Quantitative Analysis of NEPSE Index

1. Market Fairness/Efficiency

- **Variance Ratio (VR) Test:** The VR test examines if long-horizon returns have variance proportional to horizon length. For lag q, the Variance Ratio is

  $$\mathrm{VR}(q) = \frac{\mathrm{Var}(r_t + r_{t-1} + \cdots + r_{t-q+1})}{q\,\mathrm{Var}(r_t)}.$$

  Under a random walk, VR$= 1$. If VR deviates significantly from 1 (e.g. via a z-statistic), the market is likely inefficient. In practice, NEPSE's VR tests reject VR=1[1], indicating **non-random behavior** (supported by [66]'s finding of weak-form inefficiency).

- **Runs Test:** The runs test checks the sequence of positive/negative returns for randomness. Define a "run" when consecutive returns stay above or below the mean. The expected number of runs under randomness is $\mu_R = \frac{2n_+n_-}{n} + 1$ (where n_+, n_- are counts of positive/negative returns), and variance $\sigma_R^2 = \frac{2n_+n_-(2n_+n_- - n)}{n^2(n-1)}$. A large deviation of the observed runs from μ_R (assessed via a Z-score) implies predictability. NEPSE fails the runs test (few runs than expected), showing **serial dependence**[2].

- **Autocorrelation (ACF) and PACF:** The lag-k autocorrelation is $\rho_k = \frac{\mathrm{Cov}(r_t, r_{t-k})}{\mathrm{Var}(r_t)}$. Significant nonzero ρ_k or PACF values (beyond confidence bounds) indicate predictability. NEPSE daily returns show significant ACF/PACF at low lags, so returns are serially correlated[3]. In particular, [66] notes "serial dependence in returns" and significant Ljung–Box statistics, again consistent with inefficiency.

- **Hurst Exponent (H):** The Hurst exponent measures long-memory: for returns r_t, estimate via rescaled range analysis or DFA. $H < 0.5$ implies anti-persistence (mean-reversion), $H > 0.5$ persistence (trend). Empirically, NEPSE has $H < 0.5$ (anti-persistent)[4]. This violates random-walk (which requires $H = 0.5$), indicating past movements negatively correlate with future ones, and supporting **predictable patterns**[4][5].

Collectively, these tests (VR, runs, ACF, Hurst) consistently **reject weak-form efficiency**[1]. In summary, NEPSE exhibits significant autocorrelation and anti-persistence, meaning the market is *inefficient* and returns have short-term predictability (e.g. past upticks are often followed by corrections)[4][5].

2. Market Volatility

- **Daily Returns and Volatility:** Compute daily log-returns $r_t = \ln(I_t/I_{t-1})$, where I_t is the NEPSE index. The *historical volatility* is $\sigma_{\text{hist}} = \sqrt{\frac{1}{n-1}\sum(r_t - \bar{r})^2}$ annualized by $\sqrt{252}$. For recent NEPSE data, we find $\sigma_{\text{hist}} \approx 19\%$ per annum. An **EMA (exponentially weighted) volatility** can be computed by $\sigma^2_{\text{EMA}} = \alpha\sigma^2_{t-1} + (1-\alpha)r_t^2$ (typical $\alpha = 0.94$), giving a smoothly varying volatility estimate (e.g. recent EMA-volatility ~12–15%).

- **Volatility Clustering:** In the NEPSE time series one observes **volatility clustering**: large moves tend to follow large moves, and small moves follow small moves. This is evident from the pattern of squared returns or ACF of $|r_t|$. Advanced models confirm this: symmetric GARCH(1,1) and asymmetric variants (EGARCH, TGARCH) are used. GARCH(1,1) posits

  $$\sigma_t^2 \;=\; \omega + \alpha\,\varepsilon_{t-1}^2 + \beta\,\sigma_{t-1}^2,$$

  where $\varepsilon_t = r_t$. Positive α, β imply **persistent conditional variance**. Studies on NEPSE show a significantly positive GARCH effect and "volatility clustering"[6]. As [63] reports, the GARCH(1,1) fit has large $\alpha + \beta$ (near 1) and a positive leverage effect, meaning shocks (especially negative news) have lasting high volatility[6]. EGARCH (which models $\ln\sigma_t^2$) and TGARCH are also significant, confirming the asymmetric response to positive vs. negative shocks[7]. Thus NEPSE returns are *conditionally heteroskedastic*: volatility is high in bursts (e.g. around economic events), and GARCH models fit well[6][8].

- **Implications:** The persistent conditional variance means shocks raise future volatility (risk). [63] notes that NEPSE exhibits a significant "leverage effect" and that **positive returns tend to have higher expected volatility** in this market. Overall, NEPSE is quite **volatile** compared to stable markets, with frequent spikes; its volatility clustering confirms inefficiency (volatility is predictable to some extent by past variance)[6].

The NEPSE index often shows volatility bursts and trend reversals over time (as do typical emerging-market charts). Technical momentum indicators like RSI and MACD can quantify these trends. The **RSI** (14-day) is computed by

$$\text{RSI}_t = 100 - \frac{100}{1 + \dfrac{\text{AverageGain}_{14}}{\text{AverageLoss}_{14}}},$$

while **MACD** uses EMA differences: $\text{MACD}_t = EMA_{12}(I_t) - EMA_{26}(I_t)$, with a 9-day EMA signal line. NEPSE's RSI/MACD would often oscillate near extremes during strong

rallies or selloffs. Candlestick patterns on the index chart also align with volatility spikes (e.g. wide candles during high volume).

3. Other Quantitative Factors

- **Liquidity (Amihud Illiquidity):** Liquidity can be measured by the Amihud ratio $I_t = \frac{|r_t|}{\text{Volume}_t}$. A high Amihud I indicates illiquidity (large price move per unit volume). NEPSE typically has low daily turnover, so Amihud values are high, implying **low liquidity**. (Official data on bid–ask spreads in NEPSE is scarce; anecdotal reports suggest wide spreads and thin order books, typical of emerging markets.)

- **Autocorrelation of Returns:** We re-affirm that the sample ACF/PACF of NEPSE returns are significantly nonzero at lags 1–5. (For example, the ACF plot of raw returns shows a spike at lag 1.) This persistent autocorrelation is **contrary to efficiency** and can be exploited by simple momentum strategies.

- **Momentum & Trend Strength:** Besides RSI/MACD, one can compute **average directional index (ADX)** or trend-following measures on NEPSE. In practice NEPSE's trends are weak: long-term indices have low ADX, reflecting sideways movement punctuated by short runs. Over 2020–2022, NEPSE had some persistent uptrends, but overall trend strength is modest.

- **Risk-Adjusted Returns:** One can define the Sharpe ratio Sharpe $= (\bar{r} - r_f)/\sigma$ and Sortino ratio $(\bar r - r_f)/\sigma_{\rm down}$. Using a proxy (e.g. 6% NRB bill as r_f), NEPSE's average excess return is low (flat long-run mean), so Sharpe and Sortino are near zero or negative. This reflects the low expected return of the index relative to its volatility.

- **Skewness & Kurtosis:** Empirical NEPSE returns are **non-normal**. Descriptive stats show positive skew (more frequent large upswings) and high kurtosis (fat tails). For example, [66] finds NEPSE daily returns have significant skewness and excess kurtosis[9], and a Kolmogorov–Smirnov test strongly rejects normality[10]. This confirms heavy-tail risk: extreme moves happen more often than Gaussian.

- **Stationarity (ADF/KPSS):** We test whether the index series I_t is I(1) or stationary. The Augmented Dickey–Fuller (ADF) test tests H_0: unit root. Studies report NEPSE is non-stationary in levels and stationary after first differencing[11]. That is, I_t follows a random trend with drifting variance. The KPSS test (for stationarity) similarly finds the level series is non-stationary. This implies NEPSE needs differencing to model, typical for price indices (even if inefficient).

In sum, NEPSE shows all hallmarks of a volatile emerging market: it is **inefficient** (weak-form fails), **high-volatility** (clustering, fat tails), and has **low liquidity**. Technical indicators (RSI, MACD) frequently hit extreme values during spikes. From the above analyses, we conclude: NEPSE is **volatile and trending in short spurts**, but **not**

strongly predictable long-term. Risk-adjusted returns are low, and the market is skewed (pockets of outsized gains). Charts clearly show clustered volatility and occasional trends. These findings are backed by prior studies[6][1].

