Project Design – Part 2: Architecture & Flow

Having defined the core purpose, features, and UI/UX of the Gemini Pro Financial Decoder, this section delves into the underlying architecture and the precise execution flow. It outlines how each component interacts to transform raw financial data into actionable insights, providing a clear roadmap for development.

🏗️ System Architecture: A Holistic View

Our system architecture is designed for clarity, modularity, and efficiency. It illustrates the logical progression of data and control, emphasizing the distinct roles of each module and their interaction with external services like Gemini Pro.

Code snippet

```
graph TD

  A[📤 User Upload (CSV/XLSX)] --> B[file_uploader.py]

  B --> C{Cleaned DataFrame}

  C --> D[llm_handler.py]

  D -- API Call --> E[Gemini Pro via LangChain]

  E -- AI Response --> D

  D --> F[🧠 AI-Generated Summary (Text)]

  C --> G[visualizer.py]

  G --> H[📈 Graphs & Statistical Summaries]

  F --> I[app.py]

  H --> I

  J[styles.py] -- CSS Injection --> I

  I --> K[✨ UI Display to User]


  subgraph Backend Logic

    B

    D
    G
  end

  subgraph Frontend/Presentation
    I
    K
    J
  end

  style A fill:#f9f,stroke:#333,stroke-width:2px
  style E fill:#ccf,stroke:#333,stroke-width:2px
  style K fill:#cfc,stroke:#333,stroke-width:2px
```

- 📤 **User Upload (CSV/XLSX):** The initial entry point for all financial data. Users interact with the Streamlit interface to provide their balance sheets, profit & loss statements, or cash flow reports. This input acts as the catalyst for the entire analysis pipeline.
- **[file_uploader.py]:** This dedicated module acts as the gatekeeper for incoming data. Its primary role is to robustly handle the uploaded files, performing crucial validation and parsing. It transforms raw file streams into structured, cleaned Pandas DataFrames, ready for analysis.
- 📊 **Cleaned DataFrame:** The standardized output from file_uploader.py. This structured data format is the common currency exchanged between the core processing modules, ensuring consistency and ease of manipulation.
- **[llm_handler.py]:** The core intelligence hub. This module takes the Cleaned DataFrame (or a summarized version of it) and crafts intelligent prompts. It manages the secure communication with the Gemini Pro API, sending the prompts and receiving the detailed AI-generated financial summaries.
- **[Gemini Pro via LangChain]:** The powerful external AI service. Gemini Pro performs the heavy lifting of natural language understanding and generation, providing the in-depth financial commentary. LangChain serves as the orchestrating layer, simplifying API interactions, managing context, and facilitating advanced prompt engineering.
- 🧠 **AI-Generated Summary (Text):** The natural language output from llm_handler.py. This is the human-readable interpretation of the financial data, offering narratives, insights, and conclusions.
- **[visualizer.py]:** The visual storytelling module. It receives the Cleaned DataFrame and intelligently generates interactive financial charts and comprehensive statistical summaries, translating numbers into compelling visual patterns.
- 📈 **Graphs & Statistical Summaries:** The visual and quantitative outputs from visualizer.py, ready for display. These provide graphical representations of trends, distributions, and key metrics, complementing the AI's textual analysis.

- **[app.py]:** The central Streamlit application file. This module orchestrates the entire user experience. It takes the AI-Generated Summary and the Graphs & Statistical Summaries and renders them dynamically in the user interface. It also manages the Streamlit layout and user interaction flow.
- **[styles.py]:** The dedicated styling module. It injects custom CSS rules into app.py, ensuring that all UI elements adhere to the defined modern and professional aesthetic, enhancing the overall user experience.
- ✨ **UI Display to User:** The final presentation layer, where all processed information—AI insights, charts, and statistics—is seamlessly integrated and displayed to the user in a clean, intuitive, and visually appealing format.

🔄 Execution Flow: Step-by-Step Data Journey

The execution flow defines the sequential and parallel processes that occur from the moment a user uploads a file until insights are presented.

