Oculis: AI Financial Assistant - Comprehensive Project Report

1. Introduction

The **Oculis: AI Financial Assistant** project addresses a critical need in financial data analysis: enabling non-technical users to quickly and accurately extract insights from structured financial documents using natural language. Developed as a take-home assignment, Oculis showcases a full-stack application built around a Retrieval-Augmented Generation (RAG) system. This report provides a comprehensive overview of the project's architectural design, implementation methodology, and key findings, highlighting the technical decisions and challenges overcome during its development.
At its core, Oculis allows users to upload balance sheet data (in CSV or Excel formats) for various companies. This data is then intelligently processed, indexed, and made accessible through a conversational AI interface. The primary goal is to deliver precise, context-aware answers that are directly verifiable from the uploaded financial records, thereby effectively mitigating the common challenge of Large Language Model (LLM) hallucination.

2. Approach

Our development approach prioritized modularity, maintainability, and user experience, while ensuring the underlying AI system was robust and accurate.

2.1. Architectural Overview

Oculis is structured as a modern client-server application, facilitating clear separation of concerns and scalability:

- **Frontend (frontend/ directory - React.js)**: This layer is responsible for all user interaction. It provides a dynamic and responsive Single Page Application (SPA) that handles user authentication, company selection, file uploads, and the real-time chat interface. Built with React.js, it leverages component-based development for reusability and maintainability.
- **Backend (backend/ directory - Flask)**: Serving as the application's brain, the Flask backend exposes a RESTful API. It manages user sessions, persists application data (companies, financial records), orchestrates the data ingestion pipeline, and integrates with the LLM and vector store to power the conversational AI.
- **Database (SQLite via SQLAlchemy)**: A lightweight SQLite database (site.db) is used for structured data persistence. It stores user accounts, company profiles, and the parsed numerical financial metrics (e.g., revenue, net income) for each company and year. SQLAlchemy, an Object Relational Mapper (ORM), provides an abstraction layer for Python interactions with the database, defined in backend/models.py.

- **Vector Store (ChromaDB)**: Located within the backend environment, ChromaDB serves as the specialized database for storing vector embeddings of financial facts. It enables efficient semantic search, allowing the system to quickly retrieve relevant textual information based on the meaning of a user's query.
- **Large Language Model (Google Gemini API)**: The gemini-1.0-pro-latest model from the Google Gemini API is the core intelligence component. It processes natural language queries, synthesizes information from retrieved contexts, and generates human-like responses.

2.2. Retrieval-Augmented Generation (RAG) Strategy

The decision to implement a RAG architecture was fundamental to achieving the project's accuracy goals:

- **Grounding Responses in Facts**: Unlike standalone LLMs that can generate plausible but incorrect information, RAG explicitly "grounds" the LLM's responses in a verifiable knowledge base. This is crucial for financial applications where precision is paramount.
- **Access to Up-to-Date Information**: LLMs have a knowledge cutoff based on their training data. RAG allows Oculis to incorporate the latest financial reports uploaded by the user, ensuring the AI can answer questions about very recent data that the base LLM would not inherently know.
- **Cost and Resource Efficiency**: Fine-tuning an LLM on proprietary data is computationally expensive and requires significant data preparation. RAG enables the use of a powerful pre-trained LLM (Gemini) augmented with dynamic content, providing high performance without the overhead of continuous model retraining.
- **Transparency and Auditability**: While not fully exposed in the current UI, the RAG architecture inherently allows for the identification of the source documents or chunks that informed an AI's response, which is vital for trust and compliance in financial contexts.

2.3. Granular Data Chunking Strategy

A cornerstone of our RAG implementation is the granular processing of financial data. Instead of treating an entire balance sheet file as a single large document, each meaningful financial fact is extracted and stored as a distinct, small "chunk" in the vector store.
For example, from a row in a balance sheet CSV like:
Metric,2022,2023,2024,2025
Revenue,10000000,12000000,15000000,18000000
Individual chunks are generated for each year and metric, such as:

- "For Company [ID], the revenue in 2022 was 10000000."
- "For Company [ID], the revenue in 2023 was 12000000."
- ...and so on for all metrics and years.

