Abstract

The AI Transaction Categorizer is a hybrid rule-based and machine learning system designed to automate financial transaction classification with high accuracy, explainability, and complete local execution. The framework integrates TF-IDF vectorization, a Logistic Regression–Random Forest ensemble, and a YAML-driven keyword rule engine to deliver both deterministic and probabilistic categorization. A Streamlit interface enables single and batch predictions, taxonomy editing, and real-time visualization of model outputs. LIME explainability provides transparent, token-level insight into individual predictions, supporting auditability and user trust. With a fully offline architecture, no external API dependencies, and strong security guarantees, the system is engineered for environments requiring privacy, control, and operational reliability. Its modular structure ensures scalability, portability, and ease of integration into personal finance tools, accounting platforms, and internal enterprise systems.

Problem Statement

Traditional financial transaction classification systems often rely on cloud-hosted machine learning models or external APIs, introducing challenges around privacy, data retention, latency, and operational dependency. Additionally, rule-only systems struggle to generalize, while ML-only systems lack interpretability and require large labeled datasets. This creates a gap for a solution that is **accurate**, **explainable**, **fully local**, and **easy to customize** without deep ML expertise.

The problem is to design and implement a **hybrid local-only transaction categorization framework** that:

1. Combines deterministic, explainable rules with machine learning for unknown transactions.
2. Delivers fast, high-confidence classifications without external API calls.
3. Allows editable, YAML-based taxonomies for business-specific customization.
4. Provides transparency via explainability techniques like LIME.
5. Supports scalable processing for both single and batch transaction data.
6. Ensures user data never leaves the device, maintaining strong privacy and compliance postures.

Table of Contents

1. Introduction

Financial transaction classification is a core component of modern personal finance tools, accounting systems, and financial analytics platforms. Categorizing transaction descriptions into meaningful spending categories enables downstream capabilities such as expense tracking, fraud detection, budgeting, tax reporting, and behavioral insights. However, real-world transaction data is noisy, unstructured, and highly variable across merchants, banks, and user contexts. This makes automated classification challenging, especially when relying solely on rule-based systems or machine learning models operating in isolation.

The **AI Transaction Categorizer** addresses these challenges by combining deterministic rules, machine learning, and explainable AI within a fully local, privacy-preserving pipeline. The system is engineered to produce accurate, interpretable, and scalable transaction classifications without requiring cloud services or external APIs. Through its modular design, it merges practicality, transparency, and extensibility—making it suitable for both personal use and enterprise-grade financial workflows.

1.1 Background

Traditional financial categorization tools often use one of two approaches:

1. **Rule-based systems**, which rely on keyword lookup tables;
2. **Machine learning classifiers**, which infer categories from historical labeled data.

Rule-based engines offer transparency but require constant manual maintenance and fail when encountering new merchants or variations in transaction strings. Machine learning models generalize better but are often opaque, require large datasets, and may rely on cloud-hosted inference, posing privacy and compliance risks.

Recent trends in AI emphasize **hybrid architectures**, combining rules with ML to achieve the best aspects of both: interpretability, adaptability, and reliability. At the same time, user awareness of data privacy has increased the demand for systems that operate **entirely locally**, ensuring full data control.

The AI Transaction Categorizer builds on this evolution by integrating hybrid classification, explainability (via LIME), local-only execution, and a user-friendly Streamlit interface.

1.2 Motivation

The motivation for this project stems from several real-world limitations in existing financial categorization systems:

- **Privacy concerns:** Many platforms send transaction data to cloud APIs, exposing sensitive financial information.
- **Lack of explainability:** Users cannot see *why* a category was assigned, reducing trust in automated decisions.

- **Rigid rule systems:** Hard-coded rules cannot scale with new merchant patterns or transaction formats.
- **Dependency on large datasets:** ML-only models require extensive labeled data to perform well.
- **Operational fragility:** Cloud-dependent solutions fail in offline environments or restricted enterprise networks.

To address these gaps, the system aims to deliver:

- A fully local, offline-capable classifier
- A hybrid pipeline (rule-based + ML)
- Transparency through LIME explanations
- Customizable taxonomy using YAML
- A streamlined, interactive UI for both individual and batch use

This directly benefits personal finance users, small businesses, enterprise teams, and developers who need a dependable, interpretable, and privacy-safe classification engine.

