly three logic-based or comprehension-focused questions. These questions are designed to test higher-order thinking skills and are strictly answerable from the document content.
2. **Difficulty Levels**: The generated questions include a difficulty attribute (e.g., "medium", "hard") to give users an idea of their complexity.
3. **Answer Evaluation**: When a user submits an answer to a challenge question, the evaluate_answer method is called. This method sends the user's answer, the original question, the expected answer (generated by the LLM during question creation), and relevant document context to another LLM call using the EVALUATION_PROMPT.
4. **Feedback Loop**: The LLM provides a comprehensive evaluation, including:
   - A numerical score (out of 100).
   - Detailed feedback on the accuracy, reasoning quality, and document alignment of the user's answer.
   - A document_reference indicating where the correct information can be found.
   - suggestions for improvement. This structured feedback aims to promote deeper learning and understanding.

## 📊 API Endpoints

The FastAPI backend exposes the following RESTful API endpoints:

**Document Upload**

- **Endpoint**: POST /upload
- **Description**: Uploads a PDF or TXT document for processing.
- **Content-Type**: multipart/form-data
- **Request Body**:
    - file: The document file to upload.
- **Response (200 OK)**: UploadResponse (JSON)

```
{
 "session_id": "string",
 "summary": "string",
 "challenge_questions": [
   {
     "id": "string",
     "question": "string",
     "expected_answer": "string",
     "difficulty": "string"
   }
 ],
 "status": "success"
}
```

**Ask Question**

- **Endpoint**: POST /ask
- **Description**: Asks a free-form question about the uploaded document.
- **Content-Type**: application/json
- **Request Body**: QuestionRequest (JSON)

```
{
 "question": "What is the main argument of this document?",
 "session_id": "your_session_uuid"
}
```

- **Response (200 OK)**: QuestionResponse (JSON)

```
{
 "answer": "string",
 "session_id": "string",
 "source_snippets": [
    {
      "page_number": "string",
      "content": "string"
    }
  ]
}
```

### Submit Challenge Answer

- **Endpoint**: POST /challenge/answer
- **Description**: Submits an answer to a generated challenge question for evaluation.
- **Content-Type**: application/json
- **Request Body**: ChallengeAnswer (JSON)

```
{
  "question_id": "id_of_the_challenge_question",
  "answer": "Your detailed answer to the challenge question.",
  "session_id": "your_session_uuid"
}
```

- **Response (200 OK)**: EvaluationResponse (JSON)

```
{
  "evaluation": {
    "score": 85,
    "feedback": "Your answer is accurate and well-supported...",
    "document_reference": "Page 5, Paragraph 2",
    "suggestions": "Consider elaborating on..."
  },
  "question": "string"
}
```

### Get Conversation History

- **Endpoint**: GET /session/{session_id}/history
- **Description**: Retrieves the complete conversation history and document summary for a specific session.
- **Response (200 OK)**: ConversationHistoryResponse (JSON)

```
{
  "history": [
    {
      "question": "string",
      "answer": "string",
      "source_snippets": [
        {
          "page_number": "string",
          "content": "string"
        }
      ],
      "timestamp": "string"
    }
  ],
  "summary": "string"
}
```

## 🎯 Usage Examples

### 1. Basic Question & Answer Flow

```
import requests
import json

API_BASE_URL = "http://localhost:8000" # Ensure your backend is running

# Step 1: Upload a document
# Assuming you have a PDF file named 'sample_document.pdf'
with open("Intern Task GenAI.pdf", "rb") as f:
    files = {"file": ("Intern Task GenAI.pdf", f.read(), "application/pdf")}
    upload_response = requests.post(f"{API_BASE_URL}/upload", files=files)

if upload_response.status_code == 200:
    upload_data = upload_response.json()
    session_id = upload_data["session_id"]
    print(f"Document uploaded. Session ID: {session_id}")
    print(f"Summary: {upload_data['summary']}")

    # Step 2: Ask a question
    question_payload = {
        "question": "What are the functional requirements for this assistant?",
        "session_id": session_id
    }
    ask_response = requests.post(f"{API_BASE_URL}/ask", json=question_payload)

    if ask_response.status_code == 200:
        ask_data = ask_response.json()
        print(f"\nAnswer: {ask_data['answer']}")
        print("Source Snippets:")
        for snippet in ask_data['source_snippets']:
            print(f"  - Page {snippet['page_number']}: \"{snippet['content']}\"")
    else:
        print(f"Error asking question: {ask_response.status_code} - {ask_response.text}")
else:
    print(f"Error uploading document: {upload_response.status_code} -
{upload_response.text}")
```

**2. Challenge Mode Interaction**

```
import requests
import json

API_BASE_URL = "http://localhost:8000" # Ensure your backend is running

# Assume a document has already been uploaded and you have a session_id
# For demonstration, let's re-upload to get challenge questions
with open("Intern Task GenAI.pdf", "rb") as f:
    files = {"file": ("Intern Task GenAI.pdf", f.read(), "application/pdf")}
    upload_response = requests.post(f"{API_BASE_URL}/upload", files=files)

if upload_response.status_code == 200:
    upload_data = upload_response.json()
    session_id = upload_data["session_id"]
    challenge_questions = upload_data["challenge_questions"]
    print(f"Session ID: {session_id}")
    print("\nGenerated Challenge Questions:")
    for i, q in enumerate(challenge_questions):
        print(f"  Q{i+1} (ID: {q['id']}, Difficulty: {q['difficulty']}): {q['question']}")

    # Step 3: Submit an answer to the first challenge question
    if challenge_questions:
        first_question = challenge_questions[0]
        user_answer = "The assistant should be able to answer questions requiring
comprehension and inference, pose logic-based questions, and justify every answer
with document references. It must also support PDF/TXT upload and provide an
auto-summary." # Example answer

        answer_payload = {
            "question_id": first_question["id"],
            "answer": user_answer,
            "session_id": session_id
        }
        submit_response = requests.post(f"{API_BASE_URL}/challenge/answer",
json=answer_payload)

        if submit_response.status_code == 200:
            evaluation_data = submit_response.json()
            print(f"\nEvaluation for Question: {evaluation_data['question']}")
            print(f"Score: {evaluation_data['evaluation']['score']}/100")
            print(f"Feedback: {evaluation_data['evaluation']['feedback']}")
            print(f"Document Reference:
{evaluation_data['evaluation']['document_reference']}")
            print(f"Suggestions: {evaluation_data['evaluation']['suggestions']}")
        else:
            print(f"Error submitting answer: {submit_response.status_code} -
{submit_response.text}")
    else:
        print("No challenge questions generated.")
else:
    print(f"Error uploading document: {upload_response.status_code} -
{upload_response.text}")
```

