Evaluation Documentation

Team Name: Salamancas

Environment Variables	Value
VITE_ADOBE_CLIENT_ID	1a27f057ace94416b2fd19b3d35ab3f4
GOOGLE_API_KEY	Will be set by Adobe
AZURE_TTS_API_VERSION	2025-03-01-preview
AZURE_TTS_DEPLOYMENT	tts
AZURE_TTS_KEY	Will be set by Adobe
AZURE_TTS_ENDPOINT	Will be set by Adobe

Link to demo video:
https://drive.google.com/file/d/1E-gNLQdyoYgy1ebSUbpB4Zb44jJ1Jo6X/view?usp=sharing

To get started, follow these simple steps…

1. **Build the Docker Image:**

   docker build -t my-app .

   This command compiles your application into a Docker image, ready for deployment.

2. **Run the Docker Container with API Keys and Credentials:**

   docker run -e GOOGLE_API_KEY=”Your Gemini API key” -e AZURE_TTS_KEY="Your Azure TTS Key" -e AZURE_TTS_ENDPOINT="Your Azure TTS Endpoint" -e AZURE_TTS_DEPLOYMENT="tts" -e AZURE_TTS_API_VERSION="2025-03-01-preview" -e VITE_ADOBE_CLIENT_ID="1a27f057ace94416b2fd19b3d35ab3f4" -p 8080:8080 -p 8000:8000 my-app

   This command launches your application in a Docker container, exposing ports 8080 and 8000. Remember to replace the placeholder API keys and credentials with your actual values.

Technical Documentation: Document Insight & Engagement System

Project: Adobe India Hackathon 2025 - Finale

Theme: From Brains to Experience - Make It Real

Team Name: Salamancas

1. Introduction & Project Vision

1.1. The Core Problem: Information Overload and Disconnected Knowledge

In today's information-dense world, professionals such as researchers, students, and business analysts are inundated with a vast quantity of digital documents. A significant challenge arises from this volume: knowledge becomes fragmented and siloed within individual files. Recalling specific details, identifying thematic connections, or spotting contradictory information across a large personal library is a daunting, often impossible, manual task. This cognitive burden hinders deep understanding and prevents the synthesis of novel insights. Standard search tools offer simple keyword matching but fail to grasp the contextual and semantic nuances required for true knowledge discovery.

1.2. Our Vision: A Context-Aware Document Intelligence Engine

Our project, the **Document Insight & Engagement System**, was conceived to solve this problem by transforming a static collection of PDFs into a dynamic, interconnected knowledge base. The vision was to create an experience where a user's personal library becomes an active participant in their research process.

The system is designed to allow a user to:

1. **Read a document** seamlessly within an integrated viewer.
2. **Select any piece of text** that sparks their interest.
3. **Instantly receive contextually-aware insights**—such as enhancements, thematic connections, and direct contradictions—surfaced from every other document they have ever uploaded.
4. Optionally, transform these textual insights into a **rich media experience**, such as a persona-driven audio podcast, for on-the-go consumption.

Our goal was to move beyond simple Retrieval-Augmented Generation (RAG) and build a system that understands documents on a structural level, retrieves information with surgical precision, and delivers insights with speed and relevance, all while being grounded exclusively in the user's own content.

2. The Journey: Overcoming Technical Hurdles

Our development path was iterative and marked by a crucial pivot away from conventional methods that proved inadequate for the task.

2.1. The Failure of a Naive RAG Approach

Our initial design followed a standard RAG pipeline:

1. Extract raw text from PDFs.
2. Split the text into uniform, fixed-size chunks.
3. Embed these chunks and store them in a vector database.
4. Perform a similarity search on the user's query to retrieve the top-k chunks.
5. Feed the query and retrieved chunks to a Large Language Model (LLM) for synthesis.

