**Earnings Call Analyzer- AI-powered transcript analysis with topic extraction, summarization, and intelligent Q&A**

**How We Built the Earnings Call Analyzer:**

This document explains the technical architecture, design decisions, and implementation details of the AI-powered Earnings Call Analyzer.

**Table of Contents**

1. [System Architecture]

2. [Component Deep Dive]

3. [AI/ML Pipeline]

4. [Database & Search]

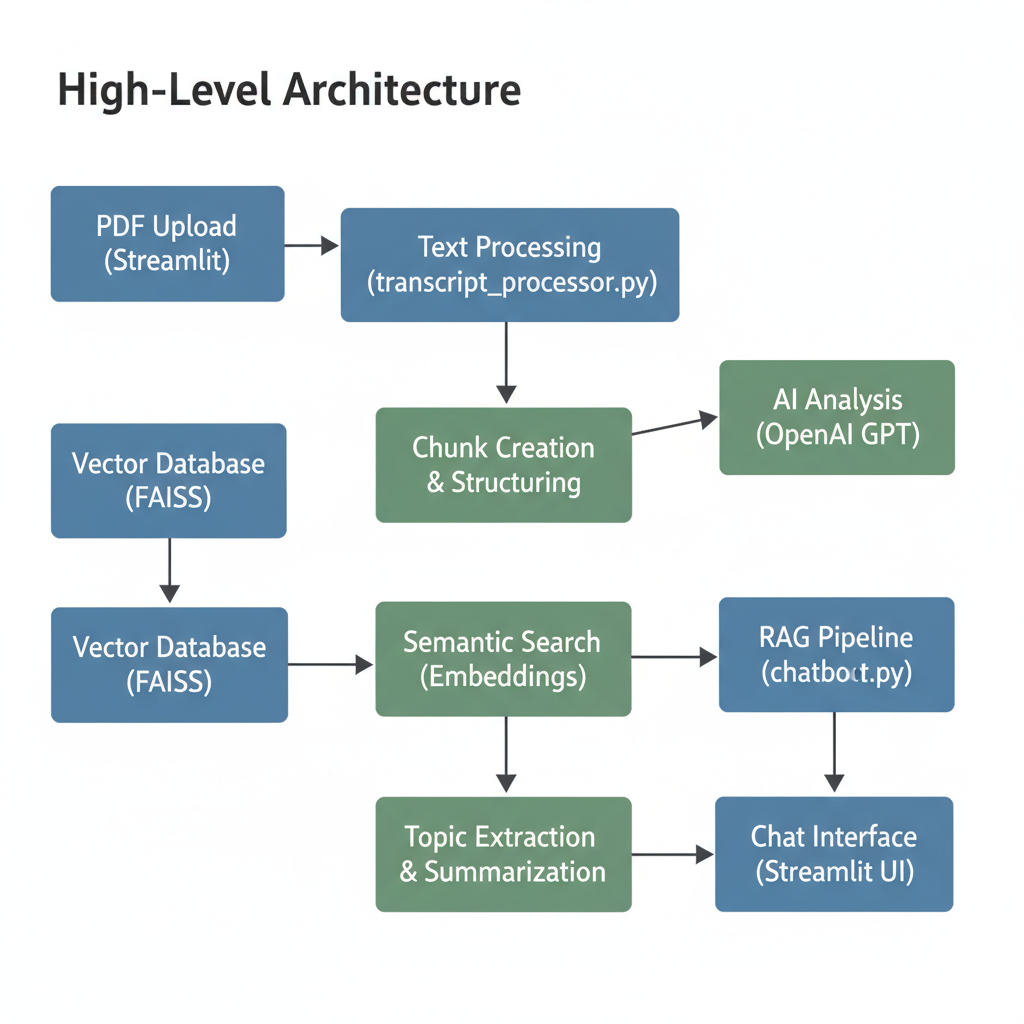
5. [User Interface]

6. [Configuration & Setup]

7. [Performance Optimization]

8. [Pros & cons in my solution]

**System Architecture:**



**Data Flow**

**1. Input:** PDF earnings call transcript (Laurus demo data given in OS path)

**2. Processing**: Extract text, identify speakers, separate sections

**3. Structuring:** Create chunks with metadata (speaker, type, content) (chunking using regex pattern)

**4. AI Analysis**: Generate topics and summaries using GPT models (By integrating Open AI API key

**5. Indexing:** Create vector embeddings and FAISS index

**6.Interaction:** User queries trigger semantic search + GPT generation

**Component Deep Dive in**:  
   
So, basically I have 4 major modules and 1 main app interface as my component. I didn’t separate front end backend and integration instead I want to keep it minimal because of considering the time constraint and also for front end it has mentioned Streamlit so I inputed the whole frontend part in main app itself which acts as a major gateway for frontend + backend + integration.  
The 4 mojor modules are:

**1. PDF Processor (transcript\_processor.py)**

* Extracts raw text with **PyPDF2**
* Identifies speakers and separates **Opening Remarks vs Q&A**
* Uses **regex pattern + fuzzy matching** for speaker consistency
* Cleans up noise (page numbers, whitespace)

**2. Topic & Summary Generator (summarizer.py)**

* Uses **OpenAI GPT** to extract 5–10 business topics
* Generates concise, investor-focused summaries
* Includes speaker names, metrics, and business strategies

**3. Vector Search (vectorstore.py)**

* Embeds transcript chunks using **text-embedding-3-small**
* Stores them in **FAISS (cosine similarity)** for semantic search
* Enables retrieval of the most relevant sections

**4. RAG Chatbot (chatbot.py)**

* Implements **Retrieval-Augmented Generation**
* Query → embedding → top chunks retrieved → GPT response
* Enforces **grounded answers only from transcript**
* Provides citations (speaker + snippet)

**Main app(app.py)**

* It is the **Streamlit front-end driver** of the Earnings Call Analyzer.
* **Initializes session state** (chunks, topics, summaries, chatbot context).
* **Displays the sidebar navigation** (Welcome, Opening Remarks, Q&A, AI Assistant).
* **Calls the right page function** (show\_welcome\_page, show\_opening\_remarks\_page, etc.).
* **Connects backend logic** (from transcript\_processor.py, summarizer.py, vectorstore.py, chatbot.py).
* **Handles errors gracefully** (like OpenAI quota or missing API key).

**AI/ML Pipeline:**The core intelligence of the application is powered by OpenAI’s GPT and Embedding models. The pipeline is designed for efficiency, cost-effectiveness, and reliability, ensuring that users receive accurate, business-focused insights from lengthy earnings call transcripts**.**

* **Models Used**
  + GPT-3.5-Turbo is used for generating summaries and handling Q&A because it balances speed, accuracy, and cost.
  + text-embedding-3-small is used to convert transcript chunks into dense vectors for semantic search**.**
* **Prompt Engineering**
  + Carefully designed prompts encourage factual, concise, and business-oriented answers.
  + Temperature is kept low (0.2) to minimize hallucinations and keep responses stable.
* **Processing Flow**
  + Transcript chunks → embeddings generated.
  + Queries converted into embeddings.
  + Most relevant transcript segments retrieved from FAISS.
  + GPT-3.5-Turbo generates context-aware summaries or answers.  
    (Note: This is what supposed to do but didn’t happen fully because of Open AI API valididty)

**Database and Search:**Efficient retrieval is critical for a smooth user experience, especially with large transcripts. The application relies on vector similarity search as the primary method, with a keyword-based fallback to ensure robustness**.**

