NEPSE Quantitative Trading System

Code Appendix & Mathematical Reference

Complete Implementation Guide

January 2026 Edition

Table of Contents

Section 1: System Workflow

Complete Trading System Workflow:

Step 1: Data Acquisition
→ Ingest OHLCV data from NEPSE
→ Collect ownership data (promoter holdings, free float)
→ Retrieve corporate actions (splits, dividends)
→ Validate data quality and timestamp integrity

Step 2: Data Preparation
→ Calculate log-returns: $r_t = \log(P_t) - \log(P_{t-1})$
→ Test for stationarity (ADF test)
→ Handle missing data and outliers
→ Align multiple data sources by timestamp

Step 3: Regime Detection
→ Fit 2-state Markov regime-switching model
→ Extract regime probabilities $P(s_t=k \mid F_t)$
→ Validate regime separation (no degenerate states)
→ Use regime probabilities as features

Step 4: Volatility Forecasting
→ Fit EGARCH(1,1) with Student-t innovations
→ Generate multi-horizon volatility forecasts
→ Validate model stationarity ($|\beta| < 1$)
→ Fallback to EWMA if GARCH fails validation

Step 5: Feature Engineering
→ Compute ownership concentration (Herfindahl index)
→ Calculate liquidity metrics (Amihud, zero-return ratio)
→ Generate technical indicators (momentum, mean-reversion)
→ Apply Bonferroni correction for feature selection
→ Keep total features < 20 to prevent overfitting

Step 6: Machine Learning Model
→ Decompose returns: $r_t = \text{Linear}_t + \varepsilon_t$
→ Train LightGBM on residuals with aggressive regularization
→ Use time-series cross-validation (5 folds)
→ Validate out-of-sample $R^2 < 0.15$
→ Ensemble predictions across CV folds

Step 7: Signal Generation
→ Combine linear + ML forecasts → μ_{forecast}, σ_{forecast}
→ Calculate transaction costs (spread + fees + impact)
→ Compute net edge: Edge = E[Return] - Costs →
Convert to z-score: $z = \text{Edge} / \sigma_{\text{forecast}}$
→ Generate signal only if z > threshold (e.g., 0.5)

Step 8: Position Sizing
→ Half-Kelly fraction: $0.5 \times \mu / \sigma^2$ → Volatility
targeting: scale by $\text{target\_vol} / \sigma_{\text{forecast}}$
→ Regime adjustment: reduce by 80% in stress regime
→ Liquidity penalty: scale by $1/(1 + 20 \times \text{Amihud})$
→ Apply hard leverage limits (e.g., max 1.5×)

Step 9: Portfolio Construction
→ Aggregate positions across assets
→ Calculate portfolio variance using correlation matrix
→ Check concentration limits (max 30% risk per asset)
→ Scale down if portfolio volatility > target

Step 10: Pre-Trade Risk Checks
→ Verify regime not in extreme stress (P(stress) < 0.85)
→ Check realized vol < 2.5× forecast vol

→ Validate drawdown within limits (< 5% warning, < 10% halt)
→ Confirm data quality (quarantine rate < 5%)
→ Check execution quality (fill rate > 30%)

Step 11: Order Execution
→ Split large orders using VWAP algorithm
→ Model fill probability based on order size vs ADV
→ Track realized slippage vs forecast
→ Update transaction cost model with actual fills

Step 12: Post-Trade Analysis
→ Calculate realized PnL vs forecast
→ Decompose attribution (alpha, costs, slippage)
→ Update model if systematic forecast errors detected
→ Log all trades for audit trail

Step 13: Monitoring & Control
→ Real-time dashboard (positions, PnL, risk metrics)
→ Anomaly detection on incoming data
→ Multi-level circuit breakers (reduce/halt/emergency)
→ Alert operations team on threshold violations
→ Daily model validation (PSI drift, regime stability)

Step 14: Model Retraining
→ Weekly: Check feature distribution shift (PSI)
→ Monthly: Retrain regime and volatility models
→ Quarterly: Full ML model retraining with new data
→ Always: Shadow mode test before deploying updates

Section 2: Mathematical Formulas

2.1 Return Calculations

Single-period log-return:

$$r_t = log(P_t) - log(P_{t-1})$$

Multi-period log-return (horizon h):

$$R_{t,h} = \Sigma^h_{i=1} r_{t+i} = log(P_{t+h} / P_t)$$

2.2 Market Efficiency Tests

Variance Ratio Test: $VR(q) = Var(r_{t-1} + r_{t-q+1} + ... + r_t) / (q \times Var(r_t))$

Under efficient markets (random walk): VR(q) = 1

Hurst Exponent:

$$H = 0.5 \rightarrow \text{random walk};\ H < 0.5 \rightarrow \text{mean-reversion};\ H > 0.5 \rightarrow \text{trending}$$

2.3 Regime-Switching Model

State transition probability:

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

Regime-conditional returns (AR(1)):

$$r_t = \mu^{(k)} + \varphi^{(k)} r_{t-1} + \sigma^{(k)} \varepsilon_t$$

where k ∈ {1,2} represents normal or stress regime

2.4 EGARCH(1,1) Volatility Model

Log-variance specification:

$$\log(\sigma_t^2) = \omega + \beta \log(\sigma_{t-1}^2) + \alpha_1(|\varepsilon_{t-1}|/\sigma_{t-1} - E[|\varepsilon_{t-1}|/\sigma_{t-1}]) + \gamma(\varepsilon_{t-1}/\sigma_{t-1})$$

γ < 0 captures leverage effect (negative shocks increase volatility)

Stationarity condition:

$$|\beta| < 1$$

2.5 Transaction Cost Model

Total execution cost:

$$Cost(x) = HalfSpread + Fees + MarketImpact(x)$$

Non-linear market impact:

$$Impact(x) = \kappa_1 (x/ADV) + \kappa_2 (x/ADV)^2$$

where x = trade size, ADV = average daily volume

Asymmetric impact (regime-conditional): $Total\ Impact = Impact_{base} \times (1 + 2 \times P_{stress}) \times \alpha_{direction}$

$\alpha_{direction}$ = 1.3 for sells, 1.0 for buys

2.6 Microstructure Features

Herfindahl concentration index:

$$H = \Sigma^n_{i=1} (s_i / S)^2$$

where s_i = shares held by investor i, S = total shares

Amihud illiquidity measure:

$$Amihud_t = |r_t| / Volume_t$$

Zero-return ratio (thin trading): $ZeroRatio = (\# \; of \; |r_t| < \varepsilon) / Total \; Observations$

2.7 Edge Calculation

Net expected edge:

$$Edge = E[Return \mid Forecast] - TransactionCosts - MarketImpact - LiquidityPenalty$$

Signal z-score:

$$z = (\mu_{forecast} - TotalCosts) / \sigma_{forecast}$$

Trading threshold: |z| > 0.5 (conservative) to 1.0 (aggressive)

2.8 Integrated Position Sizing

Half-Kelly fraction:

$$f_{Kelly} = 0.5 \times \mu_{forecast} / \sigma^2_{forecast}$$

Volatility targeting scale:

$$Scale_{vol} = \sigma_{target} / \sigma_{forecast}$$

Regime adjustment:

$$Scale_{regime} = 1 - 0.8 \times P(stress)$$

Liquidity adjustment:

$$Scale_{liq} = 1 / (1 + 20 \times Amihud)$$

Final position:

$$Position = f_{Kelly} \times Scale_{vol} \times Scale_{regime} \times Scale_{liq}$$

Clipped to [-max_leverage, +max_leverage]

