Architecture: End-to-End Fraud Detection System

This document outlines the complete architecture of the automated fraud detection system. The system is designed not just to build a model, but to *discover the optimal model* through intelligent hyperparameter tuning and then deploy it for real-time use.

The system is broken into two primary phases:

1. **Phase 1: The "Search & Build" Phase** (The Training Pipeline)
2. **Phase 2: The "Deploy & Predict" Phase** (The Inference App)

Phase 1: The "Search & Build" Phase (Training)

This phase is handled by the `fraud_pipeline_complete.py` script. Its goal is to run an exhaustive search to find the best possible combination of pipeline steps and model parameters, then build and save the single best-performing model.

The entire process is managed by **Optuna**, a hyperparameter optimization framework.

The Optuna Tuning Loop

The script executes 50 "trials" (or more). In each trial, Optuna intelligently suggests a new set of parameters and builds an *entire* ML pipeline from scratch to test them.

Here is the architecture of a **single trial**:

1. **Optuna Suggests Parameters:**
   - `n_features_to_select` : How many features (10-25) should RFE keep?
   - `smote_k_neighbors` : How many neighbors (3-7) should SMOTE use?
   - `knn_n_neighbors` : How many neighbors (3-15) for the KNN model?
   - `rf_n_estimators` : How many trees (100-300) for the Random Forest?
   - `rf_max_depth` : How deep (10-30) can the trees be?
   - `meta_C` : How strong (0.01-10.0) should the meta-model's regularization be?
2. **A Temporary Pipeline is Built:** This pipeline defines the *order of operations*, which is critical for preventing data leakage.

   ```
   [Input Data (X_train)] -> [Step 1] -> [Step 2] -> [Step 3] -> [Step 4] -> [Score]
   ```

   - **Step 1:** `RobustScaler`
     - Scales the `Time` and `Amount` features, making them robust to outliers.
   - **Step 2:** `RFE` **(Recursive Feature Elimination)**
     - Uses a simple `LogisticRegression` model to analyze all 30 features.
     - It recursively removes the weakest features until only the `n_features_to_select` (e.g., 13) best ones remain.
   - **Step 3:** `SMOTE` **(Synthetic Minority Over-sampling)**
     - Analyzes the rare "Fraud" cases.

    - Creates new, synthetic "Fraud" data points based on its neighbors (`smote_k_neighbors`). This balances the dataset so the model can learn what fraud looks like.
  - **Step 4: `StackingClassifier` (The "Brain" of the Model)**
    - This is a 2-level ensemble model that runs its own internal parallel processes (`n_jobs=-1`) to be fast.
    - See the **Stacking Architecture** diagram below.
3. **Evaluation:**
   - This entire pipeline is evaluated using 3-fold cross-validation on the training data.
   - The outer loop (`cross_val_score`) runs in serial (`n_jobs=1`), but the inner `StackingClassifier` runs in parallel (`n_jobs=-1`). This is the "parallelism fix" that allows the CPU to hit 100% in bursts for maximum efficiency.
   - The average `roc_auc` score (a metric for imbalance) is returned to Optuna.
4. **Learn & Repeat:**
   - Optuna records the result and uses it to make a smarter guess for the next trial.
   - This loop repeats 50 times.

End of Phase 1: Building the Final Model

After the loop, Optuna knows the single best set of parameters (e.g., the 13 features, 7 KNN neighbors, etc. from your output).

1. **Build Final Pipeline:** The script builds *one last pipeline* using these winning parameters.
2. **Train:** It trains this final pipeline on the *entire* training dataset.
3. **Save:** The fully trained `final_pipeline` object is saved to `fraud_detection_pipeline.joblib` , and the list of feature names is saved to `feature_names.joblib` .

The Stacking Architecture (The "Brain")

This is the detailed view of "Step 4" from the pipeline. The `StackingClassifier` itself is an ensemble of models.

1. **Level 0: The "Workers" (Base Models)**
   - The same data (after SMOTE) is fed to three different models in parallel.
   - `KNeighborsClassifier` (KNN)
   - `RandomForestClassifier` (RF)
   - `AdaBoostClassifier` (AdaBoost)
   - These models don't give the final answer. They just give their own "opinion" (prediction).
2. **Level 1: The "Manager" (Meta-Model)**
   - A simple `LogisticRegression` model.
   - Its only job is to look at the predictions from the three "workers."
   - It *learns* which workers to trust. For example, it might learn "When RF and AdaBoost agree, I should trust them. When KNN disagrees, I should ignore it."

- It makes the final decision, which is more robust than any single model.

Phase 2: The "Deploy & Predict" Phase (Inference)

This phase is handled by the `app.py` (Streamlit) file. It *uses* the saved files from Phase 1 to make live predictions.

1. **On Startup:**
    - The Streamlit app loads `fraud_detection_pipeline.joblib` and `feature_names.joblib` into memory.
2. **User Input:**
    - A user enters transaction data (`Amount` , `V4` , `V10` , etc.) into the web-page sliders and input boxes.
3. **Real-Time Prediction:**
    - The app creates a single-row `DataFrame` with the user's data, using `feature_names.joblib` to ensure the column order is correct.
    - This DataFrame is fed into the loaded `final_pipeline.predict_proba()` method.
4. **The "Inference" Pipeline:**
    - The loaded pipeline automatically runs all its saved steps on the new data:
    - `[New Data] -> [Step 1] -> [Step 2] -> [Step 3] -> [Prediction]`
    - **Step 1:** `RobustScaler`
        - Transforms the new `Time` and `Amount` using the *same* scaling it learned from the training data.
    - **Step 2:** `RFE`
        - *Instantly* selects only the 13 best features it learned to pick during training.
    - **Step 3:** `SMOTE`
        - **This step is intelligently SKIPPED.** SMOTE is only for *training*, not for predicting. The pipeline knows this automatically.
    - **Step 4:** `StackingClassifier`
        - The new data goes to the "Workers" (KNN, RF, AdaBoost), who all make a prediction.
        - Their predictions go to the "Manager" (LogisticRegression), which makes the final decision.
5. **Display Result:**
    - The app gets the final fraud probability (e.g., 87%) and displays either a "FRAUD DETECTED" error or a "Normal Transaction" success message to the user in their browser.