Architecture & System Design Overview

Abstract

This platform gives you a fast, lightweight system for classifying transactions, and it works right out of the box—even when you're offline. It was made for places where speed, transparency, and control over your own data aren't just nice to have—they're absolutely essential. Under the hood, it runs on Python 3.13 and uses a hybrid inference engine. Basically, it mixes straightforward, rule-based logic with good old machine learning. It leans on TF-IDF for vectorization and combines Logistic Regression with Random Forest models, so you get solid predictions and can actually see why the system made a call, thanks to LIME's token-level explanations.

The whole thing is modular from top to bottom. The user interface, inference engine, training pipeline, storage—each part stands on its own. This keeps maintenance simple, cuts down on headaches, and makes it easy to plug into strict enterprise setups or on-premises environments. Since everything—data, models, settings—stays local and runs locally, you don't have to worry about privacy or compliance. It fits right in with frameworks like GDPR, SOC-2, and those tough financial data rules.

For storage, it relies on a clear, file-based approach. That means you can audit everything, repeat your training cycles without hassle, and manage how your models change over time. It handles big batches quickly and delivers fast results, all while staying efficient on regular CPUs. So, whether you're an analyst working solo or running a huge enterprise cluster, this thing scales up without breaking a sweat. Plus, the design's ready for the future—microservices, distributed inference, transformers, even GPU-accelerated training—it's all on the roadmap. Put it all together, and you've got a rock-solid foundation built for both heavy workloads and whatever your business throws at it down the line.

1. Technology Stack

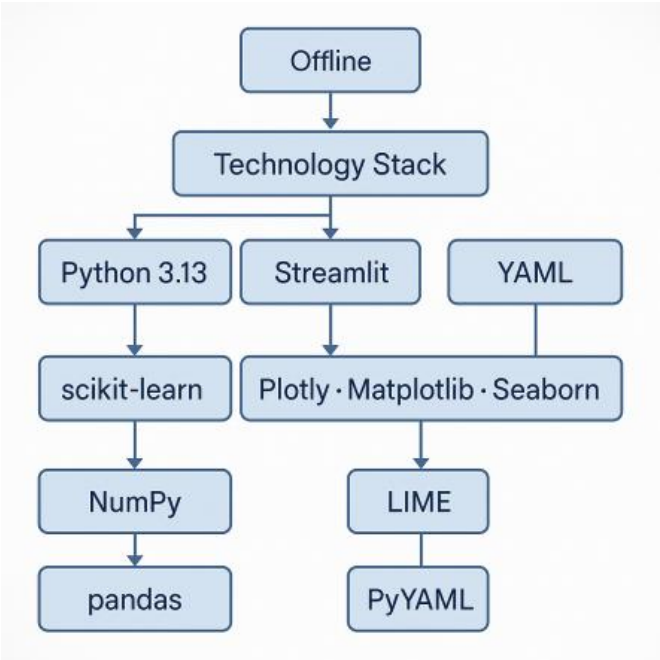
This platform runs on a modular, offline-first tech stack that's built for consistent performance, stability, and easy upkeep over time. At its core is Python 3.13—a rock-solid, well-supported language that keeps everything running smoothly. Python ties together the machine learning workflow: scikit-learn handles model management, NumPy powers fast number crunching, and pandas takes care of data engineering and pipelines.

For the front end, it uses Streamlit. That means analysts get simple, fast dashboards without the usual web development headaches. Streamlit's real-time engine keeps things snappy, even when crunching through big batches of data. On the visualization side, Plotly brings interactive analytics, while Matplotlib and Seaborn handle the static, high-quality charts needed for reports, diagnostics, and compliance.

The system also has explainability built in. LIME breaks down model behavior at the token level, offering clear, local explanations for every prediction. This is key for audits, building trust, and keeping up with strict financial regulations.

Configuration lives outside the codebase. All the taxonomies, rules, thresholds, and settings sit in YAML files, managed with PyYAML. This setup lets domain experts tweak business logic without waiting on developers—updates happen fast, no code changes required.