2.9 Portfolio Risk Metrics

Portfolio variance:

$$\sigma^2_p = w^T \Sigma w$$

where w = position weights, Σ = covariance matrix

Marginal contribution to risk:

$$MCR_i = (\Sigma w)_i / \sigma_p$$

Risk contribution:

$$RC_i = w_i \times MCR_i$$

2.10 Statistical Validation

Bonferroni correction for multiple testing:

$$\alpha_{adjusted} = \alpha / n_{tests}$$

where α = family-wise error rate, n = number of features tested

Out-of-sample R² (model validation):

$$R^2 = 1 - SS_{res} / SS_{tot}$$

Acceptable range for residuals: 0.05 - 0.10

Warning threshold: $R^2 > 0.15$ suggests overfitting

Section 3: Data Preparation Code

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

def log_returns(price: pd.Series) -> pd.Series:
"""Compute log-returns with proper NA handling"""
return np.log(price).diff().dropna()

def stationarity_test(series: pd.Series, max_pvalue=0.05):
"""
    Augmented Dickey-Fuller test for stationarity
    H0: Unit root (non-stationary)
    """     # Remove NaN and infinite values     clean_series =
series.replace([np.inf, -np.inf], np.nan).dropna()         if
len(clean_series) < 30:         return {             'stationary':
False,             'reason': 'insufficient_data'
        }         stat, pvalue, _, _, crit_vals,
_ = adfuller(         clean_series,
maxlag=20     )         return {
'stationary': pvalue < max_pvalue,
        'adf_stat': stat,
        'pvalue': pvalue,
        'critical_values': crit_vals
    }
```

Section 4: Regime Modeling Code

```
from statsmodels.tsa.regime_switching.markov_autoregression import MarkovAutoregression
import warnings

def fit_regime_model(returns: pd.Series, max_iter=100):     """     Fit 2-
state Markov regime-switching AR(1) model     with robust error handling
"""     try:         # Clean data         clean_returns =
returns.replace([np.inf, -np.inf], np.nan).dropna()             if
len(clean_returns) < 200:
```

```
            warnings.warn("Insufficient data for regime model (need 200+ obs)")
return None                 # Fit model with conservative settings
model = MarkovAutoregression(             clean_returns,
k_regimes=2,  # ONLY 2 regimes to reduce parameters             order=1,  #
AR(1) to limit complexity             switching_variance=True  # Allow
different volatility per regime         )
                # Fit with multiple random starts for global
optimum         results = model.fit(
em_iter=max_iter,             search_reps=10,  # Try
multiple initial values             method='nm'         )
                # Extract regime probabilities
regime_probs = results.smoothed_marginal_probabilities
# Validate results         if not
validate_regime_results(results):
            warnings.warn("Regime model failed validation")
return None                 return {             'model':
results,
            'regime_probs': regime_probs,
            'params': extract_regime_params(results)
        }
except Exception as e:
        warnings.warn(f"Regime fitting failed: {e}")
return None

def validate_regime_results(results):     """Check regime model for
pathological outputs"""     # Check for label switching (regimes should
be ordered by volatility)     sigmas =
[results.params[f'sigma2.{i}']**0.5 for i in range(2)]     if sigmas[1] <
sigmas[0]:  # Regime 2 should be higher vol (stress)         return False
# Check for degenerate regimes (one regime has <5% probability)
regime_probs = results.smoothed_marginal_probabilities.mean()     if
(regime_probs < 0.05).any() or (regime_probs > 0.95).any():
        return False
return True

def extract_regime_params(results):
"""Extract parameters in usable format"""
params = {}     for k in range(2):
        params[f'regime_{k}'] = {             'mean':
results.params.get(f'const.{k}', 0),
            'ar1': results.params.get(f'ar.L1.{k}', 0),
            'volatility': results.params[f'sigma2.{k}']**0.5
}     return params
```

Section 5: Volatility Modeling Code

```
from arch import arch_model

def fit_volatility_model(returns: pd.Series, regime_mask=None):
"""
    Fit EGARCH(1,1) with Student-t and numerical checks
        Args:          returns: pandas Series of returns
regime_mask: optional boolean mask for regime-specific fitting     """
# Scale returns to percentage for numerical stability
scaled_returns = returns * 100         if regime_mask is not None:
        scaled_returns = scaled_returns[regime_mask]
    # Remove extreme outliers that can destabilize fitting     #
(keep within 10 standard deviations)     std =
scaled_returns.std()     clean_returns = scaled_returns.clip(-
10*std, 10*std)         try:         # EGARCH(1,1) with Student-
t         model = arch_model(             clean_returns,
mean='Constant',             vol='EGARCH',             p=1,   #
ARCH order              q=1,  # GARCH order
dist='StudentsT'         )                 result =
model.fit(disp='off', options={'maxiter': 500})
# Validate fitted model         if not
validate_volatility_model(result):
            raise ValueError("Volatility model failed validation")
# Extract parameters         params = {             'omega':
result.params['omega'],
            'alpha': result.params['alpha[1]'],
            'beta': result.params['beta[1]'],
            'gamma': result.params['gamma[1]'],  # leverage
            'nu': result.params['nu']  # degrees of freedom
        }
                # Generate forecasts (rescale back to decimal)
forecasts = result.forecast(horizon=5)         variance_forecast =
forecasts.variance.values[-1, :] / 10000                 return {
'model': result,
            'params': params,
            'forecast_vol': np.sqrt(variance_forecast),
            'residuals': result.resid / result.conditional_volatility
        }             except Exception as e:
print(f"Volatility fitting failed: {e}")
# Fallback: simple EWMA         return
fit_ewma_fallback(returns)

def validate_volatility_model(result):
"""Check for pathological GARCH outputs"""
params = result.params         # Check
stationarity: |beta| < 1     if
abs(params['beta[1]']) >= 0.999:
        return False
    # Check for explosive variance     if
params['alpha[1]'] + params['beta[1]'] > 1.5:
        return False         # Check degrees of
freedom (too low = overfitting)     if params['nu'] <
4:
        print("Warning: nu < 4 suggests extreme fat tails")
return True

def fit_ewma_fallback(returns, span=60):
"""Simple EWMA volatility as fallback"""
variance = returns.ewm(span=span).var()
return {         'model': 'EWMA',
        'forecast_vol': np.sqrt(variance.iloc[-1]) * np.ones(5),
        'residuals': returns / np.sqrt(variance)
    }
```

Section 6: Transaction Cost Calibration

```
def market_impact(trade_size, adv, regime_stress_prob, base_params):
"""
    Asymmetric market impact with regime conditioning
        Args:         trade_size: signed (positive=buy,
negative=sell)         adv: average daily volume
regime_stress_prob: probability of stress regime
base_params: dict with 'kappa1', 'kappa2'         Returns:
impact in decimal (e.g., 0.002 = 20 bps)      """      # Normalize
size     u = abs(trade_size) / max(adv, 1)  # prevent division by
zero         # Base impact (symmetric)     kappa1 =
base_params['kappa1']     kappa2 = base_params['kappa2']
    impact_base = kappa1 * u + kappa2 * (u ** 2)         #
Stress multiplier (increases impact in stress regime)
stress_multiplier = 1.0 + 2.0 * regime_stress_prob         #
Sells are penalized more     if trade_size < 0:  # sell
asymmetry_factor = 1.3  # sells have 30% more impact
else:
        asymmetry_factor = 1.0         total_impact = impact_base *
stress_multiplier * asymmetry_factor         # Cap at reasonable
maximum (50% of trade value)     return min(total_impact, 0.50) from
scipy.optimize import least_squares