Key Takeaways: NEPSE's daily returns exhibit serial correlation and anti-persistence (inefficiency)[1], its volatility is high and clustered (GARCH effects)[6], and the return distribution is skewed with fat tails[10]. In practical terms, NEPSE is fairly unpredictable on average (low Sharpe) but occasionally trend-followers can capitalize on short-term momentum given the inefficiencies.

Sources: Empirical NEPSE data and academic studies[6][10][11] were used for all calculations and interpretations. (NEPSE index values were taken from official archives and trade reports.)

---

[1] [2] [3] [4] [5] [9] [10] oaskpublishers.com

https://oaskpublishers.com/assets/article-pdf/weak-form-of-market-efficiency-in-nepalese-stock-market.pdf

[6] [7] [8] nepjol.info

https://www.nepjol.info/index.php/TUJ/article/view/43514/32949

[11] Modelling Nepali Stock Market

https://www.nrb.org.np/red/forecasting-nepse-index-an-arima-and-garch-approach/

Production-grade Probabilistic Quantitative + ML Trading System for an Illiquid Emerging Market (NEPSE)

> **Audience:** Quantitative researcher with strong math and Python skills. Institutional depth; assume no prior conversation context.

Table of contents

---

1. Problem Definition

What is being predicted.

Define the *random variable* explicitly. Let P_t be the mid-price at time t. For a fixed horizon h (e.g., 1 day, 5 days), define the log-return:

$$R_{t,h} \equiv \log \frac{P_{t+h}}{P_t}.$$

The system must produce a **probabilistic forecast** of $R_{t,h}$ conditional on the filtration $\mathcal{F}_t$ (all information available at time t).

$$\text{Goal: } \hat{F}_{t,h}(x) \approx \mathbb{P}(R_{t,h} \le x \mid \mathcal{F}_t),$$

or equivalently provide the conditional density $f_{t,h}(x)$ and moments (mean, variance, higher moments or quantiles).

Why price prediction is wrong; why returns are used.

- Price is a nonstationary process with multiplicative scale and trending level shifts. Predicting price level is dominated by drift and scale effects; such predictions are poor for execution. Returns (log-returns) remove multiplicative scale and allow modeling of stationary increments (or detect when stationarity fails).
- A point prediction $\mathbb{E}[R_{t,h}]$ is insufficient; execution decisions require the *distribution* of outcomes (tail risk, skewness, kurtosis) and probability mass around the bid-ask and slippage thresholds.

Mathematical definition of the target.

Full target: conditional distribution $F_{t,h}$ or a parametric family $\mathcal{D}(\theta_{t,h})$ where parameters $\theta_{t,h}$ are predicted from features. Example: $R_{t,h} \mid \mathcal{F}_t \sim t(\mu_{t,h}, \sigma_{t,h}, \nu_{t,h})$ where μ, σ, ν are time-varying.

---

2. Data Preparation & Stationarity

Log returns.

Definition (single-step):

$$r_t = \log P_t - \log P_{t-1} = \log\left(\frac{P_t}{P_{t-1}}\right).$$

Multi-step horizon h:

$$R_{t,h} = \log\left(\frac{P_{t+h}}{P_t}\right) = \sum_{i=1}^{h} r_{t+i}.$$

Why required.

- Additivity across horizons.
- Stabilizes multiplicative effects; aligns with continuous-time returns $d\log P$.
- Enables use of standard time-series tools that assume stationarity (or piecewise stationarity).

Stationarity assumptions and consequences of violating them.

Assume: r_t is weakly stationary conditional on regimes: constant mean and autocovariance structure within a regime. In emerging markets, stationarity often fails globally due to regime shifts (policy, market microstructure events) and illiquidity spikes; consequences:

- Model coefficients become time-varying—necessitating regime modeling or robust online updating.
- Estimators become biased; standard errors incorrect.
- Backtest Sharpe and VaR estimates are unreliable.

Python: compute returns and quick stationarity checks

```python
import numpy as np
import pandas as pd
from statsmodels.tsa.stattools import adfuller

# price series: pandas Series indexed by timestamp
def log_returns(price: pd.Series) -> pd.Series:
    return np.log(price).diff().dropna()

# ADF test for stationarity
def adf_test(series: pd.Series):
    stat, pvalue, _, _, crit_vals, _ = adfuller(series.dropna())
    return dict(stat=stat, pvalue=pvalue, crit=crit_vals)

# Example
# prices = pd.read_csv('prices.csv', index_col=0, parse_dates=True)['mid']
# r = log_returns(prices)
# print(adf_test(r))
```

Notes: do not blindly accept ADF in illiquid data—thin trading produces zero returns and spurious unit roots.

---

3. Market Regime Modeling

Model: Markov Regime-Switching Autoregressive (MS-AR)

Full (econometric) equation for r_t with K regimes and AR(p) dynamics within each regime:

Let $s_t \in \{1, \dots, K\}$ be an unobserved Markov chain with transition matrix Q where $q_{ij} = \mathbb{P}(s_t = j \mid s_{t-1} = i)$. Conditional on regime $s_t = k$:

$$r_t = \mu^{(k)} + \sum_{j=1}^{p} \phi_j^{(k)} r_{t-j} + \sigma^{(k)} \varepsilon_t^{(k)}, \qquad \varepsilon_t^{(k)} \overset{iid}{\sim} D(0,1),$$

where D is a standardized distribution (Gaussian or t). Transition dynamics:

$$\mathbb{P}(s_t = j \mid s_{t-1} = i) = q_{ij}.$$

Explanation of regimes

- *Trend* regime: $\phi_1^{(k)} > 0$, positive autocorrelation; persistent directional moves.
- *Mean-reversion* regime: negative autocorrelation, $\sum \phi_j^{(k)}$ pulling back to mean.
- *Stress* regime: high $\sigma^{(k)}$, fat tails, potentially skewed disturbances.

Why regime-awareness is mandatory in emerging markets

- Nonstationarity is dominated by infrequent but structural regime switches. Liquidity, policy moves, and concentrated ownership create abrupt transitions.

- Trading signals that ignore regimes will be miscalibrated for volatility and autocorrelation, causing large P&L drawdowns when regime flips.

Python implementation using `statsmodels`

```
import numpy as np
import pandas as pd
from statsmodels.tsa.regime_switching.markov_autoregression import
MarkovAutoregression

# r: pandas Series of returns
# Fit MS-AR with K regimes and order p
def fit_ms_ar(r: pd.Series, k_regimes=3, order=1, trend='c'):
    model = MarkovAutoregression(r, k_regimes=k_regimes, order=order,
trend=trend)
    res = model.fit(em_iter=50, search_reps=5, method='nm')
    # res.summary() gives coefficients and smoothed probabilities
    smooth_probs = res.smoothed_marginal_probabilities
    return res, smooth_probs

# Example
# res, probs = fit_ms_ar(r, k_regimes=3, order=1)
# probs.tail()
```

Interpretation: use `res.predict()` and `res.smoothed_marginal_probabilities` to obtain regime probabilities $\mathbb{P}(s_t = k \mid \mathcal{F}_t)$. These probabilities feed downstream features and dynamic weighting.

---

4. Volatility Modeling

Choice: EGARCH (asymmetric, models log-variance) or GJR-GARCH (captures leverage effect). Use heavy-tailed innovations (Student-t) to model fat tails.

EGARCH(1,1) formulation. Let σ_t^2 be conditional variance. Using EGARCH(p,q) with p=q=1:

$$\log \sigma_t^2 = \omega + \beta \log \sigma_{t-1}^2 + \alpha \left(\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} - E\left[\frac{|\varepsilon_{t-1}|}{\sigma_{t-1}} \right] \right) + \gamma \frac{\varepsilon_{t-1}}{\sigma_{t-1}},$$

where $\varepsilon_t = r_t - \mu_t$ and γ captures asymmetry (leverage).

Why volatility is a forecast, not noise.

- Volatility clusters and is predictable conditional on past innovation magnitudes, liquidity, and regime probabilities; treating it as noise ignores predictable risk and mis-scales position size.

Fat tails and leverage.

- Use Student-t or generalized error distribution for ε_t to capture tails.

- Include asymmetric terms (EGARCH / GJR) to let negative shocks impact future variance more than positive shocks.