Everything runs fully offline—no outside connections, no external dependencies. This protects data privacy, shrinks the attack surface, and meets tough enterprise and regulatory standards. In the end, you get a platform that's tough, transparent, and ready to deploy anywhere: isolated networks, air-gapped systems, and secure enterprise environments.



2. System Architecture

The architecture breaks down into four layers, each handling its own job but working together as a tight, predictable whole. This setup isn't just neat for its own sake—it keeps everything modular, easy to scale, and straightforward to manage. You get a clear separation of concerns, which makes life easier when you need to maintain or update things, and it sets you up nicely if you ever want to move toward microservices or distributed computing later on.

User Interface Layer (Streamlit)

Here's where users actually interact with the system. Streamlit powers the UI, making it quick and lightweight. From this layer, analysts can do just about everything they need: run predictions one at a time or in bulk, upload transaction files, edit and check taxonomy definitions, see dashboards or diagnostics, and kick off training or refresh jobs. All the heavy lifting—business logic and machine learning—happens behind the scenes, so users never have to deal with the messy stuff. Even when you're working with big datasets, the interface stays snappy, since it talks to the prediction engine directly and doesn't waste time.

Prediction Engine (Hybrid ML + Rule Layer)

This is the brains of the operation. The predict.py module runs the inference show, combining classic rules with machine learning. It works like this:

- First, it looks for obvious matches based on taxonomy rules, keyword patterns, or merchant logic—so if it can answer right away, it does.
- If nothing fits, it hands things over to a hybrid ML setup: a TF-IDF vectorizer plus a VotingClassifier that mixes Logistic Regression and Random Forest. That way, you get a good balance between accuracy and being able to explain the results.
- For transparency, it uses LIME to break down why each prediction happened, which is great when you need to audit or explain things in regulated settings.
- This whole process makes the system accurate and trustworthy, and you don't get any weird surprises no matter how messy the transactions look.

Training Pipeline

The train.py module handles the machine learning lifecycle from start to finish. It can spin up synthetic data if you're starting cold, vectorize text data (using TF-IDF, bi-grams, and vocab rules), train models, tune hyperparameters, and score performance with cross-validation. It

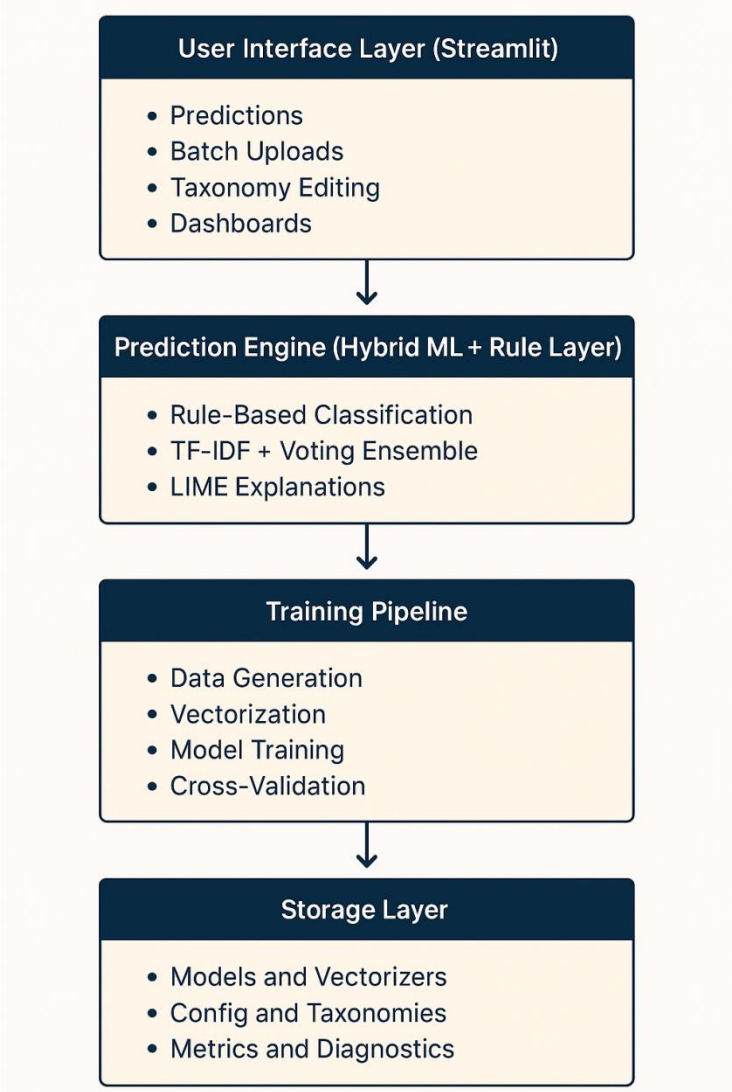
also handles generating and updating taxonomy, then packages everything up and stores it with versioning. The result? You always get reproducible, reliable results, which is a must when you're working in enterprise or locked-down environments.