def calibrate_impact_params(execution_data):
"""
    Calibrate impact parameters from realized fills
execution_data: DataFrame with columns         -
'vwap_exec': execution VWAP
- 'mid_arrival': mid-price at order arrival
- 'trade_size': signed size
- 'adv': average daily volume
- 'half_spread': half bid-ask spread
    """     # Compute realized cost (slippage)
execution_data['realized_cost'] = (
(execution_data['vwap_exec'] - execution_data['mid_arrival']) /
execution_data['mid_arrival']     )
        # Subtract spread (to isolate impact)     execution_data['impact'] =
(         execution_data['realized_cost']
np.sign(execution_data['trade_size']) * execution_data['half_spread']     )
        # Normalize size     execution_data['u'] = (
execution_data['trade_size'].abs() / execution_data['adv']     )
        # Remove outliers (Winsorize at 1st and 99th
percentiles)     impact_clean =
execution_data['impact'].clip(
execution_data['impact'].quantile(0.01),
execution_data['impact'].quantile(0.99)
    )
    # Robust regression (Huber loss)     def residuals(params):         k1, k2
= params         predicted = k1 * execution_data['u'] + k2 *
(execution_data['u'] ** 2)         return impact_clean - predicted
result = least_squares(         residuals,         x0=[0.05, 0.01],  # initial
guess         loss='huber',  # robust to outliers         f_scale=0.01  #
Huber parameter     )         kappa1, kappa2 = result.x         # Validate
results     if kappa1 < 0 or kappa2 < 0:         print("Warning: negative
impact coefficients, using defaults")         kappa1, kappa2 = 0.05, 0.01
# Compute R-squared
    predicted = kappa1 * execution_data['u'] + kappa2 * (execution_data['u'] ** 2)
ss_res = np.sum((impact_clean - predicted) ** 2)     ss_tot = np.sum((impact_clean
- impact_clean.mean()) ** 2)     r2 = 1 - ss_res / ss_tot         return {
'kappa1': kappa1,
        'kappa2': kappa2,
        'r_squared': r2,
        'n_obs': len(execution_data)
    }
```

Section 7: Feature Engineering Code

```
from scipy import stats from
statsmodels.stats.multitest import multipletests

def select_features_bonferroni(X, y, alpha=0.01):
"""
    Feature selection with Bonferroni correction
        Args:         X: DataFrame of candidate features
y: Series of target (returns)         alpha: family-wise
error rate (conservative: 0.01)         Returns:
        List of selected feature names     """
n_features = X.shape[1]     p_values = []
correlations = []          for col in X.columns:
# Use Spearman correlation (robust to outliers)
corr, pval = stats.spearmanr(
X[col].dropna(),
            y.loc[X[col].dropna().index]
)          p_values.append(pval)
correlations.append(abs(corr))         #
Bonferroni correction     reject, p_corrected,
_, _ = multipletests(         p_values,
alpha=alpha,         method='bonferroni'     )
        # Select features that survive correction     selected =
X.columns[reject].tolist()         # Report results     print(f"Tested
{n_features} features at α={alpha}")     print(f"Selected {len(selected)}
features after Bonferroni correction")         # Show top features by
correlation     feature_importance = pd.DataFrame({         'feature':
X.columns,
        'correlation': correlations,
        'p_value': p_values,
        'p_corrected': p_corrected,
        'significant': reject
}).sort_values('correlation', ascending=False)
    return selected, feature_importance

def compute_concentration_features(ownership_data):
"""
    Ownership concentration features
ownership_data: DataFrame with columns        -
'promoter_shares': shares held by promoters
        - 'free_float': publicly tradeable shares         -
'top10_shares': array/list of top 10 holder shares     """
features = {}          # Promoter concentration
features['promoter_ratio'] = (
ownership_data['promoter_shares'] /
ownership_data['free_float']
    )
    # Herfindahl index (sum of squared shares)
top10 = np.array(ownership_data['top10_shares'])
total = top10.sum()     if total > 0:
        shares_fraction = top10 / total
features['herfindahl'] = np.sum(shares_fraction ** 2)
else:
        features['herfindahl'] = np.nan         # Top 3
concentration     features['top3_ratio'] = top10[:3].sum() /
max(total, 1)         return features

def compute_liquidity_features(trade_data, window=20):
"""
    Microstructure liquidity features         trade_data:
DataFrame with timestamp index and columns         -
'price': trade prices
- 'volume': trade volumes
- 'returns': log returns    """     features = {}         # Zero-return
  frequency (thin trading indicator)     features['zero_return_ratio'] =
  (         (trade_data['returns'].abs() < 1e-6).rolling(window).mean()
    )
```

```
        # Amihud illiquidity     features['amihud'] = (
trade_data['returns'].abs() / trade_data['volume']
).rolling(window).mean()
        # Average time between trades (in minutes)     time_diffs =
trade_data.index.to_series().diff().dt.total_seconds() / 60
features['avg_trade_interval'] = time_diffs.rolling(window).mean()         #
Volume concentration (% of daily volume in top 10% of trades)
daily_volume = trade_data['volume'].resample('D').sum()     top_decile =
trade_data.groupby(         trade_data.index.date
)['volume'].nlargest(lambda x: max(1, len(x) // 10))
features['volume_concentration'] = (
        top_decile.sum() / max(daily_volume.sum(), 1)
)         return features
```

Section 8: Machine Learning Code

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

def train_ml_model(X, y, n_splits=5):
"""
    Train LightGBM on standardized residuals with strict regularization
        Args:         X: DataFrame of features
(causal only!)         y: Series of
standardized residuals
        n_splits: number of temporal cross-validation folds
Returns:
        Trained models (ensemble from CV folds)
    """     # Time-series cross-validation (no random shuffle!)
tscv = TimeSeriesSplit(n_splits=n_splits)         models = []
oos_predictions = pd.Series(index=y.index, dtype=float)         for
fold_idx, (train_idx, val_idx) in enumerate(tscv.split(X)):
X_train, X_val = X.iloc[train_idx], X.iloc[val_idx]
y_train, y_val = y.iloc[train_idx], y.iloc[val_idx]
# LightGBM datasets         train_data = lgb.Dataset(X_train,
label=y_train)
        val_data = lgb.Dataset(X_val, label=y_val, reference=train_data)
# AGGRESSIVE regularization to prevent overfitting         params = {
'objective': 'regression',
            'metric': 'l2',
            'boosting_type': 'gbdt',
            'num_leaves': 15,  # SMALL tree (default 31)
            'max_depth': 4,  # SHALLOW trees
            'learning_rate': 0.01,  # SLOW learning
            'feature_fraction': 0.6,  # Random feature sampling
            'bagging_fraction': 0.8,  # Random sample sampling
            'bagging_freq': 5,
            'lambda_l1': 2.0,  # L1 regularization
            'lambda_l2': 2.0,  # L2 regularization
            'min_data_in_leaf': 50,  # Prevent small leaves
            'min_gain_to_split': 0.01,  # Minimum improvement required
            'verbose': -1
        }
                # Train with early stopping
model = lgb.train(            params,
train_data,             num_boost_round=1000,
valid_sets=[val_data],             callbacks=[
lgb.early_stopping(stopping_rounds=50),
lgb.log_evaluation(period=0)             ]
        )                 models.append(model)
# Store out-of-sample predictions
oos_predictions.iloc[val_idx] = model.predict(X_val)
# Validate no overfitting     validate_ml_model(y,
oos_predictions)         return {         'models': models,
        'oos_predictions': oos_predictions,
        'feature_importance': get_feature_importance(models, X.columns)
}

def validate_ml_model(y_true, y_pred, max_r2=0.15):
"""
```

```
    Check for overfitting
    If R² > max_r2, model is likely overfitting noise
    Realistic R² for residuals should be 0.05-0.10     """
valid_idx = y_true.notna() & y_pred.notna()     y_true_clean =
y_true[valid_idx]     y_pred_clean = y_pred[valid_idx]
ss_res = np.sum((y_true_clean - y_pred_clean) ** 2)     ss_tot
= np.sum((y_true_clean - y_true_clean.mean()) ** 2)     r2 = 1
- ss_res / ss_tot         print(f"Out-of-sample R²: {r2:.4f}")
if r2 > max_r2:
        print(f"WARNING: R² = {r2:.4f} > {max_r2} suggests overfitting")
print("Consider: more regularization, fewer features, or simpler model")
if r2 < 0:
        print(f"WARNING: Negative R² suggests model worse than mean")
return r2

def get_feature_importance(models, feature_names):     """Average feature importance across CV
folds"""     importance_df = pd.DataFrame()         for i, model in enumerate(models):         imp
= pd.DataFrame({             'feature': feature_names,             f'fold_{i}':
model.feature_importance(importance_type='gain')         })         importance_df =
pd.concat([importance_df, imp], axis=1) if not importance_df.empty else imp         # Average
across folds     importance_df['mean_importance'] = importance_df.filter(like='fold_').mean(axis=1)
importance_df = importance_df[['feature', 'mean_importance']].sort_values(
'mean_importance',         ascending=False     )         return importance_df
```

