quantumquant inc. Reinforcement Learning Trading System Overview

System Architecture

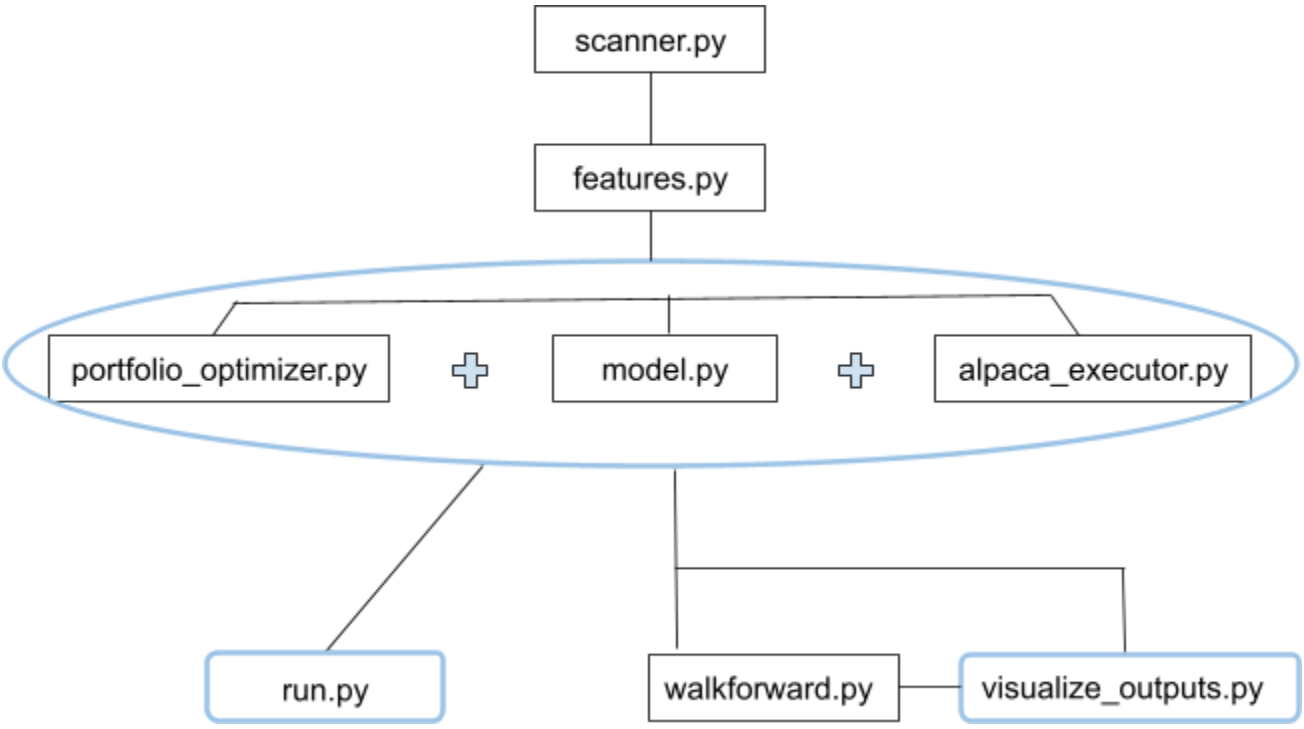


Table of Contents

Abstract

This paper presents a regime-aware reinforcement learning (RL) trading system designed to exploit directional movements using vanilla stocks and leveraged exchange-traded funds (ETFs). The system integrates automated trading universe selection, feature engineering, probabilistic regime detection, and portfolio optimization to dynamically allocate capital between bullish and bearish exposures. A hybrid architecture combining statistical modeling and reinforcement learning enforces strict regime-consistent constraints, reducing exposure to adverse market conditions. Empirical simulations demonstrate strong outperformance, with improved drawdown characteristics and enhanced directional accuracy.

Introduction

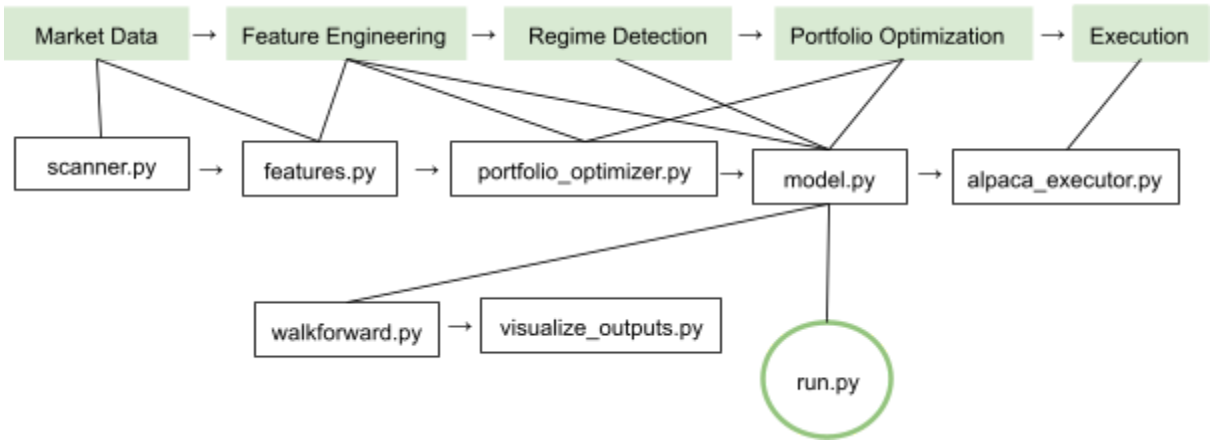
Financial markets are inherently noisy, adaptive, and non-stationary, making them difficult environments for purely rule-based or purely data-driven systems. Traditional algorithmic strategies often fail when market conditions shift, while unconstrained machine learning models may produce unstable or unsafe behavior in live trading environments. This paper presents a real-time trading system designed to operate under practical constraints. Rather than relying on reinforcement learning as a standalone solution, the system integrates learning-based signals within a structured decision framework that includes market regime classification, portfolio-level allocation, and execution-time safeguards.

