Problem Statement:

Investors and analysts face difficulty in efficiently analyzing large volumes of stock and ETF data, understanding complex market indicators, and validating trading strategies before deployment. Manual analysis is time-consuming, error-prone, and lacks real-time responsiveness. Additionally, many users struggle to interpret technical metrics and backtesting results, limiting informed decision-making. There is a need for an integrated platform that automates signal generation, validates strategies through backtesting, delivers timely alerts, and explains insights in a user-friendly manner.

Proposed System:

To address these challenges, we propose a **Python-based AI-powered stock and ETF signal generation platform** built using a modular architecture. The system integrates financial data APIs to fetch historical market data, applies machine learning models to generate buy/sell/hold signals, and performs backtesting to evaluate strategy performance over the past five years. Real-time alerts are generated based on ML signals and backtesting confidence scores and delivered via email according to user preferences. A Streamlit-based dashboard provides user authentication, visual analytics, and GenAI-driven explanations, enabling users to clearly understand trading signals, performance metrics, and alerts. The entire system is integrated using FastAPI-based backend services for scalable and efficient communication between modules.

Tools and Technologies Used:

- **Programming Language:**
  - Python
- **Data Ingestion**
  - yfinance API (historical stock & ETF data)
  - Supabase (ticker storage and retrieval)
- **Machine Learning**
  - Random Forest
  - XGBoost
  - LSTM
  - GenAI (for explaining stock data, metrics, and results)

Backtesting

- VectorBT (strategy backtesting and performance evaluation)

Alerts & Notifications

- Gmail SMTP (email alerts)
- Confidence scoring from the backtesting engine
- User-defined alert scheduling

Backend & Integration

- FastAPI (RESTful backend services)
- Modular API-based architecture

Dashboard & Visualization

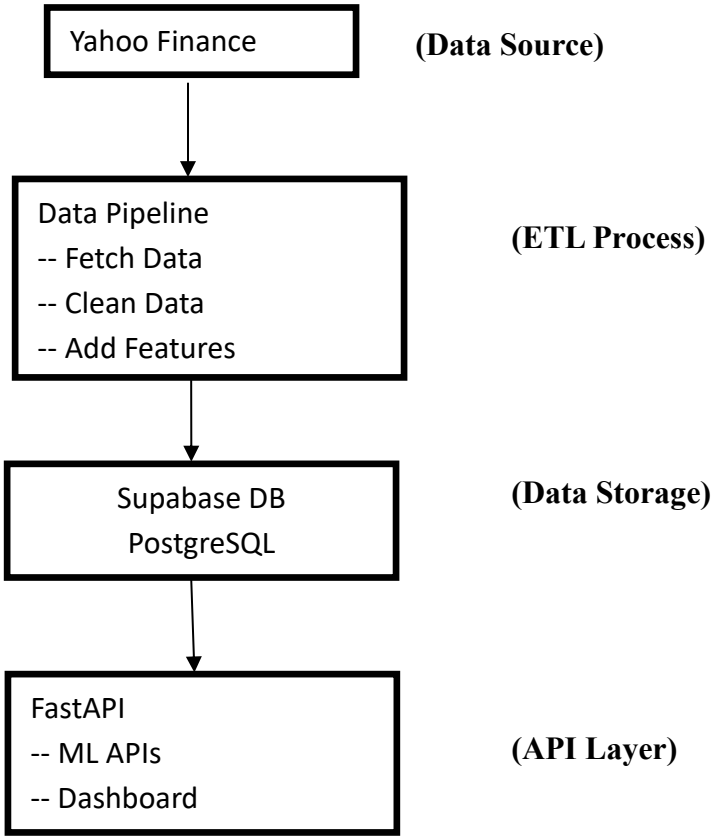
- Streamlit
- ML - Signals
- OHLCV charts
- Equity Curve Graph
- Profit & Loss Graph
- Cumulative returns Graph
- Alert management controls

DATA PIPELINE

Overview:

This implements an automated backend infrastructure for **real-time stock market** analytics. It handles **the ingestion, validation, and feature enrichment** of **60+ Indian stock tickers** to support downstream Machine Learning (ML) operations without any manual intervention.

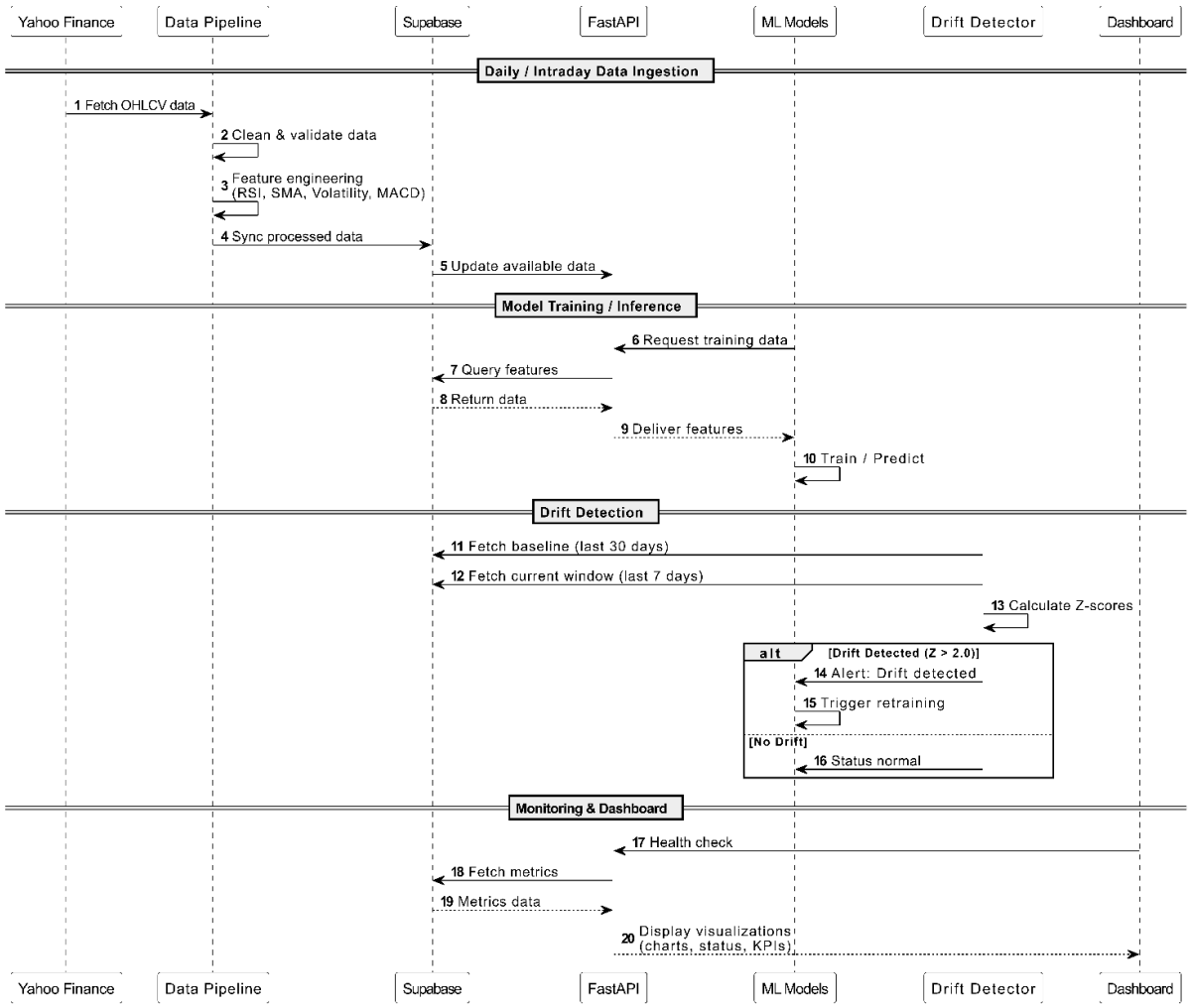
Architecture:



Tech Stack:

- python
- FastAPI
- Supabase
- pandas
- yfinance
- numpy
- pyarrow
- uvicorn
- python-dotenv

Flow Diagram:



Technical Architecture:

The system follows a **Medallion Architecture** pattern to ensure data quality and lineage:

- **Bronze Layer (Raw):** Immutable ingestion of OHLCV data from Yahoo Finance.
- **Silver Layer (Cleaned):** Application of business rules, OHLC consistency checks, and outlier detection.
- **Gold Layer (Enriched):** Dynamic feature engineering (RSI, SMA, MACD) stored for model consumption.

Technical Indicators Added for ML model:

- **Daily Return :** Measures day-to-day price movement and is a foundational feature for return-based prediction models.
- **RSI (14-period) :** Indicates momentum strength and potential reversal zones based on recent gains and losses.

- **Simple Moving Averages (SMA 20, SMA 50) :** Smooth price fluctuations to reveal underlying trends and act as dynamic support/resistance levels.
- **Rolling Volatility (20) :** Represents market instability and helps ML models adapt to changing risk regimes.
- **MACD :** Combines trend-following and momentum indicators to detect buy/sell crossover signals.
- **Volume Moving Average :** Helps validate price action by analyzing participation strength in the market.

Data Modeling (PostgreSQL):

Data is persisted in a **Supabase PostgreSQL** database using a time-series optimized schema.

- **Primary Table:** stock_features
- **Keys:** Composite Primary Key on (ticker, date).
- **Indexes:** B-Tree indexing on date and ticker for high-performance retrieval.

ETL & Orchestration Logic:

- **Incremental Loading:** The pipeline checks the database for existing records and only fetches missing date ranges.
- **Parallel Processing:** Uses ThreadPoolExecutor to process multiple tickers concurrently, significantly reducing execution time.
- **Dual Persistence:** Data is stored locally as **Apache Parquet** (for speed) and in the **Cloud** (for accessibility).

Data Quality & ML outputs Governance:

- **Drift Detection:** Implements Kolmogorov-Smirnov (KS) tests to monitor feature distribution shifts.
- **Automated Alerts:** If data drift is detected (p-value < 0.05), the system logs an alert for model retraining.

STOCK TRADING SIGNAL PREDICTION SYSTEM

Overview:-

The machine learning–based system for predicting stock trading signals such as **BUY**, **HOLD**, and **SELL**. The system analyzes historical stock price data, technical indicators, and real-time news sentiment to make informed trading decisions.By combining traditional machine learning models, deep learning techniques, and sentiment analysis, the system aims to improve prediction accuracy and robustness in a highly dynamic financial market.

Strategy:-

Core Approach:

1. **Data-Driven Signals**
   Trading signals are generated using historical stock prices and technical indicators.
2. **Multi-Model Ensemble**
   Multiple machine learning models are combined to reduce bias and improve reliability.
3. **Sentiment Integration**
   Real-time news sentiment is incorporated to capture market reactions to external events.
4. **Threshold-Based Labeling**
   Percentage price change thresholds are used to classify trading signals.

Signal Generation Logic

- **BUY**: Next day price increase greater than 1%
- **SELL**: Next day price decrease greater than 1%
- **HOLD**: Price change between −1% and +1%

Feature Engineering and Technical Indicators:

After data collection, several **technical indicators** were identified and engineered based on financial relevance and model compatibility.

Selected Technical Indicators

1. **Daily Return**
    - Percentage change in closing price between consecutive days
    - Captures short-term price momentum
2. **Volatility (14-day rolling standard deviation)**
    - Measures market risk and price fluctuation intensity
3. **SMA Ratio (20-day Simple Moving Average Ratio)**
    - Ratio of current price to 20-day SMA
    - Indicates trend direction and overbought/oversold conditions
4. **EMA Ratio (20-day Exponential Moving Average Ratio)**
    - Gives more weight to recent prices
    - Reacts faster to recent market changes
5. **MACD (Moving Average Convergence Divergence)**
    - Difference between 12-day EMA and 26-day EMA
    - Used to identify trend strength and momentum shifts
6. **RSI** is a momentum indicator that measures the speed and strength of recent price movements to identify overbought or oversold conditions.