Storage Layer

Everything—models, configs, diagnostics—gets saved in a simple but tough file-based storage setup:

- Model files and vectorizers go in models/
- Taxonomy config lives in config/taxonomy.yaml
- Metrics, confusion matrices, and diagnostics end up in outputs/
- Logs and system snapshots are stored too

This keeps deployments portable and transparent. You can move the setup around—local machine, enterprise network, containers—without messing with databases or cloud storage. It just works, wherever you need it.


User Interface Layer (Streamlit)
Predictions
Batch Uploads
Taxonomy Editing
Dashboards
Prediction Engine (Hybrid ML + Rule Layer)
Rule-Based Classification
TF-IDF + Voting Ensemble
LIME Explanations
Training Pipeline
Data Generation
Vectorization
Model Training
Cross-Validation
Storage Layer
Models and Vectorizers
Config and Taxonomies
Metrics and Diagnostics

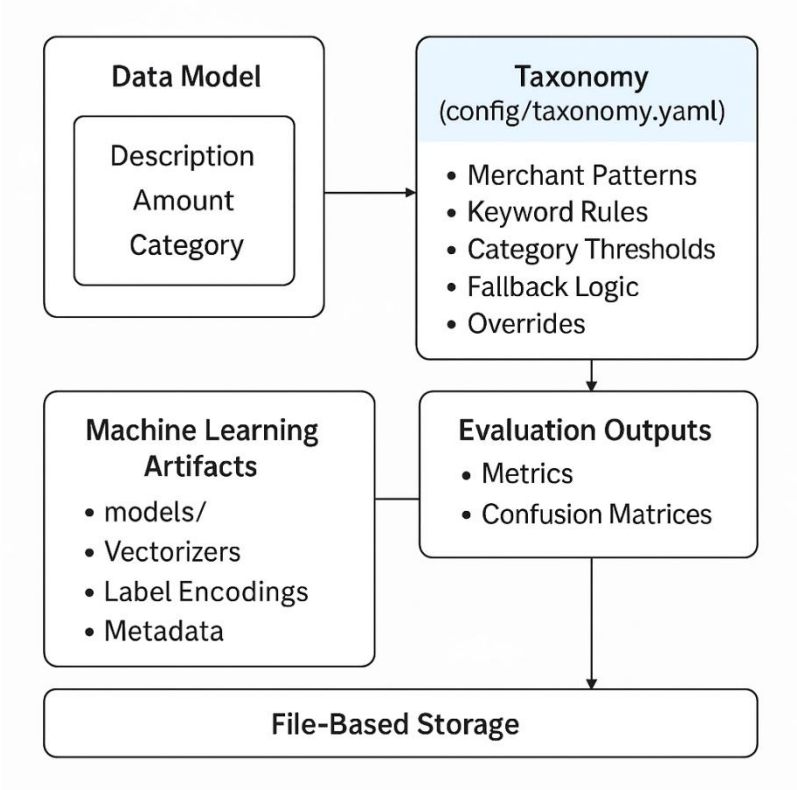
3. Data Model & Storage

I kept the data model simple on purpose. By focusing on just three key details—description, amount, and category—each record stays easy to handle, no matter where the data comes from. Banks, payment processors, ERP systems, ledgers—you name it, they all export data in this basic format. So everything fits right in, and you don't have to fuss with custom mapping or complicated conversions.