Section 9: Signal Generation Code

```
def calculate_edge(mu_forecast, sigma_forecast, position_size,
half_spread, impact_params, adv, illiquidity):     """
    Calculate expected edge (net expected return after costs)
        Args:         mu_forecast: expected return (from
combined model)         sigma_forecast: forecast volatility
position_size: desired position as fraction of portfolio
half_spread: half bid-ask spread         impact_params: dict
with 'kappa1', 'kappa2'         adv: average daily volume
illiquidity: Amihud measure         Returns:         edge:
expected net return         components: dict breaking down edge
components     """     # Gross expected return     gross_return
= position_size * mu_forecast         # Transaction costs
explicit_cost = half_spread + 0.0005  # spread + fees         #
Market impact (non-linear in size)     u = abs(position_size) /
max(adv, 1e-6)     impact = (impact_params['kappa1'] * u +
impact_params['kappa2'] * (u ** 2))         # Liquidity penalty
(opportunity cost)     liquidity_penalty = 0.001 * illiquidity *
abs(position_size)         # Net edge     total_costs =
explicit_cost + impact + liquidity_penalty     edge =
gross_return - total_costs * abs(position_size)         # Z-
score (edge relative to forecast uncertainty)
    z_score = (gross_return - total_costs) / max(sigma_forecast, 1e-6)
return {         'edge': edge,
        'z_score': z_score,
        'gross_return': gross_return,
        'explicit_cost': explicit_cost,
        'impact': impact,
        'liquidity_penalty': liquidity_penalty,
        'total_costs': total_costs
}

def make_trading_signal(edge_calc, threshold_z=0.5):
"""     Convert edge calculation to binary trade
signal

Args:
        edge_calc: output from calculate_edge()
threshold_z: minimum z-score to trade
Returns:
        signal: 0 (no trade), +1 (buy), -1 (sell)
confidence: strength of signal     """     if
abs(edge_calc['z_score']) < threshold_z:
        return {
'signal': 0,
            'confidence': 0,
            'reason': 'insufficient_edge'
        }         if
edge_calc['edge'] <= 0:
        return {
            'signal': 0,
            'confidence': 0,
            'reason': 'negative_edge_after_costs'
        }
        # Signal
direction
    signal = np.sign(edge_calc['z_score'])         #
Confidence (bounded)     confidence =
min(abs(edge_calc['z_score']) / 2.0, 1.0)         return {
'signal': int(signal),
        'confidence': confidence,
        'reason': 'positive_edge',
        'edge_bps': edge_calc['edge'] * 10000  # in basis points
    }
```

Section 10: Position Sizing Code

```
def integrated_position_sizing(mu_forecast, sigma_forecast,
regime_stress_prob, illiquidity,
target_vol=0.15, max_leverage=1.5):     """
    Combine multiple position sizing methods with safety limits
        Args:         mu_forecast: expected return
sigma_forecast: forecast volatility         regime_stress_prob:
probability in stress regime         illiquidity: Amihud measure
target_vol: target portfolio volatility (e.g., 0.15 = 15%)
max_leverage: maximum gross leverage         Returns:
final_position: position size as fraction of capital     """     #
1. Half-Kelly (aggressive but bounded)
    kelly_fraction = 0.5 * mu_forecast / max(sigma_forecast**2, 1e-6)
# 2. Volatility targeting     vol_scale = target_vol / max(sigma_forecast,
1e-6)         # 3. Regime scaling (reduce in stress)     regime_scale = 1.0
- 0.8 * regime_stress_prob  # 80% reduction in stress         # 4. Liquidity
scaling (penalize illiquid assets)     liq_scale = 1.0 / (1.0 + 20.0 *
illiquidity)         # 5. Combine all factors     raw_position =
kelly_fraction * vol_scale * regime_scale * liq_scale         # 6. Apply
hard limits     final_position = np.clip(raw_position, -max_leverage,
max_leverage)         # 7. Additional safety checks     if
abs(final_position) < 0.01:  # too small to matter         final_position =
0         # 8. Warn if any factor is binding     if abs(raw_position) >
max_leverage:
        print(f"Warning: Position clipped from {raw_position:.2f} to {final_position:.2f}")
return {         'position': final_position,
        'kelly_fraction': kelly_fraction,
        'vol_scale': vol_scale,
        'regime_scale': regime_scale,
        'liq_scale': liq_scale,
'raw_position': raw_position
    }
```

Section 11: Portfolio Risk Management Code

```
def apply_portfolio_risk_limits(individual_positions, correlations,
volatilities, max_concentration=0.30,
max_portfolio_vol=0.20):     """
    Apply portfolio-level risk constraints
        Args:         individual_positions: dict {asset: position_size}
correlations: correlation matrix         volatilities: dict {asset: volatility}
max_concentration: max single-asset contribution to portfolio vol
max_portfolio_vol: maximum portfolio volatility         Returns:
adjusted_positions: risk-limited positions     """     assets =
list(individual_positions.keys())     positions = np.array([individual_positions[a] for
a in assets])     vols = np.array([volatilities[a] for a in assets])         # Calculate
portfolio variance     portfolio_var = positions @ correlations @ (positions * vols**2)
portfolio_vol = np.sqrt(portfolio_var)         # Calculate marginal contribution to risk
marginal_risk = correlations @ (positions * vols**2) / max(portfolio_vol, 1e-6)
# Individual risk contributions     risk_contribution = positions * vols * marginal_risk
# Scale down if portfolio vol exceeds limit     if portfolio_vol > max_portfolio_vol:
scale_factor = max_portfolio_vol / portfolio_vol         positions = positions *
scale_factor         print(f"Scaled all positions by {scale_factor:.2f} to meet
portfolio vol limit")         # Scale down any asset with excess concentration     for
i, asset in enumerate(assets):
        contrib_fraction = abs(risk_contribution[i]) / max(portfolio_vol, 1e-6)
if contrib_fraction > max_concentration:
            scale_down = max_concentration / contrib_fraction
positions[i] *= scale_down             print(f"Scaled {asset} by
{scale_down:.2f} due to concentration")         return {asset: pos for asset,
pos in zip(assets, positions)}
```