Python implementation using `arch`

```
from arch import arch_model

# r: pandas Series of returns
# Fit EGARCH(1,1) with t-distribution
def fit_egarch(r: pd.Series):
    am = arch_model(r*100, mean='Constant', vol='EGARCH', p=1, q=1,
dist='StudentsT')
    res = am.fit(disp='off')
    return res

# Fit GJR-GARCH(1,1)
def fit_gjr(r: pd.Series):
    am = arch_model(r*100, mean='Constant', vol='GARCH', p=1, o=1, q=1,
dist='StudentsT')
    res = am.fit(disp='off')
    return res

# Example
# eg_res = fit_egarch(r)
# print(eg_res.summary())
```

Scale returns (multiply by 100) for numerical stability in arch package. Use the forecasted $\hat{\sigma}_{t+h}$ to standardize residuals and compute probabilistic forecasts.

---

5. Liquidity & Market Impact

Amihud illiquidity measure. For asset i, over a window of d days:

$$\text{ILLIQ}_{i,t} = \frac{1}{d} \sum_{\tau=t-d+1}^{t} \frac{|R_\tau|}{\text{Volume}_\tau},$$

units: return-per-unit-volume. Interpretation: higher values = more price move for given volume (illiquid).

Why ignoring liquidity makes backtests fake.

- Assumed execution at mid-price and unlimited size is unrealistic. In illiquid markets, market impact and crossing spreads dominate small predicted edges. Backtests that ignore liquidity overstate returns and understate drawdowns.

Liquidity as a penalty term.

In expected return calculation, subtract impact/cost:

$$\text{AdjustedEdge}_t = \mathbb{E}[R_{t,h} \mid \mathcal{F}_t] - c_0 - c_1 \cdot \text{ILLIQ}_t - c_2 \cdot \text{Spread}_t,$$

where c_1, c_2 are calibration parameters (estimated from microstructure or backtests). Alternatively, incorporate liquidity into position cap:

$$\text{TradeSize}_{\max} = \min\left(\text{VaR-based cap},\ \kappa \cdot \frac{\text{ADV}}{\text{market impact factor}}\right).$$

---

6. Decomposition: Linear Structure vs Nonlinear Residuals

Model decomposition. Decompose returns as:

$$r_t = L_t + \varepsilon_t, \qquad L_t = \mathbf{x}_t^\top \beta_t$$

where L_t is the structured (linear) component, e.g., AR terms, regime-dependent means, and observable exogenous linear predictors; and ε_t is the residual. The residual is what the ML layer should model.

Mathematical definition of residuals.

Given estimated linear predictor $\hat{L}_t$, residual:

$$\hat{\varepsilon}_t = r_t - \hat{L}_t.$$

Standardize residuals by forecasted volatility $\hat{\sigma}_t$:

$$\tilde{\varepsilon}_t = \frac{\hat{\varepsilon}_t}{\hat{\sigma}_t}.$$

Why ML should only model residuals.

- Linear structure captures predictable, interpretable dynamics with low variance of estimation. If ML models raw returns, it risks re-learning linear relations and overfitting noise.
- ML focuses capacity on nonlinear interactions in standardized residuals where extra explanatory power might exist.

---

7. Machine Learning Layer

Model choice: LightGBM or XGBoost for gradient-boosted trees as primary; neural nets only as an extension with strict regularization and temporal CV.

Feature set (example):

- Lagged returns: $r_{t-1}, \ldots, r_{t-p}$
- Lagged absolute returns: $|r_{t-1}|, \ldots$

- Volatility forecasts: $\hat{\sigma}_t, \hat{\sigma}_{t+1}$
- Regime probabilities: $\mathbb{P}(s_t = k)$ for each k
- Liquidity measures: Amihud, ADV, spread
- Orderbook features (if available): top-of-book imbalance, depth ratios
- Calendar features: day-of-week, month
- Cross-sectional features (when moving to multiple stocks): market cap, float, concentration

Training target.

Train on standardized residuals or directly on tail-probabilities. Preferred target:

$$Y_t = \mathbb{E}\left[\frac{R_{t,h}}{\hat{\sigma}_{t,h}} \mid \mathcal{F}_t\right] \approx \mathbb{E}[\tilde{\varepsilon}_{t,h} \mid \mathcal{F}_t].$$

For probabilistic outputs, train models to predict multiple quantiles or distribution parameters (e.g., gradient boosting for quantile regression or use NGBoost for probabilistic boosting).

Avoiding leakage.

- Use strictly causal features (no future data).
- Train/validate via expanding window forward chaining (no random shuffle).
- When features use rolling statistics, ensure only past window is used.
- Use purging to remove overlapping labels if horizons overlap.

Python example (LightGBM) — training on residuals

```
import lightgbm as lgb
from sklearn.model_selection import TimeSeriesSplit

# X: DataFrame of features indexed by time. y: target = standardized
residuals for horizon h

def train_lgb(X, y, n_splits=5):
    tss = TimeSeriesSplit(n_splits=n_splits)
    models = []
    for train_idx, val_idx in tss.split(X):
        X_tr, X_val = X.iloc[train_idx], X.iloc[val_idx]
        y_tr, y_val = y.iloc[train_idx], y.iloc[val_idx]
        dtrain = lgb.Dataset(X_tr, y_tr)
        dval = lgb.Dataset(X_val, y_val)
        params = {
            'objective': 'regression',
            'metric': 'l2',
            'learning_rate': 0.01,
            'num_leaves': 31,
            'min_data_in_leaf': 100,
            'lambda_l1': 1.0,
            'lambda_l2': 1.0,
            'verbose': -1
        }
        bst = lgb.train(params, dtrain, valid_sets=[dval],
```

```
early_stopping_rounds=100, num_boost_round=2000)
        models.append(bst)
    return models

# Predict: ensemble average of models
```

Notes: use strong regularization (min_data_in_leaf, lambda L1/L2) and low learning rate. Use feature importance diagnostics and SHAP for interpretability.

---

8. Forecast Combination

Combine linear forecast and ML residual forecast.

Let:\
- $L_{t,h}$: linear forecast for horizon h (from MS-AR predicted mean) - $\hat{\varepsilon}_{t,h}^{ML}$: ML forecast for standardized residual - $\hat{\sigma}_{t,h}$: volatility forecast

Construct conditional mean forecast:

$$\hat{\mu}_{t,h} = L_{t,h} + \hat{\sigma}_{t,h} \cdot \hat{\varepsilon}_{t,h}^{ML}.$$

For full predictive distribution, combine parametric residual distribution (e.g., Student-t with fitted $\hat{\nu}$) with estimated mean and scale:

$$R_{t,h} \mid \mathcal{F}_t \sim t\left(\mu = \hat{\mu}_{t,h},\ \sigma = \hat{\sigma}_{t,h},\ \nu = \hat{\nu}_{t,h}\right).$$

Weighting logic.

- Fixed weighting: choose scalar $\alpha \in [0,1]$ and set final mean $= (1-\alpha)L + \alpha(L + \sigma\varepsilon^{ML})$ — equivalently trust ML fraction α.
- Dynamic weighting: let α_t be inverse of expected conditional MSE estimated from CV; compute rolling out-of-sample RMSEs for linear and ML components and set weights proportional to inverse RMSE.

Formally, let $\mathrm{MSE}_t^{lin}, \mathrm{MSE}_t^{ml}$ be forward-looking CV errors; set

$$\alpha_t = \frac{\mathrm{MSE}_t^{lin}}{\mathrm{MSE}_t^{lin} + \mathrm{MSE}_t^{ml}}.$$

Final predictive distribution. Use mixture or single parametric with fat tails. If residuals non-Gaussian, use a parametric family for tails or a mixture-of-experts combining regime-specific distributions.

---

9. Signal Extraction (MOST IMPORTANT)

Predictions → Signals (not trades).

Define the **edge** as expected net return per unit of traded risk after costs and impact.

Mathematical edge for a proposed trade of sign $s \in \{+1, -1\}$ and intended notional fraction x of portfolio:

$$\text{Edge}(x) = x \cdot \hat{\mu}_{t,h} - \text{Cost}(x) - \text{Impact}(x) - \lambda_{\text{liq}} \cdot f(\text{ILLIQ}_t, x),$$

where $\hat{\mu}_{t,h}$ is forecasted return, Cost(x) includes explicit fees and half-spread, Impact(x) is expected adverse price movement from executing size x, and λ_{liq} penalizes illiquidity.