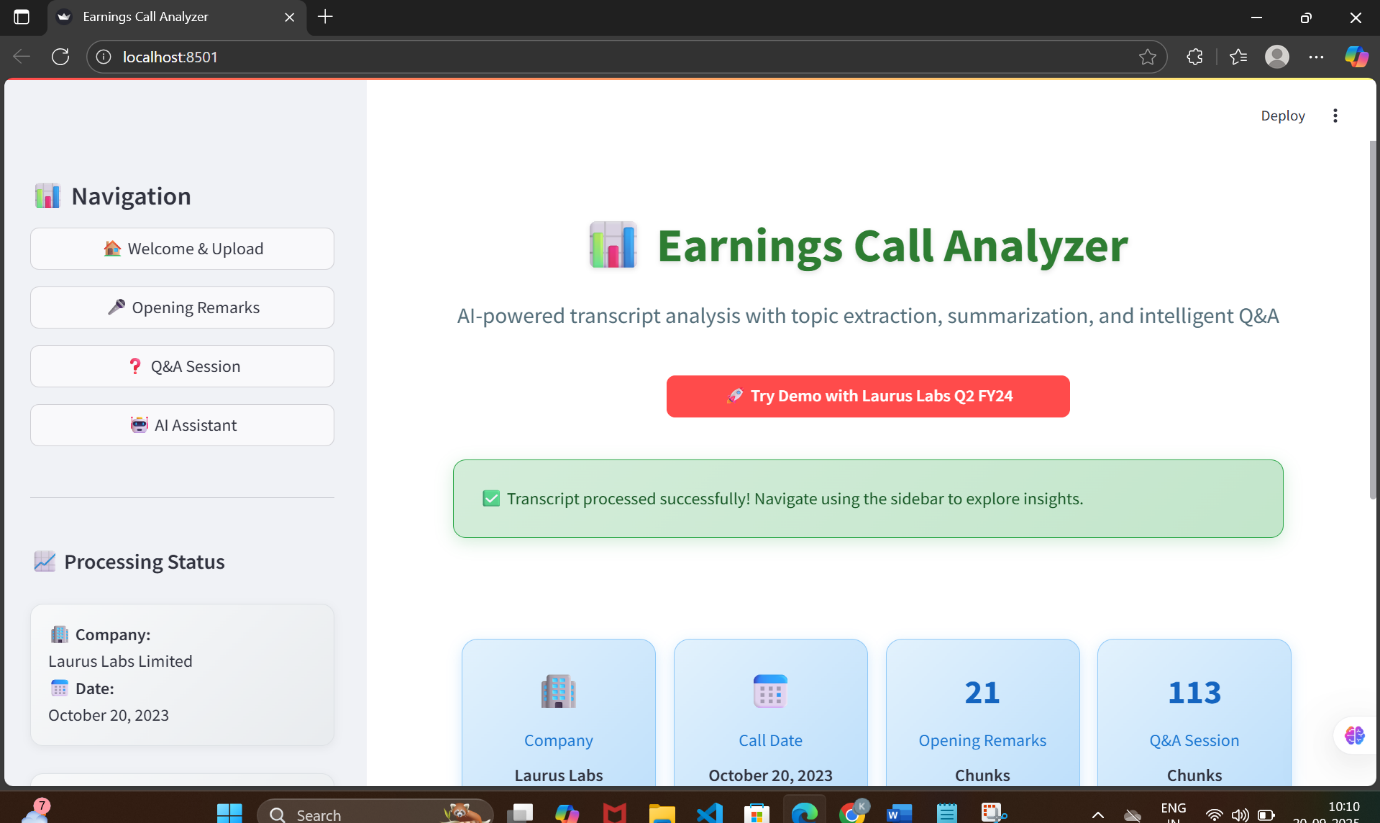
* **Transcript Chunking**
  + The uploaded PDF transcript is first processed and split into smaller text chunks (e.g., 500–1000 tokens each).
  + Chunking ensures that embeddings capture local context while keeping input sizes manageable for the model.
  + Each chunk is tagged with metadata (section type: *Opening Remarks* or *Q&A*) for better organization during search**.**
* **Primary Search**
  + FAISS (IndexFlatIP) is used with cosine similarity for efficient, high-performance vector search.
  + Each chunk is converted into embeddings using text-embedding-3-small, then indexed in FAISS.
  + To reduce API overhead, embeddings are generated in batches (10 chunks per request).