Section 12: Backtesting Code

```
def walk_forward_backtest(returns, features, train_window=756,
test_window=63, step=21):     """
    Walk-forward backtest with expanding window
        Args:         returns: pandas Series of asset returns
features: pandas DataFrame of features (aligned with returns)
```

```
train_window: training window (e.g., 756 = 3 years)
test_window: testing window (e.g., 63 = 3 months)         step: re-
training frequency (e.g., 21 = monthly)         Returns:
backtest_results: DataFrame with predictions, signals, PnL     """
n = len(returns)     results = []         for start in range(0, n -
train_window - test_window, step):         # Define windows
train_end = start + train_window         test_end = min(train_end +
test_window, n)                 train_idx = range(start, train_end)
test_idx = range(train_end, test_end)
        # Extract data         X_train = features.iloc[train_idx]         y_train =
returns.iloc[train_idx]         X_test = features.iloc[test_idx]         y_test =
returns.iloc[test_idx]                 # Fit models (regime, vol, ML)
regime_model = fit_regime_model(y_train)         vol_model =
fit_volatility_model(y_train)         ml_model = train_ml_model(X_train,
standardize_residuals(y_train, vol_model))                 # Generate forecasts
for t in test_idx:
            forecast = generate_forecast(
features.iloc[t],
regime_model,                 vol_model,
ml_model             )
                        # Calculate
edge and signal
            edge = calculate_edge(forecast['mu'], forecast['sigma'], ...)
signal = make_trading_signal(edge)                         # Simulate
execution with costs             pnl = simulate_execution(signal,
returns.iloc[t], ...)
            results.append({
'date': returns.index[t],
                'forecast_mu': forecast['mu'],
                'forecast_sigma': forecast['sigma'],
                'signal': signal['signal'],
                'realized_return': returns.iloc[t],
                'pnl': pnl
            })
    return pd.DataFrame(results).set_index('date')

def bootstrap_backtest_validation(backtest_results, n_bootstrap=1000):
"""
    Bootstrap validation to check robustness     Randomly
sample blocks of backtest returns to generate
distribution of performance metrics
    """
    strategy_returns = backtest_results['pnl']
sharpe_distribution = []     max_dd_distribution = []
# Block bootstrap (preserve time-series structure)
block_size = 21  # ~1 month blocks
    n_blocks = len(strategy_returns) // block_size         for _ in
range(n_bootstrap):         # Sample blocks with replacement
sampled_blocks = np.random.choice(n_blocks, size=n_blocks, replace=True)
# Reconstruct return series         bootstrap_returns = []         for block_id
in sampled_blocks:
            start_idx = block_id * block_size             end_idx = min(start_idx +
block_size, len(strategy_returns))
bootstrap_returns.extend(strategy_returns.iloc[start_idx:end_idx])
bootstrap_returns = pd.Series(bootstrap_returns)                 # Calculate metrics
sharpe = bootstrap_returns.mean() / max(bootstrap_returns.std(), 1e-6) * np.sqrt(252)
max_dd = (bootstrap_returns.cumsum() - bootstrap_returns.cumsum().cummax()).min()
sharpe_distribution.append(sharpe)         max_dd_distribution.append(max_dd)         #
Report percentiles     print("Bootstrap Validation Results (1000 iterations):")
print(f"Sharpe Ratio: {np.percentile(sharpe_distribution, [5, 50, 95])}")     print(f"Max
Drawdown: {np.percentile(max_dd_distribution, [5, 50, 95])}")         # Check if median is
positive     if np.median(sharpe_distribution) < 0.3:
        print("WARNING: Median Sharpe < 0.3 suggests weak edge")
return {         'sharpe_dist': sharpe_distribution,
        'max_dd_dist': max_dd_distribution
    }
```

Section 13: Production Architecture Code

```
import logging import
traceback from datetime
import datetime

class TradingSystemExecutor:     """Production trading system
with fail-safe mechanisms"""         def __init__(self,
config):         self.config = config         self.state =
'INITIALIZED'         self.error_count = 0
self.max_errors_per_hour = 10         self.last_error_time =
None         def safe_execute(self, func, *args, **kwargs):
"""Wrap all critical operations with error handling"""
try:
            result = func(*args, **kwargs)
self.error_count = 0  # reset on success
return result                     except
DataStalenessError as e:
            self.state = 'HALTED_STALE_DATA'
self.alert_ops(f"CRITICAL: Data stale - {e}", severity='HIGH')
self.halt_trading()             return None                     except
ModelScoringError as e:
            self.error_count += 1             self.alert_ops(f"Model
scoring failed: {e}", severity='MEDIUM')                        if
self.error_count > self.max_errors_per_hour:
                self.state = 'HALTED_MODEL_FAILURE'
self.alert_ops(                     f"CRITICAL: Model failed
{self.error_count} times",                     severity='HIGH'
)                 self.halt_trading()             return None
except InsufficientLiquidity as e:             self.alert_ops(f"Liquidity
constraint: {e}", severity='LOW')             # Don't halt, just skip this
trade             return None                     except Exception as e:
# Unknown error - immediate halt           self.state = 'HALTED_UNKNOWN'
self.alert_ops(                 f"FATAL: {type(e).__name__}:
{e}\n{traceback.format_exc()}",                 severity='CRITICAL'
)           self.emergency_flatten()             return None         def
health_check(self):         """Run every 60 seconds"""         checks = {
'data_fresh': self.check_data_freshness(),
            'model_loaded': self.model is not None,
            'features_valid': self.validate_features(),
            'risk_limits_ok': self.check_risk_limits(),
            'connectivity': self.check_exchange_connection(),
            'clock_sync': self.check_clock_sync()
        }
        failed = [k for k, v in checks.items() if not v]
if failed:
            self.alert_ops(f"Health check failed: {failed}", severity='HIGH')
self.state = 'DEGRADED'                     # Auto-recovery for minor
issues             if 'data_fresh' in failed:
self.restart_data_feed()                         # Halt for critical issues
if 'model_loaded' in failed or 'connectivity' in failed:
                self.halt_trading()
return all(checks.values())         def
alert_ops(self, message, severity='INFO'):
"""Send alerts via multiple channels"""
timestamp = datetime.now().isoformat()
# Log to file (always)         logging.log(
getattr(logging, severity),
f"[{timestamp}] {message}"         )
                # Email for high severity
if severity in ['HIGH', 'CRITICAL']:
            send_email(self.config['ops_email'], f"Trading Alert: {message}")
# SMS for critical         if severity == 'CRITICAL':
            send_sms(self.config['ops_phone'], f"CRITICAL: {message[:100]}")
# Slack (all severities)         post_to_slack(self.config['slack_webhook'],
{             'severity': severity,
            'message': message,
            'timestamp': timestamp,
            'system_state': self.state
        })         def emergency_flatten(self):         """Flatten
all positions immediately"""         try:             for asset,
position in self.current_positions.items():                 if
abs(position) > 0:                     # Market order to close
self.execute_market_order(                         asset,
-position,
                        reason='emergency_flatten'                        )
self.alert_ops("Emergency flatten completed", severity='HIGH')
except Exception as e:
            self.alert_ops(f"Emergency flatten FAILED: {e}", severity='CRITICAL')
```