All the rules and logic for categorizing data sit outside the code, in config/taxonomy.yaml. That one file holds everything: merchant and keyword patterns, category thresholds, fallback rules, overrides, plus any extra metadata you need for category hierarchies. If someone needs to tweak business logic, they just update the YAML—no code changes or engineering help required. That means faster updates and way less red tape.

Machine learning stuff—models, vectorizers, label encodings, metadata—lives under the models/ folder. We version everything, so deployments stay consistent and portable, no matter the environment. Any evaluation outputs, like metrics or confusion matrices, go in outputs/. This setup makes it easy to monitor results, spot drift, or handle audits.

The whole system is file-based—no external databases, no cloud services. It runs anywhere, even offline or in locked-down, high-compliance settings. That keeps things simple to deploy and makes it much easier to track changes, reproduce results, and handle audits. All of this matters a lot when you're dealing with enterprise or regulated financial data.



4. AI / ML / Automation Components

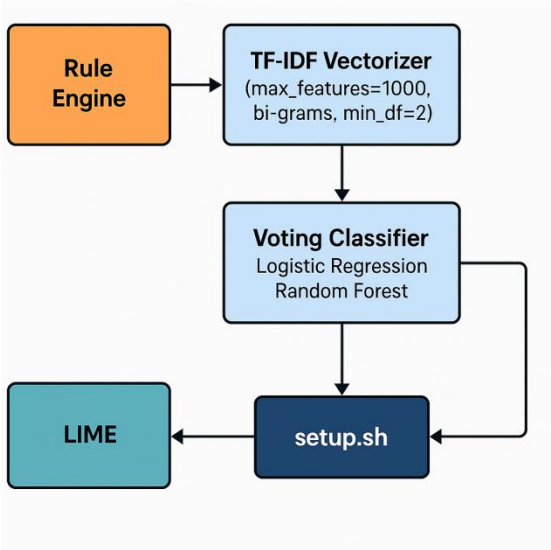
This machine learning pipeline is all about being easy to understand, reliable, and simple to run. At the heart of it, there's a TF-IDF vectorizer set up with 1,000 max features, bi-gram extraction, and a minimum document frequency of two. Basically, it's tuned to pick up on useful language patterns in transaction descriptions without slowing things down—so it runs fast, even on regular hardware.

For the classifier, there's a VotingClassifier that combines Logistic Regression and Random Forest. You get the best of both worlds: Logistic Regression is stable and easy to explain, great for categories that are clear-cut. Random Forest steps in for the weird edge cases, complicated interactions, or messy merchant descriptions. The result? Good accuracy, without needing the heavy machinery of deep learning.

To make sure things stay precise when it matters most, the system uses a rule engine before handing anything over to the ML model. This rule layer checks for known merchant patterns, certain keywords, and any hard-coded exceptions defined by the business. So, when a case is obvious, the rule engine takes over; when it's not, the model steps in. This way, you get reliable results for routine stuff, but the system can still handle surprises.

For transparency, there's LIME. It breaks down every prediction, showing which tokens tipped the scales. This helps analysts, auditors, and compliance teams see exactly why the system made a certain call—making it a lot easier to trust and debug.

Setting up the whole environment is automatic. The setup.sh script does everything: installs dependencies, sets up the environment, creates configs, trains the model, and gets everything ready to use. This makes onboarding a breeze—no headaches or surprises—and keeps everything reproducible, whether you're training again or moving to another machine.