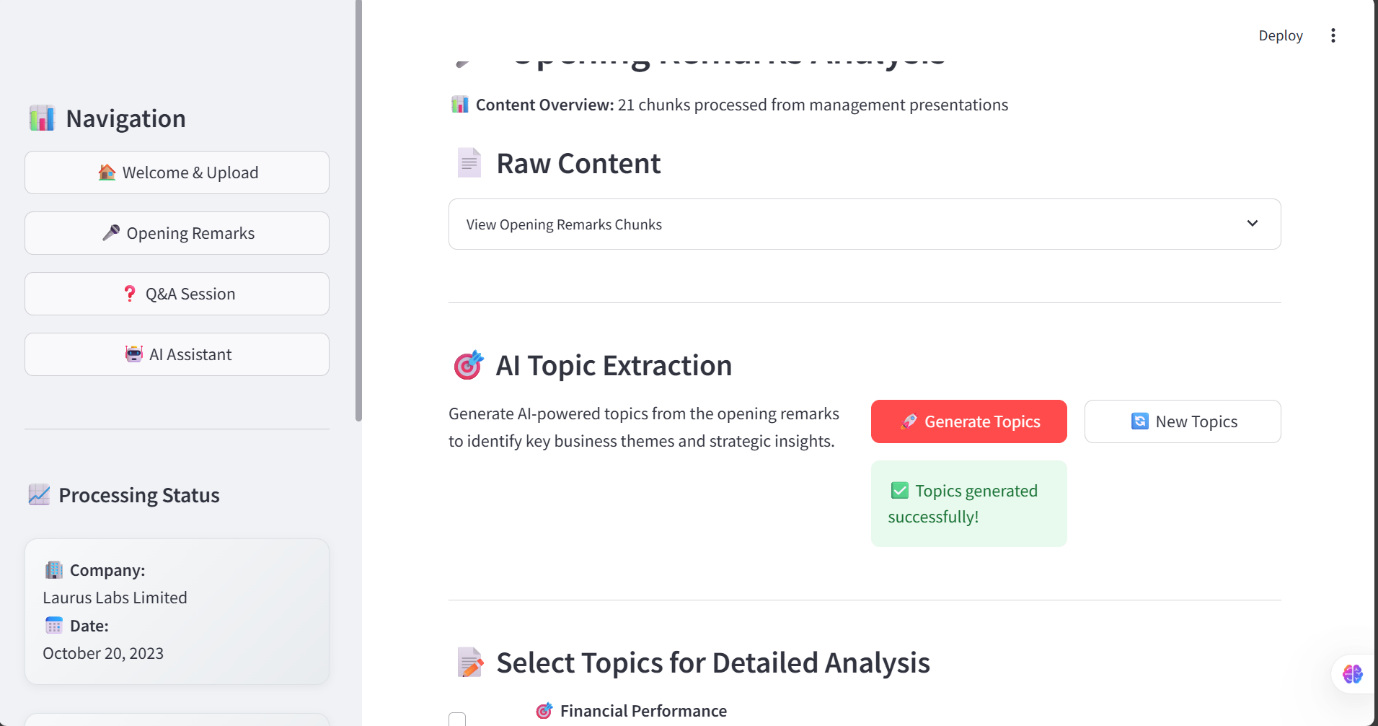
**User Interface:**

The front-end is built with **Streamlit**, offering a clean and interactive interface so users can analyze transcripts with minimal effort.  
I kept in minimalistic also step by step processing for UI .  
In navigation I had all the 4 sections welcome & upload, Opening remarks , Q&A, AI assistant if we press each button it will take it to the respective pages .  
In welcome page after pressing Try demo data it will show all the details   
In opening remarks page first chunks will be showed after that generate topics will be showed down only after pressing a topic generate summary will be shown where LLM goes generates from chunks same for Q&A.  
In AI assistant all sample question, Text box for prompting, ask question button and clear cache button there. For each section reset button kept to regenerate.

**Configuration and setup:**I’ll share my document to you open it with vs code , activate virtual environment with .venv

**.\.venv\Scripts\Activate.ps1** in terminal paste this to activate  
next it will show .\.venv with ps now paste **Pip install -r requirements.txt**check for success status  
next paste **cd backend**  
next paste **streamlit run app.py**   
main thing: in .env file paste the openai\_API\_KEY to see the full outcome (any Openai API key or if you’ve org id paste it in Openai\_ORG\_ID)  
now our webinterface would connect a local host and show our web interface   
another option cloning the github repository and following the same steps as mentioned .  
Next after web interface open press try demo data and move to each section to see chunks, topics and summary also chat with chatbot to have an interactive chat to know more.