Each of these chunks is then embedded and stored in ChromaDB with rich metadata, including company_id, year, metric, and value. This strategy offers:

- **High Precision Retrieval**: When a user asks a specific question (e.g., "What was the total assets in 2024?"), the vector retriever can directly fetch the most relevant, concise chunk(s) (e.g., the chunk about "Total Assets for 2024").
- **Effective Metadata Filtering**: The attached metadata allows for powerful pre-retrieval filtering. For instance, when chatting about Company A, the retriever can be configured to *only* search documents associated with company_id = A, drastically reducing the search space and improving relevance.
- **Optimized LLM Context Window Usage**: By providing the LLM with only highly relevant, compact pieces of information, we minimize the total token count in the prompt. This leads to faster response times and more cost-efficient API calls.
- **Simplified Data Updates**: If a financial report is revised for a specific year, only the relevant chunks for that year need to be updated or re-embedded, rather than re-processing an entire large document.

2.4. Authentication and Data Persistence

- **JWT-Based Authentication**: The backend (backend/auth.py) implements JSON Web Token (JWT) based authentication using Flask-JWT-Extended. This provides a secure, stateless mechanism for user login and protecting API routes.
- **SQLAlchemy ORM**: backend/models.py defines the database schema using SQLAlchemy. This includes models for User, Company, and BalanceSheet. The BalanceSheet model stores the extracted numerical financial data, ensuring structured and queryable persistence.
- **Database Migrations**: Flask-Migrate is used to manage database schema changes, allowing for version-controlled and reproducible database setups.
- **Database Seeding**: A flask seed-db command is provided to populate the database with initial user credentials, facilitating quick setup for demonstration and testing.

2.5. Frontend Technologies

- **React.js**: Chosen for its component-based architecture, declarative UI, and efficient rendering, enabling a dynamic and interactive user interface.
- **Tailwind CSS**: A utility-first CSS framework that accelerates UI development by providing low-level utility classes directly in the markup. This promotes consistency and reduces the need for custom CSS.
- **Vite**: A fast build tool that provides an excellent development experience for React applications, offering rapid hot module replacement and optimized builds.
- **React Icons**: Used for a visually appealing and consistent set of icons across the application.

3. Methodology

3.1. Data Ingestion Pipeline

The data ingestion pipeline is designed for both user convenience and backend processing efficiency:

1. **File Upload (frontend/src/components/ChatInterface.jsx)**:
    - The user selects a CSV or Excel file via an <input type="file"> element.
    - The handleFileUpload function captures the selected file and the companyId (from the currently selected company).
    - A FormData object is used to send the file and companyId to the backend's /api/balance_sheets endpoint, allowing for multi-part form data submission.
    - Crucially, the frontend no longer requires the user to input a "year" for the upload; this is now dynamically determined by the backend.
2. **Backend File Reception and Processing (backend/app.py & backend/ai_model.py)**:
    - The Flask route receives the file and company_id.
    - The file is temporarily saved to the uploads directory (backend/uploads/).
    - The process_structured_financial_data function in backend/ai_model.py takes the filepath and company_id.
    - **Dynamic Year Detection**: Pandas (pd.read_csv or pd.read_excel) is used to load the financial sheet into a DataFrame. The function then dynamically identifies columns that represent years (e.g., '2022', '2023') by checking if column headers are four-digit numbers.
    - **Data Extraction and Validation**: For each identified year, the function extracts specific financial metrics (Revenue, Net Income, Total Assets, Total Liabilities) based on predefined row labels. Robust try-except blocks are used to convert these extracted values to floats, gracefully handling any non-numeric or missing data.
    - **Duplicate Data Handling**: Before storing new data for a specific company_id and year, the system performs a crucial step:
        - It calls delete_vectors_for_balance_sheet to remove any existing vector embeddings in ChromaDB associated with that company_id and year.
        - It then deletes any existing records in the SQL BalanceSheet table for the same company_id and year. This ensures data freshness and prevents redundant or conflicting entries.