1.3 Objectives

The core objectives of the **AI Transaction Categorizer** are:

1. **Develop a hybrid classification engine** that combines deterministic keyword rules with probabilistic ML classification.
2. **Ensure full local execution** with no external API or cloud dependency, preserving privacy and compliance.
3. **Provide high classification accuracy** using TF-IDF vectorization and an ensemble model.
4. **Enable explainability** through LIME, delivering token-level insights into classification decisions.
5. **Support scalability** through single and batch prediction pathways.
6. **Offer a user-friendly interface** with Streamlit for prediction, configuration, and visualization.
7. **Make the taxonomy system editable**, allowing users to customize rules without modifying code.
8. **Generate reproducible outputs** including metrics, confusion matrices, and model artifacts.

1.4 Scope

The scope of this project includes:

- **Designing and implementing** a local-only ML + rule-based classification system.
- **Building a training pipeline** that generates synthetic data, trains the model, and exports artifacts.
- **Developing an inference pipeline** that processes both single and batch transactions.

- **Constructing a Streamlit UI** for user interaction, visualization, and configuration editing.
- **Integrating explainability tools** to support human-understandable model reasoning.
- **Providing automation tools** (setup.sh) for easy project initialization.

Out of Scope

- Real bank API integrations
- Live merchant database lookups
- Cloud-hosted inference
- Deep learning / transformer-based NLP
- Fraud detection or anomaly detection
- Multi-language transaction support (future enhancement)

1. Technology Stack

Core Languages & Runtime

Python 3.13 (Latest Stable Version)

The system is implemented entirely in **Python 3.13**, leveraging its mature ecosystem for machine learning, data processing, and rapid application development. Python provides:

- High compatibility with ML libraries
- Strong community support
- Easy integration with YAML, visualization tools, and web frameworks
- Excellent performance for CPU-based ML when optimized with vectorized operations

Local-Only Execution

The project follows a **privacy-first, offline-capable architecture**, meaning:

- **No external APIs**
- **No cloud calls**
- **No external data dependencies**

All computation, storage, and inference occurs on the user's device, ensuring robust security, data sovereignty, and compliance readiness (GDPR/SOC-2-aligned).

Backend & Machine Learning Stack

scikit-learn – Machine Learning Engine

(Used in `train.py`)
The classification engine is built using **scikit-learn**, chosen for its stability, interpretability, and suitability for structured/tabular NLP tasks. Components include:

- **TF-IDF Vectorizer**
  Converts text descriptions into weighted features using bi-gram modeling.
- **Logistic Regression Classifier**
  Provides linear decision boundaries with strong interpretability and robustness.
- **Random Forest Classifier**
  Captures non-linear relationships and enhances ensemble stability.
- **Soft-Voting Ensemble (VotingClassifier)**
  Combines Logistic Regression + Random Forest probabilities, improving overall classification quality and reducing variance.

Why scikit-learn?
Fast training, lightweight runtime, fewer dependencies, CPU-friendly, proven reliability for production workloads.

pandas and numpy – Data Foundation

(Used in `train.py`)
These libraries power all data engineering tasks:

- **Data loading & preprocessing** (CSV parsing, string cleanup, normalization)
- **Synthetic dataset generation** for training
- **Vectorized numerical operations** to optimize performance
- **Seamless integration with scikit-learn** pipelines

pandas and numpy ensure efficient and reproducible data operations across all environments.

LIME (lime.lime_text) – Explainable AI Module

(Used in `predict.py`)
LIME provides human-interpretable explanations for machine learning predictions.

Capabilities:

- Builds a **local surrogate model** around each prediction
- Highlights **token-level contributions** (top_words)
- Offers transparency for stakeholders, auditors, and debugging workflows
- Integrates directly into the Streamlit UI for analyst-friendly visibility

This transforms the classifier into a trust-enhanced system suitable for financial applications.

PyYAML – Rule Engine & Taxonomy System

(Used in `predict.py`)
The rule-based layer uses YAML for simple, robust, and editable configurations.

Purpose:

- **Merchant keyword matching** for high-confidence rule-based predictions
- **Category definitions** with metadata & thresholds
- **Hybrid logic control** between rule-based and ML-based classification

This allows domain experts to update classification rules **without modifying code**.

Frontend / UI Layer

Streamlit – Interactive Application Interface

(Used in `app.py`)

Streamlit provides a modern, reactive, and intuitive dashboard enabling:

- **Single transaction classification**
- **Batch CSV uploads** for bulk processing
- **Live taxonomy editing** through YAML controls
- **Model metrics visualization** (accuracy, F1 scores, category breakdowns)
- **Confusion matrix and probability charts**
- Streamlit abstracts UI development so the focus remains on ML and logic rather than frontend engineering.