The central idea is to separate prediction from decision-making. The reinforcement learning model generates directional signals, but the final execution is governed by deterministic rules that enforce risk control and market awareness. This hybrid design improves interpretability, robustness, and safety while maintaining adaptability to changing market conditions.

System Architecture Overview

The system is organized as a sequential pipeline in which raw market data is transformed into actionable trading decisions.

This process can be expressed as:



Market data is first processed into a structured representation through feature engineering. The resulting feature set is then used by a regime detection module to classify the current market state. This classification, along with the feature vector and portfolio state, is passed into a reinforcement learning policy that outputs discrete trading actions. These actions are subsequently executed through Alpaca Markets API interface.

scanner.py

Is essentially a **full end-to-end ML-powered stock scanner** that finds **high-probability "continuation trades"**—stocks likely to keep going up after showing strength.

"Which stocks are most likely to continue trending up tomorrow?"

What it does

☑ Builds Trading Universe

Downloads constituents list from web:

- S&P 500 (Wikipedia)
- NASDAQ listings (official file)
- Adds core ETFs like SPY, QQQ, IWM

Then:

- Cleans symbols
- Limits total count (`MAX_SYMBOLS`)
- Saves universe to CSV
- Outputs a large pool of tradable stocks.

☑ Downloads Market Data (Alpaca)

Uses Alpaca API to fetch:

- Daily OHLCV (Open, High, Low, Close, Volume)
- ~520 days of history

Then:

- Downloads in chunks (for reliability)
- Retries failed symbols
- Saves raw data to disk
- Result: clean historical dataset for all symbols

☑ Market Regime Detection

Computes overall market conditions using core indexes like:

SPY, QQQ, IWM, VIXY

Creates features like:

- `bull` / `bear`
- `risk_on_score`
- market returns + volatility

📌 Example:

- If QQQ + SPY are up → bullish
- If VIXY rising → bearish

Result: context-aware trading (not blind signals)

☑ Feature Engineering (Core Alpha Layer)

For each stock it computes:

📈 Trend Features ○ SMA5, SMA20, SMA50, SMA200 ○ SMA crossovers ○ Bullish stack alignment	**⚡ Momentum** ○ Returns (1d, 5d, 10d, etc.) ○ MACD (trend acceleration) ○ RSI (momentum strength)	**📊 Volatility** ○ Rolling std dev (10d, 20d)
📉 Bollinger Bands ○ Position in band ○ Breakouts / rejections	**💧 Liquidity** ○ Dollar volume ○ Volume spikes	**⚠️ Risk Signals** ○ Overbought RSI ○ Bollinger rejection ○ Weak MACD

☑ Trains a ML Model

Uses:

- `HistGradientBoostingClassifier` (very strong tabular model)
- Median imputation for missing data

Inputs:

- Engineered features (`daily_features.csv`)

Output:

- Probability stock continues upward

☑ Scores & Ranks Stocks

For the latest day, it computes the continuation **probability**. ML model outputs a **composite confidence score which is a**

Weighted mix of:

🧮 ML probability **+** Trend **+** Momentum **+** Liquidity **+** Regime **=** ***Confidence Score***

☑ Applies Strict Filters

It only keeps stocks that meet conditions like:

- Positive daily move
- Strong trend (SMA signals)
- Not overbought/reversing
- Good liquidity
- Not in bearish regime

If too strict → fallback logic relaxes filters

☑ Select Top Trades

Final selection TOP_N = 30

Sorted by:

- confidence
- score
- probability

☑ Results

👉 Scans the entire market
👉 Finds stocks already moving up
👉 Confirms they are strong (trend + momentum + volume)
👉 Removes risky setups
👉 Uses ML to estimate continuation probability
👉 Outputs the **top high-conviction trades**

Output:

- 📊 CSV with full data
- 📄 Text list of symbols:
- 📈 Charts:

    - ➪ continuation probability
    - ➪ composite score
- File names:
    - ➪ `live/latest_breakout_momentum_symbols.txt`
    - ➪ `live/latest_breakout_momentum_symbols_csv.txt`
    - ➪ `live/latest_breakout_momentum_candidates.csv`
    - ➪ `ive/latest_breakout_momentum_candidates.txt`
    - ➪ `live/latest_breakout_momentum_score_chart.png`
    - ➪ `live/latest_breakout_momentum_probability_chart.png`

DATA AND FEATURE ENGINEERING

Feature engineering captures multiple dimensions of market behavior, including momentum, trend, volatility, and structural characteristics. Momentum is represented through indicators such as relative strength index, moving average convergence divergence, and stochastic oscillators. Trend is captured via moving averages and their derivatives, including slope-based measures. Volatility is modeled using average true range and related expansion metrics. To stabilize learning, features are normalized and transformed into a continuous observation space suitable for reinforcement learning. Additional derived quantities, such as momentum agreement and dispersion, provide higher-level representations of market consensus.

features.py

This file is mainly the data prep layer. It creates the daily feature table the trading system needs. It's basically a daily data downloader + feature builder for the trading system. It prepares the dataset that the walk-forward/trade scripts can use. It downloads recent daily price/volume data, calculates technical indicators, adds market regime awareness, scores each symbol, and saves everything for the next part of the pipeline.

1. **Loads symbols**
    - Pulls S&P 500 symbols from Wikipedia.
    - Pulls NASDAQ-listed symbols from NasdaqTrader.
    - Adds core/regime ETFs like `SPY`, `QQQ`, `DIA`, `IWM`, `VIXY`.
    - Respects Alpaca's streaming limits by keeping the universe to `MAX_SYMBOLS = 30`, while forcing the core/regime symbols to stay included.