3.2. Data Storage

- **SQL Database (site.db)**:
    - For each company and each year identified in the uploaded file, a new record is created or updated in the BalanceSheet table (backend/models.py).
    - The BalanceSheet model includes fields for company_id, year, revenue, net_income, assets, and liabilities. This structured storage allows for direct SQL queries and ensures data integrity.
- **Vector Store (ChromaDB)**:

- After storing in the SQL DB, each individual financial fact is transformed into a natural language sentence (e.g., "For Company 'Acme Corp', the total liabilities in 2025 was 2800000.0.").
- These sentences are then converted into high-dimensional numerical vectors (embeddings) using a pre-trained embedding model (part of the LangChain setup).
- Each vector is stored in ChromaDB along with rich metadata: company_id, year, metric (e.g., "Total Liabilities"), value, and source (e.g., FinancialData_8_2025.csv). This metadata is critical for targeted retrieval.

3.3. RAG Pipeline Implementation (LangChain)

The conversational AI is powered by a sophisticated RAG pipeline orchestrated by LangChain, connecting the user's query to the relevant financial data and the Gemini LLM.

1. **User Query Reception (backend/app.py)**:
   - The frontend sends the user's chat message (input) and the companyId to the /api/chat endpoint.
   - The generate_chat_response function in backend/ai_model.py is invoked.
2. **Conversational Memory Management**:
   - A ConversationBufferMemory from LangChain is used to store the ongoing chat history for each unique user-company session. This memory (_memory_store[memory_key]) ensures the LLM has context from previous turns, enabling natural follow-up questions (e.g., "What about next year?").
   - The chat_history (list of HumanMessage and AIMessage objects) is passed to the LLM along with the current user query.
3. **Context Retrieval**:
   - A retriever is initialized from the ChromaDB vector store.
   - **Crucial Filtering**: The retriever is configured with a filter that uses the company_id of the current conversation ({"company_id": {"$eq": company_id}}). This ensures that the vector search *only* considers documents relevant to the currently selected company, significantly narrowing the search scope and improving accuracy.
   - The user's query is embedded, and a similarity search is performed against the filtered vectors in ChromaDB. The k (e.g., 5) most relevant document chunks are retrieved.
4. **Prompt Engineering**:
   - A highly refined and restrictive system_prompt is dynamically constructed and passed to the LLM. This prompt is critical for guiding the LLM's behavior:
     - It explicitly instructs the LLM to use *only* the context provided by the RAG system.
     - It forbids the use of external knowledge or hallucination.
     - It specifies the desired behavior when information is not found (e.g., "I do not have that specific information...").

        - It emphasizes providing exact numerical values without modification.
        - It clearly defines the LLM's role as a "precise and factual financial analyst AI assistant."
    - This iterative refinement of the prompt was key to overcoming initial challenges with LLM hallucination.
5. **Generation (Google Gemini API)**:
    - The ChatGoogleGenerativeAI model (using gemini-1.0-pro-latest) is instantiated with a low temperature (e.g., 0.2 or 0.0) to encourage deterministic and factual responses.
    - The create_stuff_documents_chain from LangChain combines the retrieved context documents with the system_prompt, chat_history, and the current user_query into a single, comprehensive prompt for the Gemini LLM.
    - The LLM processes this augmented prompt and generates a natural language answer.
6. **Response Handling & Memory Update**:
    - The LLM's answer is extracted and returned to the frontend.
    - The user's input and the LLM's answer are saved back into the ConversationBufferMemory using memory.save_context, ensuring the chat history is updated for subsequent interactions.