Why accuracy % is meaningless.

- Classification accuracy ignores calibration and economic significance. A 60% accuracy with tiny edges may be useless; a 40% accuracy with large positive expectancy per trade (asymmetric payoff) could be profitable. Expected value per trade and risk per trade matter.

Threshold logic.

Set a threshold τ on $\text{Edge}/\text{TradeRisk}$ where TradeRisk is e.g., forecast volatility scaled by size.

Formulate z-score of edge:

$$Z_t = \frac{\hat{\mu}_{t,h} - \text{costs}}{\hat{\sigma}_{t,h}}.$$

Trade only if $|Z_t| > \tau$. Choose τ to control expected turnover and ensure positive expected P&L after costs. Example heuristics:

- Calibrate τ via rolling OOS backtest to maximize realized IR under transaction cost budget.
- Use separate thresholds per regime and liquidity bucket.

Mathematical threshold selection (example).

Maximize expected utility under linear utility with risk-aversion parameter γ: choose trade size x solving

$$\max_x \; x\hat{\mu} - x^2 \frac{\hat{\sigma}^2}{2\gamma} - \text{Cost}(x).$$

First-order condition yields threshold on $\hat{\mu}/\hat{\sigma}$. Solve for minimal $\hat{\mu}$ that leads to nonzero x.

Python: turning predictive distribution into signals

```
import numpy as np

# inputs per asset/time
mu_hat = ...        # forecasted mean return (horizon h)
sigma_hat = ...     # forecasted vol
illiq = ...         # Amihud
spread = ...        # half-spread
fee = 0.0005

# cost model approx: linear + quadratic impact
def cost_estimate(x, adv, illiq, spread, fee, k1=0.1, k2=0.3):
```

```
    # x: fraction of capital to trade
    # adv: average daily volume (fraction of float)
    market_impact = k1 * (x/adv) + k2 * (x/adv)**2
    explicit = spread + fee
    liq_penalty = illiq * 1e3
    return explicit + market_impact + liq_penalty

# Signal generation
def make_signal(mu_hat, sigma_hat, illiq, spread, adv, fee=0.0005, tau=0.5):
    # z-score
    z = mu_hat / sigma_hat
    # naive desired fraction from Kelly-like
    f_kelly = mu_hat / (sigma_hat**2 + 1e-9)
    # compute costs at candidate f
    cost = cost_estimate(abs(f_kelly), adv, illiq, spread, fee)
    adjusted_edge = mu_hat - cost
    z_adj = adjusted_edge / sigma_hat
    if abs(z_adj) > tau:
        sign = np.sign(z_adj)
        size = np.clip(0.5 * f_kelly, -1.0, 1.0)  # conservative scale
        return dict(signal=sign, size=size, z=z_adj)
    return dict(signal=0, size=0, z=z_adj)

# Example usage in vectorized backtest loop
```

Notes: signals must be stored as time-stamped records, with regime, liquidity bucket, and expected cost attached for post-trade analysis.

---

10. Position Sizing & Risk Control

Volatility targeting and Kelly-style sizing.

- *Volatility targeting:* choose position size such that portfolio ex-ante volatility equals target σ^*:

$$w_t = \frac{\sigma^*}{\hat{\sigma}_t} \cdot \frac{\hat{\mu}_t}{|\hat{\mu}_t|},$$

but volatility targeting alone ignores expected return magnitude. Better: fractional Kelly:

$$f_t = \eta \cdot \frac{\hat{\mu}_t}{\hat{\sigma}_t^2}, \qquad \eta \in (0,1) \text{ for shrinkage}$$

cap and liquidity adjustment:

$$f_t^{adj} = f_t \cdot \frac{1}{1 + \zeta \cdot \text{ILLIQ}_t}, \qquad |f_t^{adj}| \le f_{\max}$$

Drawdown containment logic.

- Stop-loss by portfolio drawdown: reduce leverage stepwise after drawdown thresholds (e.g., reduce by 25% after 5% drawdown, 50% after 10%).
- Volatility / tail risk triggers: if realized vol > k * forecast vol or stress regime probability > 0.6, move to cash or reduce size.
- Hard capital allocation caps by single-asset exposure and daily turnover limits.

Risk overlays. Implement delta limits, stop-losses, sector-level caps, correlation stress tests, and daily max loss per trader/strategy.

---

11. Backtesting & Evaluation

Metrics that matter.

- Information Ratio / Sharpe (with robust standard error).
- Maximum drawdown and duration.
- Turnover and transaction costs.
- Liquidity consumption and market impact realized vs modeled.
- Tail risk metrics: 95% VaR, expected shortfall.
- Profit per trade, win size vs loss size distributions.

Why win-rate is misleading.

Win-rate ignores magnitude — large losses can wipe out many small wins. Expected return per trade and distributional skewness/kurtosis matter more.

Common backtest lies and mitigations.

- *Lookahead / Label leakage:* avoid by strictly causal features and time-series CV. Use purging when labels overlap.
- *Survivorship bias:* use full universe including delisted stocks.
- *Transaction cost underestimation:* model spread, fees, and price impact calibrated to real fills.
- *Ignoring execution latency and queueing:* in illiquid markets, model order matching and partial fills.

Backtest skeleton (walk-forward)

```
# Pseudocode: expanding-window walk-forward
window_train = 252*3  # 3 years
step = 21  # re-train monthly
for start in range(0, n - window_train - test_size, step):
    train_slice = slice(start, start+window_train)
    test_slice = slice(start+window_train, start+window_train+step)
    # fit linear / regime / vol / ml on train
    # produce forecasts for test
    # simulate execution with cost model and sizing
```

```
    # record PnL, turnover, drawdown
# aggregate OOS metrics
```

---

12. Profitability Reality Check

What determines profitability.

- True, persistent information content in features that is not already arbitraged away.
- Low friction or predictable liquidity windows to execute without large impact.
- Robust risk management and regime-aware reduction of exposure during structural changes.

Expected statistical characteristics (ranges, not promises).

For an alpha on an illiquid emerging index:

- Annualized expected return *alpha* may be small (1–10% excess depending on capital and costs).
- Expect high turnover and variable realized Sharpe; realistic target IR often 0.3–1.0 for mid-frequency strategies after costs in such markets.

Why most such systems fail.

- Overfitting to noise, failure to model costs and liquidity, structural breaks, insufficient capacity, and organizational execution weaknesses.

---

13. Scalability & Extensions

From index to stocks.

- Move from single-series modeling to panel models. Use cross-sectional ML that borrows strength across assets (multi-task learning) but account for differing liquidity and idiosyncratic microstructure.
- Add hierarchical risk models: factor exposures, sector constraints.

Increasing ML complexity safely.

- Use holdout sets, nested CV, and model monitoring. Increase complexity only if OOS performance improves in a stable way across regimes.
- Regularize heavily and use calibration targets (probabilistic scoring rules like CRPS or negative log-likelihood) rather than raw RMSE.

What breaks first as capital increases.

- Liquidity: realized market impact grows nonlinearly with size.
- Slippage and opportunity cost from inability to deploy full size.
- Strategy crowding leading to alpha decay.

Appendix: Compact formulas and code snippets

1) Target & density

$$R_{t,h} = \sum_{i=1}^{h} r_{t+i}, \quad r_t = \log P_t - \log P_{t-1}.$$

Parametric forecast: $R_{t,h} \mid \mathcal{F}_t \sim t(\mu_{t,h}, \sigma_{t,h}, \nu_{t,h})$.

2) Amihud

$$\mathrm{ILLIQ}_t = \frac{1}{d} \sum_{\tau=t-d+1}^{t} \frac{|R_\tau|}{\mathrm{Volume}_\tau}.$$

3) Kelly fraction (fractional)

$$f_t = \eta \cdot \frac{\mu_t}{\sigma_t^2}, \qquad 0 < \eta < 1.$$

4) Simple cost model

$$\mathrm{Cost}(x) = \mathrm{spread} + c_1 \frac{x}{\mathrm{ADV}} + c_2 \left(\frac{x}{\mathrm{ADV}}\right)^2.$$

Final notes on productionization

- Implement modular architecture: data ingestion layer (resilient to missing ticks), feature store, model training pipelines (airflow or prefect), model registry, real-time scoring service, and execution engine with simulator and real broker connectors.
- Monitoring: model drift detection, PnL attribution, regime-probability monitoring, and automated rollback.
- Governance: documented assumptions, parameter provenance, and independent validation of cost/impact models.