  
 Fig: welcome and Upload page

  
Fig: Opening remarks page

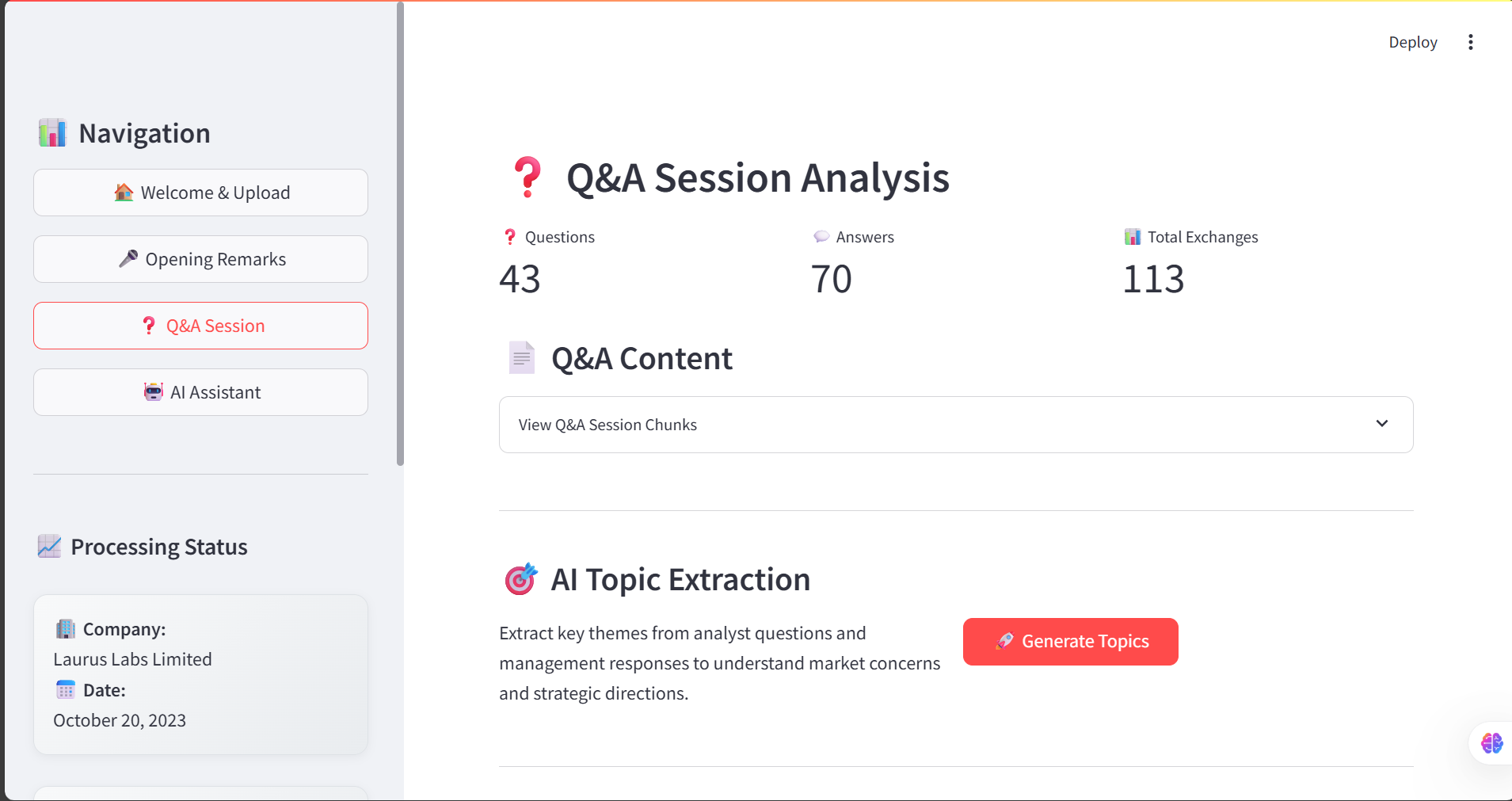
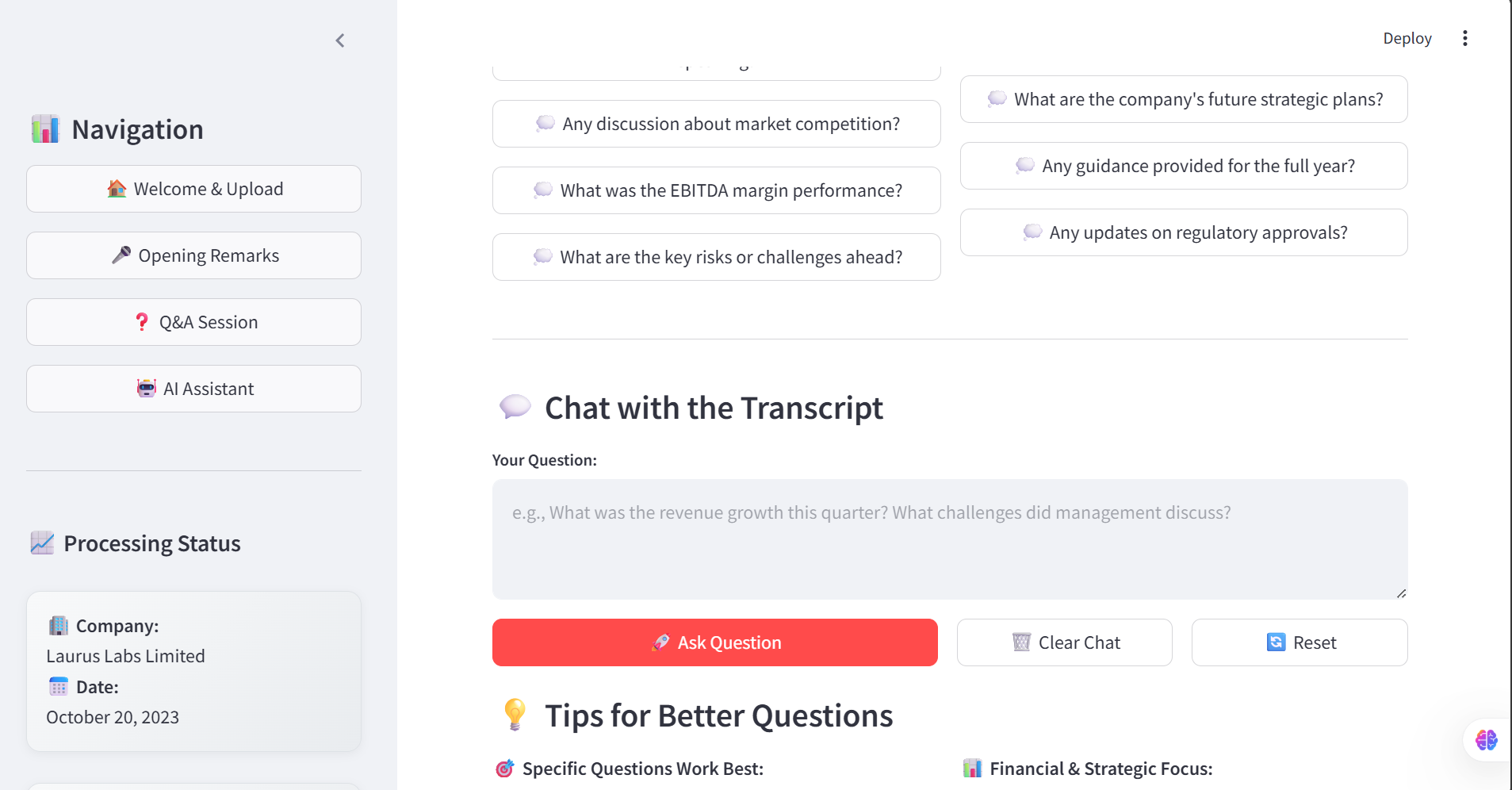