Visualization Stack

Plotly

Used for:

- Dynamic probability distributions
- Category frequency visualizations
- Interactive charts for analysis and debugging

Matplotlib & Seaborn

Used for:

- **Confusion matrix heatmaps**
- Static diagnostic plots

These libraries provide a balance of interactivity (Plotly) and stability (Matplotlib/Seaborn).

Packaging & Environment Management

requirements.txt – Dependency Control

Lists all project dependencies, ensuring:

- Reproducible installations
- Version consistency across systems
- Clean environment setup
- Supports both local and cloud/container deployments.

setup.sh – Automated Bootstrap Script

This script automates complete environment initialization:

- **Creates a virtual environment**
- **Installs packages** from requirements
- **Builds directory structure** (`models/`, `outputs/`, `config/`, `data/`)
- **Runs the training pipeline**, generating:
    - TF-IDF vectorizer
    - Trained ensemble model
    - Categories list
    - Taxonomy
    - Metrics
    - Confusion matrix

This ensures a **single-command setup** for new users or environments.

2. System Architecture

The system is built using a **modular, layered architecture** that ensures clarity, maintainability, extensibility, and secure local-only execution. Each layer is isolated in functionality but collaborates seamlessly through well-defined data flows. This design makes the platform easy to scale, debug, iterate on, and deploy in privacy-sensitive environments.

2.1 High-Level Architectural Components

a. User Interface Layer — `app.py`

The user interface is implemented with Streamlit, offering a clean and interactive experience. It serves as the primary access point for analysts, testers, and end-users.

Key Capabilities

- **Single-Transaction Input Form**
  Users can type or paste a transaction description and instantly view:
  - predicted category
  - confidence score
  - rule-match or ML-based classification
  - optional LIME explanation
- **Batch CSV Uploader**
  Accepts bulk transaction files and outputs:
  - predicted categories
  - confidence levels
  - failure/flagged transactions
  - downloadable annotated CSV
- **Taxonomy YAML Editor**
  Direct editing of the keyword-based rule engine through a built-in YAML panel. Enables:
  - adding new keywords
  - adjusting thresholds
  - renaming labels
  - updating descriptions
- **Visualization Dashboard**
  Provides interactive charts including:
  - model metrics
  - confusion matrix heatmap
  - category distribution
  - prediction probability plots

The UI layer abstracts system complexity and enables low-code interaction for non-technical users.

b. Prediction Service — `predict.py`

This layer contains the **core decision engine** responsible for real-time transaction categorization. It combines rules with machine learning for accuracy, transparency, and robustness.

Resources Loaded

- `models/model.pkl` – trained VotingClassifier
- `models/vectorizer.pkl` – TF-IDF vectorizer
- `models/categories.pkl` – ordered list of categories
- `config/taxonomy.yaml` – YAML rule-based keyword definitions

Prediction Flow (Hybrid Pipeline)

1. **Apply Rule-Based Keyword Matching**
   The system checks the transaction text for keywords defined in the taxonomy.
   If matched → classification is returned immediately with high confidence (≈0.95).
2. **Fallback to Machine Learning**
   If no rule match is found, the TF-IDF + Ensemble classifier generates:
   - prediction
   - probability distribution across all categories
3. **Confidence Scoring**
   The model's probability is used to assign a confidence score and determine if the result should be flagged.
4. **Optional LIME Explainability**
   For single predictions, LIME generates a token-level explanation, helping users understand how the model arrived at its decision.

Returned Prediction Object

The final output includes:

- `category` – final predicted label
- `confidence` – probability or rule confidence
- `method` – `"rule_match"` or `"ml_model"`
- `probabilities` – full softmax-style distribution
- `explanation` – LIME "top contributing words" list

c. Training Pipeline — `train.py`

The training pipeline is designed for full automation, enabling reproducibility and consistent behavior across environments.

Responsibilities

- **Synthetic Dataset Generation**
  Creates a balanced training dataset simulating real transaction patterns.
- **TF-IDF Vectorization**
  Constructs bi-gram text features and transforms input descriptions.
- **Model Training**
  Uses a VotingClassifier combining:
  - Logistic Regression
  - Random Forest
- **Cross-Validation Scoring**
  Evaluates performance and stores metrics for transparency.
- **Artifact Exporting**
  Outputs include:
  - **Model artifacts** → `models/model.pkl`, `vectorizer.pkl`, `categories.pkl`
  - **Metrics** → `outputs/metrics.json`
  - **Confusion Matrix** → `outputs/confusion_matrix.png`
  - **Taxonomy** → `config/taxonomy.yaml` (auto-generated if missing)

This pipeline ensures consistency, reproducibility, and ease of retraining.

