Project Overview

This document provides a high-level explanation of a social-support eligibility application, aimed at readers who have not seen the source code. It outlines the major components, their responsibilities, and how they interact. The goal is to convey what each part of the system does—what input it expects, what output it produces, and how that output is used—without delving into implementation details or quoting specific code.

1. Purpose and Scope

The application’s main goal is to help applicants determine whether they qualify for social-support benefits based on their personal details and submitted documents. It does this through four key phases:

1. Front-End User Interface

2. Document Parsing and Data Ingestion

3. Validation and Feature Preparation

4. Eligibility Prediction and Explanation

5. Conversational Assistance

Throughout, the system relies on a combination of traditional data‐processing techniques (for tables and numeric computations) and modern large‐language‐model (LLM) capabilities (for unstructured text parsing and explanations). The result is a seamless flow from the moment a user fills out a form and uploads files, all the way through to a final decision and an interactive, chat‐style explanation.

2. High-Level Architecture

At a glance, the application is organized into distinct layers, each responsible for one logical piece of work. Below is a simplified diagram:

- The Front-End Interface collects personal details and file uploads from the user.

- Uploaded files are handed off to Parsing & Ingestion components, which convert each document into structured data (tables or key‐value fields).

- Structured data is then fed through Validation & Feature Engineering, which ensures consistency and computes the set of inputs (features) that the prediction engine needs.

- The Eligibility Prediction component uses a pre‐trained machine‐learning pipeline to determine a “yes/no” outcome.

- Simultaneously, an Explanation Generation component uses the model internals (e.g., feature importances) and an LLM to craft a human‐readable rationale for the decision.

- Finally, the Conversational Agent ties everything together, providing an interactive chat interface where applicants can ask follow‐up questions (e.g., “Why did I fail validation?” or “What next steps should I take?”).

Each box above corresponds to a logical module or set of modules. In the sections that follow, we describe them conceptually—what each one receives and what it returns.

3. Front-End Interface

What it does

- Presents a multi‐section form where an applicant enters personal details (name, date of birth, contact information, demographic data, household size, declared income, etc.).

- Provides file-upload controls for the required supporting documents: bank statements, a resume, an assets/liabilities spreadsheet, and a credit report.

- Displays feedback, parsed data, eligibility results, and a chat window once submission and processing have occurred.

How it works, conceptually

1. Form Rendering

- The applicant sees four distinct sections:

1. Personal Information

2. Demographic Details

3. Household & Income

4. File Uploads

- Each section shows input fields (text, dropdowns, date selectors, numeric entries) that the user must complete.

2. Submission Trigger

- When the user clicks a “Submit” button, the interface gathers all entered values and the uploaded files into one “application package.”

- This package is then forwarded to the backend logic for parsing and validation.

3. Stateful Display

- As soon as submission occurs, the interface shifts into a read-only view that shows:

- Any validation errors (highlighted, if validation fails).

- The tabular results of parsing each uploaded file (so the user can confirm that the system read the bank transactions, resume fields, etc., correctly).

- Computed metrics (employment summary, asset/liability totals).

- The final eligibility outcome (clear “Eligible” or “Not Eligible” banner).

- A natural-language explanation of that outcome (e.g., “You were found ineligible because your average monthly credits fell below your declared income by more than 10%”).

- A chat interface at the bottom where the user can ask additional questions.

4. Calling Sequence

- Step A: Collect inputs → send to Parsing & Ingestion

- Step B: After Parsing returns structured data → send to Validation & Feature Engineering

- Step C: If validation passes, send features to Eligibility Prediction; else, skip prediction

- Step D: Take the prediction result and feed both the raw inputs and the model’s output into Explanation Generation

- Step E: Initialize the chat agent’s first message, based on either “validation failed,” “ineligible,” or “eligible”

All of these actions happen behind a single “Submit” event from the user’s perspective. The front end simply waits for each backend component to return before updating the view.