3.4. Frontend Interaction (frontend/src/)

The React frontend (frontend/src/) provides a highly interactive and user-friendly experience:

- **Routing (main.jsx, pages/)**: react-router-dom is used to manage client-side routing, enabling navigation between LoginPage.jsx and DashboardPage.jsx.
- **Authentication Context (components/AuthContext.jsx)**: A React Context API is used to manage user authentication state globally across the application, making user information and login/logout functions easily accessible.
- **Company Management (components/CompanyManagement.jsx)**: Allows users to add new companies and select the active company for financial analysis.
- **Chat Interface (components/ChatInterface.jsx)**: This is the primary interaction hub.
    - **State Management**: Extensive use of useState for messages, input, loading, error, file, uploading, etc., ensures a reactive and predictable UI.
    - **Lifecycle Management (useEffect)**:
        - Automatically scrolls the chat window to the latest message using useRef and scrollIntoView.
        - Clears chat messages and upload form states when the companyId changes, providing a fresh context for each company.
    - **API Integration**: Uses axios via api/api.js to communicate with the backend for chat messages and file uploads.
    - **Dynamic UI Elements**: Input fields and buttons are dynamically enabled/disabled based on loading states, companyId selection, and input validity, guiding the user.
    - **Visual Feedback**: Clear loading indicators (spinners, dot pulse animation),

success messages (FaCheckCircle), and error messages (FaTimesCircle) are displayed. System messages (FaInfoCircle) provide contextual information about uploads and chat resets.
    - **Aesthetic Design**: Tailwind CSS is extensively used to create a modern, clean, and responsive design. This includes custom scrollbars (index.css), subtle shadows, gradients, rounded corners, and animate-fade-in transitions for chat bubbles, enhancing the overall user experience.
- **Financial Charts (components/FinancialCharts.jsx)**: (Placeholder for future visualization capabilities).
- **User Management (components/UserManagement.jsx)**: (Placeholder for future user administration features).

4. Findings and Results

4.1. Successful Outcomes

- **High Accuracy in Financial Queries**: The RAG pipeline, combined with granular chunking and precise prompt engineering, consistently enabled the AI to retrieve and present accurate numerical financial data from uploaded documents. Queries like "What were the total liabilities for 2025?" yielded exact figures, demonstrating the system's reliability.
- **Enhanced User Data Ingestion Experience**: The backend's ability to automatically parse multiple years from a single CSV/Excel file significantly streamlined the data upload process. This eliminates redundant user input and improves efficiency, addressing a key UX concern.
- **Natural Conversational Flow**: The integration of LangChain's ConversationBufferMemory allowed the AI to maintain context across turns, facilitating more natural and intuitive interactions where users could ask follow-up questions without repeating prior information.
- **Robust Error Handling and User Feedback**: Comprehensive error handling mechanisms were implemented across both frontend and backend. Users receive clear, actionable feedback for issues such as file upload failures, missing data, or AI communication problems, significantly improving usability.
- **Modular and Scalable Architecture**: The project's well-defined separation of concerns (frontend, backend, database, vector store, LLM) resulted in a modular codebase that is easy to understand, debug, and extend. This architecture lays a strong foundation for future features and scaling.
- **Modern and Aesthetic User Interface**: The React frontend, styled with Tailwind CSS, provides a visually appealing, responsive, and intuitive user experience. Attention to detail in animations, typography, and layout contributes to a polished feel.