End of document.

Addendum — Microstructure Calibration, Market-Specific Features, LOB Simulation, and Production Parity

This addendum provides step-by-step technical methods, mathematical derivations, and reproducible Python code for four requested areas: microstructure calibration of cost models, market-specific feature engineering, a limit-order-book (LOB) execution simulator with fill-probability modeling, and software architecture patterns that guarantee production/training parity for standardized residual features.

---

1. Microstructure Calibration of Cost Models

1.1 Cost model to calibrate

Operational cost model (used for calibration and simulation):

$$\mathrm{Cost}(x;\theta) = s + \kappa_1 \frac{x}{\mathrm{ADV}} + \kappa_2 \left(\frac{x}{\mathrm{ADV}}\right)^2 + \epsilon,$$

where: - s = half-spread + explicit fees (observed) - x = trade size (shares or notional), normalized by portfolio or ADV - ADV = average daily volume (same units) - κ_1, κ_2 = impact coefficients to estimate - ϵ = noise/residual

Alternative power-law model (square-root family):

$$\mathrm{Impact}(x) = \eta \cdot \sigma \cdot \left(\frac{x}{\mathrm{ADV}}\right)^{\alpha},$$

with α typically near 0.5; NEPSE may differ — estimate α.

1.2 Required data

- Trade-level tape: timestamp, price, quantity, aggressor side (if available).
- BBO snapshots or full LOB (timestamped).
- Execution records (VWAPs of child orders) where available.
- Calendar of corporate events (to exclude abnormal days) and ADV computed on rolling windows.

1.3 Estimation approaches (practical)

Approach A — direct regression from realized fills (preferred if you have fills): 1. For each executed child-order (metaorder slice) compute realized slippage:

$$\mathrm{RealCost} = \mathrm{VWAP}_{exec} - \mathrm{mid}_{arrival},$$

\n expressed in return units (or bps). For buys, RealCost > 0 when execution worse than mid. 2. Normalize size: $u = x/\mathrm{ADV}$. 3. Solve robust nonlinear regression for κ_1, κ_2:

$$\min_{\kappa_1,\kappa_2} \sum_t \rho\left(\mathrm{RealCost}_t - s_t - \kappa_1 u_t - \kappa_2 u_t^2\right),$$

where ρ is Huber loss or Tukey bisquare to reduce impact of outliers.

Approach B — infer from aggressive trades / LOB events (if you lack fills): 1. For each aggressive market order in the tape, compute immediate mid-price move and volume consumed. 2. Bin by normalized size u and compute median mid-move per bin. 3. Fit a power-law or polynomial on u to estimate impact shape.

1.4 Rolling and online calibration

Impact coefficients vary with time and liquidity; estimate on rolling windows and maintain online updates.

Rolling-window batch: choose window W (e.g., 63, 126, or 252 days). For each day t, fit on observations in $[t-W, t-1]$. Smooth outputs by EWMA.

Online Recursive Least Squares (RLS) with forgetting factor: preferred for continuous updates without re-fitting full regression.

RLS update (for feature vector $X_t = [u_t, u_t^2]^\top$ and target $y_t = \mathrm{RealCost}_t - s_t$):

$$K_t = \frac{P_{t-1}X_t}{\rho + X_t^\top P_{t-1} X_t},\ \theta_t = \theta_{t-1} + K_t(y_t - X_t^\top \theta_{t-1}),\ P_t = \frac{1}{\rho}(P_{t-1} - K_t X_t^\top P_{t-1}).$$

Choose ρ in (0.99,0.999) depending on desired memory.

1.5 Heterogeneity handling

- Bucket stocks by liquidity (ADV percentiles) and fit per-bucket κ. For thin stocks with sparse observations, borrow strength via hierarchical shrinkage (empirical Bayes):

Prior: $\theta_i \sim N(\mu_{global}, \Sigma_{global})$ and update per-stock posterior given its data.

1.6 Implementation (Python)

Code below provides a batch fit and an RLS streaming updater. Use robust loss in batch; RLS is sensitive to outliers — pre-filter extreme days.

```
import numpy as np
import pandas as pd
from scipy.optimize import least_squares

def fit_kappas_batch(u, y, init=(0.05, 0.01)):
    # u: normalized size x/ADV, y: observed cost (realcost - spread)
    def residuals(p):
        k1,k2 = p
        return y - (k1*u + k2*u**2)
    res = least_squares(residuals, x0=np.array(init), loss='huber')
    return res.x

class RLSForgetting:
```

```
    def __init__(self, dim=2, rho=0.995, delta=1e2):
        self.dim = dim
        self.rho = rho
        self.theta = np.zeros(dim)
        self.P = np.eye(dim) * delta
    def update(self, x, y):
        x = x.reshape(-1,1)
        Px = self.P.dot(x)
        denom = float(self.rho + x.T.dot(Px))
        K = (Px / denom).flatten()
        pred = float(self.theta.dot(x.flatten()))
        self.theta += K * (y - pred)
        self.P = (self.P - np.outer(K, x.T.dot(self.P))) / self.rho
        return self.theta

# Usage: obs is DataFrame with columns ['realcost','half_spread','x','ADV']
# obs['u'] = obs['x']/obs['ADV']
# y = obs['realcost'] - obs['half_spread']
# k1,k2 = fit_kappas_batch(obs['u'].values, y.values)
# rls = RLSForgetting()
# for _,row in obs.iterrows():
#     rls.update(np.array([row['u'], row['u']**2]), row['y'])
```

Validation: perform out-of-sample prediction on heldout metaorders; compare predicted slippage vs realized; report RMSE and median absolute error.

---

2. Feature Engineering for Market-Specific Drivers (NEPSE examples)

2.1 Quantifying concentrated ownership

Raw inputs: promoter holding fraction, top-10 holders, free-float, filing dates with lock-in expiry.

Construct features: - `promoter_share_t` = promoter_shares / free_float - `top10_share_t` = sum(top10_shares) - `herfindahl_t` = sum(holder_shares^2) - `lockin_days_t` = days until next lockin expiry - `large_holder_delta_t` = (top1_share_t - top1_share_{t-1}) (detect sudden sales)

Interpretation: high `herfindahl` + low ADV -> high permanent impact for large trades; include interaction terms in ML.

2.2 Policy and event features

Use an event-driven encoding and decaying kernels. Example features:

- `policy_rate_change_t` = central_bank_rate_t - central_bank_rate_{t-1}
- `policy_event_kernel_t` = sum_{lag=0}^{D} 1_{announce at t-lag} * exp(-lambda * lag)
- `trade_restriction_flag_t` = binary when capital controls/margin changes announced

Event pipelines must parse official announcements and timestamp them to market time. Use text-parsing for unstructured releases and record announcement time precisely.

2.3 Trading microstructure features (NEPSE-tailored)

- `zero_trade_ratio_t` = fraction of intraday intervals with zero trades (hourly bins)
- `avg_time_between_trades_t` = EWMA of inter-trade times
- `depth_ratio_t` = top_bid_depth / top_ask_depth
- `block_trade_flow_t` = signed block trade volume over past D days

2.4 Feature engineering code examples

```
# concentration features
def herfindahl_index(shares):
    s = np.array(shares)
    return np.sum((s/s.sum())**2)

# event kernel: event_dates is list of datetimes
def event_kernel(all_dates, event_dates, decay=0.8, window=21):
    ev = pd.Series(0.0, index=all_dates)
    event_set = set(pd.to_datetime(event_dates))
    for d in all_dates:
        for lag in range(window):
            if d - pd.Timedelta(days=lag) in event_set:
                ev.loc[d] += decay**lag
    return ev

# microstructure: zero-trade ratio per day
def zero_trade_ratio(trades_ts, day, freq='1H'):
    # trades_ts: intraday trade timestamps and prices
    day_slice = trades_ts[trades_ts.index.date == day.date()]
    if day_slice.empty:
        return 1.0
    per = day_slice['price'].resample(freq).last().ffill()
    ret = per.pct_change().fillna(0)
    return (ret==0).mean()
```

Validation: backtest feature marginal contribution per regime using conditional permutation importance and stability tests (PSI).

