EMIPredict AI: Technical Documentation

1. Introduction

This document provides a technical overview of the **EMIPredict AI** platform. The project's primary goal is to solve a dual machine learning problem: (1) classifying a loan applicant's EMI eligibility and (2) regressing their maximum affordable EMI. The solution is delivered as an interactive, multi-page Streamlit application with a complete MLOps lifecycle managed by MLflow.

2. Solution Architecture

The application is built on a modular, decoupled architecture to separate concerns and ensure maintainability.

- **app.py**: The main entry point and home page for the Streamlit application. It provides navigation and introduces the project.
- **pages/**: A directory containing each distinct page of the application:
  - **1_Data_Explorer.py**: Handles visualization of the cleaned dataset.
  - **2_Model_Training.py**: Contains the UI and logic to execute the entire training pipeline on demand.
  - **3_Live_Prediction.py**: Provides the user-facing form for real-time predictions.
- **utils.py**: The core logic module. This file contains all helper functions for data loading, cleaning, feature engineering, and model evaluation. This modularity allows the training page and notebook to share the exact same processing logic.
- **mlruns/**: A local directory automatically generated by MLflow to store all experiment data, including parameters, metrics, and model artifacts.
- **Generated Artifacts (.joblib, .txt)**: The training pipeline saves three key artifacts (standard_scaler.joblib, label_encoder.joblib, feature_names.txt) to disk. The Live_Prediction page loads these to ensure that new, raw data is processed *exactly* the same way as the training data.
- **Streamlit Cloud**: The platform is deployed on Streamlit Cloud, which links directly to the GitHub repository and manages all dependencies via requirements.txt.

Data Flow Diagram

```
[Data: emi_prediction_dataset.csv]
           |
           v
[Page 2: Model Training] --(calls)--> [utils.py: load_data,
clean_data, engineer_features]
           |
           v
[Splits & Scales Data]
           |
           v
```

```
[Trains 6 Models] --(logs to)--> [MLflow Server (mlruns/)]
          |                      |
          v                      v
[Saves: .joblib, .txt]      [Registers Best Models (XGBoost)]
          |
          |
[Page 3: Live Prediction] --(loads)--> [Best Models from MLflow]
          |                      |
          '--(loads)--> [Saved: .joblib, .txt]
          |
          v
[User Input] -> [Preprocess Input] -> [Predict] -> [Display Results]
```

3. Methodology

3.1. Data Preprocessing Pipeline

The pipeline is defined in utils.py to ensure consistency.

1. **Data Loading (load_data)**: Loads the 400,000+ record CSV using pd.read_csv(low_memory=False) to prevent DtypeWarning.
2. **Data Cleaning (clean_data)**:
    - Converts text-based numeric columns (age, monthly_salary, bank_balance) to numeric using pd.to_numeric(errors='coerce'). This step creates new NaN values where non-numeric text was present.
    - Handles messy categorical data (e.g., gender column with M, Male, female, F) by standardizing them to a clean format (Male, Female).
    - **Imputation**: Uses SimpleImputer to fill all NaN values (both original and newly created). strategy='median' is used for numerical features and strategy='most_frequent' for categorical features.
3. **Feature Engineering (engineer_features)**:
    - **New Financial Ratios**: Creates several new features critical for risk assessment, including total_monthly_expenses, debt_to_income_ratio, savings_to_income_ratio, and loan_to_income_ratio.
    - **Encoding**:
        - Binary Encoding: Converts existing_loans (Yes/No) to 1/0.
        - Ordinal Encoding: Converts education (e.g., High School -> 1, Graduate -> 2) to a numeric format.
        - One-Hot Encoding: Uses pd.get_dummies for all other nominal categorical features (gender, marital_status, employment_type, etc.) to create binary True/False columns.
4. **Data Splitting**:
    - The data is split into Train (70%), Validation (15%), and Test (15%) sets.
    - stratify=y['class'] is used during the split to ensure that all three sets have the same proportional distribution of Eligible, High_Risk, and Not_Eligible classes.
5. **Scaling**:

- A StandardScaler is fit *only* on the X_train set.
- The scaler is applied to transform the X_train, X_val, and X_test sets.
- Crucially, the scaler is only applied to *continuous numerical features*. The one-hot encoded True/False columns are (correctly) left unscaled.

3.2. Model Training and MLflow

1. **Experiment**: A single MLflow experiment named EMIPredict_AI_v1 is used.
2. **Run Strategy**: To avoid parameter conflicts, a **separate MLflow run** is created for each of the 6 models (e.g., run_name="XGBoostClassifier").
3. **Logging**: For each run, the following are logged:
   - **Parameters**: All model hyperparameters (e.g., max_depth, n_estimators).
   - **Metrics**: All validation metrics (e.g., val_accuracy, val_rmse).
   - **Artifacts**: The trained model object itself is logged using mlflow.sklearn.log_model() or mlflow.xgboost.log_model().
4. **Model Registry**: The registered_model_name argument is used during logging (e.g., registered_model_name="XGBoostClassifier"). This automatically versions the best-performing models, allowing the prediction page to programmatically load the "latest" version.

3.3. Development Challenges & Resolutions

During development, two key technical challenges were identified and resolved.

1. **Challenge: Input X contains NaN Error**:
   - **Problem**: The initial model training failed with a NaN error, even after an imputation step.
   - **Root Cause**: The data cleaning step (pd.to_numeric(errors='coerce')) was creating *new* NaN values *after* the imputation step had already run.
   - **Resolution**: The preprocessing pipeline was re-ordered. Data cleaning and NaN creation (errors='coerce') are now performed *first*, followed by a comprehensive imputation step that catches both original and newly created NaN values.
2. **Challenge: MLflow Parameter Conflict (got multiple values for keyword argument 'objective')**:
   - **Problem**: The XGBoost training script failed when attempting to log both the classifier and regressor in a single run.
   - **Root Cause**: The script was trying to log the objective parameter twice in the same run (once as multi:softprob and once as reg:squarederror), which MLflow does not permit.
   - **Resolution**: The training logic was refactored to follow MLflow best practices. Each of the 6 models is now trained and logged in its own **separate MLflow run**. This prevents all parameter conflicts and provides a much cleaner, more organized experiment dashboard.

4. Production Deployment

4.1. Platform: Streamlit Cloud

The application is deployed to production using **Streamlit Cloud**. This platform was chosen for its direct integration with GitHub, which enables a seamless Continuous Deployment workflow.

- **Repository**: https://github.com/Viplove0114/EMI_prediction_AI/
- **Workflow**: When new code is pushed to the main branch of the GitHub repository, Streamlit Cloud automatically detects the changes, rebuilds the application container, and deploys the new version.
- **Dependencies**: All required Python packages are managed via the requirements.txt file, which Streamlit Cloud uses to build the environment.

4.2. Live Application URL

The final, customer-facing application is live and accessible at the following URL:
https://emipredictionai-007.streamlit.app/