Fig:Q&A analysis page

  
Fig: AI assistant page  
  
so, after loading the code and giving the correct commands in terminal this is how our output will be.   
  
**Perfomance Optimization:** As mentioned all the success indicator are fulfilled as far as my knowledge

* **Process the demo PDF** without errors- done
* **Generate meaningful topics** relevant to business/finance-done
* **Provide accurate answers** to questions about the transcript-partial output because of API key issue
* **Demonstrate understanding** of RAG architecture and vector search-done
* **Show clean code practices** and proper documentation- done

So, I hope I met 85% of success criteria and did proper coding like mentioned components are there, UI using streamlit, Rag model, Faiss indexing, Chunking data, API integration, LLM working(partial) and also developed with 5 days of allotted time.

**Pros and cons of my web interface:  
 Pros:** End-to-End Automation

* Upload a PDF → text extraction → chunking → AI analysis → semantic search → chatbot Q&A.
* Minimal manual work; very user-friendly.

Structured Analysis

* Splits transcripts into opening remarks, Q&A, and speaker-based chunks.
* Metadata (speaker, type, content) allows targeted queries.

AI-Powered Insights

* Uses GPT-3.5-turbo for summaries, topic extraction, and Q&A.
* Provides concise, factual, business-oriented responses.

Semantic Search

* FAISS + embeddings enable contextual retrieval.
* Even complex or paraphrased queries can find relevant transcript parts.

User Interface

* Streamlit tabs make uploading, analysis, and Q&A intuitive.
* Clean separation of features: Upload → Topics → Chat.

**Cons:**

* No valid API key because of that LLM cannot give full output
* Dependency on openai API we can build an own llm but that would be the greatest task ever
* Can opted to huggingface API or any open source API but if I opted then I need to change my whole code which is not possible because of time constraint as well as I do not have practice on it .
* No advanced analytics dashboards or visualization of metrics yet.

**Summary:  
Every website or app as their own pros and cons if I decided I would never listed my cons and showcased my pros alone but a developer job is to develop a thing that has cons and make it versatile to use that’ s what I believe on my insight . I also feel proud of myself because I did a great job on developing this web interface within the limited time constraint. I also made a demo video of my implementation. I also believe that I fulfilled all the mentioned criteria and I hope you would like my project.   
Thank you!  
feel free to contact me for any clarification**