- **Range:** 0 to 100
- **RSI > 70:** Stock is **overbought** → possible price correction
- **RSI < 30:** Stock is **oversold** → possible price rebound

These indicators were chosen because they are:

- Widely used in financial analysis
- Computationally efficient
- Suitable for tree-based machine learning models

Models Used:

1. LSTM Model

Purpose:

The Long Short-Term Memory (LSTM) model is used for time-series prediction by learning sequential patterns in stock price data.

Features Used:

- Open, High, Low, Close, Volume (OHLCV)
- Company encoding
- Technical indicators:
    - Daily Return
    - Volume Change
    - Moving Averages (MA20, MA50)
    - Volatility
    - RSI
    - EMA12, EMA26
    - MACD

Architecture:

- Bidirectional LSTM layers
- Dropout layers for regularization
- Dense output layer with softmax activation
- Lookback window of 10 days

2. Random Forest Model

Purpose:

Random Forest is a tree-based ensemble model that provides robust and interpretable predictions.

Features Used:

- Relative Strength Index (RSI)
- Daily Return
- 10-day Simple Moving Average (SMA_10)
- 50-day Simple Moving Average (SMA_50)

Configuration:

- Number of estimators: 100
- Maximum depth: 5
- Random state: 42

Reason for Choosing Depth = 5:

- Shallow trees reduce overfitting
- Boosting compensates by sequential learning
- Improves generalization on unseen market data

3. XGBoost Model

Purpose:

XGBoost is a gradient boosting model optimized for performance and speed.

Features Used:

- RSI
- Daily Return
- SMA_10
- SMA_50

Configuration:

- Number of estimators: 300
- Maximum depth: 6
- Learning rate: 0.05

- Subsample: 0.8
- Column sample by tree: 0.8

Reason for Choosing Depth = 6:

- Prevents overfitting on noisy stock data
- Ensures trees learn meaningful patterns rather than memorizing data
- Balances model complexity and generalization

SENTIMENT ANALYSIS:

GenAI-Based Sentiment Integration:

Purpose of GenAI:

While machine learning models provide numerical predictions, they lack interpretability. GenAI is integrated to:

- Explain predictions in human-readable language
- Improve trust and transparency
- Assist decision-making

How GenAI Works in the System:

1. ML ensemble generates a prediction score
2. The score and trading signal are passed to a GenAI model
3. GenAI generates a concise explanation describing the market outlook

Example Output:

"The stock shows positive momentum with supportive technical indicators, suggesting a potential buying opportunity."

Benefits of GenAI Integration:

- Converts complex ML outputs into understandable insights
- Bridges the gap between technical analysis and user interpretation
- Enhances usability for non-technical stakeholders

Ensemble Learning Approach:

An ensemble model is created by combining the predictions from:

- Random Forest
- XGBoost
- LSTM

Ensemble Logic:

- Both models predict the next-day return
- The final prediction is calculated as the average of 3 model outputs
- Trading signal is generated as:
    - **STONG BUY** -> All 3 models say buy
    - **BUY** → Positive predicted return
    - **SELL** → Negative predicted return
    - **HOLD**

Why Ensemble?

- Reduces bias from individual models
- Improves stability and robustness
- Performs better in volatile financial markets

Requirements:

Software:

- Python 3.8 or above
- pip package manager

Python Libraries:

- s++ensorflow
- keras

- matplotlib

Library	Purpose
yfinance	Provides market context
pandas	Data manipulation and preprocessing
numpy	Numerical computations
yfinance	Stock market data retrieval
scikit-learn	Random Forest model and preprocessing
xgboost	Gradient boosting model
joblib	Model serialization
FastAPI	API development
Uvicorn	ASGI server
GenAI (LLM)	Prediction explanation

- **Ollama** – Runs a local large language model to generate natural-language explanations for ML predictions
- **requests** – Sends API requests to GenAI services
- **python-dotenv** – Manages API keys securely(optional)

Mistral (GenAI Library / Model):

- Mistral is a lightweight, high-performance large language model (LLM).
- Used to generate natural-language explanations for ML model outputs.
- Runs locally via Ollama, so no data is sent to external servers.
- Converts ensemble predictions (Random Forest + XGBoost) into clear trading insights like BUY / SELL / HOLD with reasoning.

APIs:

- Google Gemini API for sentiment analysis
- NewsAPI for fetching financial news
- Ngrok for Fastapi

Environment Setup:

1. Clone the project repository
2. Create and activate a virtual environment
3. Install required Python libraries
4. Set API keys as environment variables
5. Place historical stock data (output.csv) in the project directory or use yfinance library

System Workflow:

1. User provides stock ticker
2. Historical data is fetched using yfinance
3. Technical indicators are computed
4. ML models generate predictions
5. Ensemble combines predictions
6. GenAI explains the final decision
7. BUY/SELL signal is returned via API

Files Description:

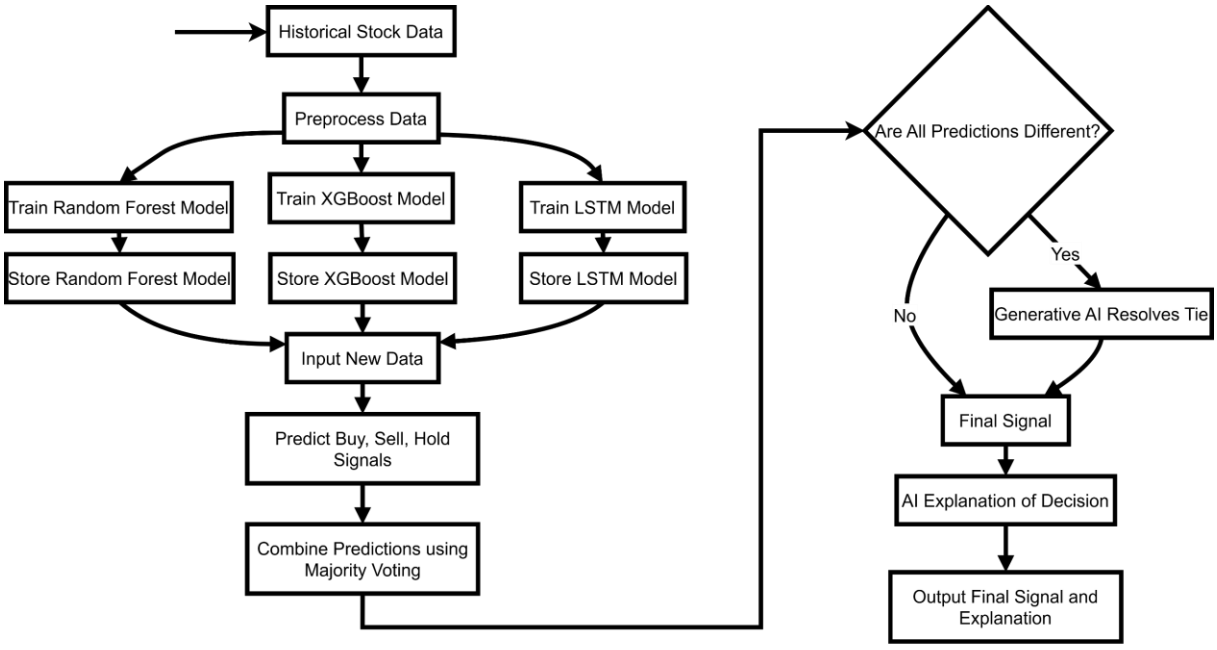
- **model (3).ipynb** – LSTM model training and evaluation
- **rf_model.ipynb** – Random Forest training
- **xgb_model.ipynb** – XGBoost training
- **genAI (1).ipynb** – News sentiment analysis
- **ensemble (1).ipynb** – Ensemble prediction system

- **output.csv** – Historical stock data
- **.pkl files** – Saved trained models

Model Performance:

Expected Metrics:

- Accuracy: 60–70%
- Precision and recall vary by signal type



BACKTESTING & ALERTS

Overview :

The Backtesting & Alerts Module is the validation and evaluation layer of the AI-powered Stock and ETF Signal Generation Platform. Its primary responsibility is to objectively test ML-generated trading signals on historical market data and determine whether those signals are profitable, stable, and risk-aware before being shown to users or used for alerts. This module bridges the gap between prediction (using ML models) and decision-making (through dashboards and alerts). It ensures that strategies are not judged solely by accuracy, but by their actual trading performance.

Why Backtesting is Critical:

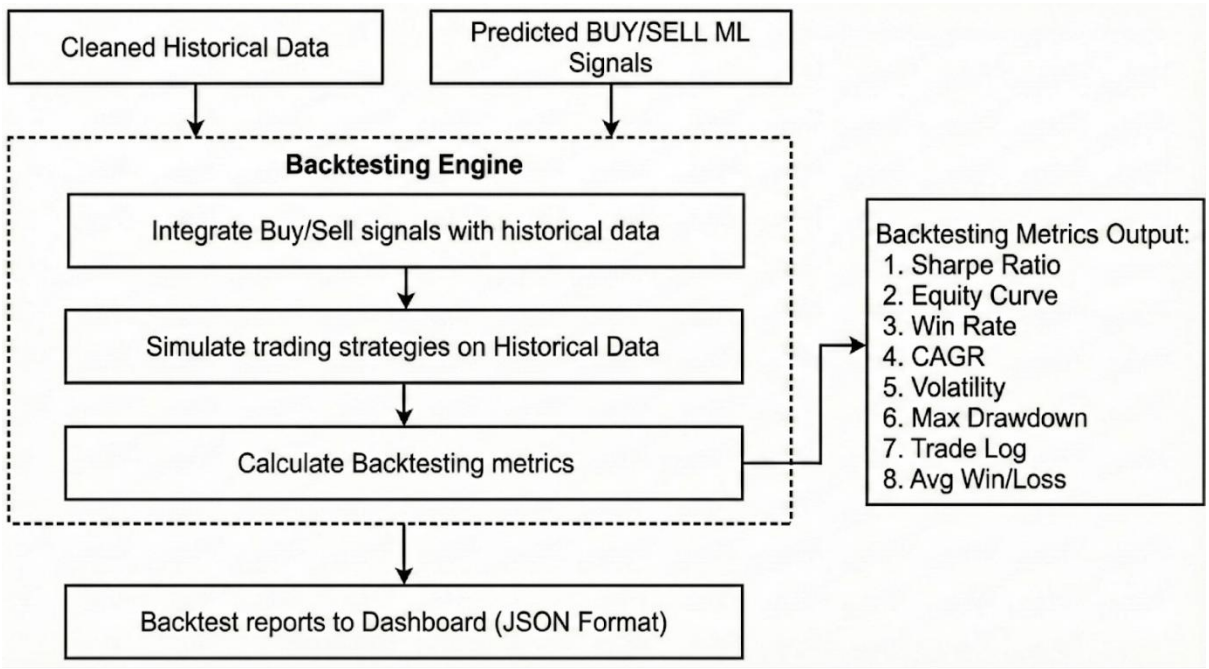
- Machine Learning models can generate **BUY/SELL** signals with high predictive accuracy, but still fail financially due to:
  - Poor risk management
  - Large drawdowns
  - Overtrading
  - Market regime changes
- Backtesting prevents these failures by answering:
  - Would this strategy have made money?
  - How risky was it?
  - How consistent were the returns?

Without backtesting, deploying ML signals directly is equivalent to gambling with historical ignorance.

Core Objectives of the Backtesting:

- Validate ML strategies on 5 years of historical data
- Quantify returns, risk, and consistency
- Compare ML strategy against Buy & Hold benchmark
- Produce API-ready outputs for dashboards
- Trigger alerts based on performance thresholds

Backtesting System Architecture:



System Workflow:

The workflow moves through three distinct phases:

1. **Data Ingestion (Input):** Aggregating clean market data, algorithmic strategies, and ML predictions.
2. **Processing (The Engine):** Integrating signals and simulating trade execution.
3. **Analytics & Reporting (Output):** Calculating performance metrics and formatting data for the frontend dashboard.

Component Breakdown:

A. Inputs:

The backtesting engine requires three critical inputs to function:

- **Cleaned Historical Data:** Time-series market data (OHLCV - Open, High, Low, Close, Volume) that has been pre-processed to remove outliers and fill gaps.
- **Predicted BUY/SELL ML Signals:** Output from machine learning models that classify market conditions or predict future price movements to generate trade signals.