1. **User Initiates Upload:**
    - The user interacts with the intuitive file upload components located in the **Sidebar** of the Streamlit application.
    - They select and upload their desired financial statement: **Balance Sheet**, **Profit & Loss Statement**, or **Cash Flow Statement**.
    - This action triggers the file_uploader.py module.
2. **File Ingestion and Validation (file_uploader.py):**
    - file_uploader.py receives the uploaded file stream.
    - It immediately performs **rigorous validation checks**:
        - Confirms the file type is either .csv or .xlsx.
        - Checks for file integrity (e.g., non-empty, readable).
        - Attempts to parse the file into a Pandas DataFrame.
        - Validates essential column headers (e.g., 'Account', 'Amount' for typical financial statements) and data types.
    - In case of errors (invalid format, corrupted file, missing data), a clear st.error() notification is displayed to the user via app.py, and the process halts.
    - Upon successful validation and parsing, file_uploader.py returns a **cleaned and standardized Pandas DataFrame**.
3. **Intelligent Analysis (llm_handler.py):**
    - The app.py passes the Cleaned DataFrame to llm_handler.py.
    - **LLM Initialization:** llm_handler.py first ensures a secure and active connection to the **Gemini Pro LLM** via the langchain_google_genai library, handling API key authentication.
    - **Prompt Generation:** It then dynamically crafts a highly specific **prompt template**. This template incorporates a structured representation of the Cleaned DataFrame (e.g., key rows/columns, aggregated summaries, or even the entire DataFrame if concise enough) and clear instructions for the AI on the desired financial analysis (e.g., "Analyze the liquidity from this Balance Sheet," "Identify profitability trends from this P&L").
    - **AI Inference:** The crafted prompt is sent to the Gemini Pro API.
    - **Response Handling:** Gemini Pro processes the request and returns a **natural language summary** of the financial statement. llm_handler.py receives this response,

performs any necessary parsing or formatting, and returns the AI-generated textual insights.

4. **Visual Data Storytelling (visualizer.py):**
    - Concurrently or sequentially (depending on integration design), the Cleaned DataFrame is also passed from app.py to visualizer.py.
    - **Data Extraction:** visualizer.py identifies and extracts relevant **numeric data** columns suitable for visualization.
    - **Chart Generation:** It then intelligently generates appropriate **Plotly charts**:
        - **Line charts** for time-series data (e.g., monthly revenue trends, cash flow over quarters).
        - **Bar charts** for comparisons (e.g., expense categories, asset breakdown).
        - Potential for **pie charts** for proportional analysis.
    - **Statistical Summary:** It also generates a **statistical summary** of key numeric columns (e.g., using DataFrame.describe()) to provide quick quantitative insights.
    - These generated charts and statistics are then returned to app.py.
5. **User Interface Display (app.py & styles.py):**
    - app.py receives the **AI-generated summary** from llm_handler.py and the **charts and statistical summaries** from visualizer.py.
    - It dynamically updates the **Content Area** of the Streamlit UI to display these outputs.
    - All elements in the UI are rendered with the custom styling rules defined in styles.py (which are injected using st.markdown(..., unsafe_allow_html=True)), ensuring a consistent, modern, and professional aesthetic (e.g., gradient backgrounds, card-based layouts, button animations).
    - **Notification Cards:** Throughout this process, app.py will display appropriate st.success(), st.warning(), or st.error() messages as **Notification Cards** to keep the user informed about the status of their request.

📁 Final Folder Structure: Organized for Development

A well-defined folder structure is paramount for team collaboration, code maintainability, and project scalability.

Bash

```
gemini_financial_decoder/

│

├── app.py              # Main Streamlit application entry point and UI orchestration.

├── llm_handler.py       # Module responsible for all LLM interactions: API calls, prompt management, and AI response processing.

├── file_uploader.py      # Module for handling file uploads: parsing CSV/XLSX, data validation, and returning cleaned DataFrames.

├── visualizer.py         # Module dedicated to generating interactive Plotly charts and statistical summaries from financial data.
├── styles.py            # Centralized repository for all custom CSS styling injected into the
Streamlit application.

├── requirements.txt       # Lists all Python package dependencies, ensuring consistent
environment setup across development machines.

└── .streamlit/            # (Optional but recommended) Streamlit specific configuration files.

   └── config.toml        # For setting app title, favicon, theme, etc.
```

- **gemini_financial_decoder/**: The root directory for the entire project.
- **app.py**: As the central nervous system, this file orchestrates the user interface, manages data flow between modules, and acts as the streamlit run target.
- **llm_handler.py**: Encapsulates all logic related to interacting with the Gemini Pro API, including prompt engineering, context management, and parsing LLM responses.
- **file_uploader.py**: Handles all aspects of file input, from raw file stream to validated and cleaned Pandas DataFrames, ensuring data integrity.
- **visualizer.py**: Focuses solely on transforming processed financial data into insightful and interactive visualizations using Plotly, along with summary statistics.
- **styles.py**: Contains all custom CSS to apply the desired aesthetic theme, ensuring a polished and branded user experience.
- **requirements.txt**: A crucial file for reproducibility, listing all Python libraries and their exact versions that the project depends on. This ensures all team members and deployment environments use the same dependencies.
- **.streamlit/ (Optional but Recommended)**: A standard directory for Streamlit-specific configurations.
    - **config.toml**: Can be used to set global Streamlit configurations like the application title, favicon, default theme, and more, providing a consistent setup without needing to hardcode these in app.py.