## 🔧 Configuration

### Environment Variables

- GOOGLE_API_KEY: **Required** for accessing the Google Gemini API for text generation and embeddings.
- LANGCHAIN_TRACING_V2: **Optional**. Set to true to enable LangChain's tracing for debugging and monitoring, typically used with LangSmith.
- LANGCHAIN_API_KEY: **Optional**. Your API key for LangSmith if you are using it for tracing and observability.

**Customization Options**

You can modify various parameters to fine-tune the assistant's behavior:

- **Chunk Size**: Adjust the chunk_size and chunk_overlap parameters in RecursiveCharacterTextSplitter within src/workflow.py to control how documents are split.
  ```
  # In src/workflow.py, inside DocumentProcessor.__init__
  self.text_splitter = RecursiveCharacterTextSplitter(
      chunk_size=1000,  # Adjust this value
      chunk_overlap=200, # Adjust this value
      length_function=len
  )
  ```

- **Model Temperature**: Modify the temperature parameter in ChatGoogleGenerativeAI within src/workflow.py to control the creativity/randomness of the LLM's responses. Lower values (e.g., 0.1) make responses more deterministic, higher values (e.g., 0.7) make them more creative.
  ```
  # In src/workflow.py
  llm = ChatGoogleGenerativeAI(
      model="gemini-1.5-flash",
      temperature=0.1, # Adjust this value
      max_tokens=2048,
      timeout=30,
      max_retries=3,
  )
  ```

- **Vector Search Results**: Change the k parameter in the similarity_search method within the retrieve_context function in src/workflow.py to control how many top-k relevant document chunks are retrieved.
  ```
  # In src/workflow.py, inside retrieve_context function
  docs = vector_store.similarity_search(question, k=5) # Adjust 'k' value
  ```

- **UI Theme**: The frontend's visual theme can be modified by adjusting the CSS variables defined at the top of streamlit_app.py. This includes colors, fonts, shadows, and border radii.

## 🐛 Troubleshooting

### Common Issues

1. **API Key Error (GOOGLE_API_KEY not found)**
   - **Problem**: The application cannot find your Google Gemini API key.
   - **Solution**: Ensure you have created a .env file in the root directory of your project (next to main.py and streamlit_app.py). Verify that it contains GOOGLE_API_KEY=your_gemini_api_key_here with your correct key. Also, ensure python-dotenv is installed (uv add python-dotenv or pip install python-dotenv).
2. **Document Processing Fails (No readable text found or Error processing PDF/text file)**
   - **Problem**: The application could not extract meaningful text from the uploaded document.
   - **Solution**:
     - Check the document format: Ensure it's a valid PDF or plain TXT file.
     - Verify document content: Some PDFs might be image-based (scanned documents) without selectable text, which PyPDF2 cannot extract. Try a different document or ensure the text is selectable.
     - Check logs: Look for more specific error messages in the terminal where your FastAPI backend is running.
3. **Port Already in Use (Port 8000 already in use or Port 8501 already in use)**
   - **Problem**: Another application or a previous instance of your backend/frontend is already using the required port.
   - **Solution**:
     - **Identify and Kill Process**: On Linux/macOS, use lsof -i :8000 (or 8501) to find the process ID (PID), then kill -9 <PID>. On Windows, use netstat -ano | findstr :8000 (or 8501) to find the PID, then taskkill /PID <PID> /F.
     - **Use Different Port**: You can specify a different port when running the applications:
       - For FastAPI: uv run fastapi dev main.py --port 8001
       - For Streamlit: uv run streamlit run streamlit_app.py --server.port 8502

### Debug Mode

To get more detailed logging and insights into the application's behavior, you can enable debug logging.
Add or modify the logging configuration at the top of main.py and src/workflow.py:

```
import logging
logging.basicConfig(level=logging.DEBUG) # Change INFO to DEBUG
logger = logging.getLogger(__name__)
```

This will print more verbose messages to your console, which can be helpful for diagnosing issues.

## 🤝 Contributing

Contributions are welcome! If you'd like to contribute to this project, please follow these steps:

1. **Fork the repository**: Create your own copy of the project.
2. **Create a feature branch**: git checkout -b feature/your-feature-name
3. **Make your changes**: Implement your new features or bug fixes.
4. **Add tests if applicable**: Ensure your changes are well-tested.
5. **Submit a pull request**: Open a pull request to the main branch of the original repository, describing your changes in detail.

## 📄 License

## 🙏 Acknowledgments

- **LangChain**: For powerful document processing and LLM integration capabilities.
- **Google Gemini**: For advanced AI reasoning and embeddings.
- **Streamlit**: For rapid and intuitive web application development.
- **FastAPI**: For building a robust and high-performance API backend.

For more information or support, please open an issue on the GitHub repository.