Project Walkthrough & Technical Interview Guide

1. Project Overview

This project is an AI-Powered Lead Prediction Engine. Its primary goal is to help sales teams prioritize leads by analyzing historical data to predict which leads are most likely to convert.

Key Features:

- Predictive Scoring: Assigns a convert probability (0-100%) to every lead using a Random Forest model.
- Automated Insights: Uses LLMs (Gemini/Llama) to explain “why” a lead is high priority and suggest next actions.
- Interactive Chat: A RAG-based (Retrieval Augmented Generation) chat interface allows users to ask questions about their data (e.g., "Why are referrals converting better?").
- Hybrid Architecture: A Python FastAPI backend for heavy ML/AI lifting + a React frontend for a responsive, interactive dashboard.

2. Backend Architecture (Python/FastAPI)

The backend is structured as a modular API. It handles file processing, model training, inference, and LLM communication.

Component Breakdown

Component	Role	Tech Stack
API Layer	Exposes REST endpoints for the frontend.	FastAPI
ML Engine	Handles data cleaning, training, and prediction.	Scikit-Learn, Pandas
LLM Service	Generates human-readable insights and chat responses.	Gemini API, Llama.cpp
Orchestrator	Connects raw data to ML and LLM services.	Python Logic
Caching/State	Optimizes performance (especially for demos).	In-Memory & File-based

Detailed File-by-File Walkthrough (.py Files)

Entry & Configuration

- main.py: The application entry point. It initializes the FastAPI app, sets up CORS (Cross-Origin Resource Sharing) to allow the frontend to connect, and registers the routers (endpoints).
- backend/core/config.py: Central configuration. Stores constants like `UPLOAD_DIR`, API keys, and toggle flags like `DEMO_MODE`.
- backend/core/database.py: Manages the connection to the persistent store (SQLite) to save prediction history and lead results.

API Endpoints (backend/api/)

- upload.py:
  Role: Handles CSV file uploads.
  Key Logic: Checks file size and type. In `DEMO_MODE`, it calculates a file hash (fingerprint) to check if this exact file was processed before. If yes, it returns cached results instantly to avoid re-running heavy ML tasks.
- predict.py:
  Role: Triggers the prediction pipeline for a specific file.
  Key Logic: Loads the CSV, calls `orchestrator` to run predictions, and allows for an optional `explain` parameter to generate SHAP-like explanations for specific leads.
- train.py: Endpoint to manually trigger model retraining, though `upload.dpy` often handles "lazy training" on new datasets.
- chat.py: Endpoint for the AI Chat interface. Receives user queries, retrieves relevant data context, and calls `llm_service`.
- History.py: Retrieves past prediction runs from the database so users can revisit previous analyses.

Core Services (`backend/services/`)

- `ml_service.py` (CRITICAL):

Role: The "Brain" of the prediction engine.

Key Methods:

- `train(df)`: cleans data, encodes categories, finds the target column (fuzzy match), helps selects features, and fits a Random Forest Classifier.
- `predict_score(df)`: Uses the trained model to generate probabilities for new data. Handles missing columns by filling with 0s.

- `preprocess(df)`: Feature engineering logic-”calculates things like `EngagementScore` (TimeOnSite * PagesVisited).

- `llm_service.py`:
  Role: Abstraction layer for AI.
  - Key Logic: Implements a "Fallback System". It tries to use Gemini (Google) first. If that fails or is unconfigured, it falls back to a local Llama (GGUF) model. This ensures reliability.
  - Functions: `generate_insights` (for specific leads) and `chat_with_data` (for dataset-wide Q&A).

prediction_orchestrator.py:
Role: The conductor. It ties everything together:
1.Receives a dataframe.
2.Calls `ml_service` to get scores.
3.Calls `llm_service` to get insights for top leads.
4.Merges everything into a final JSON response.

- `csv_service.py`: Utility for safe CSV reading (handling encoding errors, different delimiters).
- `cache_service.py`: Manages the "Fingerprint Cache" for the demo mode, saving results to disk so the app feels instant for repeated files.
- `result_processor.py`: Formats the raw ML output into the clean JSON structure the frontend expects (sorting leads, assigning 'High/Medium/Low' labels).

3. Frontend Architecture (React)

The frontend is a Single Page Application (SPA) built with React and Tailwind CSS. It is component-based for reusability.

Key Components

- `App.jsx`: The main controller. It holds the global state (current file, metrics, leads list) and manages navigation (switching between Home, Dashboard, and Chat views).
- `pages/HomePage.jsx`: The landing experience.
- Contains the File Uploader.
- Shows a "Processing" animation while the backend works.
- `pages/DashboardPage.jsx`: The main analytical view.
- `StatCard` Components: Display high-level metrics (Total Leads, Conversion Rate).
- Charts: Visualizes lead distribution (High vs Low priority).
- Leads Table: A sortable, filterable list of leads with their scores and AI reasons.
- Drift Alert: Shows a warning if the new data looks very different from training data.
- `pages/ChatPage.jsx`: A chat interface similar to ChatGPT.
- Maintains message history.
- Sends user questions to the backend `chat` endpoint and streams the response.
- `components/layout/Sidebar.jsx`: The persistent navigation menu on the left.

4. End-to-End Data Flow (The "Story")

1. User uploads `leads.csv` on the Frontend.
2. `upload.py` receives the file.
   - Check: Have we seen this file before? (Hash check).
   - If New:
   `ml_service.py` reads it, cleans it, and trains a Random Forest model to understand patterns (e.g., "High TimeOnSite = Conversion").
   `ml_service.py` then runs predictions on the same data.
   `llm_service.py` looks at the top 5 leads and generates a sentence like "High priority because they visited the pricing page 3 times."
3. Backend returns a structured JSON to the Frontend.
4. App.jsx updates the state.
5. DashboardPage.jsx renders the charts and the list of leads, sorted by priority.
6. User clicks "Chat" and asks "Why is lead #101 high priority?".
7. chat.py sends this question + lead data to Gemini/Llama, which replies

"Lead 101 is high priority because they have a high engagement score of 98%..."

5. Interview Key Talking Points

- Robustness: Mention how the system handles missing data (imputation), missing columns (filling with 0s), and API failures (fallback from Gemini to Llama).
- Performance: Highlight the `DEMO_MODE` cache mechanism that makes the app feel instant by hashing file contents.
- Scalability: The modular separation of `ml_service` and `api` allows the ML engine to be scaled independently (e.g., moved to a GPU worker) without rewriting the API.
- Explainability: The project doesn't just give a score; it uses LLMs to explain the *reasoning*, which is crucial for user trust in AI tools.