B. The Backtesting Engine:

This is the logic core of the system. It performs the following sequential operations:

1. **Signal Integration:** It merges the "Predicted ML Signals" with the "Cleaned Historical Data", effectively mapping a decision (Buy/Sell/Hold) to a specific timestamp in history.
2. **Strategy Simulation:** It runs the trading strategies against the historical timeline using Vectorbt. This step simulates account balance, transaction costs, and trade execution as if they were happening in real-time.
3. **Metrics Calculation:** Once the simulation is complete, the engine aggregates the results to compute standard financial performance metrics.

C. Outputs

The system generates two types of output:

1. **Backtesting Metrics:** A statistical breakdown of performance.
2. **Dashboard Integration:** The final report is serialized into **JSON Format**. This standardized format allows the frontend dashboard to easily parse and visualize the results (charts, tables, and summary cards).

Input Requirements:

Historical Market Data:

The module requires OHLCV (Open, High, Low, Close, Volume) data.

Why OHLCV?

- Indicators depend on OHLCV
- Accurate trade simulation requires High/Low prices
- Volume is critical for volatility and liquidity signals

ML-Generated Trading Signals:

Signals are received from the ML pipeline in numeric format:

- 1 → BUY

- 0 → HOLD
- -1 → SELL

Signals are assumed to be time-aligned with market data.

Backtesting Configuration

- Initial capital
- Transaction costs (brokerage, slippage)
- Backtesting duration (5 years)
- Asset type (Stock / ETF)

These configurations ensure realism and reproducibility.

Backtesting Engine Design:

The engine simulates trades using Vectorbt as if executed in real markets:

1. Merge ML signals with historical prices
2. Convert signal changes into buy/sell orders
3. Track portfolio state:
    - Cash
    - Holdings
    - Equity value
4. Apply realistic constraints:
    - No over-buying
    - No selling without holdings
    - Transaction costs

Why VectorBT was Chosen:

Limitations of Traditional Loop-Based Backtesting:

- Slow execution for large datasets

- Difficult to scale to multiple assets
- Inefficient for ML-generated numerical signals

VectorBT Design Philosophy:

VectorBT is built on:

- NumPy vectorization
- Numba acceleration
- Parallel simulation

This allows thousands of trades to be simulated simultaneously without explicit loops.

Suitability for ML-Based Strategies:

VectorBT excels at:

- Handling numerical ML outputs
- Aligning predictions with price arrays
- Portfolio-level simulations
- Deterministic and reproducible results

Performance Metrics Generated :

- **Confidence Score**
- **ML Strategy Metrics**
  - Total Return (%)
  - CAGR (%)
  - Volatility (%)
  - Sharpe Ratio
  - Max Drawdown (%)
  - Total Trades
  - Win Rate (%)
  - Profit Factor

Metrics Definitions:

Metric	Definition	Purpose
Confidence Score	A value that shows how sure the system is about a Buy, Sell, or Hold signal based on past data and model predictions.	Helps users understand how much they can trust the generated signal before making a decision.
Sharpe Ratio	The average return earned in excess of the risk-free rate per unit of volatility or total risk.	Determines if returns are due to smart decisions or excessive risk.
Equity Curve	A graphical representation of the change in the value of a trading account over a time period.	Visualizes the stability of growth and drawdowns.
Win Rate	The percentage of trades that resulted in a profit.	$\frac{\text{Winning Trades}}{\text{Total Trades}}$. A high win rate doesn't guarantee profit if losses are large.
CAGR	Compound Annual Growth Rate. The mean annual growth rate of an investment over a specified period of time longer than one year.	Measures the smoothed annual growth of the portfolio.
Volatility	A statistical measure of the dispersion of returns for a given security or market index.	Indicates the level of risk/price fluctuation associated with the strategy.

Max Drawdown	The maximum observed loss from a peak to a trough of a portfolio, before a new peak is attained.	Highlights the worst-case scenario and downside risk.
Avg Win/Loss	The ratio of the average profit from winning trades to the average loss from losing trades.	Helps assess the risk/reward ratio of the strategy.
Profit Factor	Represents the overall percentage gain or loss generated over a specific period	Indicates the total performance of the strategy in percentage terms.

Market Benchmark (Buy & Hold) -

Buy & Hold serves as a baseline to evaluate whether ML adds value beyond simply holding the asset.

Metric Selection Rationale -

These metrics jointly measure:

- Profitability
- Risk
- Consistency
- Strategy robustness

Outputs :

- Confidence Score
- Metrics summary table
- Equity curve graph
- Trade PnL distribution graph

- Trade visualization graph

These outputs support debugging, auditing, and visualization.

API Layer and Dashboard Integration:

The module exposes results through a FastAPI service.

- High performance
- Automatic validation
- JSON-native responses

The API decouples computation from visualization. The dashboard consumes:

- Metrics JSON
- Time-series equity data
- Trade statistics

This enables flexible UI development without backend changes.

Environment & Execution Setup:

The module runs inside a Python virtual environment to ensure:

- **Dependency isolation -** Prevents conflicts between project-specific libraries (vectorbt, FastAPI, numpy, etc.) and system-wide Python packages.
- **Reproducibility -** Guarantees that the same library versions are used across development, testing, and deployment environments.
- **Clean deployment -** Makes the module portable and easy to deploy on servers, cloud platforms, or containers.
- **Core Dependencies Used -** The following libraries form the backbone of the Backtesting & Alerts module.

1. FastAPI - FastAPI is used to expose backtesting results and performance metrics as REST APIs for the dashboard and alert systems. It is fast, lightweight, and designed for production-grade APIs.

2. Pydantic - Pydantic provides automatic data validation and serialization for API inputs and outputs. It ensures that metrics, configurations, and responses follow a strict schema.

3. NumPy - NumPy is the foundation for numerical computation. VectorBT relies heavily on NumPy arrays for fast, vectorized backtesting operations.

4. Pandas - Pandas is used for handling time-series market data, aligning ML signals with price data, and preparing structured inputs for VectorBT.