5. Security & Compliance

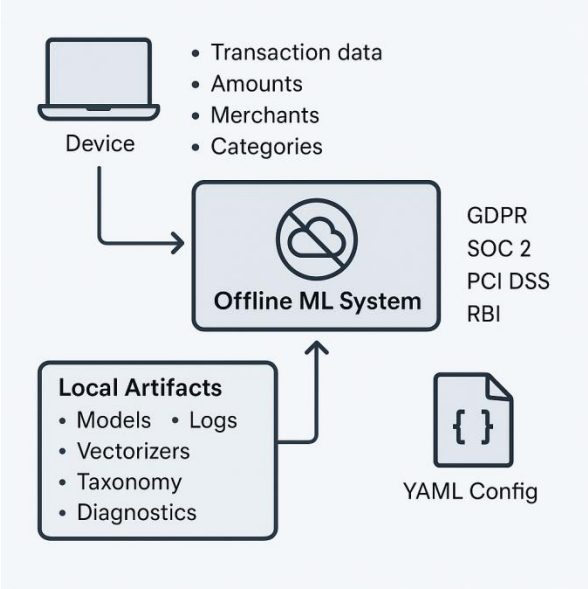
Everything runs directly on the device—no API calls flying out, no cloud dependencies. Sensitive transaction details like amounts, merchant info, and categories all stay local, locked down from start to finish. Because nothing leaves the device, the platform naturally fits strict privacy rules and cuts down the risk that comes with sending data over a network.

This offline-first setup lines up with all the big regulatory frameworks—GDPR, SOC-2, PCI-DSS, RBI—you name it. Since no personal or financial data ever leaves the user's environment, the system dodges headaches like compliance bottlenecks, outside vendor reviews, or questions about where the data actually lives.

Every operational piece—models, vectorizers, logs, metrics, taxonomy files, diagnostics—sits in a clean, file-based directory structure. It's predictable and easy to audit. You can track exactly which model version did what and when, which makes governance and repeatable audits straightforward.

When it's time to update classification rules or taxonomies, you just swap out YAML config files. There's no need to mess with the source code, so organizations can tweak logic safely. This setup keeps configuration and logic apart, tightens up change control, and lets compliance or risk teams stay hands-on without waiting for engineers to jump in.

In short, this architecture is built for security and control. It's made for banks, enterprises, or any high-stakes environment where privacy and auditability aren't just nice-to-haves—they're requirements.



6. Scalability & Performance

This platform is built for speed and flexibility, so you get reliable performance no matter where you deploy it. The ML setup is lightweight and modular, with vectorized preprocessing that keeps things moving fast, even when traffic spikes. Low-latency responses aren't just a goal—they're the norm here.

Performance

The inference pipeline uses vectorized TF-IDF transformations and a tight VotingClassifier ensemble. So predictions come quick, without putting extra strain on your CPUs. There's no heavy deep learning overhead in the way, which means you get fast results, even at scale.

The batch_predict workflow is all about throughput. It easily handles thousands of transactions per second on regular multi-core CPUs. Thanks to Streamlit's in-process design, user actions—like uploads, predictions, or drilling down into results—stay snappy, even during big batch jobs. You won't run into network lag that slows everything down.

Scaling Options

You get a lot of ways to scale up or out, depending on what you need.

Vertical Scaling

- More CPU cores mean more parallel inferences.
- Extra RAM lets you use bigger vocabularies and richer taxonomies.
- Switching to NVMe or SSD storage speeds up artifact loading and startup.

Horizontal Scaling

- Containerization makes it easy to clone the prediction engine and its environment.
- You can run multiple inference nodes behind load balancers to scale out as much as you want.
- Training jobs can run on their own compute nodes, so they never slow down real-time inference.

Service Decomposition

The architecture is modular, so you can break out the UI, prediction engine, storage, and training pipeline into their own microservices. That makes distributed execution, autoscaling, and fault isolation straightforward. You can go hybrid—cloud and on-prem—without a headache.

