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.