5. VectorBT - VectorBT is the core backtesting engine. It enables parallel, vectorized simulation of ML-based strategies over large historical datasets with high performance.

6. Uvicorn - Uvicorn is an ASGI server used to run the FastAPI application efficiently in development and production environments.

Commands & Execution Flow:

- **Create a virtual environment**
  - Virtual Environment # Create a virtual environment
    python -m venv env
  - Activate the virtual environment:
    env\Scripts\activate #Windows
    source env/bin/activate # Linux / macOS
  - Deactivate the environment when finished:
    deactivate
- **Command to install all the dependencies**
  pip install fastapi pydantic numpy pandas vectorbt uvicorn

- **Dependency Management & Package Verification**
  - To inspect all currently installed Python packages inside the virtual environment, use the following command:
    pip list
  - Once the environment is set up and all dependencies are installed, the exact package versions are frozen into a requirements.txt file:
    pip freeze > requirements.txt
  - Installing Dependencies from requirements.txt
    pip install -r requirements.txt

- Verifying a Specific Package Installation

  pip show <package-name>

- Run backtests

- Start FastAPI server

  uvicorn src.api_client:app --reload

- Accessing the Running Server

Once started, the API will be available at:

http://127.0.0.1:8000

Interactive API documentation (Swagger UI):

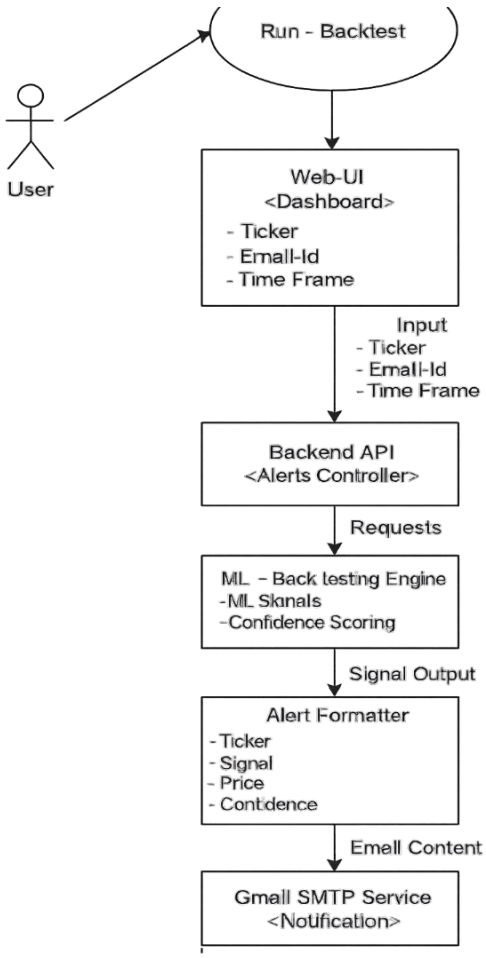
http://127.0.0.1:8000/docs

Alerts System Design :

The Alerts Service is responsible for notifying users when a strong and reliable BUY or SELL signal is detected for a stock.

Since raw ML signals can be noisy and risky, this service ensures that alerts are sent only when the ML signal is validated using historical performance and meets a confidence threshold.

Alerts System Architecture:



Purpose of Alerts Service -

The goal of the Alerts Service is to:

- Allow users to register stock alert preferences
- Continuously monitor ML trading signals
- Validate those signals using back testing
- Send alerts only when signals are reliable

Alert Objectives -

Strategy Success or Failure:

The system continuously evaluates how well the ML trading strategy is performing compared to the market.
If the strategy is consistently performing well, it allows alerts to be sent.
If the strategy performs poorly, alerts are suppressed to avoid misleading users.

Risk Threshold Breaches:

The system monitors risk metrics such as drawdown and volatility.
If the trading strategy becomes too risky (for example, large losses or unstable performance), alerts are blocked even if the ML model gives BUY or SELL signals.

Performance Degradation:

The Alerts System tracks how the ML model's performance changes over time. If the model starts performing worse than the market, it is considered unreliable, and alerts are not sent until performance improves.

Alert Channel -

Email Alert-

When a valid and high-confidence trading signal is generated, the system sends an email notification to the user. This allows traders to receive alerts even when they are not logged into the dashboard.

Trigger Conditions -

An alert is generated only when all required conditions are satisfied.

1. Sharpe Ratio Below or Above Threshold:

The Sharpe ratio measures risk-adjusted returns of the ML strategy.
If the Sharpe ratio falls below a defined safe level, the strategy is considered risky and alerts are stopped.

2. Drawdown Above Limit:

Drawdown measures how much the strategy has fallen from its highest value. If the drawdown crosses a defined limit, it indicates large losses, and alerts are blocked to protect users.

3. Profit Targets Achieved:

If the ML strategy is producing profits above a defined target compared to the market, the system allows alerts to be sent, indicating strong trading opportunities.

Inputs to the Alerts System-

The Alerts System receives data from two main sources:

From Dashboard:

- **Ticker symbol** (e.g., AAPL, TSLA)
- **User email ID**
- **Time Frame**

These are provided when a user registers for alerts.

From ML & Backtesting Engines:

- ML trading signal (BUY, SELL, HOLD)

Backtesting performance metrics such as:

- Strategy return
- Market return
- Risk measures

These are used to evaluate whether the signal is reliable.

Outputs from the Alerts System-

When all conditions are satisfied, the system generates an alert containing:

- Ticker name
- BUY / SELL / HOLD signal
- Confidence score
- Current price

Sample output:

Ticker: AAPL

Signal: BUY

Price: 150.00

Confidence: High

17. Error Handling & Edge Cases :

- No trades generated
- Flat or sideways markets
- Extreme volatility
- Missing or misaligned data

DASHBOARD & VISUALIZATION

Overview:

The Dashboard & Visualization Hub is the user-facing control center of the AI-Powered Stock & ETF Signal Generation Platform. It is responsible for transforming raw outputs from the Data Engineering, Machine Learning, and Backtesting modules into clear, interactive, and decision-ready visual insights.

While backend systems generate predictions, indicators, and performance metrics, the Dashboard Hub ensures that this intelligence is understandable, interpretable, and actionable for users.