4. File-Based Storage Layer

A lightweight storage model ensures simplicity and portability without needing a database.

Directory Structure

Folder	Purpose
`data/`	Synthetic training CSV
`models/`	ML model + vectorizer artifacts
`config/`	YAML taxonomy rules
`outputs/`	Metrics JSON + confusion matrix PNG

Why No Database?

- Reduces infrastructure complexity
- 100% local and offline
- Easy to package, distribute, and deploy
- DB integration can be added later if needed

2.2 Processing Logic

The architecture supports two major flows: Single prediction and batch processing.

Single Prediction Path

```
UI → TransactionPredictor.predict()
       → rule_match()
           → keyword hit? → high-confidence result
       → else ml_predict()
       → confidence score
       → optional LIME explanation
UI displays final output
```

This flow is optimized for real-time inference and explainability.

Batch Prediction Path

```
UI CSV upload → batch_predict()
               → rule engine + ML engine applied row-by-row
               → get_statistics()
               → output:
                     - category distribution chart
                     - low-confidence items list
                     - downloadable annotated CSV
```

Batch mode enables high-volume transaction processing at scale.

3. Data Model and Storage (Expanded & Professional)

The system uses a **lightweight, file-based data model** specifically designed for local, offline, and secure execution. Each component is optimized for transparency, portability, and rapid retraining.

3.1 Core Data Entities

Transactions Dataset (CSV) – Training Foundation

Used during model training and evaluation.

Fields:

- **description** – free-text merchant/transaction string
- **amount** – numeric transaction value
- **category** – ground-truth label (e.g., *Coffee/Dining*, *Fuel*, *Groceries*, etc.)

This format ensures flexibility and compatibility with pandas, ML pipelines, and batch uploads.

3.2 Taxonomy / Rule Engine – `config/taxonomy.yaml`

This YAML file defines the **rule-based classifier**, enabling deterministic categorization before ML fallback.

Each category block contains:

- **name** – category label
- **threshold** – confidence requirement (default: 0.7)
- **keywords** – merchant/brand patterns
- **description** – human-readable category definition

Purpose:
Rules enable precise, explainable, and ultra-fast classification for known patterns.

3.3 Model Artifacts – `models/`

Files produced after training:

- **model.pkl** – VotingClassifier (Logistic Regression + Random Forest)
- **vectorizer.pkl** – fitted **TfidfVectorizer**
- **categories.pkl** – ordered category list

These files are consistently loaded by `predict.py` during inference.

3.4 Metrics & Diagnostics – `outputs/`

- **metrics.json** – macro F1, weighted F1, CV scores, sample counts
- **confusion_matrix.png** – visualization of per-category accuracy

This folder supports model transparency and enables ongoing monitoring.

3.5 Storage Approach

- **Lightweight, local-first design**
  - CSV (data)
  - YAML (taxonomy)
  - pickle (models)
  - JSON (metrics)
  - PNG (confusion matrix)
- **No database dependency**
  - Zero operational overhead
  - Ideal for offline & secure deployments
- **Easily migratable** to:
  - S3 / MinIO object storage
  - PostgreSQL / MySQL
  - BigQuery / Snowflake

The architecture is intentionally simple yet future-proof.

4. AI / ML / Automation Components

4.1 Machine Learning Model Design

TF-IDF Vectorization

Used for converting transaction text into numeric feature vectors.

Parameters (from `train.py`):

- `max_features=1000`
- `ngram_range=(1, 2)`
- `min_df=2`
- Automatic lowercase normalization

This provides robust text representation without requiring deep learning.

Ensemble Model Architecture

Logistic Regression

- `max_iter=1000`
- `class_weight='balanced'`
- Strong linear baseline with explainable decision boundaries.

Random Forest

- `n_estimators=100`
- `class_weight='balanced'`
- Captures non-linear relationships and complex merchant patterns.

VotingClassifier (Soft Voting)

- Averages probability outputs from LR + RF
- Produces stable, calibrated class probabilities
- Avoids overfitting while improving accuracy

This hybrid ensemble ensures both interpretability and robustness.

4.2 Rule Engine

A simple but powerful heuristic-based classification layer.

