Volatility Prediction with Online Machine Learning

Ayush Singh

April 24, 2025

Abstract

This project develops a scalable and efficient framework for predicting realized volatility using high-frequency Limit Order Book (LOB) data from 127 stocks, each stored in an individual CSV file with $1-second$ resolution. The objective is to forecast realized volatility over configurable $10-minute$ windows by leveraging fine-grained microstructural features and applying online learning techniques.

Unlike traditional batch training methods, we implement an incremental training pipeline that processes each stock sequentially, making the system compatible with resource-constrained environments, such as a local MacBook. Central to this approach is robust feature engineering tailored to the LOB domain, capturing signals such as bid-ask spreads, order imbalances, and mid-price movements.

The model also addresses irregular data intervals and missing timestamps, ensuring consistent volatility estimation across varying trading activity. We benchmark online learning frameworks such as Vowpal Wabbit, River, and scikit-learn's partial_fit, selected for their lightweight memory footprint and adaptability to streaming data.

Performance is evaluated using metrics like Mean Squared Error (MSE) and Quasi-Likelihood (QLIKE) loss within a simulated online prediction protocol. This solution combines microstructural insight, real-time adaptability, and computational efficiency to advance short-term volatility forecasting in high-frequency financial markets.

Contents

1 Data Description and Target Calculation

Each stock's CSV contains a time-series of LOB snapshots divided into $10 - minute$ buckets (intervals). Within each $10 - minute$ interval, data is recorded at up to $1 - second$ frequency. The **order book** typically provides information like the best bid price, best ask price, and corresponding sizes (volumes), and possibly deeper levels of the book (e.g., second best bid/ask, etc.). In our case, we have "snapshots" rather than every single order event, meaning the data is sampled each second (if there was any change). Some seconds might be missing if no changes occurred; in those cases the order book state remains the same as the last update.

Realized Volatility Definition: Realized volatility for a time interval is a measure of the total variability of price within that interval. Formally, it can be defined as the square root of the sum of squared high-frequency returns over the interval. If we denote by P_t a price at time t (we will specify which price in the order book to use), then one common definition is:

$$\text{Realized Variance}_{[t_0,t_0+T]} = \sum_{t=t_0+1}^{t_0+T} (\log P_t - \log P_{t-1})^2$$

and **realized volatility** is the square root of this variance sum. In simpler terms, we take the log returns at each $1 - second$ step in the $10 - minute$ window, square them, sum them up, and then take a square root. This yields the volatility over that $10 - minute$ period.

- *Choice of Price for Returns:* In an order book, there is no single "traded price" every second unless a trade happens each second. A good proxy for the price is the mid-price (the average of best bid and best ask) or a **weighted average price (WAP)** that accounts for both bid and ask prices. We will use the mid-price or WAP as the price series P_t for computing returns. (The Kaggle Optiver volatility competition, for example, used the weighted average price of top-of-book quotes for return calculations.) Using $mid{-}price/WAP$ ensures we consider the information from both sides of the book even if no trade occurred.

- *Handling Missing Seconds:* When an interval has some seconds with no snapshot (no change), we assume the price stayed the same during those seconds. In practice, we can **forward-fill** the last seen bid and ask prices for missing timestamps so that our price series has a value for every second in the 600-second window. Forward-filling implies a zero return for those seconds (since if price didn't move, the log return is 0). This approach makes the realized volatility calculation accurate because if the market was inactive for a few seconds, it contributes zero to volatility, which is intuitive.

Flexibility in Volatility Window: While the default is a $10 - minute$ realized volatility, our pipeline can be designed to allow different window sizes. For instance, we could compute realized volatility over $5 - minute$ or $30 - minute$ windows by adjusting the interval length and number of returns summed. The formula remains the same - sum of squared log returns over the chosen horizon. We just need to parameterize the window length in the data processing stage. Similarly, one could compute realized volatility using different sampling frequencies (e.g., if we had data at $1 - second$, we could also experiment with $2 - second$ or $5 - second$ returns to see the effect).

Target Alignment (Prediction Task): We will train the model to predict the realized volatility for each $10 - minute$ interval. Typically, in a forecasting setup, features from one interval might be used to predict volatility in the next interval. For example, use order book data from *12:00-12:10* to predict volatility for *12:10-12:20*. This is a sensible approach because it avoids using future information. In our strategy, we can set it up such that for each time bucket (except perhaps the very first for each stock), the input features come from that bucket's LOB data and the target is the realized volatility computed over that same bucket if we assume a nowcasting approach or the next bucket if we assume a forecasting approach. *We will assume a forecasting approach* (as was done in the Optiver Kaggle competition), meaning the *features from a $10 - minute$ window are used to predict the realized volatility of the following $10 - minute$ window*. This way, the model is genuinely predicting future volatility. In implementation, this means when processing a stock's data sequentially, we will pair each interval's features with the next interval's realized volatility as the training target. (If needed, the framework could be adjusted to nowcast the same interval's volatility, but that would be an easier problem since one could directly compute it; forecasting is more useful.)