This does not perform predictions or calculations. Its responsibility is to present, organize, and explain all analytical outputs in a professional trading-platform interface.

Why the Dashboard is Critical:

The most advanced ML models and backtesting engines are ineffective if users cannot see, trust, and interpret their results.

The Dashboard & Visualization Hub ensures:

- Signals are clearly visible (BUY / SELL / HOLD)
- Model confidence is shown visually
- Risk is displayed through drawdowns and volatility
- Performance is validated using equity curves and metrics
- AI reasoning is transparent through explanations and feature importance

Core Objectives of the Dashboard & Visualization Hub:

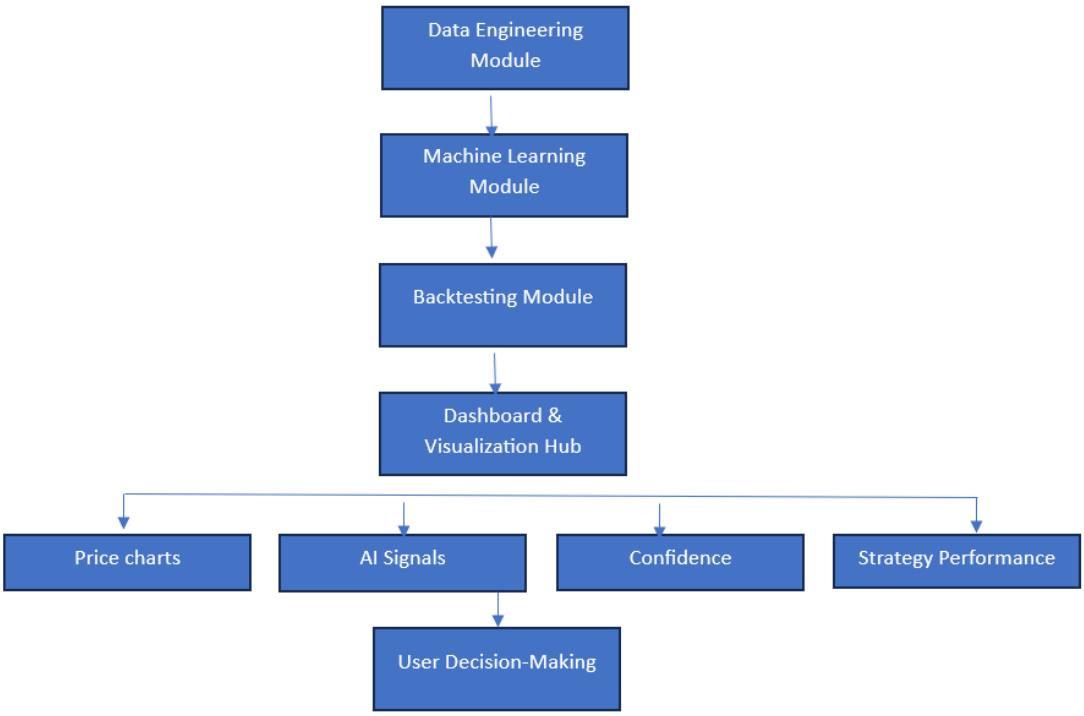
The module is designed to:

- Display real-time and historical stock and ETF data
- Visualize technical indicators and price trends
- Present ML-generated trading signals
- Show backtesting and strategy performance

- Provide risk and benchmark comparison
- Enable data export for reporting and analysis
- Act as the single interaction point for all users

System Position & End-to-End Workflow Architecture:

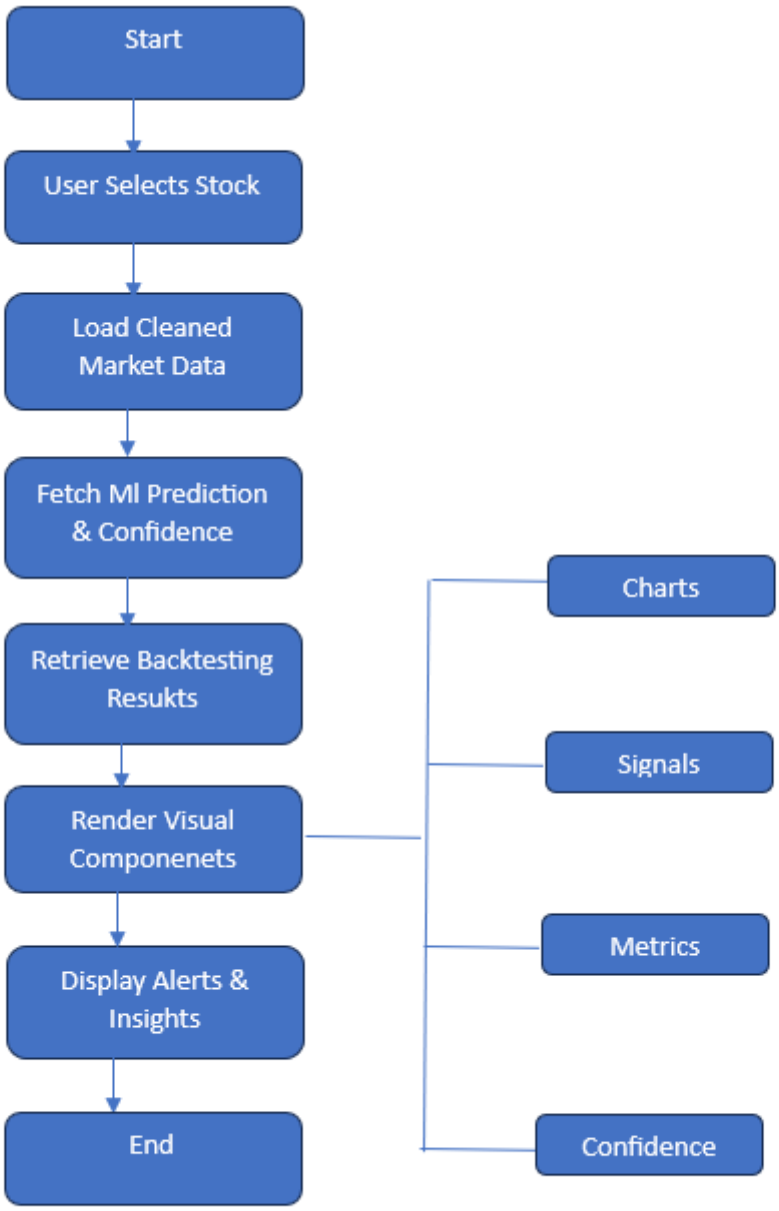
The Dashboard & Visualization Hub is the final presentation layer in the system. The platform follows a modular workflow where each component has a clearly defined role. Dashboard Hub – Visualizes, explains, and presents all outputs.