4.2. Challenges Encountered & Solutions

- **ChromaDB where Clause for Multiple Conditions**:
    - **Challenge**: Initially, attempts to filter ChromaDB documents using multiple metadata conditions (e.g., company_id and year) in a simple dictionary format resulted in Expected where to have exactly one operator errors. ChromaDB's where clause requires explicit logical operators for compound conditions.
    - **Solution**: The filtering logic was updated to explicitly use the $and and $eq operators within the where dictionary: {"$and": [{"company_id": {"$eq": company_id}}, {"year": {"$eq": year}}]}. This resolved the filtering issue, ensuring precise retrieval.
- **LLM Hallucination and Context Adherence**:
    - **Challenge**: Despite providing relevant context, the LLM occasionally generated incorrect numerical values or relied on its general knowledge rather than strictly adhering to the provided financial data. This was evident when the LLM returned "$315,000" for liabilities instead of the correct "$2,800,000" from the uploaded data.
    - **Solution**: This was primarily addressed through rigorous **prompt engineering**. The system prompt was made highly restrictive, explicitly instructing the LLM to "MUST be based SOLELY on the provided financial data context," "Do NOT use any external knowledge," and to state "I do not have that specific information" if the answer wasn't in the context. Debugging by logging the source_docs confirmed that the correct data was indeed being retrieved, validating the prompt as the key area for improvement.
- **Frontend UX for Multi-Year File Upload**:
    - **Challenge**: The initial design required users to manually specify the "year" for each balance sheet upload, which was inefficient and poor UX when a single file contained data for multiple years.
    - **Solution**: The backend's process_structured_financial_data function was refactored to dynamically identify all year columns within the uploaded CSV/Excel file using Pandas. Correspondingly, the frontend's ChatInterface.jsx was updated to remove the manual "year" input field, allowing users to upload a single comprehensive file.
- **CSS Specificity Conflicts in Styling**:
    - **Challenge**: When applying specific styles (e.g., bg-white to a "Login" button), these were sometimes overridden by more general CSS rules (e.g., global button styles or component library defaults) that made other buttons black.
    - **Solution**: The immediate fix involved using Tailwind's !important prefix (!bg-white) on the specific button's class to force the desired style. For a long-term, cleaner solution, it was recommended to refactor global CSS to apply styles via more specific classes (e.g., .btn-dark) rather than generic element selectors, thus allowing Tailwind utilities to apply without conflict.
- **LangChain Deprecation Warnings**:

    - **Challenge**: Noticed LangChainDeprecationWarning messages for components like ConversationBufferMemory.
    - **Solution**: These were identified as warnings, not errors, indicating older but still functional API usage. For the scope of this project, they were deemed acceptable. For future development, migrating to newer LangChain memory management patterns (e.g., ConversationSummaryBufferMemory for token efficiency in long conversations) would be a logical next step.

4.3. Key Takeaways

- **Data Granularity is Paramount for RAG Accuracy**: For structured, factual data like financial reports, breaking down information into small, contextually rich chunks with comprehensive metadata is the most effective strategy for precise retrieval and minimizing LLM errors.
- **Prompt Engineering is an Iterative Art**: The quality of LLM responses in a RAG system is highly dependent on the clarity, specificity, and restrictiveness of the system prompt. Continuous refinement and testing of prompts are essential.
- **Seamless UX Requires Full-Stack Coordination**: Delivering a user-friendly experience, especially for data ingestion, often necessitates intelligent processing on the backend to abstract away complexity from the user interface.
- **Debugging RAG: Verify Context First**: When LLM responses are inaccurate, the first step in debugging should always be to inspect the exact context (retrieved documents) that the LLM received. This helps differentiate between retrieval failures and LLM misinterpretation.
- **Tailwind CSS and Specificity**: While powerful, understanding CSS specificity rules is crucial when mixing Tailwind utility classes with custom or global stylesheets to ensure styles are applied as intended.

5. Conclusion

Oculis successfully demonstrates a robust and intelligent AI financial assistant, showcasing the power of Retrieval-Augmented Generation in providing accurate, context-driven insights from structured data. The project effectively integrates modern frontend and backend technologies, implements a sophisticated RAG pipeline, and addresses common challenges in LLM application development. This comprehensive approach has resulted in a functional, user-friendly application that serves as a strong foundation for further development and exploration in AI-powered financial analysis.