This approach failed spectacularly for several reasons:

1. **Destructive Chunking**: Uniformly splitting text (e.g., every 512 characters) destroyed the document's inherent structure. A single chunk could contain half a sentence, severing its connection to its parent heading or the surrounding context, leading to retrieved results that were semantically similar but contextually useless.
2. **Context-Blind Retrieval**: Simple vector similarity search is "dumb." It finds text with similar wording but cannot distinguish between a main heading and a footnote. It struggled to identify nuanced relationships like contradictions or enhancements, often returning tangentially related but ultimately unhelpful information.
3. **High Latency**: The process of sequentially asking an LLM to find different types of insights (contradictions, then connections, etc.) for every user query was unacceptably slow, failing the "instant" requirement of the user journey.

2.2. Our Resolution: A Pivot to a Multi-Stage, Context-Aware Architecture

Recognizing these limitations, we re-architected our entire backend around a new philosophy: **context is king**. This led to the development of our innovative, multi-stage pipeline that intelligently processes, indexes, and retrieves information, forming the core of our final solution.

3. Our Approach: A Phased System Architecture

Our final solution is a sophisticated, full-stack application built on a foundation of intelligent document processing and a highly optimized retrieval workflow.

3.1. Phase 1: Hybrid-Machine-Learning-Solution-for-PDF-Structure-Extraction (Round 1A)

The cornerstone of our system is its ability to understand a PDF's structure before processing its content. We abandoned naive text extraction in favor of a custom-trained machine learning model.

- **Technology**: We use a scikit-learn classifier (heading_classifier_model.joblib), which was

trained on a labeled dataset of PDF text blocks.

- **Process**: When a PDF is uploaded, the document_parser.py script, powered by the PyMuPDF library, iterates through every text block on every page. For each block, our feature_extractor.py module calculates a set of features based on its visual and positional properties:
  - font_size: The point size of the text.
  - is_bold: A boolean indicating if the font weight is bold.
  - x_position and y_position: The coordinates of the text block on the page.

    etc.
- **Classification**: These features are fed into our pre-trained classifier, which accurately predicts the structural role of the text (e.g., H1, H2, H3, body_text, other). This classification is the first and most critical step in preserving the document's logical hierarchy.

3.2. Phase 2: Intelligent Document Parsing & Contextual Chunking (Round 1A & 1B)

With the structural role of every text block identified, we can now perform intelligent chunking.

- **Logic**: Instead of arbitrary splits, our DocumentParser groups text blocks logically. It iterates through the classified blocks and aggregates consecutive body_text blocks under their last seen parent heading.
- **Output**: The result is a list of "smart chunks." Each chunk is a dictionary containing not just the text but also a rich set of metadata:
  - original_content: The full text of the chunk.
  - text_type: The classified role (e.g., heading2).
  - source_document: The filename of the PDF.
  - page_number: The page where the chunk is located.
    This process ensures that the semantic and structural integrity of the information is maintained throughout the entire pipeline.

3.3. Phase 3: Document-Intelligence & Vector Indexing (Round 1B)

Once the document is deconstructed into smart chunks, we prepare it for fast and efficient retrieval.

- **Embedding**: Each chunk's original_content is passed to a Sentence Transformer model (google text-embedding-004), which converts the text into a high-dimensional vector embedding. This model is chosen for its excellent balance of performance and speed.
- **Indexing**: The resulting vector embedding and the full metadata dictionary for each chunk are stored in a **ChromaDB** collection. We chose ChromaDB for its lightweight nature, persistence, and powerful metadata filtering capabilities. This creates a rich, queryable knowledge base where every piece of information is indexed by both its

semantic meaning (the vector) and its structural context (the metadata).

3.4. Phase 4: Advanced Multi-Stage Insight Retrieval

This is where our system's intelligence truly shines. When a user selects text in the frontend, the retrieval_handler.py module executes a highly optimized, multi-stage process.