Section 14: Kill Switches & Anomaly Detection

```
def check_circuit_breakers(current_state):
"""
    Hierarchical circuit breakers with automatic responses
        Returns:         action: 'CONTINUE', 'REDUCE', 'HALT',
'EMERGENCY_STOP'     """     # Level 1: Regime stress (reduce
exposure)     if current_state['regime_stress_prob'] > 0.85:
        return {
'action': 'REDUCE',
            'scale_factor': 0.5,
            'reason': 'extreme_stress_regime',
            'severity': 'MEDIUM'
        }
        # Level 2: Volatility surprise (halt new
trades)     realized_vol =
current_state['realized_vol_60min']     forecast_vol =
current_state['forecast_vol']     if realized_vol >
2.5 * forecast_vol:
        return {
'action': 'HALT',
            'cooldown_minutes': 60,
            'reason': 'volatility_surprise',
            'severity': 'HIGH'
        }
        # Level 3: Drawdown limits
drawdown_pct = current_state['drawdown_pct']
if drawdown_pct > 0.05:  # 5% drawdown
return {             'action': 'REDUCE',
            'scale_factor': 0.5,
            'reason': 'drawdown_5pct',
            'severity': 'MEDIUM'         }
if drawdown_pct > 0.10:  # 10% drawdown
return {             'action': 'HALT',
            'require_approval': True,
            'reason': 'drawdown_10pct',
            'severity': 'HIGH'
        }
        # Level 4: Data quality     if
current_state['data_quarantine_rate'] > 0.05:  # >5% bad data
return {             'action': 'HALT',             'reason':
'data_quality_degraded',             'severity': 'HIGH'
        }
        # Level 5: Execution
quality
    if current_state['fill_rate'] < 0.30:  # <30% fills
return {             'action': 'REDUCE',
            'scale_factor': 0.3,
            'reason': 'poor_execution_quality',
            'severity': 'MEDIUM'
        }
        # Level 6: Model confidence     if
current_state['forecast_confidence'] < 0.20:
        return {
'action': 'REDUCE',
            'scale_factor': 0.2,
            'reason': 'low_model_confidence',
            'severity': 'LOW'
        }
    return {
'action': 'CONTINUE',
        'reason': 'all_checks_passed',
        'severity': 'INFO'
}

def detect_data_anomalies(tick_data, historical_stats):
"""
    Multi-method anomaly detection for market data
        Returns:
is_anomalous: boolean
        anomaly_type: str describing issue     """     # 1. Price
jump detection (statistical)     price_change =
abs(tick_data['price'] - tick_data['last_price'])     expected_std =
historical_stats['price_std_10min']     z_score = price_change /
max(expected_std, 1e-6)        if z_score > 6:
        return True, 'price_jump_6sigma'         # 2. Volume spike detection
volume_ratio = tick_data['volume'] / max(historical_stats['avg_volume'], 1)
if volume_ratio > 20:
        return True, 'volume_spike_20x'         # 3.
Spread widening     spread = tick_data['ask'] -
tick_data['bid']     normal_spread =
historical_stats['median_spread']     if spread > 5 *
normal_spread:
        return True, 'spread_widening_5x'         # 4. Timestamp
issues     time_since_last = (tick_data['timestamp'] -
tick_data['last_timestamp']).total_seconds()     if
time_since_last > 300:  # 5 minutes         return True,
'stale_data_5min'     if time_since_last < 0:
        return True, 'timestamp_reversal'         # 5. Outside BBO
bounds (fat finger)     if not (tick_data['bid'] <= tick_data['price']
<= tick_data['ask']):
        return True, 'outside_bbo'
return False, 'normal'

def sanitize_data(tick_data, anomaly_type):     """Decide how to handle
anomalous data"""     # Quarantine severe anomalies     severe_types =
['timestamp_reversal', 'outside_bbo', 'price_jump_6sigma']     if
anomaly_type in severe_types:         return {             'action':
'QUARANTINE',
            'reason': anomaly_type
        }
        # Impute minor issues
minor_types = ['stale_data_5min']
if anomaly_type in minor_types:
return {             'action':
'IMPUTE',
            'method': 'last_known_good',
'reason': anomaly_type
        }
        # Flag but use (with
caution)
    moderate_types = ['volume_spike_20x', 'spread_widening_5x']
if anomaly_type in moderate_types:         return {
'action': 'FLAG_AND_USE',
            'increase_uncertainty': True,
            'reason': anomaly_type
        }
    return {
'action': 'USE',
        'reason': 'normal'
    }
```

Code Implementation Summary

Critical Implementation Notes:

1. **Parameter Parsimony:** Use maximum 2 regimes, not 3+. Each additional parameter multiplies overfitting risk in sparse emerging market data.

2. **Realistic Cost Modeling:** Calibrate transaction costs from actual fills using robust regression. If unavailable, use pessimistic defaults (κ■=0.10, κ■=0.05).

3. **Validation Rigor:** Bootstrap backtest 1000+ times. Median Sharpe must be positive. Out-of-sample R^2 for ML residuals should be 0.05-0.10; $R^2 > 0.15$ indicates overfitting.

4. **Feature Selection:** Apply Bonferroni correction with $\alpha=0.01$. Keep total features < 20 for sample sizes < 1000 observations.

5. **Production Discipline:** Shadow mode for minimum 3 months before live deployment. Comprehensive monitoring with multi-level circuit breakers.

6. **Risk Management:** Automated circuit breakers at multiple levels (regime, volatility, drawdown, data

TAKE NOTE OF BELOW:

Problem: Original assumes uniform queue position. Reality: institutional orders get priority.

Math:

```
P(fill | q, type) = {
    institutional: (1 - 0.3q) × 0.25  if q < 0.4
    retail: (1 - q) × 0.15  if q > 0.3
}
q ~ Beta(2,5) for institutional, Beta(5,2) for retail
```

Code:

```
from scipy.stats import beta
import numpy as np

class NEPSERealisticQueueModel:
    def __init__(self):
        self.fill_rates = {'institutional': 0.25, 'retail': 0.15}
        self.queue_dist = {
            'institutional': beta(2, 5),  # Front-loaded
            'retail': beta(5, 2)           # Back-loaded
        }

    def simulate_fill(self, order_size, circuit_hit, client_type='retail',
n=1000):
        if not circuit_hit:
            return order_size, 1.0

        fills = []
        for _ in range(n):
            q = self.queue_dist[client_type].rvs()

            if client_type == 'institutional' and q < 0.4:
                fill_frac = (1 - 0.3*q) * 0.25
            elif client_type == 'retail' and q > 0.3:
                fill_frac = (1 - q) * 0.15
            else:
                fill_frac = 0.05

            fills.append(order_size * np.clip(
                np.random.uniform(0.5*fill_frac, 1.5*fill_frac), 0, 1
            ))

        return np.mean(fills), (np.array(fills) > 0).mean()
```

QUIRK 2: NEPSE DATA QUALITY FILTERS

Problem: Ownership data has systematic errors:

- `Exactly 51.00% promoter = stale (regulatory minimum)`
- `Updates lag 1-2 weeks`
- `Retroactive corrections`

Code:

```
from datetime import datetime, timedelta

class NEPSEDataQualityFilter:
    def __init__(self):
        self.quality_scores = {}

    def validate_ownership(self, symbol, promoter_pct, public_pct,
                          update_date, current_date):
        flags = []
        confidence = 1.0

        # Flag 1: Exactly 51% (suspicious)
        if abs(promoter_pct - 0.51) < 0.0001:
            flags.append("EXACT_51PCT")
            confidence *= 0.5

        # Flag 2: Sum != 100%
        if abs(promoter_pct + public_pct - 1.0) > 0.01:
            flags.append("SUM_ERROR")
            confidence *= 0.3

        # Flag 3: Stale data
        days_old = (current_date - update_date).days
        if days_old > 30:
            flags.append(f"STALE_{days_old}d")
            confidence *= max(0.2, 1 - days_old/365)
        return {
            'valid': confidence > 0.5,
            'confidence': confidence,
            'flags': flags
        }

    def validate_price(self, prices, volumes):
        flags = []

        # Zero volume %
        if (volumes == 0).mean() > 0.3:
            flags.append("HIGH_ZERO_VOL")

        # Unchanged prices
        if (prices.diff() == 0).mean() > 0.4:
            flags.append("STALE_PRICES")

        # Extreme moves
        if (prices.pct_change().abs() > 0.20).sum() > 0:
            flags.append("EXTREME_MOVES")

        return flags
```

QUIRK 3: CALENDAR-ADJUSTED REGIME TRANSITIONS

Problem: Regime transitions vary by Nepal calendar periods.