2. **Downloads daily Alpaca bars**
    - Utilizes the `.env` Alpaca keys.
    - Downloads about `520` days of daily OHLCV data.

- Uses `DataFeed.IEX`.
- Downloads symbols in chunks of `50`.
- If a chunk fails, it retries each symbol individually.

3. **Saves raw data**

   Writes:
   - `daily_system_outputs/daily_raw_bars.csv`
   - `daily_system_outputs/download_universe_symbols.csv`

4. **Builds features per symbol**
   For each stock/ETF, it calculates:
   - Returns: `return_1d`, `return_3d`, `return_5d`, `return_10d`, `return_20d`
   - Volume: `dollar_volume`, `relative_volume_20`
   - Moving averages: `sma_5`, `sma_20`, `sma_50`, `sma_200`
   - RSI: `rsi_14`
   - MACD: `macd`, `macd_signal`, `macd_hist`
   - Bollinger Bands: `bb_high`, `bb_low`, `bb_position`
   - ATR/risk: `atr_14`, `atr_pct`, `volatility_20d`
   - Breakout flags: `closed_above_prev_high`, `closed_below_prev_low`
   - Future returns: `future_return_1d`, `future_return_3d`, `future_return_5d`

5. **Adds market regime context**
   It merges proxy features from:

   ```
   SPY, QQQ, DIA, IWM, VIXY
   ```

   So every symbol row gets context like:

   ```
   qqq_return_5d
   spy_return_5d
   vixy_return_5d
   qqq_volatility_20d
   ```

   Then it classifies the market into:

   ```
   TREND_RISK_ON
   VOLATILE_RISK_ON
   CHOP_NEUTRAL
   DEFENSIVE_RISK_OFF
   ```

And maps that to a route:

`BREAKOUT_TREND_ROUTE`
`CONFIRMED_MOMENTUM_ROUTE`
`SELECTIVE_WATCH_ROUTE`
`DEFENSIVE_OR_CASH_ROUTE`

---

6. **Builds scoring columns**
   It creates:
   - `momentum_score`
   - `trend_score`
   - `breakout_score`
   - `risk_score`
   - `high_conviction_score`
   - `risk_on_score`

These are rough ranking/decision-support features for downstream scripts.

---

7. **Saves final feature dataset**
   It writes:

   `daily_system_outputs/daily_features.csv`

   Then prints the first 20 and last 20 rows.

REINFORCEMENT LEARNING POLICY

The decision-making component is implemented using Proximal Policy Optimization (PPO), a policy gradient algorithm designed to maintain stability through clipped updates. The policy operates on a discrete action space consisting of holding, entering, and exiting positions chosen by the rl model. The observation space includes the engineered feature vector, regime-related variables such as regime confidence and directional scores, and internal portfolio state variables including current position, unrealized profit and loss, and trade duration. The reward function is defined in terms of incremental changes in portfolio value while penalizing unfavorable trade dynamics. Formally, the reward at time t is given by

$$Rt = \Delta PnLt - \lambda \cdot Drawdownt,$$

where λ controls the penalty applied to adverse price movements. This formulation encourages profitable trades while discouraging excessive risk-taking. The system also includes a comprehensive monitoring layer that tracks all aspects of operation, including positions, profit and loss, and decision pathways. Each decision is logged with contextual information, including the original model signal, any applied overrides, and the final executed action. This provides a complete audit trail and enables detailed analysis of system behavior. Such transparency is critical for debugging, performance evaluation, and building trust in automated decision-making systems.

model.py

This file is the **supervised ML decision layer** for the trading system. It takes the `daily_features.csv` file from `features.py`, trains the model, predicts high-probability continuation setups, assigns actions, and produces a position-sizing plan. It **does not place trades**.

It reads:

`daily_system_outputs/daily_features.csv`

Then outputs:

`regime_predictions.csv`
`regime_actions.csv`
`regime_universe_30.csv`
`regime_universe_30.txt`
`regime_position_plan.csv`
`regime_feature_importance.csv`
`regime_threshold_table.csv`

First, it loads the feature dataset and makes sure required columns exist, especially future returns. Those are used only for training labels, not live prediction. Then it creates a target called:

`explosive_continuation_label`

A row is labeled positive only if the stock later has:

- `+0.30%` next day
- `+1.00%` over 3 days
- `+1.50%` over 5 days
- enough range expansion
- enough relative volume
- tradable liquidity/movement

So the model is not just learning “green tomorrow.” It is learning **strong continuation with upside expansion**.

Model logic

It trains an ensemble:

```
75% HistGradientBoostingClassifier
25% RandomForestClassifier
```

The model predicts:

- **explosive_continuation_probability**

That probability estimates how likely each stock is to produce the explosive continuation target.

It avoids leakage. The script explicitly blocks future-return columns, predictions, actions, target weights, and ID columns from becoming model features. That matters because otherwise the model could accidentally “cheat” by learning from future data. It uses market regime routing to classify the market into:

```
TREND_RISK_ON
VOLATILE_RISK_ON
CHOP_NEUTRAL
DEFENSIVE_RISK_OFF
```

Then maps those to routes:

```
BREAKOUT_TREND_ROUTE
CONFIRMED_MOMENTUM_ROUTE
SELECTIVE_WATCH_ROUTE
DEFENSIVE_OR_CASH_ROUTE
```

This controls whether the model is allowed to open longs, watch, hold, or liquidate.

It creates trade actions. Each symbol gets one of these:

```
OPEN_LONG
HOLD_LONG
WATCH_3_BARS
CONTINUE_HOLD_WAIT_CONFIRMATION
LIQUIDATE
NO_TRADE
```

The action depends on:

- model probability
- market regime
- tradability
- momentum state
- late reversal risk
- last 3 bars confirmation

It sizes positions. For the latest trading day, it builds a top 30 universe and assigns:

- `target_weight`
- `target_dollars`
- `recommended_shares`
- `stop_pct`
- `risk_budget_per_trade`

Sizing is regime-aware:

- **TREND_RISK_ON**: up to 60% gross exposure, more aggressive sizing
- **VOLATILE_RISK_ON**: reduced exposure
- **CHOP_NEUTRAL**: smaller/selective exposure
- **DEFENSIVE_RISK_OFF**: 0% exposure / cash route

Basically, this script takes your daily feature table and turns it into:

- Which stocks look explosive?
- Should I open, hold, watch, or liquidate?
- How much should each position be?
- What is the current market route?
- What 30 symbols should my trading universe use?