Dashboard Workflow:

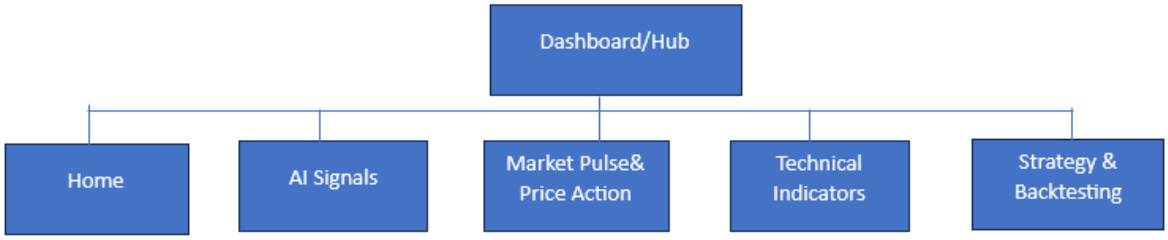
- User selects stock or ETF
- Cleaned price and indicator data is loaded
- ML prediction is fetched
- Strategy performance is retrieved
- Charts, metrics, and signals are rendered

- Alerts and confidence indicators are shown
- Data can be exported



Page Architecture of the Dashboard & Visualization Hub:

The Dashboard is implemented as a multi-page Streamlit application that replicates the experience of a professional trading terminal.



Home / Overview Page:

Purpose:

- Introduce the platform
- Provide high-level system understanding
- Enable navigation

Components:

- Core Capabilities (Trading Dashboard, Technical Analysis, Settings & AI Config)
- System highlights (50+ indicators, real-time feed, uptime, open-source status)
- Quick action buttons (Open Dashboard, View Analysis, Configure)

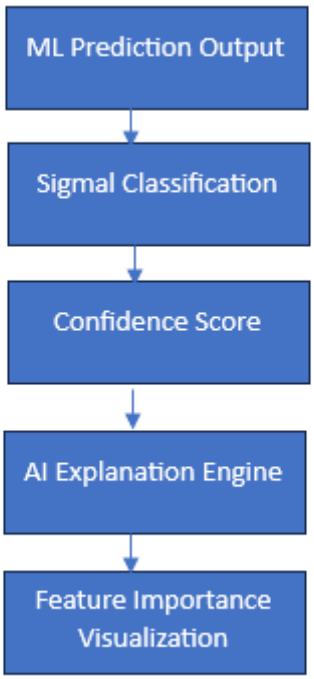
AI Signals Page:

Purpose:

- Display AI-generated BUY / SELL / HOLD
- Explain model reasoning

Components:

- Primary signal card
- Confidence bar
- AI explanation
- Key market insights (RSI, MACD, Trend)
- Feature importance bars



Market Pulse & Price Action Page:

Components:

- Price, High, Low, Volume cards
- Candlestick chart
- Volume bars
- Moving averages

Technical Indicators Page:

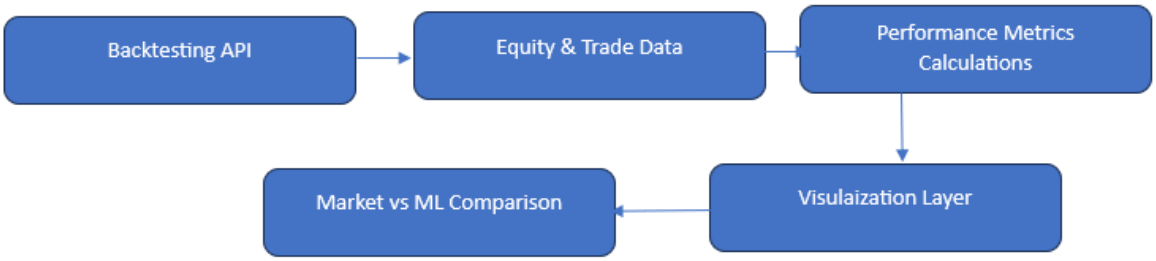
Components:

- RSI value and status
- MACD momentum and cross
- Trend status
- RSI momentum chart

Strategy Analysis & Backtesting Page:

Components:

- Equity curve (Market vs ML Strategy)
- Buy/Sell markers
- Trade activity
- Daily P/L
- Volume
- Metrics: total return, CAGR, Sharpe, drawdown, win rate.



Export Trading Data Page:

Components:

- Export price data
- Export AI signals
- Export backtest results

Input Sources:

Source	Data Received
Data Engineering API	Current price, prediction, signal, confidence
OHLC, Volume, Indicators	Backtesting API
ML API	Equity, returns, risk metrics

Visualization Components:

The platform uses professional financial visualizations:

- Candlestick charts
- Equity and indicator line charts
- Volume and P/L bar charts
- Summary metric cards
- BUY / SELL / HOLD badges
- Confidence bars
- Feature importance bars
- Trade markers

Technology Stack:

- Streamlit
- Plotly
- Pandas, NumPy
- Python
- Figma
- Git & GitHub

UI Blueprint & Design:

All pages and flows were first designed in Figma.

The blueprints defined page structure, navigation, and data placement before development.

Risk & Trust Layer:

- The dashboard displays:
- ML confidence
- Drawdowns

- Volatility
- Equity curves
- Strategy comparison

This prevents blind trust in predictions and promotes data-driven trading.

Dashboard Conclusion:

The Dashboard & Visualization Hub transforms data, ML, and backtesting outputs into a single professional trading interface. It enables users to see what the AI predicts, why it predicts it, and how well it performs. This makes the system usable, trustworthy, and investment-ready.