4. Document Parsing & Data Ingestion

Once the form is submitted, the system must transform four different user-supplied documents into a structured format. These documents include:

1. Bank Statement (could be CSV, Excel, or PDF)

2. Resume (PDF or DOCX)

3. Assets & Liabilities (CSV or Excel)

4. Credit Report (PDF)

Below is how each one is handled at a high level:

4.1 Bank Statement Parser

- Input: The raw file bytes from the uploaded bank statement.

- Process:

1. Determine file type (CSV/Excel vs. PDF).

2. If it is a spreadsheet, read it into a tabular structure (rows and columns). Normalize column names so that “Date,” “Description,” “Credit,” “Debit,” and “Balance” can be found reliably, even if exported from different banks.

3. If it is a PDF, attempt to extract any embedded tables. If a neat table is detected, parse that into rows and columns. If no table structure is found (for example, the bank statement is just text lines), fall back to a simple line-parsing strategy (splitting lines by whitespace or known delimiters).

4. Convert the resulting data into a uniform table with exactly these columns:

- Date (converted to a standard date format)

- Description (merchant or transaction memo)

- Amount (positive for credits, negative for debits)

- Running Balance (account balance after each transaction)

- Output: A table (in memory) listing every transaction. If parsing fails entirely (for example, a scanned image PDF), the parser returns an empty or “invalid” flag, prompting the validation step to raise an error.

4.2 Resume Parser

- Input: The raw file bytes (either PDF or DOCX) of the user’s resume.

- Process:

1. Read and concatenate all textual content from the resume.

2. Send that full text into an LLM (large language model) chain with a template that asks:

- “Extract the applicant’s full name, email address, education summary, list of past employers, key projects, and skills.”

3. The LLM responds in a strictly structured JSON format (e.g., `{ "Name": "...", "email\_address": "...", "Companies": "...", ... }`).

4. Convert that JSON response back into an in-memory dictionary so downstream components can access fields like “Companies” or “Skills” directly.

- Output: A structured object (dictionary) containing keys such as:

- Name

- Email Address

- Education summary

- Companies worked for (as a textual list)

- Projects

- Skills

4.3 Assets & Liabilities Parser

- Input: The raw file bytes (CSV or Excel) of an assets/liabilities spreadsheet.

- Process:

1. Read the spreadsheet into a table (if Excel, this may produce multiple sheets).

2. Identify a column (usually labeled “Type”) that indicates whether each row is an “Asset” or a “Liability.”

3. Normalize that column so that every row is clearly marked.

4. For spreadsheets with multiple sheets, either pick the sheet(s) that list assets and liabilities or combine them under one table.

- Output: A single table where each row is tagged as either “Asset” or “Liability” with numerical values. Downstream logic will compute totals, ratios, and breakdowns by category (e.g., “Property” vs. “Investment”).

4.4 Credit Report Parser

- Input: The raw PDF bytes of the credit report.

- Process:

1. Extract all textual content from every page of the PDF.

2. Provide the combined text to an LLM chain, requesting extraction of key fields such as credit score, applicant’s name, date of birth, address, phone number, and summary account information (delinquencies, credit limits, etc.).

3. Enforce that the LLM returns a JSON object matching a predefined schema (e.g., `{ "credit\_score": "...", "Name": "...", "DOB": "...", ... }`).

- Output: A structured object containing fields for credit score and personal details, which can be used both in validation (to compare the name on the credit report with the form/resume) and in any follow-up questions.

5. Validation & Feature Engineering

With all documents parsed into structured data, the system moves on to two related but distinct tasks: a) ensuring that what the user entered in the form matches what the documents show, and b) computing numerical “features” that a machine-learning model needs in order to make an eligibility decision.

5.1 Validation

Purpose: Catch blatant mismatches or inconsistencies before attempting any predictions. If validation fails, the user is immediately informed (both via a visible error message and via the conversational agent) so they can correct or clarify their submission.

Types of Checks:

1. Name Consistency

- Compare the name entered in the form with the name parsed from the resume and the name parsed from the credit report.