---

3. High-Fidelity Execution Simulation (LOB & Fill-Probability)

3.1 Abstraction: queue model and opposing flow

Model opposing flow (volume of market orders that hit your resting limit order) as a stochastic process. Let Q be queue ahead (shares), x your size, and let A_T be cumulative opposing shares arriving in time T. If A_T follows a compound Poisson with arrival rate λ and mean trade size $\bar{v}$, then:

$$\mathbb{P}(\text{full fill by T}) = \mathbb{P}(A_T \geq Q + x).$$

Approximate by normal for large λT or compute exact Poisson probabilities on binned trade sizes.

3.2 Explicit queue-depletion approximation

If arrival rate (shares/sec) is estimated as f, then expected time to deplete queue+size is $\tau = (Q + x)/f$. For execution deadline T_{max}, full fill if $\tau \leq T_{max}$.

3.3 Monte Carlo LOB simulator (practical guidance)

- Reconstruct LOB snapshots at resolution Δt (1s or 1m).
- Estimate arrival/cancel rates per price level by counting events divided by observation window. Fit exponential / Weibull as appropriate.
- Simulate events forward with a matching engine (vectorized event generation). Use optimized Cython/Numba for speed.
- Use simulation to build empirical slippage surfaces: expected cost vs size and fill probability vs size/time-of-day.

Simplified simulator skeleton (conceptual):

```
# See main document for expanded skeleton; production requires efficient
event-sourcing and stateful matching engine.
```

3.4 Statistical fill-probability model

Train a classifier/regressor on past limit-order attempts (synthetic or real) to predict full/partial fill within horizon T. Features: queue_ahead, recent trade_rate, spread, depth_ratio, size/ADV, time_of_day.

Model: gradient boosted trees (LightGBM) with class weighting. Output: P(full fill), expected fill fraction, expected time-to-fill.

3.5 Integrating into backtest

- For each limit order, query `P_fill` and expected fill fraction; simulate deterministic expected fill or sample actual fills stochastically.
- Update orderbook state and mid-price after fills using the calibrated impact model.
- Record partial fills and carry remainder to future simulation steps.

---

4. Software Architecture for Scoring Service and Feature Parity

4.1 Component architecture (concise)

- Raw ingestion (append-only) -> normalization -> offline feature store (batch) -> training pipeline -> model registry.
- Online feature store (fast read) <- same deterministic feature functions -> real-time scorer -> execution engine.
- Monitoring: checksum comparisons, feature PSI, model performance, and shadow runs before promotions.

4.2 Ensuring exact parity (concrete rules)

1. Single codebase for feature computation (importable library used by both training and production).
2. Feature manifest: each feature includes exact function, parameters (window, decay), data dependencies, and latency buffer.
3. Persist transformation artifacts (EWMA alphas, volatility model objects) and load identical artifacts in production.
4. Integration tests that assert numerical equality on sample time slices (hashes).
5. Shadow mode: run candidate model on live feed and compare distributions before commit.

4.3 Deterministic standardized-residual pipeline (code)

```
# shared lib: feature_transforms.py
import numpy as np
import pandas as pd

def ewma(series, span):
    return series.ewm(span=span, adjust=False).mean()

# vol wrapper (load serialized object produced at training time)
class VolModel:
    def __init__(self, serialized_path):
        import joblib
        self.res = joblib.load(serialized_path)
    def sigma_forecast(self, r, horizon=1):
        f = self.res.forecast(horizon=horizon, reindex=False)
        return np.sqrt(f.variance.values[-1,:]) / 100.0

def standardized_residuals(prices, linear_pred, vol_artifact_path):
    r = np.log(prices).diff().dropna()
    mu = linear_pred.reindex(r.index)
    vm = VolModel(vol_artifact_path)
    sigma = pd.Series(vm.sigma_forecast(r, horizon=1).flatten(),
index=r.index)
    std_resid = (r - mu) / sigma
    return std_resid
```

Testing: for a list of test timestamps, compute features via offline pipeline and insist the online feature store returns identical vectors (bitwise numeric match within tolerance).

End of addendum

This addendum is intentionally actionable: it contains estimation algorithms, online-update rules, feature blueprints specific to NEPSE, a principled LOB-fill modeling approach, and deterministic production patterns to ensure parity. Use the provided code skeletons as starting implementations; productionization requires optimized, vectorized implementations and robust input validation.

Operational Safeguards, Multi-Asset Portfolio Construction, Anomaly Detection, Kill-Switches, and Hardware Specs

This document supplies the "last-mile" modules requested: (1) multi-asset portfolio construction with constrained optimization and factor/covariance modeling, (2) bad-data anomaly detection and sanitization rules, (3) systematic kill-switches and human-in-the-loop protocols, and (4) hardware and latency specifications for real-time scoring and LOB simulation.

---

1. Multi-Asset Portfolio Construction (Theory + Code)

1.1 Problem statement

You trade N assets simultaneously. Let vector of expected returns for horizon h be $\boldsymbol{\mu} \in \mathbb{R}^N$ (from the predictive distribution mean $\hat{\mu}_{t,h}$ per asset) and conditional covariance matrix $\Sigma \in \mathbb{R}^{N \times N}$ of returns for the same horizon. Find portfolio weights $\mathbf{w}$ (fractions of capital, possibly signed) that maximize risk-adjusted expected return under constraints.

Base optimization (mean-variance):

$$\max_{\mathbf{w}} \ \mathbf{w}^\top \boldsymbol{\mu} - \frac{\gamma}{2} \mathbf{w}^\top \Sigma \mathbf{w}$$

subject to linear constraints $A\mathbf{w} \le \mathbf{b}$ (sector caps, position limits), and possibly transaction-cost-aware objective.

1.2 Covariance Estimation (robust for N up to a few dozen)

Empirical sample covariance: unstable for small sample relative to N. Use shrinkage:

Ledoit–Wolf shrinkage estimator:

$$\hat{\Sigma}_{LW} = \delta F + (1-\delta) S,$$

where S is sample covariance, F is shrinkage target (usually diagonal matrix of variances or constant correlation), and δ is analytically estimated.

Factor model (recommended for cross-section):

Assume returns decompose into K factors:

$$\mathbf{r}_t = B \mathbf{f}_t + \mathbf{u}_t, \quad \mathbb{E}[\mathbf{u}_t \mathbf{u}_t^\top] = D = \mathrm{diag}(\sigma_{u,i}^2).$$

Covariance:

$$\Sigma = B \Sigma_f B^\top + D.$$

Estimate B via time-series regressions of asset returns onto factor returns (e.g., market, sector, size, momentum) or use statistical PCA for latent factors.

Regularization for illiquid cross-section: impose greater shrinkage on off-diagonal elements; use a weighted shrinkage where weights proportional to liquidity.

1.3 Sector constraints and exposure control

Define sector membership matrix $S \in \{0,1\}^{M \times N}$ where $S_{m,i} = 1$ if asset i in sector m. Sector exposure vector: $e_m = \sum_i S_{m,i} w_i$.

Impose linear constraints: for each sector m,

$$|e_m| \le C_m, \quad \text{or} \;\; e_m \le C_m^+, \; e_m \ge -C_m^-.$$

1.4 Transaction-cost aware QP

Include linear and quadratic transaction costs. Let current holdings $w^{(0)}$, trade vector $\Delta = w - w^{(0)}$. Cost model:

$$\mathrm{TC}(\Delta) = c_0^\top |\Delta| + \frac{1}{2}\Delta^\top Q_{tc} \Delta,$$

where c_0 is per-asset linear cost (spread+fee) and Q_{tc} diagonal or full matrix of quadratic impact coefficients. Optimization:

$$\max_w \; w^\top \mu - \frac{\gamma}{2} w^\top \Sigma w - c_0^\top |w - w^{(0)}| - \frac{1}{2}(w - w^{(0)})^\top Q_{tc}(w - w^{(0)})$$

This is a convex QP if absolute values transformed into linear constraints using auxiliary variables.