- **Stage 1: Broad Candidate Search**: The user's selected text is embedded using the same Sentence Transformer model. This query vector is used to perform an initial, broad similarity search against ChromaDB, retrieving the top 100 potentially relevant document chunks. This "wide net" approach ensures we capture all possible candidates for the next stage.
- **Stage 2: High-Precision Re-ranking**: A simple vector search can be noisy. To refine the results, we introduce a crucial re-ranking step. The initial 100 candidates are passed to a **Cross-Encoder model** (cross-encoder/ms-marco-MiniLM-L-6-v2). Unlike the first search, which compares vectors independently, a cross-encoder processes the user's query and each candidate chunk *together*, yielding a much more accurate and contextually aware relevance score. This allows us to re-rank the candidates with high precision and select the top 30, which are then de-duplicated.
- **Stage 3: Parallel LLM Categorization for Speed**: To meet the demand for "instant" insights, we avoid slow, sequential LLM calls. Instead, we make three **parallel, asynchronous calls** to the Google Gemini LLM. Each call is given the same context (the re-ranked chunks) but a different, highly specialized prompt:
  - **Contradiction Prompt**: Asks the LLM to find *only* the top 5 sections that directly oppose or challenge the user's selection.
  - **Enhancement Prompt**: Asks for *only* the top 5 sections that provide more detail, specific examples, or build directly upon the user's selection.
  - **Connection Prompt**: Asks for *only* the top 5 sections that are thematically related but are not direct enhancements or contradictions.

By running these tasks concurrently using Python's asyncio, we reduce the insight generation time to roughly that of a single LLM call, a massive performance gain that is critical to the user experience. The LLM is instructed to return a clean JSON object, which is then parsed and sent to the frontend.

4. System Architecture & Implementation Details

4.1. Backend (Python, Flask)

The backend is a modular Flask application designed for scalability and maintainability.

- **app.py**: The main entry point, defining the Flask routes (/upload, /retrieve, /generate-podcast, etc.) and handling CORS.
- **file_api_handler.py**: Manages the logic for file uploads, saving PDFs, and triggering the

asynchronous indexing pipeline.

- **indexing_pipeline.py**: Orchestrates the entire process of converting a raw PDF into indexed, searchable data by coordinating the parser, feature extractor, and ChromaDB client.
- **retrieval_handler.py**: Contains all the logic for the multi-stage retrieval process described in Phase 4, including embedding, querying, re-ranking, and the parallel LLM calls.
- **generate_podcast.py**: Handles the generation of persona-based podcast scripts by sending the retrieved context to the Gemini LLM with specialized conversational prompts.
- **generate_audio.py**: Interfaces with the Azure Text-to-Speech API to convert the generated podcast scripts into playable MP3 audio files.

4.2. Frontend (React, Vite, TypeScript)

The frontend is a modern, responsive single-page application designed for an intuitive user experience.

- **Index.tsx**: The main page component that orchestrates the entire UI, managing state for the document list, the selected PDF, the user's text selection, and the retrieved insights.
- **ResizablePanel.tsx**: Implements the core three-panel layout (document list, PDF viewer, insights panel), allowing the user to resize each section to their preference.
- **PDFViewer.tsx**: The heart of the user interaction. It integrates the **Adobe PDF Embed API** to render documents. Its most critical function is listening for the SELECTION_END event from the API.
- **useDebounce.tsx**: A custom React hook that is crucial for performance. It ensures that when a user selects text, we wait for a brief pause (e.g., 500ms) before sending a request to the backend. This prevents the system from being flooded with API calls while the user is still adjusting their selection.
- **RightPanel.tsx**: The "Insights Bulb." This component conditionally renders loading indicators or the categorized insights (Contradictions, Enhancements, Connections) received from the backend. It also contains the UI for selecting a podcast persona and the AudioPlayer.tsx component for playback.

5. Conclusion

Our Document Insight & Engagement System goes beyond traditional document management. It uses deep structural understanding and a multi-stage retrieval pipeline with re-ranking and parallelized LLM calls to deliver fast, relevant, and context-aware insights, transforming passive document libraries into active knowledge partners.