It is the **brain and sizing layer**, but not the broker/execution layer.

Order Execution

A defining characteristic of the system is the enforcement of regime-consistent trading behavior. In bullish regimes, the system restricts exposure to long positions, whereas in bearish regimes, exposure is limited to inverse long positions. Positions that conflict with the current regime are immediately liquidated upon detection of a regime transition. During transitional regimes, the system reduces trading activity by requiring a high confidence threshold before allowing new entries. This mechanism acts as a filter that mitigates noise and prevents overtrading in ambiguous market conditions. By constraining the action space in this manner, the

reinforcement learning policy is guided toward structurally valid decisions, improving stability and interpretability.

run.py

The Alpaca execution layer takes order CSVs created by `model.py` and actually submits market orders to Alpaca.

What it reads

```
daily_system_outputs/daily_walkforward_emulated_rebalance_orders.csv
daily_system_outputs/daily_walkforward_emulated_exit_orders.csv
```

These files should already contain planned buys/sells from the model or portfolio logic.

What it does

1. Loads Alpaca credentials from `.env`
2. Connects to Alpaca trading API
3. Checks account equity, cash, buying power, and current positions
4. Verifies the market is open unless overridden
5. Checks a daily-loss kill switch
6. Builds an executable order plan
7. Sells before buys
8. Applies confidence-based sizing to buys
9. Enforces cash buffer and max order limits
10. Submits Alpaca market orders
11. Tracks fills and logs slippage
12. Saves before/after account and position snapshots

Main protections

It has several production controls:

```
cash_buffer_pct = 8%
max_order_dollars = $15,000
max_daily_loss_pct = 2%
max_daily_loss_dollars = $2,000
sell_first = True
require_market_open = True
paper = True by default
```

So it tries to avoid:

- buying when cash would get too low
- trading after large daily losses
- buying during bad intraday windows
- oversized orders
- executing when market is closed

Intraday timing logic

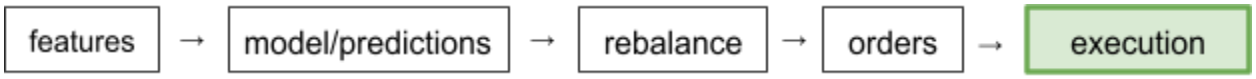
It changes behavior depending on time of day:

- skips buys during first 5 minutes
- skips buys during final 10 minutes
- reduces buy size early in session
- reduces buy size late in session
- still allows sells near close

Important detail

This script executes orders. The earlier model files only generated predictions, actions, target weights, or order CSVs. This one is the actual broker execution script.

So in the pipeline, this is the final step:



Experimental Results

Backtesting results indicate substantial performance improvements over a passive benchmark. The strategy achieves a final equity multiple significantly greater than that of the benchmark while maintaining lower drawdowns. The system demonstrates strong alignment with prevailing market trends, entering positions during sustained directional movements and exiting promptly during reversals. Performance gains are particularly pronounced during periods of high volatility, where regime detection provides a clear directional signal.

walkforward.py

Is the **walk-forward backtest simulator** for the system. Essentially it's the **research/backtesting layer**. It tests whether the model would have made money historically using rolling train/test windows.

It reads

`daily_system_outputs/daily_features.csv`

It trains an ML model

It builds a supervised ML model to predict next day price trend continuation:

- future 3-day return >= 1%
- or future 5-day return >= 1.5%
- or top 20% return for that date

It uses a soft-voting ensemble:

```
75% HistGradientBoostingClassifier
25% RandomForestClassifier
```

How walk-forward works

It repeatedly does this:

- ☑ Train on 252 trading days
- ☑ Test on next 42 trading days
- ☑ Step forward 42 days
- ☑ Repeat

That simulates retraining over time instead of training once on all history.

What the simulated strategy does

The simulator is aggressive and return-focused:

- trades only the strongest setups
- concentrates into max 3 positions
- allows max 2 new entries per day
- can allocate up to 45% into one position
- uses up to 95–100% gross exposure in strong regimes
- pyramids into winners
- lets winners run
- exits mostly by trailing ATR stops or major breakdowns

Entry logic

A stock is considered when it has:

- high explosive_alpha_probability
- strong explosion_setup_score
- volume participation
- range expansion
- breakout/trend alignment
- no late reversal risk
- tradable liquidity

Then it sizes positions using:

- Kelly-style expected value
- volatility adjustment
- regime multiplier
- convex ranking

So higher-confidence explosive names get much larger allocations.

Exit logic

It uses wide stops early, then locks profit later.

Main exits include:

- `TRAILING_ATR_STOP`
- `MODEL_DISASTER`
- `HARD_MOMENTUM_BREAKDOWN`
- `BIG_PROFIT_LOCK`
- `TIME_DECAY`
- `FORCED_WINDOW_END_EXIT`

It also has delayed profit locks:

+10% profit → lock at least +4%
+16% profit → lock at least +9%
+25% profit → lock at least +16%