Flow:

1. Check transaction description against each category's keyword list.
2. If a match is found:
    - `category = category_name`
    - `confidence ≈ 0.95`
    - `method = "rule_match"`
3. If no rule is triggered:
    - Use ML classifier
    - `method = "ml_model"`

Advantages

- Highly interpretable
- Extremely fast
- Deterministic for common merchants
- Reduces load on the ML classifier

This hybrid rule + ML pipeline improves accuracy and transparency.

4.3 Explainability (XAI)

LIME Text Explainer

Used primarily in single prediction mode.

Provides:

- The strongest contributing words
- Positive/negative influence indicators
- Local surrogate model approximations

UI Visualization Includes:

- Token list
- Contribution weights
- Optional bar-style visuals

This ensures model decisions are auditable, compliant, and user-friendly.

4.4 Automation

`setup.sh` automates:

- Virtual environment creation
- Package installation
- Folder setup (data/models/outputs/config)
- Full training pipeline
- Taxonomy file generation
- Metrics and confusion matrix export

Batch Prediction Automation – `batch_predict()`

Handles:

- Bulk classification
- Confidence thresholds
- Automatic rule + ML processing
- Throughput reporting
- Downloadable annotated CSV

This enables efficient end-to-end operation with minimal user input.

5. Security and Compliance

5.1 Local-First Security Model

The system adheres to strict privacy principles:

- **No external API calls**
- **No cloud dependencies**
- **No remote requests**
- **No data transmission outside the device**

This architecture is ideal for environments that require strict data governance.

5.2 Data Handling

All files—including:

- transaction data
- configurations
- model artifacts
- predictions

are stored **locally** on the user's machine.

The Streamlit UI runs on **localhost**, ensuring no external exposure.

5.3 Security Practices

The system maintains clean separation of responsibilities:

```
data/      → input data
models/    → trained ML artifacts
config/    → YAML taxonomy rules
outputs/   → metrics & diagnostics
```

Additional safeguards:

- Taxonomy is fully auditable (YAML format)
- Git version control enables traceability

5.4 Compliance Alignment (Future-Ready)

Can align with:

GDPR

- No external data transfer
- User retains full control of data

SOC 2

Requires adding:

- Access control
- Logging of prediction requests
- Secure internal deployment (VPC)

Infrastructure Compatibility

- Enterprise-grade on-prem deployment
- Containerized microservices
- Private VPC environments

6. Scalability and Performance (Expanded & Professional)

6.1 Performance Characteristics

The TF-IDF + ensemble architecture provides:

- **Fast inference** even on CPUs
- **Low memory overhead**
- **No GPU requirements**

`batch_predict()` can process **thousands of transactions per second** on mid-range hardware.

6.2 Scalability Levers

Vertical Scaling

- More CPU/RAM
- Tensor-optimized servers
- Faster vectorizer caching

Horizontal Scaling

- Convert prediction engine into a **REST microservice**
- Deploy using:
    - Load balancers
    - Multiple replicas
- Shared model storage (NFS / S3 / MinIO / GCS)

6.3 Precomputation & Caching

- Vectorizer + model cached in memory
- Streamlit's `@st.cache_resource` accelerates UI interactions

6.4 Future Engineering Enhancements

- Full Docker containerization
- Kubernetes orchestration
- Migration from CSV to relational DBs
- Federated learning (privacy-preserving training)
- Model quantization for mobile/embedded devices
- Distillation of ensemble into a single, faster learner

3. System Architecture

The AI Transaction Categorizer is designed around a **modular, layered software architecture** that emphasizes clarity, extensibility, maintainability, and fully local execution. Each layer is isolated with a well-defined responsibility, ensuring that updates in one part of the system do not impact others. This architecture also enables secure offline operation, making the system suitable for privacy-sensitive or regulated environments.

3.1 High-Level Overview

The system architecture consists of four primary layers:

1. **User Interface Layer (Streamlit – `app.py`)**
   Provides interactive access for prediction, batch processing, taxonomy management, and visualization.
2. **Prediction Service (Hybrid Rule + ML Engine – `predict.py`)**
   Performs real-time classification using a two-stage pipeline: rule-based matching followed by machine learning fallback, including optional explainability.
3. **Training Pipeline (`train.py`)**
   Handles dataset generation, vectorization, model training, cross-validation, and artifact export.
4. **File-Based Storage Layer**
   A lightweight, local file system that stores models, data, configurations, and outputs without requiring a database.

The architecture is intentionally simple yet robust, enabling both personal use and enterprise integration while maintaining full data locality.

3.2 User Interface Layer (Streamlit – `app.py`)

The UI layer serves as the primary interaction point for users, abstracting away system complexity and providing a modern, responsive experience.