- Rather than relying on an exact string match, use a fuzzy-matching algorithm: if any pair of names scores below a high similarity threshold (e.g., 90 out of 100), flag a validation failure.

- Return both a pass/fail boolean and a human-readable string explaining which comparisons fell short and by how much.

2. Income vs. Bank Activity

- Take the “declared income” from the form (a single number, e.g., monthly salary) and compare it against the user’s bank statement data.

- Group all credit transactions (positive amounts) by calendar month, compute the average monthly credit.

- Check whether the difference between declared income and average bank credits is within a reasonable tolerance (for example, ±10%).

- Return a pass/fail boolean and a summary of the computed average credits, percentage difference, and whether it falls within tolerance.

3. Overall Validation Result

- If both name and income checks pass, return “valid” to the main system.

- Otherwise, return “invalid” plus a combined list of “name mismatch” and/or “income mismatch” reasons.

What it returns

- A boolean flag (valid vs. invalid).

- If invalid, a list of one or more human-readable error messages such as:

- “Submitted name ‘Alice B. Kumar’ does not closely match the resume name ‘Alice B K.’ (similarity 82).”

- “Declared monthly income AED 5,000 differs from average bank credits AED 3,200 by 36%, which exceeds the 10% threshold.”

If validation fails at this stage, the system stops computing features or running a prediction. The interface shows the errors on screen, and the chat agent explains exactly why validation failed and what can be done to fix it.

5.2 Feature Engineering

Once validation passes, the next step is to assemble all relevant numeric and categorical inputs into one “feature vector” for the machine-learning model. These features come from three places:

1. Personal & Household Details (from the form)

- Age (computed from date of birth vs. current date).

- Gender, marital status, disability status, and housing type (all categorical).

- Family size and number of dependents (numeric).

- Declared income (numeric).

2. Employment History Features (derived from the resume)

- The resume parser has output a textual list of previous employers (e.g., “Company A (Jan 2018–Dec 2019), Company B (Feb 2020–Present)…”).

- This chunk of text is fed into an LLM chain with a prompt like “Given the list of companies and dates, compute the total number of months of work experience, the number of distinct employers, the tenure at current employer (in months), and the earliest start year.”

- The LLM returns a small JSON object with fields:

- Number of companies (integer)

- Total experience in months (integer)

- Average tenure per employer (float)

- Current employer tenure in months (integer)

- Earliest start year (4-digit integer)

- These five numeric values become part of the feature vector.

3. Wealth Metrics (from assets & liabilities)

- The assets/liabilities table indicates line items tagged “Asset” or “Liability,” each with a numeric value and a category (e.g., “Property,” “Investment,” “Loan”).

- The feature generator computes:

- Total assets (sum of all asset values).

- Total liabilities (sum of all liability values).

- Net worth (assets minus liabilities).

- Count of asset items and count of liability items.

- Asset-to-liability ratio (assets divided by liabilities; if liabilities are zero and assets > 0, treat ratio as a large sentinel value).

- Total value of property assets (sum of all assets where category = “Property”).

- Total value of investment assets (sum of all assets where category = “Investment”).

6. Eligibility Prediction & Explanation

Once the feature vector is ready, the system performs two connected tasks:

1. Run a Trained Machine-Learning Pipeline

2. Generate a Natural-Language Explanation

6.1 Machine-Learning Prediction

- What gets called: A pre-trained classification pipeline that was previously created by an offline training process (details in Section 7).

- What it expects: A single row of data containing all numeric and categorical features described above.

- What it returns:

- A binary label: 1 → Eligible, 0 → Not Eligible.

- (Internally, the pipeline may also store the probability scores, but the front end only shows the final label.)

From the user’s perspective, as soon as all documents are parsed and validated, the “Eligibility Prediction” module is invoked automatically. The returned label is shown as a prominent banner: “Eligible for Social Support” or “Not Eligible for Social Support.”