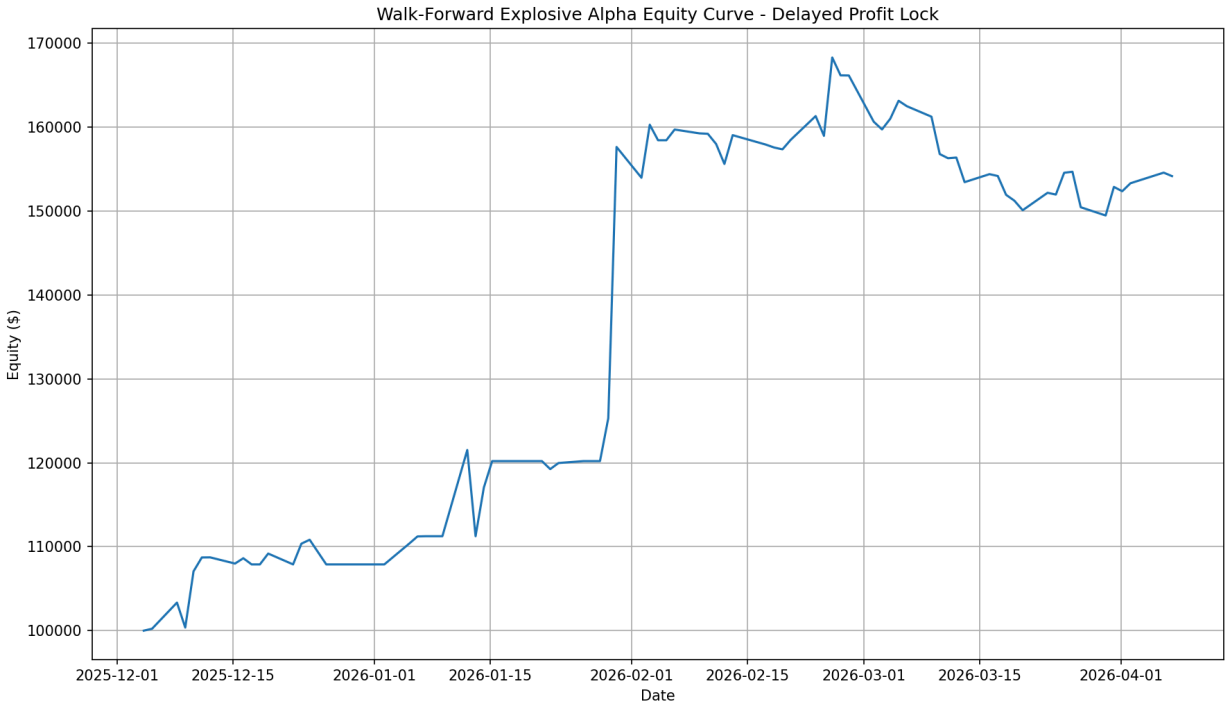
Outputs

```
wf_delayed_profit_lock_predictions.csv
wf_delayed_profit_lock_window_metrics.csv
wf_delayed_profit_lock_daily_equity.csv
wf_delayed_profit_lock_trade_log.csv
wf_delayed_profit_lock_events.csv
Wf_delayed_profit_lock_equity_curve.png
```

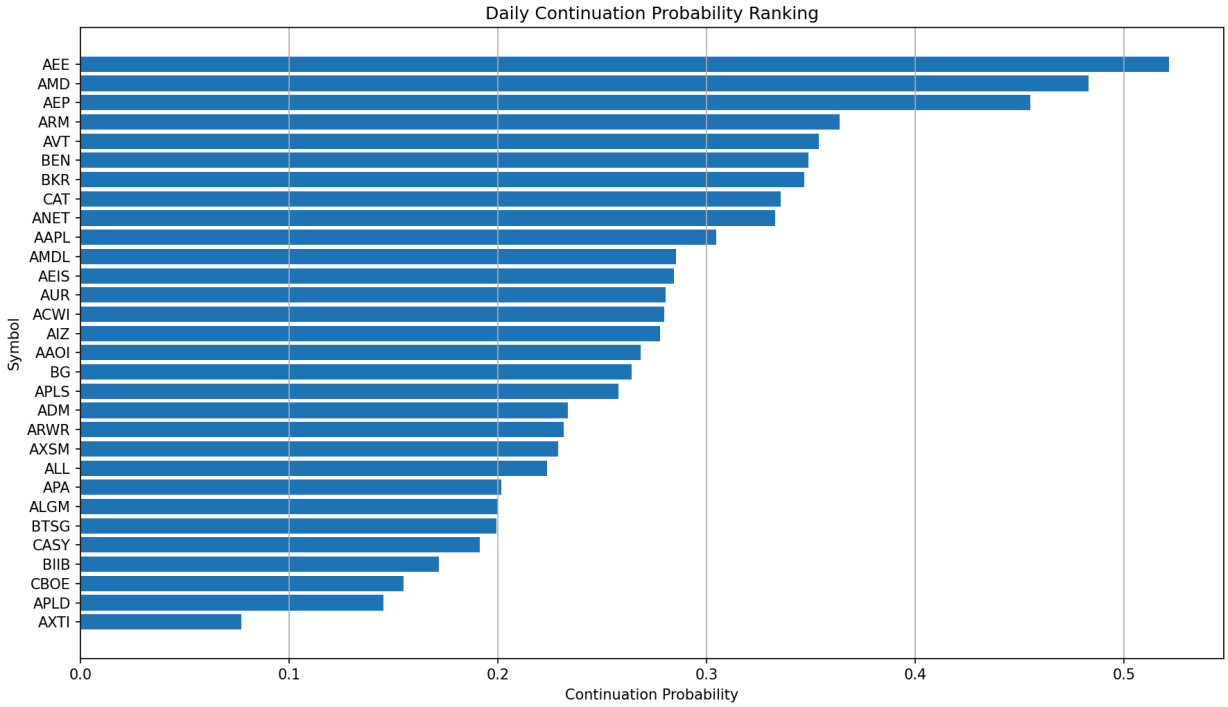
visualize_outputs.py

Is the **system image generator**. It runs after the model scripts finish, and creates visual summaries of model behavior, trade rankings, thresholds, and price history data generated by the walkforward.