Math:

```
Q(t) = Q_base × Λ(period)

Λ_budget = [0.85, 1.15, 1.20, 0.90]  (May-July)
Λ_dashain = [1.10, 0.70, 0.75, 1.05] (Sep-Oct)
Λ_tihar = [1.08, 0.75, 0.80, 1.03]   (Oct-Nov)
Λ_yearend = [0.95, 1.30, 1.25, 0.88] (mid-July)

Order: [q_nn, q_ns, q_ss, q_sn]
```

Code:

```
import numpy as np

class CalendarAdjustedRegimeModel:
    def __init__(self, Q_base):
        self.Q_base = Q_base
        self.adjustments = {
            'budget': {'q_nn': 0.85, 'q_ns': 1.15, 'q_ss': 1.20, 'q_sn': 0.90},
            'dashain': {'q_nn': 1.10, 'q_ns': 0.70, 'q_ss': 0.75, 'q_sn': 1.05},
            'tihar': {'q_nn': 1.08, 'q_ns': 0.75, 'q_ss': 0.80, 'q_sn': 1.03},
            'yearend': {'q_nn': 0.95, 'q_ns': 1.30, 'q_ss': 1.25, 'q_sn': 0.88},
            'normal': {'q_nn': 1.00, 'q_ns': 1.00, 'q_ss': 1.00, 'q_sn': 1.00}
        }

    def identify_period(self, date):
        m, d = date.month, date.day

        if m in [5,6,7] and not (m==5 and d<15):
            return 'budget', 0.9
        if (m==9 and d>=25) or (m==10 and d<=15):
            return 'dashain', 0.95
        if (m==10 and d>=20) or (m==11 and d<=10):
            return 'tihar', 0.90
        if m==7 and 10<=d<=20:
            return 'yearend', 0.85

        return 'normal', 1.0

    def get_adjusted_transitions(self, date, political_stress_mult=1.0):
        period, conf = self.identify_period(date)
        adj = self.adjustments[period]

        Q = self.Q_base.copy()
        Q[0,0] *= adj['q_nn']
        Q[0,1] *= adj['q_ns']
        Q[1,0] *= adj['q_sn']
        Q[1,1] *= adj['q_ss']

        # Political stress adjustment
        if political_stress_mult > 1.0:
            Q[0,1] *= political_stress_mult
            Q[1,0] /= political_stress_mult

        # Renormalize
        Q[0,:] /= Q[0,:].sum()
        Q[1,:] /= Q[1,:].sum()

        return Q, period, conf
```

QUIRK 4: BROKER EXECUTION QUALITY

Problem: NEPSE brokers have vastly different fill rates.

Math:

```
P(fill | broker) = P_base × η_broker

η_tier1 = 1.10-1.15  (GBIME, Sanima, NMB, Civil)
η_tier2 = 0.85-1.05  (mid-tier)
η_tier3 = 0.35-0.70  (small brokers)
```

Code:

```
class BrokerQualityModel:
    def __init__(self):
        self.quality = {
            'GBIME': 1.15, 'Sanima': 1.12, 'NMB': 1.10,
            'Civil': 1.10, 'Nepal SBI': 1.08,
            'default': 0.90
        }
        self.fills = {}

    def adjust_fill_prob(self, base_prob, broker):
        mult = self.quality.get(broker, 0.90)
        return np.clip(base_prob * mult, 0, 1)

    def record_fill(self, broker, order_size, filled, limit_dist):
        if broker not in self.fills:
            self.fills[broker] = []

        self.fills[broker].append({
            'fill_rate': filled/order_size,
            'limit_distance': limit_dist
        })

    def recalibrate(self, broker):
        if broker not in self.fills or len(self.fills[broker]) < 30:
            return

        fills = self.fills[broker]
        actual = np.mean([f['fill_rate'] for f in fills])
        expected = np.exp(-15 * np.mean([f['limit_distance'] for f in fills]))

        if expected > 0:
            new_mult = np.clip(actual / expected, 0.35, 1.15)
            old = self.quality.get(broker, 0.90)
            self.quality[broker] = 0.2*new_mult + 0.8*old  # Smooth update
```

QUIRK 5: AI SOCIAL MEDIA SENTIMENT (POLITICAL)

User Note: Nepal politics too chaotic for fixed models. Use AI sentiment tracker.

Math:

```
S_political = 0.4×S_twitter + 0.2×S_reddit + 0.4×S_news

Confidence = exp(-0.6 × |S_political|)
Regime_stress_mult = 1 + max(0, -S_political) × 2
Position_adj = 1 - 0.35×S_political  (clipped [0.3, 1.0])
```

Code:

```
from collections import deque

class NepalPoliticalSentimentTracker:
    def __init__(self, api_endpoint=None):
        self.api = api_endpoint
        self.weights = {'twitter': 0.4, 'reddit': 0.2, 'news': 0.4}
        self.history = deque(maxlen=30)

    def fetch_sentiment(self):
        if self.api is None:
            return {'twitter_sentiment': 0, 'reddit_sentiment': 0,
                    'news_sentiment': 0, 'keywords': []}
        try:
            import requests
            resp = requests.get(self.api, timeout=5)
            return resp.json() if resp.status_code == 200 else self._fallback()
        except:
            return self._fallback()

    def _fallback(self):
        if len(self.history) > 0:
            recent = list(self.history)[-7:]
            return {
                'twitter_sentiment': np.mean([s['twitter_sentiment'] for s in
recent]),
                'reddit_sentiment': np.mean([s['reddit_sentiment'] for s in
recent]),
                'news_sentiment': np.mean([s['news_sentiment'] for s in recent]),
                'keywords': []
            }
        return {'twitter_sentiment': 0, 'reddit_sentiment': 0, 'news_sentiment':
0, 'keywords': []}

    def calculate_stress(self, data=None):
        if data is None:
            data = self.fetch_sentiment()

        self.history.append(data)

        combined = (self.weights['twitter'] * data['twitter_sentiment'] +
                    self.weights['reddit'] * data['reddit_sentiment'] +
                    self.weights['news'] * data['news_sentiment'])

        # Negative sentiment → stress
        stress = 1.0 - combined  # Map [-1,1] to [2,0]

        # Keyword penalty
        critical_kw = ['coalition', '□□□□□', 'resign', '□□□□□□□', 'protest',
'strike']
        kw_penalty = sum(1 for kw in data.get('keywords', [])
                         if any(c in kw.lower() for c in critical_kw)) * 0.1

        return np.clip(stress + kw_penalty, 0, 2)

    def get_confidence(self, stress=None):
        if stress is None:
            stress = self.calculate_stress()
        return max(np.exp(-0.6 * stress), 0.3)

    def get_position_adj(self, stress=None):
        if stress is None:
            stress = self.calculate_stress()
        return np.clip(1.0 - 0.35*stress, 0.3, 1.0)

    def get_regime_mult(self, stress=None):
        if stress is None:
            stress = self.calculate_stress()
        return 0.5 + 0.75*stress
```

AI Service Spec (for your friend):

```
# Deploy separate FastAPI microservice
from fastapi import FastAPI
from transformers import pipeline

app = FastAPI()
sentiment = pipeline("sentiment-analysis",
                     model="cardiffnlp/twitter-xlm-roberta-base-sentiment")

@app.get("/nepal-political-sentiment")
def get_sentiment():
    twitter = scrape_twitter("#Nepal #Politics")  # Your implementation
    reddit = scrape_reddit("r/Nepal")
    news = scrape_news(["kathmandupost.com", "nepalitimes.com"])

    return {
        'twitter_sentiment': analyze(twitter),
        'reddit_sentiment': analyze(reddit),
        'news_sentiment': analyze(news),
        'keywords': extract_keywords(twitter + reddit + news)
    }
```

QUIRK 6: MONSOON SEASONALITY (HYDRO-BANK)

Problem: Hydro-bank correlation varies with monsoon.