Key Functional Modules

- **Single-Transaction Input Form**
  Users can enter any transaction description and instantly view:
  - Predicted category
  - Confidence score
  - Whether the prediction was rule-based or ML-based

    - Optional LIME explanation for transparency
- **Batch CSV Upload Module**
  Supports high-volume classification through:
    - CSV ingestion
    - Row-wise rule + ML classification
    - Summary statistics (category distribution, low-confidence entries)
    - Downloadable output CSV
- **Taxonomy YAML Editor**
  Lets users edit:
    - Category definitions
    - Keywords
    - Thresholds
    - Descriptions

  This makes the system adaptable to new merchants or domain-specific needs without writing code.

- **Visualization Dashboard**
  Includes:
    - Confusion matrix heatmap
    - Model performance metrics
    - Category distribution plots
    - Probability charts for ML predictions

The UI provides a full platform around the prediction service, making it accessible even to non-technical users.

3.3 Prediction Service (Hybrid Engine – `predict.py`)

The prediction service executes the classification logic through a layered decision pipeline.

Artifacts Loaded at Runtime

- **`models/model.pkl`** – trained VotingClassifier
- **`models/vectorizer.pkl`** – TF-IDF vectorizer
- **`models/categories.pkl`** – ordered category list
- **`config/taxonomy.yaml`** – YAML keyword-based rule engine

Two-Stage Prediction Pipeline

1. Rule-Based Keyword Detection

- The system scans the transaction description for keywords defined in the taxonomy.

- If a match occurs:
    - `method = "rule_match"`
    - `confidence ≈ 0.95`
    - Immediate, high-precision result

This layer is fast, deterministic, and explainable.

2. Machine Learning Fallback

If no rules are triggered:

- TF-IDF vectorization transforms the input text into numerical features.
- VotingClassifier (Logistic Regression + Random Forest) predicts the category.
- Confidence is computed from soft-voting probabilities.

Optional: LIME Explainability

For single predictions, the system returns:

- Top influencing tokens
- Contribution weights
- Interpretable local surrogate explanation

Output Structure

Returned predictions include:

- `category`
- `confidence`
- `method` ("rule_match" or "ml_model")
- `probabilities` (per-class distribution)
- `explanation` (for LIME-enabled requests)

3.4 Training Pipeline (`train.py`)

This pipeline is designed for **automated, repeatable, and transparent training** of the model.

Responsibilities

- **Synthetic Data Generation**
  Produces a balanced labeled dataset simulating real-world patterns.
- **TF-IDF Vectorization**
  Converts textual descriptions into bi-gram numeric representations.
- **Model Training**

    - Logistic Regression
    - Random Forest
    - Combined using soft-voting VotingClassifier
- **Cross-Validation**
  Ensures model reliability and generalization across categories.
- **Artifact Exporting**
  Outputs include:
    - `models/model.pkl`
    - `models/vectorizer.pkl`
    - `models/categories.pkl`
    - `outputs/metrics.json`
    - `outputs/confusion_matrix.png`
    - `config/taxonomy.yaml` (generated if missing)

By using synthetic training data, the system avoids the need for external datasets while maintaining consistent learning behavior.

3.5 File-Based Storage Layer

The entire system runs without a database, making it highly portable and easy to deploy.

Directory Structure

Folder	Purpose
`data/`	synthetic training CSV
`models/`	ML models and vectorizer artifacts
`config/`	YAML-based taxonomy rules
`outputs/`	metrics.json + confusion matrix PNG

Benefits

- Fully local
- Zero infrastructure complexity
- Ensures privacy and compliance
- Can be migrated to DBs or cloud storage if needed

3.6 Processing Logic

The processing logic describes how data flows from input to output across prediction modes.

Single Prediction Flow

```
User Input (UI)
      ↓
TransactionPredictor.predict()
      ↓
rule_match()
      → if keyword hit → deterministic high-confidence category
      ↓ else
ml_predict()
      ↓
confidence calculation
      ↓
(optional) LIME explanation
      ↓
UI displays final result
```

This flow supports high-speed interactive predictions with full explainability.

Batch Processing Flow

```
CSV Upload (UI)
      ↓
batch_predict()
      ↓
Row-wise Hybrid Classification (rule + ML)
      ↓
get_statistics()
      → category distribution
      → low-confidence items
      → summary metrics
      ↓
UI renders visualizations
      ↓
Downloadable annotated CSV
```

Batch mode is optimized for large-scale classification workloads.