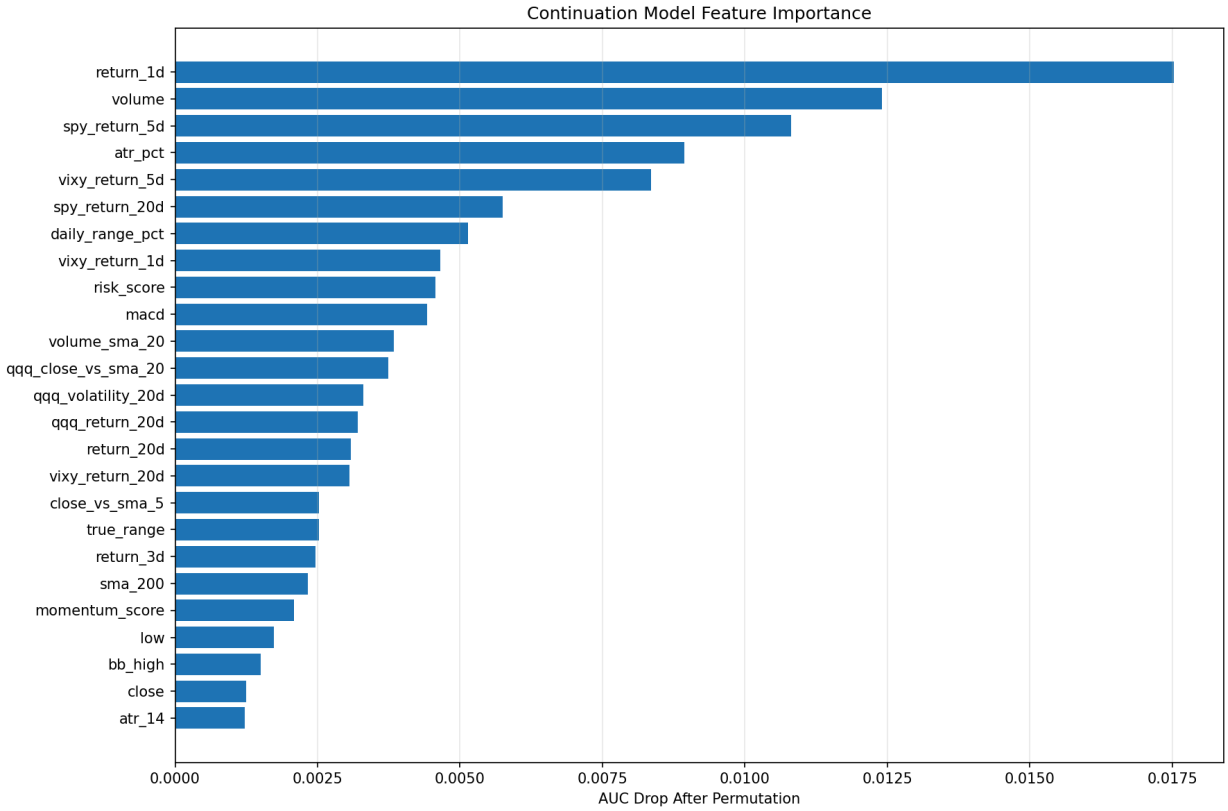
Portfolio Equity Curve



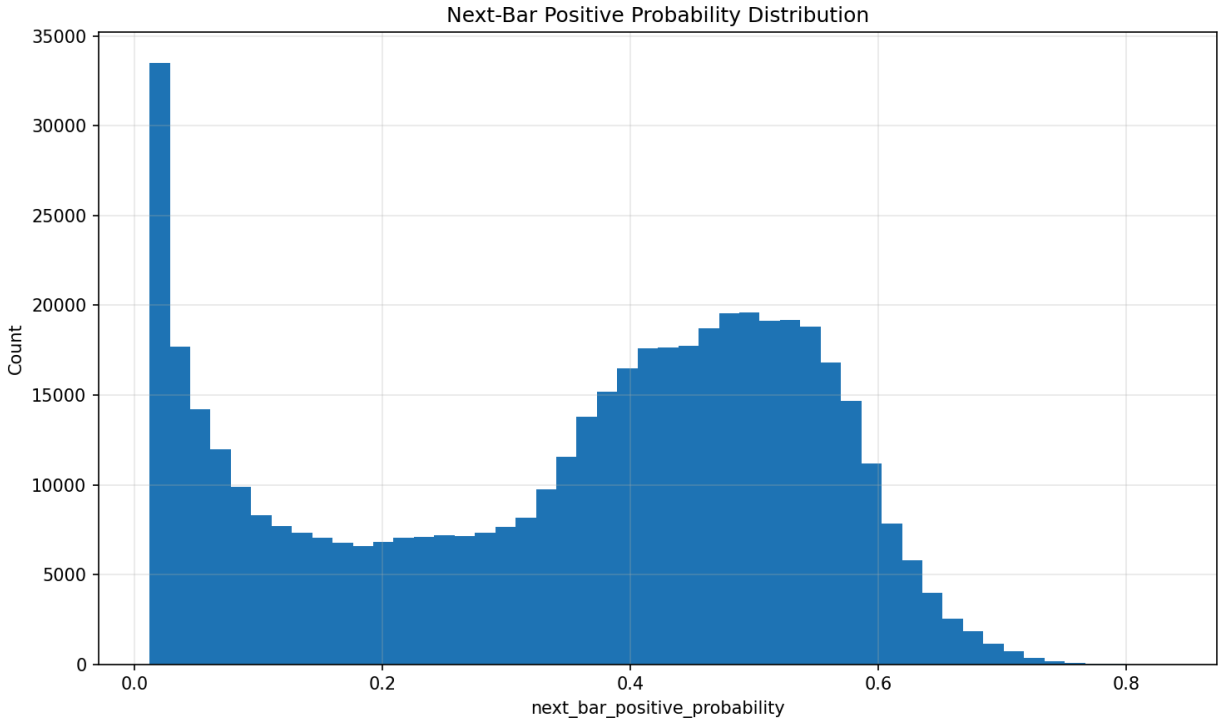
Continuation Probability Ranking



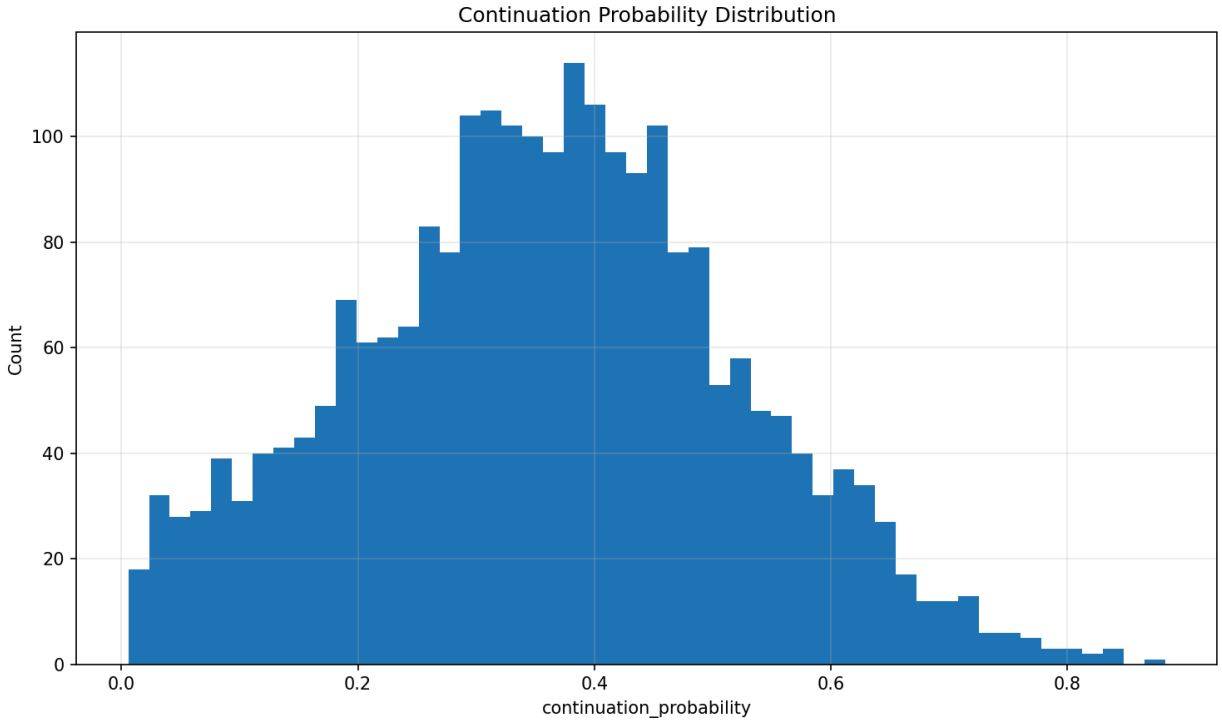
Feature Importance



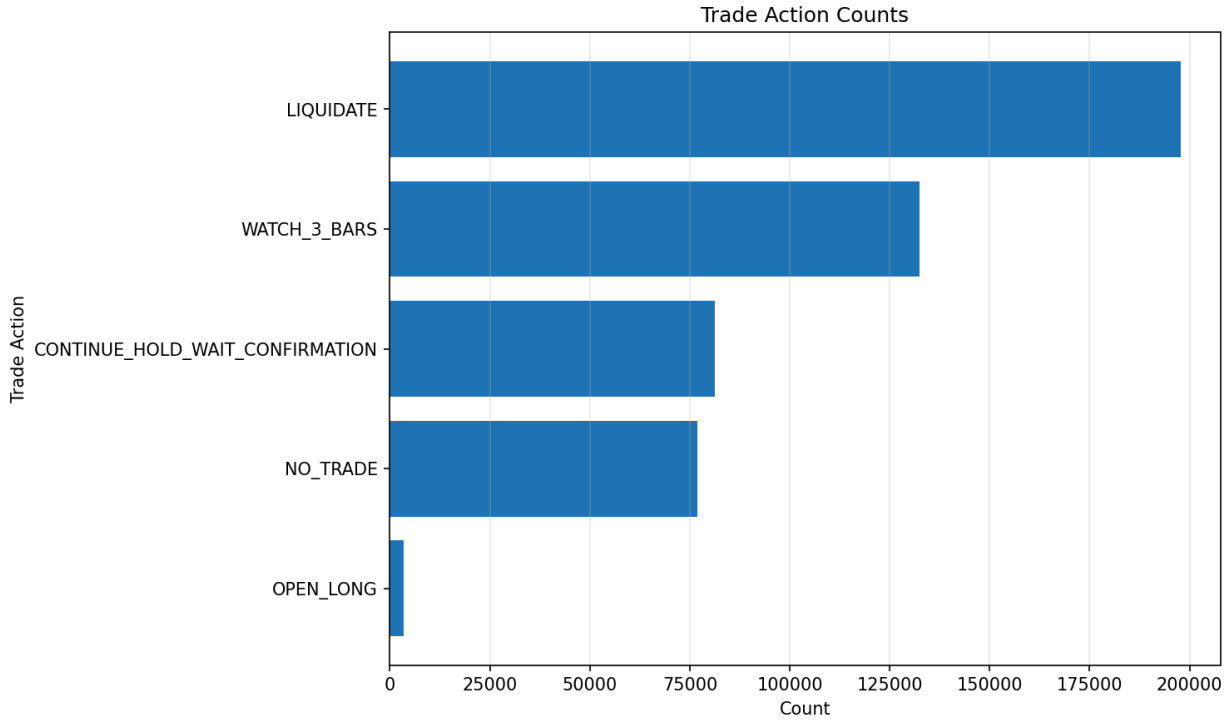
Next-bar Positive



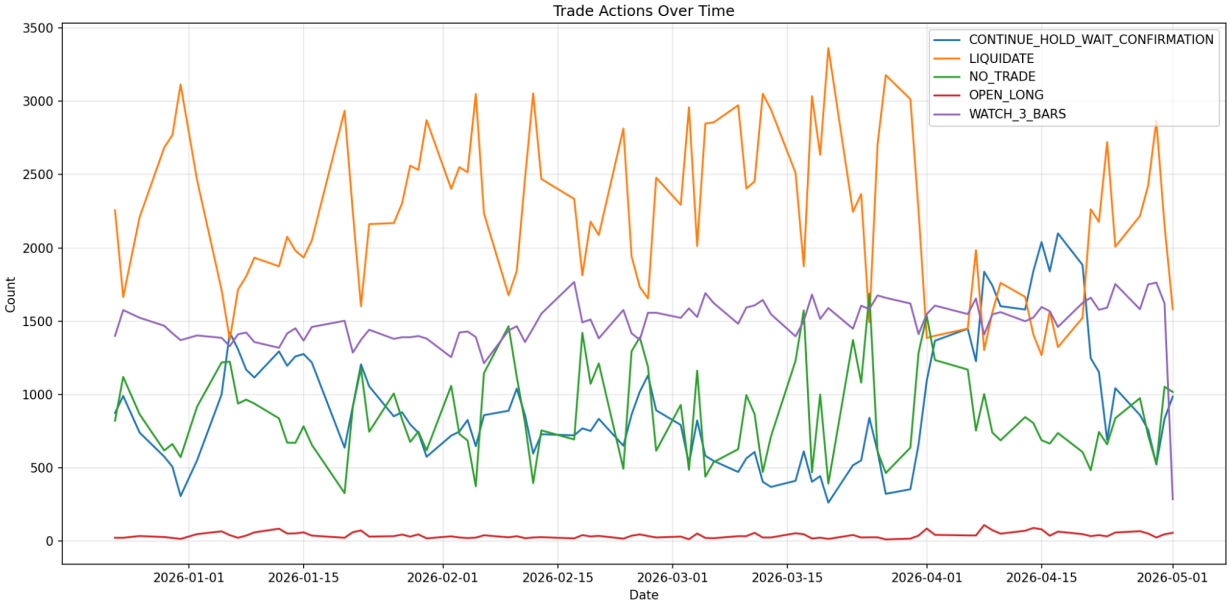
Continuation Probability Distribution



Trade Actions Count



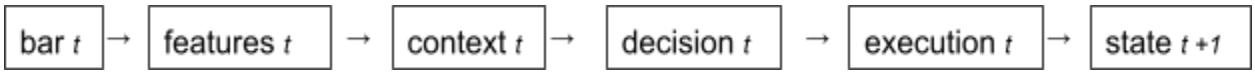
Trade Actions Overtime



Discussion

`Run.py` orchestrates the live trading loop. On each incoming bar, it validates the symbol and price, updates feed logs and heartbeat counters, sends proxy bars to the regime engine, updates the monitor, constructs the latest feature row, and retrieves market context. It then obtains current position and risk state, suppresses trading until the regime snapshot is ready, recomputes target weights when needed, builds per symbol position and risk context, skips symbols that are neither confirmed nor already held, and finally calls the signal engine.

This event flow can be summarized as:



This architecture offers three practical strengths. First, it places a top-down market-state model ahead of trade admission, which reduces the chance of taking local signals that conflict with broader market conditions. Second, it separates policy inference from executable action, making the system