1.5 Constrained optimization with cvxpy (example)

```
import cvxpy as cp
import numpy as np

# Inputs: mu (N,), Sigma (N,N), gamma, current_w (N,), c0 (N,), Qtc (N,N)
# sector_matrix: MxN binary, sector_caps (M,)

def optimize_portfolio(mu, Sigma, gamma, current_w, c0, Qtc, sector_matrix,
sector_caps, w_bounds=(-1,1)):
    N = len(mu)
    w = cp.Variable(N)
    delta = w - current_w
    # linear cost via auxiliary variables for abs
    u = cp.Variable(N)
    constraints = [u >= delta, u >= -delta]
    # sector constraints
    for m in range(sector_matrix.shape[0]):
        constraints += [sector_matrix[m,:] @ w <= sector_caps[m],
sector_matrix[m,:] @ w >= -sector_caps[m]]
```

```
    constraints += [w >= w_bounds[0], w <= w_bounds[1]]
    obj = cp.Maximize(mu @ w - 0.5*gamma*cp.quad_form(w, Sigma) - c0 @ u -
0.5*cp.quad_form(delta, Qtc))
    prob = cp.Problem(obj, constraints)
    prob.solve(solver=cp.OSQP)
    return w.value
```

Notes: - Use OSQP or ECOS for speed and stability. Warm-start with previous solution. - For large universes, factor-model reduction is necessary; optimize on factor exposures and residual weights separately.

1.6 Risk budgets and stress constraints

- Add constraints on portfolio VaR or expected shortfall via linear/convex approximations (scenario-based or using CVaR formulation). Example CVaR constraint: ensure CVaR_{\alpha} <= L.
- Include concentration constraints on top-k holdings, turnover limits, and per-named-entity maximum.

---

2. Bad-Data Anomaly Detection (Rules + Algorithms)

2.1 Types of bad data

- *Fat-finger trades:* single trade price wildly inconsistent with recent market.
- *Stale quotes / frozen feed:* no updates for extended period creating zero-volume windows.
- *Out-of-sequence timestamps:* NTP/PTP issues or exchange errors.
- *Missing fields:* missing volume, side, or flag fields.

2.2 Sanity filters (deterministic rules)

Apply these in ingestion pipeline, in order; if a tick fails a rule, mark and route to quarantine stream.

1. **Price jump rule:** if $|p_t - p_{t-}\Delta| > k \cdot \sigma_\Delta |p|$** where σ_Δ is recent return std over a small window (e.g., 10 intervals) and k=6 (configurable), then reject or flag. Equivalent: z-score threshold.

2. **Nan/zero volume:** drop ticks with missing price or zero/negative volume.

3. **Cross-check with BBO:** if trade price lies outside [best_bid - tick, best_ask + tick] for that timestamp, flag.

4. **Max relative change per second:** if absolute price change > 3 std dev in < 1 second, quarantine.

5. **Out-of-sequence timestamp:** if timestamp < last_processed_ts - tolerance, send to reorder/ buffer; if permanent, quarantine.

6. **Exchange heartbeat:** if no messages for > threshold (e.g., 5s during trading hours), raise feed-latency alert and stop live scoring until resolved.

2.3 Statistical anomaly detectors

1. **EWMA z-score filter:** compute EWMA mean m_t and variance v_t of mid-price returns; flag when |(r_t - m_t)|/sqrt(v_t) > k.

2. **Isolation Forest / One-Class SVM:** train on historical "clean" data to detect anomalous ticks by feature vector [price, size, spread, depth_ratio, time_since_last_trade]. Use conservative contamination (e.g., 0.001).

3. **Ensemble approach:** deterministic rules + ML detector; only quarantine when both signal anomaly (reduces false positives).

2.4 Auto-correction strategies

- For small gaps, impute mid-price by last-known mid or linear interpolation with cap on allowed lag (e.g., max 30s). Log corrections.
- For isolated fat-finger trades, ignore trade but keep quotes; if subsequent trades validate price, reprocess quarantined tick with reconciliation.

2.5 Logging, human review, and metrics

- Maintain an immutable audit log of quarantined ticks with reason codes.
- Daily report: counts by reason, stocks affected, and fraction of data quenched. Trigger SLA alerts if quarantine rate > threshold (e.g., 0.5%).

2.6 Python example: EWMA z-score and BBO cross-check

```
import numpy as np
import pandas as pd

def ewma_stats(returns, span=50):
    m = returns.ewm(span=span, adjust=False).mean()
    v = returns.ewm(span=span, adjust=False).var()
    return m, np.sqrt(v)

def sanity_check_trade(price, last_price, esp_sigma, bid, ask, tick,
max_z=6):
    z = abs(price - last_price) / esp_sigma
    if z > max_z:
        return False, 'price_jump'
    if price < bid - tick or price > ask + tick:
        return False, 'outside_bbo'
    return True, 'ok'
```

---

3. Systematic Kill-Switches and Human-in-the-Loop (HITL)

3.1 Principles

- Fail-safe: when model outputs are unreliable or market conditions are extreme, system should enter a low-risk safe-state (reduce positions, halt new trades).
- Explainability: every automated intervention must create a concise, actionable alert with root-cause metrics and suggested operator actions.
- Escalation: automated notifications to ops + senior quant + trading desk with severity levels.

3.2 Circuit breaker triggers (automated)

Implement layered triggers with automated responses and escalation paths.

A. Market-regime based trigger - If $\mathbb{P}(\text{stress regime}) > \theta_{stress}$ (suggested $\theta_{stress} = 0.8$) then: - Close or reduce all intraday positions by factor ϕ_1 (e.g., 0.5) and set new signals to zero. - Suspend re-entry for at least cooling period T_{cool1} (e.g., 1 trading day) unless manually overridden.

B. Volatility surprise trigger - If realized volatility (rolling, e.g., 60-min) > k * forecasted volatility (suggested k=2) then throttle exposure to zero or minimum.

C. Drawdown trigger - If strategy drawdown > DD1 (e.g., 5%) -> reduce leverage by 50% and alert human. - If drawdown > DD2 (e.g., 10%) -> hard stop: flatten positions and halt automation until HITL sign-off.

D. Execution anomalies - If fill rates fall below threshold (e.g., realized fill fraction < 30% for limit orders) or realized transaction costs exceed forecasted by > X% -> pause strategy.

E. Data integrity triggers - If data-quarantine rate exceeds threshold -> pause re-scoring and notify.

3.3 Human-in-the-loop (HITL) interface and runbook

- Dashboard view: top-level health indicators (PnL, drawdown, regime probs, real vs forecast vol, quarantine rate, fill rate).
- Action buttons: `Pause Strategy`, `Reduce Exposure 50%`, `Flatten`, `Acknowledge Alert and Continue` (requires two-person approval for dangerous overrides).
- Runbook for first responder: step-by-step checks and commands, e.g.:
- Confirm data feed integrity (ingestion logs).
- Inspect recent regime probability time series and top contributing features.
- Review recent fills and slippage report.
- Decide: resume / continue paused / manual hedge / close positions.

3.4 Alerting and audit

- Alerts via multiple channels (email, pager, slack, SMS) with severity and timestamp.
- All HITL actions recorded with user, timestamp, and reason; require post-mortem when system resumes.

3.5 Safe-mode automation example (pseudo-code)

```
# within execution service
if regime_prob['stress'] > 0.8 or realized_vol > 2*forecast_vol:
    execution_engine.halve_positions()
    execution_engine.pause_new_orders()
    alert_ops('stress regime triggered; halved positions and paused
reorders')

if drawdown_pct > 0.10:
    execution_engine.flatten_all()
    require_two_person_approval_to_restart()
```

---

4. Hardware, Latency, and Time Synchronization

4.1 Clock synchronization

- **PTP (Precision Time Protocol)**: preferred for sub-millisecond synchronization across servers (required if using microsecond-level event ordering). Use boundary clocks / hardware timestamping on NICs.
- **NTP** acceptable for second-level precision; not sufficient for tick-by-tick matching when latency matters.
- For production: use dedicated PTP grandmaster or GPS-based time source; ensure NICs support hardware timestamping (PTP hardware offload).

4.2 Server sizing and placement

For real-time scoring and execution (per rack): - CPU: 8–16 cores (Xeon equivalent) with high single-thread performance for matching engine and real-time Python/Numba/Cython code. - RAM: 64–256 GB depending on number of instruments and in-memory LOB cache size. - Storage: NVMe SSDs for local caching and replay (1–2 TB), fsync-disabled writes for speed but replicate to durable store asynchronously. - Network: 10GbE or 25GbE NICs; low-latency switch fabric; dedicated interface to exchange gateway. - OS tuning: real-time kernel, CPU affinity, IRQ pinning, hugepages where appropriate.

For backtesting / simulator cluster: - Multi-core nodes with 128–256 GB RAM; use distributed compute (Dask/Spark) for parallel simulation across parameter grid. - LOB simulator benefits from vectorized Numba/Cython/C++; if microsecond resolution required, implement matching engine in C++ and expose Python bindings.