6.2 Explanation Generation

Simply returning “Not Eligible” is not sufficient for most applicants. They need to know why. Therefore, an explanation module constructs a short prompt that includes:

1. Which label was predicted (eligible vs. not eligible).

2. Which features were most influential (for example, “the model determined that your net worth is above the threshold” or “your monthly income was too low compared to typical qualifying levels”).

3. Exact values of those top features (e.g., “Net Worth = AED 120,000; Average Monthly Credit = AED 4,000; Total Experience = 24 months”).

4. Guidance wording, asking the LLM to produce a concise, plain-language rationale.

Under the hood, this explanation process might look like:

- Step 1: Inspect the trained model’s internal “feature importance” (for instance, if it is a tree-based model).

- Step 2: Sort those importances in descending order, pick the top two or three.

- Step 3: Create a very short bullet list summarizing for each top feature:

- Feature name

- Importance score (e.g., 0.28)

- The applicant’s value for that feature (e.g., “24 months”).

- Step 4: Stitch that into a prompt like:

The model predicted “Not Eligible.”

- Step 5: Send that prompt into an LLM chain.

- Step 6: Receive a multiline explanation

- Step 7: Display that explanation text beneath the eligibility banner.

From the user’s perspective, they see a label (“Eligible” or “Not Eligible”) plus a friendly, personalized paragraph explaining exactly why that label was chosen.

7. Conversational Assistance

After the initial decision and explanation appear, the interface automatically launches a chat window. The user can type follow-up questions such as:

- “Why did my application fail validation?”

- “Which document was missing or incorrect?”

- “How can I increase my likelihood of eligibility?”

- “What if I paid off some liabilities—how would that affect my score?”

The chat system operates as follows:

1. Determine Conversation Context

- If the application failed validation, the agent knows that it should reference validation errors (e.g., name mismatch, income mismatch).

- If validation passed but the prediction was “Not Eligible,” the agent focuses on the feature-importance explanation and suggests next steps (e.g., “Consider verifying your declared income with a formal payslip,” “You might want to reduce liabilities or submit proof of additional assets”).

- If the prediction was “Eligible,” the agent outlines next steps (e.g., “Please visit the social-support office with these documents,” “You should expect benefits starting next month,” etc.).

2. Receive Follow-Up Query

- The user’s new question is appended to the chat history.

- The system composes a prompt by combining:

- The user’s new question text

- The original application data (form details, parsed documents, feature vector)

- Any prior validation errors (if they exist)

- The machine’s explanation text

- The predicted label

3. LLM-Based Response

- That combined content is sent to an LLM with instructions like:

- “Given everything we know—application form, errors, parsed tables, features, prediction, and previous explanation—please answer the follow-up question in a helpful, concise way.”

- The LLM returns a final chat message, which is appended to the conversation and displayed.

4. Continuous, Stateful Dialogue

- The agent maintains all previous messages so that future follow-ups can refer back to earlier points. For instance, the user could ask: “In my assets sheet, did you see my investment fund?” and the agent can recall exactly what assets were parsed.

From the user’s perspective, this chat behaves much like a customer-service bot: interactive, context-aware, and grounded in the actual data they provided.

8. Offline Model Training (Background)

Although the user never sees this directly, a key part of the system is a pipeline that produces the “trained ML model” used for prediction. Here is how that is generally organized (so that anyone integrating with or improving the system understands where the prediction logic came from):

1. Synthetic Data Generation

- Because real labeled data (actual applicants who know if they truly qualified) may not be readily available, the team first generates a “synthetic dataset” with plausible distributions for all features: age, income, assets, liabilities, family size, employment history, etc.

- For example, they might simulate 500 applicants, randomly assigning years of experience, random salary within a realistic range, and employ a simple formula to label whether someone should get social support or not.

- This synthetic set is saved for auditing and for initial model benchmarking.