more interpretable and safer under live constraints. Third, it centralizes position truth in a dedicated manager while exposing rich monitoring context to user interfaces and logs. Nevertheless, the system remains sensitive to inaccuracies, which can lead to premature exits or missed opportunities. Additionally, the performance of the reinforcement learning component depends on the quality and diversity of training data.

Conclusion

This paper presents a regime-aware reinforcement learning trading system that combines statistical modeling with policy optimization to achieve robust performance across varying market conditions. By combining adaptive signal generation with deterministic risk controls and execution constraints, the system achieves a balance between flexibility and stability. The approach demonstrates that reinforcement learning can be effectively applied in financial markets when embedded within a controlled architecture. Rather than replacing traditional safeguards, learning-based models can complement them, resulting in more robust and interpretable systems.

Future work will focus on extending the framework to multi-asset portfolios, incorporating online learning for real-time adaptation, and enhancing risk management through portfolio-level optimization.

Referrences

[1] Brugière, P. Quantitative Portfolio Management: With Applications in Python. In Springer Texts in Business and Economics (pp. 99–220). Springer. Retrieved March 2026.

[2] Donadio, S., & Ghosh, S. Learn Algorithmic Trading: Build and Deploy Algorithmic Trading Systems and Strategies Using Python and Advanced Data Analysis. Packt Publishing.

[3] Jansen, S. (2020). Machine Learning for Algorithmic Trading: Predictive Models to Extract Signals from Market and Alternative Data for Systematic Trading Strategies with Python. Packt Publishing.

[4] Paleologo, G. A. (2026). Advanced Portfolio Management: A Quant's Guide for Fundamental Investors. Wiley. Retrieved March 2026.

[5] Turner, T. (2026). A Beginner's Guide to Day Trading Online (2nd ed.). Adams Media.

[6] Van Der Post, H. (2026). Algorithmic Trading: How to Effectively Profit with Python. In Python Libraries for Finance (Book 4). Reactive Publishing.

[7] Van Der Post, H. TensorFlow for Quantitative Finance: Transform Financial Analysis with TensorFlow's Cutting-Edge Machine Learning Techniques. In Python Libraries for Finance (Book 6). Packt Publishing.

[8] Van Der Post, H., & Schwarz, A. Developing and Evaluating Reinforcement Learning Strategies for Financial Trading. In Deep Learning: Advanced Techniques for Finance: Revolutionize Financial Analysis with Python (pp. 441–634). Reactive Publishing.