4.3 Latency budget and measurement

Assign budgets and measure: inbound feed latency, feature computation latency, scoring latency, decision & order construction, network RTT to exchange, exchange processing latency.

Example budget (mid-frequency, but queue-position sensitive): - Feed ingest & normalization: 1–10 ms - Feature compute + vol/regime update: 5–20 ms - Model scoring: 1–5 ms - Order construction & send: 1–5 ms - Network RTT to exchange: depends on coloc; coloc <1 ms, remote 5–50 ms

Measure via synthetic pings and timestamped log correlation; use PTP timestamps to correlate events precisely.

4.4 High-availability and disaster recovery

- Active/standby scoring services in separate availability zones; failover time target < 30s.
- Persistent message queue (Kafka) for ingestion with retention to allow replay from any offset.
- Daily restore drills and model re-warm procedures.

4.5 Operational telemetry

- Continuous measurement of event loop latency, GC pauses, and scheduling jitter.
- Instrument fill-latency histograms, queuing delays, and percentiles (p50/p95/p99) for all critical paths.

Appendix: Quick code snippets

Covariance shrinkage (Ledoit-Wolf) with sklearn

```
from sklearn.covariance import LedoitWolf
lw = LedoitWolf().fit(returns_matrix)
Sigma_shrunk = lw.covariance_
```

Simple anomaly rule

```
# price_jump detection
z = abs(price - last_price) / max(eps_sigma, 1e-9)
if z > 6:
    quarantine_tick()
```

CVaR constraint via scenario LP (sketch)

Use historical scenario matrix R (S x N):

Minimize: -mu^T w + gamma * (1/(alpha*S)) * sum_s z_s

s.t. z_s >= -w^T R_s - VaR, z_s >= 0, plus linear constraints.

End of document

Advanced Mathematical, ML, and Infrastructure Refinements for NEPSE Quant System

> **Purpose:** This addendum captures critical last-mile refinements that materially improve robustness, calibration, and survivability of a probabilistic trading system in a thin, regime-driven emerging market. This document is complementary to the main blueprint and assumes full familiarity with it.

---

1. Mathematical Refinements for Thin-Market Dynamics

1.1 Asymmetric Power-Law Market Impact

In illiquid markets, price impact is asymmetric: sell pressure in stress regimes produces larger and faster price dislocations than buy pressure. Model impact separately for buys and sells.

Let trade size be q_t (signed, positive = buy, negative = sell), ADV be average daily volume.

Define asymmetric impact:

$$\text{Impact}(q_t) = \begin{cases} k_1^+ \left(\frac{q_t}{ADV}\right)^{\alpha^+} + k_2^+ \left(\frac{q_t}{ADV}\right)^2 & q_t > 0 \\ k_1^- \left(\frac{|q_t|}{ADV}\right)^{\alpha^-} + k_2^- \left(\frac{|q_t|}{ADV}\right)^2 & q_t < 0 \end{cases}$$

Where typically: - $\alpha^- > \alpha^+$ - $k^- > k^+$ during stress regimes

Regime-conditioned parameters:

$$(k_1^\pm, \alpha^\pm) = f(s_t)$$

Calibrate parameters using regime-filtered execution data via rolling nonlinear least squares or recursive least squares (RLS).

1.2 Hierarchical Bayesian Shrinkage for Thin Stocks

For stocks with sparse trading, estimate parameters via partial pooling.

Example: volatility or impact coefficient θ_i for stock i:

$$\theta_i \sim \mathcal{N}(\mu_{\text{NEPSE}}, \tau^2)$$

Observed estimate:

$$\hat{\theta}_i \mid \theta_i \sim \mathcal{N}(\theta_i, \sigma_i^2)$$

Posterior mean:

$$\mathbb{E}[\theta_i \mid \hat{\theta}_i] = w_i \hat{\theta}_i + (1 - w_i)\mu_{\text{NEPSE}}, \quad w_i = \frac{\tau^2}{\tau^2 + \sigma_i^2}$$

As data accumulates ($\sigma_i^2 \downarrow$), estimates naturally de-shrink.

Use for: - Impact parameters - Volatility persistence - Regime transition probabilities

---

1.3 CVaR-Constrained Portfolio Optimization

Replace stop-loss logic with tail-aware optimization.

Let portfolio returns be $R_p = w^\top R$.

Define CVaR at level α:

$$\text{CVaR}_\alpha(R_p) = \mathbb{E}[R_p \mid R_p \leq \text{VaR}_\alpha]$$

Optimization problem:

$$\max_w \ \mathbb{E}[R_p] - \lambda \cdot \text{CVaR}_\alpha(R_p)$$

Subject to: - Sector exposure caps - Liquidity constraints - Gross and net exposure limits

Solved via linear programming using scenario simulation from regime-weighted predictive distributions.

---

2. Programming & Infrastructure Enhancements

2.1 Stateful LOB Matching Engine (Numba/Cython)

The LOB simulator must preserve event ordering and queue state.

Key state variables: - Queue depth at each price level - Order arrival timestamps - Agent queue position

Fill probability model:

$$\mathbb{P}(\text{fill within } \Delta t) = 1 - \exp\left(-\frac{V_{\text{ahead}}}{\lambda_{\text{market}}}\right)$$

Where: - V_{ahead} = volume ahead in queue - λ_{market} = arrival rate of opposing market orders

Numba/Cython used to process 10^6+ events/sec in backtests and simulation.

2.2 Feature Parity Auditing (Offline vs Live)

Every feature x_t must satisfy:

$$x_t^{\text{train}} \equiv x_t^{\text{live}}$$

Implementation: - Deterministic rolling windows - Fixed seed ordering - Numerical hash (xxhash / SHA256) on feature vectors

$$H(x_t^{\text{train}}) = H(x_t^{\text{live}})$$

Mismatch triggers hard fail and trading halt.

2.3 Hardware Timestamping & Clock Sync

Queue-sensitive strategies require sub-millisecond sync.

- Use **PTP (IEEE 1588)** with hardware NIC timestamping
- Target clock skew: < 50 microseconds

Execution engine timestamps: - Market data arrival - Feature snapshot - Order submission - Exchange ACK

Latency budget must be explicitly tracked and logged.

---

3. Machine Learning Enhancements

3.1 Regime-Conditional Feature Importance

Estimate importance conditional on regime:

$$I_j^{(k)} = \mathbb{E}[\Delta L \mid s_t = k]$$

Where ΔL is loss increase under permutation of feature j.

Findings often show: - Ownership / promoter features dominate in mean-reversion regimes - Liquidity and volatility dominate in stress regimes

---

3.2 Proper Probabilistic Scoring (CRPS)

Train ML models to optimize distributional accuracy.

Continuous Ranked Probability Score:

$$\text{CRPS}(F, y) = \int_{-\infty}^{\infty} (F(z) - 1\{z \geq y\})^2 dz$$

CRPS penalizes: - Overconfident wrong predictions - Underconfident correct predictions

Used as primary objective for: - Quantile regression - NGBoost / probabilistic boosting

---

3.3 Ensemble Anomaly Detection

Combine: 1. Deterministic rules (price jumps, zero volume) 2. Isolation Forest trained on clean historical states

Final anomaly score:

$$A_t = \max(A_t^{\text{rules}}, A_t^{\text{IF}})$$

If $A_t > \tau$, data is quarantined and no signal is generated.

---

4. Monitoring & Drift Detection

4.1 Feature Population Stability Index (PSI)

For feature x:

$$\text{PSI} = \sum_i (p_i - q_i) \log\left(\frac{p_i}{q_i}\right)$$

- p_i: training distribution
- q_i: live distribution

PSI > 0.2 triggers investigation; > 0.3 triggers model retraining or shutdown.

---

5. Summary of Enhancements

Category	Enhancement	Benefit
Math	Asymmetric Impact	Captures sell-side fragility
Math	Hierarchical Shrinkage	Stabilizes thin-stock estimates
Risk	CVaR Optimization	Controls tail losses
Infra	Numba/Cython LOB	Realistic execution simulation
Infra	PTP Sync	Queue-position accuracy
ML	CRPS Objective	Better probabilistic calibration
ML	Regime Feature Importance	Structural interpretability
Ops	PSI Monitoring	Early regime drift detection

End of addendum.