2. Benchmarking Multiple Classifiers

- Using an automated benchmarking tool, dozens of different classification algorithms (decision trees, random forests, gradient boosting, support-vector machines, logistic regression, etc.) are trained and evaluated on the synthetic data.

- Each candidate is scored by cross-validation or a held-out validation set, and the best one (highest accuracy or best balance between precision/recall) is chosen.

3. Final Training Pipeline

- Once the “winning” algorithm is identified (for example, “Random Forest”), a final training step is run:

- A preprocessing pipeline is built that scales or normalizes numeric inputs, one-hots categorical inputs, and then attaches the chosen classifier at the end.

- That pipeline is fit to the entire synthetic dataset and serialized to disk as a single file (sometimes called a “pickle” or “joblib” file).

4. Model Versioning

- Every time the development team wants to update the model (for instance, by tuning hyperparameters, adding a new feature, or training on real applicant data), they repeat the above steps, produce a new serialized model file, and replace the old one in the “Models” folder.

- The prediction component always picks up the newest file by checking file timestamps or a versioning registry.

Takeaway for Readers

- Conceptually, you do not need to know which specific algorithm is inside—just that it is “a standard machine-learning pipeline” that expects the feature vector we described.

- If you ever want to replace or retrain that model, you generate data, benchmark, pick an algorithm, build a preprocessing + classifier pipeline, and save it.

- The front-end prediction engine simply reads that latest pipeline and runs a single `.predict()` call.

9. Library & Tool Choices (Why They Were Used)

Below is a broad explanation of the categories of technology chosen, along with the rationale for each.

9.1 Interactive Front-End

- Streamlit-Style Framework

- Chosen because it allows rapid construction of interactive forms, file uploaders, and data-display tables with minimal boilerplate.

- It automatically bundles everything into a shareable web application, which is ideal for a small team or proof-of-concept.

- Alternative web frameworks exist (e.g., Flask, Django + custom JavaScript), but those would require more development time on the front‐end side.

9.2 Document Parsing

- Tabular Libraries (Pandas-Style)

- A well-established data library for reading and normalizing CSV and Excel files.

- Provides built-in functions to rename columns, convert date strings to date objects, filter rows, and compute group sums (e.g., monthly credits).

- If a future data volume grows substantially, a distributed processing tool (Dask, Spark) could replace this, but for typical bank statements, a “in-memory table” is sufficient.

- PDF/Text Libraries

- A dedicated PDF parsing library is used to extract tables where possible (e.g., if the bank statement is a nicely formatted bank export).

- If no table can be found (e.g., a scanned or free-text PDF), a fallback text-based parsing strategy attempts line splitting.

- A similar document reading approach is used for Word files: load all paragraphs in order, then hand that raw text to an LLM for any fields that need to be extracted.

9.3 Fuzzy Matching (Validation)

- Fuzzy String Matcher

- Names can appear in slightly different formats—middle initials, abbreviations, or minor typos. Instead of demanding an exact match, the system computes a “fuzzy match score” between 0 and 100.

- A threshold of around 90 indicates “almost identical,” while 70–80 would be suspicious. This reduces false rejections due to minor formatting differences.

9.4 Large-Language Models (LLMs)

- LLM Used: **Gemma3: 1b version**

- An on-premises or local LLM server is used to avoid external API costs and latency.

- Whenever the system needs to interpret free-text (resume paragraphs, PDF credit-report dumps) or to generate an English explanation (model rationale, chat responses), it sends a carefully designed prompt to the LLM.

- For resume and credit parsing, the LLM is constrained by a schema so that it must return a structured JSON object. This eliminates error-prone ad hoc regex parsing.

- For explanations and conversational follow-ups, the LLM is prompted with context (features, parsed tables, validation messages) and asked to produce natural, human-readable replies.

9.5 Machine-Learning Model

- Traditional ML Library

- A popular Python ML library is used to construct pipelines that combine data preprocessing (standard scaling for numeric inputs, one-hot encoding for categorical inputs) with a classifier (e.g., a decision tree or random forest).