Deployment Flexibility

This thing runs wherever you need it:

- On a local analyst's workstation when network access is limited
- In on-prem clusters for regulated enterprises
- On air-gapped networks in banks or secure sites
- On edge devices or kiosks that need lightweight, standalone inference
- Inside container-based CI/CD pipelines for automated retraining, validation, and deployment

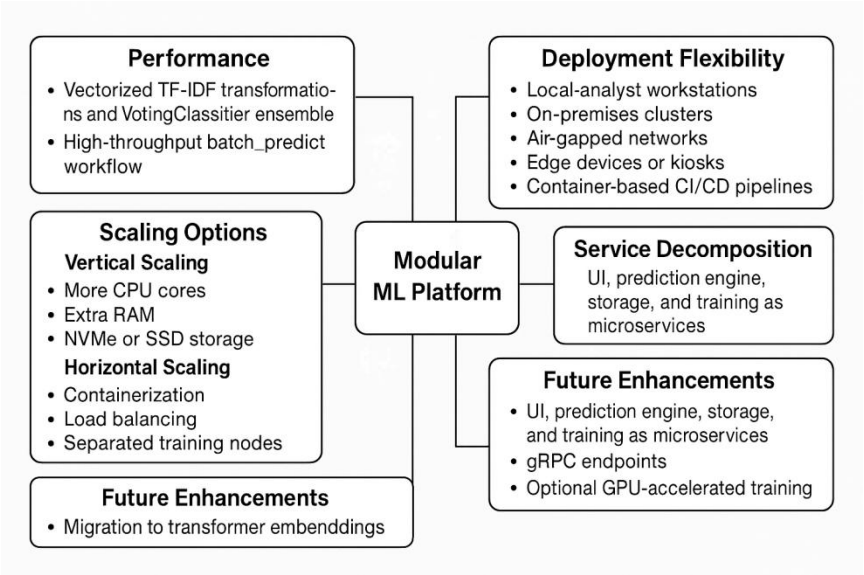
No matter where you put it, integration, scaling, and migration stay smooth.

Future Enhancements

The design leaves plenty of room for upgrades—no need to tear out the core systems. Coming improvements include:

- Easy migration to transformer-based embeddings for richer classification
- Plugging into fast stores like Redis or DuckDB for speedy lookups and preprocessing
- Automated model lifecycle management with job schedulers
- gRPC endpoints for real-time, low-latency classification
- Optional GPU acceleration for faster training, while inference stays efficient on CPUs

All this keeps the platform ready for bigger workloads, larger datasets, and more advanced ML features down the road.



7. Conclusion

This platform handles heavy financial workloads without breaking a sweat, even when transactions are flying in nonstop. It's built from the ground up with modular pieces—each one does its job on its own. That means you get steady, predictable performance no matter where you run it: online, offline, or locked down tight. Governance is rock solid. Data stays where it should, access is locked down, and every move leaves a trace. You can run this thing anywhere—on one analyst's laptop, all across a big company, or even deep inside an air-gapped network. No headaches, no endless reconfiguring.

Everything in the stack sits in its own lane. You've got the front end, the inference engine, the model training pipeline, and storage, all working independently. That separation wipes out chokepoints between components and keeps things future-proof. If your plans call for microservices, wide-scale inference, GPU-powered modeling, or the latest transformer architectures, you can pivot without any drama. Updates fit right in. No breaking what works, no scrambling to rewrite everything downstream.

Bottom line, we get a tough, flexible platform that grows with you and always plays by the rules—operationally and from a compliance angle. It delivers results right away, keeps latency in check, and is always ready for whatever comes next as your analytics get more advanced. You stay in control, scale up when you need to, and skip the usual migration mess.

GitHub Link: https://github.com/Thor-asgardian/AI-Transaction-Categorizer

Youtube link: https://youtu.be/0yoCW06De-w