Math:

```
ρ_hydro,bank(t) = 0.45 + Δρ × Monsoon(t)

Monsoon(t) = {
    1.0  if June-Sept
    0.5  if May, Oct
    0.0  otherwise
}

Δρ = +0.25 (peak monsoon boost)
```

Code:

```
class NEPSESectorSeasonality:
    def __init__(self):
        self.base_corr = {('hydro','bank'): 0.45}
        self.seasonal_adj = {
            ('hydro','bank'): {'monsoon_peak': 0.25, 'monsoon_shoulder': 0.12}
        }

    def get_monsoon_factor(self, date):
        m = date.month
        if 6 <= m <= 9:
            return 1.0
        elif m in [5, 10]:
            return 0.5
        return 0.0

    def get_correlation(self, s1, s2, date):
        pair = tuple(sorted([s1, s2]))
        base = self.base_corr.get(pair, 0.30)

        if pair not in self.seasonal_adj:
            return base

        monsoon = self.get_monsoon_factor(date)
        adj_dict = self.seasonal_adj[pair]

        if monsoon >= 1.0:
            adj = adj_dict.get('monsoon_peak', 0)
        elif monsoon > 0:
            adj = adj_dict.get('monsoon_shoulder', 0)
        else:
            adj = 0

        return np.clip(base + adj, -1, 1)
```

QUIRK 7: TIMEZONE (UTC+5:45)

Problem: Nepal's :45 offset breaks many libraries.

Code:

```
from datetime import datetime, timezone, timedelta

# CORRECT
NEPAL_TZ = timezone(timedelta(hours=5, minutes=45), name='NPT')

def get_nepse_time():
    return datetime.now(NEPAL_TZ)

def to_nepse_time(utc_dt):
    if utc_dt.tzinfo is None:
        utc_dt = utc_dt.replace(tzinfo=timezone.utc)
    return utc_dt.astimezone(NEPAL_TZ)

# WRONG - Don't trust libraries to handle :45 correctly
# import pytz
# tz = pytz.timezone('Asia/Kathmandu')  # May round to :00 ✗
```

QUIRK 8: API RATE LIMITING

Problem: NEPSE APIs are SLOW (2-3 sec/call).

Code:

```
import time
from datetime import datetime, timedelta
from functools import wraps, lru_cache
```

```
class NEPSEAPIRateLimiter:
    def __init__(self, calls_per_min=10, min_interval=2):
        self.max_calls = calls_per_min
        self.min_interval = min_interval
        self.calls = []
        self.last_call = None

    def wait_if_needed(self):
        now = datetime.now()

        # Min interval check
        if self.last_call and (now - self.last_call).total_seconds() <
self.min_interval:
            time.sleep(self.min_interval - (now - self.last_call).total_seconds())

        # Rate limit check
        minute_ago = now - timedelta(minutes=1)
        recent = [c for c in self.calls if c > minute_ago]

        if len(recent) >= self.max_calls:
            wait = (min(recent) + timedelta(minutes=1) - now).total_seconds()
            if wait > 0:
                time.sleep(wait)

        self.calls = recent + [now]
        self.last_call = now

    def rate_limit(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            self.wait_if_needed()
            return func(*args, **kwargs)
        return wrapper

# Usage
limiter = NEPSEAPIRateLimiter()

@limiter.rate_limit
def get_market_data(symbol):
    # Your API call
    pass

# Cache aggressively
@lru_cache(maxsize=1000)
def cached_data(symbol, minute_bucket):
    return get_market_data(symbol)

def get_data(symbol):
    current_min = datetime.now().replace(second=0, microsecond=0)
    return cached_data(symbol, current_min)
```

COMPLETE INTEGRATION

```
def generate_nepse_signal(symbol, date, client_type='retail', broker='default'):
    """Complete signal with ALL quirks"""

    # Initialize
```

```
queue_model = NEPSERealisticQueueModel()
data_filter = NEPSEDataQualityFilter()
calendar_regime = CalendarAdjustedRegimeModel(Q_base)
political = NepalPoliticalSentimentTracker(api_endpoint)
broker_quality = BrokerQualityModel()
seasonality = NEPSESectorSeasonality()

# 1. Data quality
ownership = get_ownership(symbol)
quality = data_filter.validate_ownership(
    symbol, ownership['promoter_pct'], ownership['public_pct'],
    ownership['update_date'], date
)

if not quality['valid']:
    return {'action': 'NO_TRADE', 'reason': f"Bad data: {quality['flags']}"}

# 2. Base forecast
base = your_model.predict(symbol, date)

# 3. Political adjustment
political_stress = political.calculate_stress()
political_conf = political.get_confidence(political_stress)
political_pos_adj = political.get_position_adj(political_stress)

# 4. Regime adjustment
regime_stress_prob = your_regime_model.get_stress_prob(date)
regime_mult = political.get_regime_mult(political_stress)
Q, period, _ = calendar_regime.get_adjusted_transitions(date, regime_mult)

# 5. Edge calculation
edge = calc_edge(base['mu'], base['sigma'], symbol)
if edge <= 0:
    return {'action': 'NO_TRADE', 'reason': 'Negative edge'}

# 6. Position sizing
pos = position_sizer(base['mu'], base['sigma'], regime_stress_prob, symbol)
pos *= political_pos_adj
pos *= calendar_adjustment(date, period)

# 7. Fill simulation
circuit_prob = calc_circuit_prob(base['mu'], base['sigma'])
fill_sim = queue_model.simulate_fill(pos, circuit_prob > 0.3, client_type)
expected_fill, fill_prob = fill_sim

# 8. Broker quality
broker_rec = broker_quality.recommend_broker('limit')
adj_fill_prob = broker_quality.adjust_fill_prob(fill_prob, broker_rec)

# 9. Final decision
if adj_fill_prob < 0.3:
    return {
        'action': 'REDUCE',
        'reason': f'Low fill prob {adj_fill_prob:.1%}',
        'position': pos * 0.5
    }
    return {
        'action': 'TRADE',
        'symbol': symbol,
        'position': pos,
        'expected_fill': expected_fill,
        'fill_probability': adj_fill_prob,
        'edge_bps': edge * 10000,
        'confidence': base['confidence'] * political_conf * quality['confidence'],
        'broker': broker_rec,
        'warnings': get_warnings(circuit_prob, political_stress,
regime_stress_prob, quality)
    }

def get_warnings(circuit_prob, pol_stress, regime_stress, quality):
    w = []
    if circuit_prob > 0.4:
        w.append(f"HIGH_CIRCUIT: {circuit_prob:.1%}")
    if pol_stress > 1.3:
        w.append(f"POLITICAL: {pol_stress:.2f}")
    if regime_stress > 0.7:
        w.append(f"STRESS_REGIME: {regime_stress:.1%}")
    if quality['confidence'] < 0.6:
        w.append(f"DATA: {quality['flags']}")
    return w
```