Data Consolidation: Because the data for each stock is in a separate file, an initial step is to iterate through each stock file and extract (or compute) the features and target for each $10 - minute$ interval. Each interval can be identified by a time index or an ID (in the Optiver data, a time_id was given to each interval). We will need to ensure data is sorted by time. We might create an index like (stock_id, time_id) to label each interval. The **training instances** will then be tuples of (features, realized volatility target). We do not need to load all data into memory at once; instead, we can stream through the files one by one (as described in the Online Learning section).

Handling Gaps at Boundaries: If there is any gap between intervals (e.g., a break between trading sessions or missing intervals for a stock), we should be cautious if using previous interval's volatility as a feature. In general, each $10-minute$ bucket is treated independently for feature extraction. Missing seconds within a bucket are handled by forward-filling as mentioned. If an entire bucket has no trading activity (unlikely in active stocks, but possible in illiquid stocks), its realized volatility would be zero; we should still include it to train the model that sometimes volatility can be zero (and learn what features correspond to that, likely large spreads and no trades, etc.).

2 Feature Engineering for LOB Microstructure

Feature engineering is arguably the most important part of this strategy. We need to distill each $10-minute$ interval of high-frequency LOB data (which could be up to 600 data points if we have one snapshot per second) into a set of informative features that help predict the upcoming volatility. The features will leverage well-known concepts from market microstructure:

2.1 Price-Based Features

- **Mid-Price and Returns:** The mid-price is defined as $(\text{best bid price} + \text{best ask price})/2$. Its log returns over short horizons reflect price movement. We can include features like the last mid-price return in the interval (i.e. $\log(P_{t_0+T}/P_{t_0+T-1})$ for the end of the interval) and summary statistics of mid-price returns over the interval (mean, standard deviation, etc.). A high standard deviation of returns in the current interval often indicates high volatility continuing into the next. We do not include the realized volatility of the interval as a feature (that would directly leak the target if we were predicting the same interval's vol), but we can include partial volatility measures from sub-periods (see volatility proxies below).

- **Bid-Ask Spread:** The *spread* = $ask\_price1 - bid\_price1$ is a fundamental measure of market liquidity. We expect that larger spreads indicate lower liquidity and often higher subsequent volatility (prices can jump more when spread is wide). We will use both the absolute spread and the relative spread (spread divided by mid-price) as features. Relative spread normalizes for price level (so that a $\$0.05$ spread on a $\$10$ stock (0.5%) is more significant than $\$0.05$ on a $\$100$ stock (0.05%)). The spread can be summarized over the $10 - minute$ window: e.g. average spread, max spread, and perhaps how the spread at the end of the interval compares to the beginning (to see if liquidity is worsening or improving).

- **Weighted Average Price (WAP):** A refined measure of price is the micro-price, which weighs the bid and ask by their quantities:

$$\text{microprice} = \frac{bid\_price1 \times ask\_size1 + ask\_price1 \times bid\_size1}{bid\_size1 + ask\_size1}$$

  This gives a price closer to whichever side has more volume, anticipating where the next trade might execute. We can compute the microprice each second and treat its changes similarly to mid-price. The WAP (volume-weighted average price) mentioned in some references is similar; since our data is just top-of-book, the microprice is essentially a volume-weighted mid-price. We can include *log returns of WAP* or microprice over the interval as features.

- **Price Trend:** Features capturing short-term trends can be useful. For instance, price change over the interval (log of last mid-price minus first mid-price in the $10 - min$ window) tells us the direction and magnitude of price movement in that interval. While the absolute movement doesn't directly equal volatility, large moves might signal volatility. Additionally, *number of price up-ticks vs down-ticks* (how many times did the mid-price increase or decrease from one second to the next) could indicate the directionality and choppiness of the market. A balanced up/down count with frequent changes could indicate high churn (volatility). If the data provides price at *multiple levels* (e.g., second best bid/ask), we might measure if the best prices are moving closer to second level (an indication of pressure).

2.2 Order Book Depth and Imbalance Features

- **Order Imbalance (Top Level):** Order imbalance measures the difference in buying vs selling pressure. A simple version at the top of the book is:

$$\text{Imbalance} = \frac{bid\_size1 - ask\_size1}{bid\_size1 + ask\_size1}$$

  which lies in $[-1, 1]$. An imbalance close to 1 means far more buy volume at the top than sell volume (potential upward pressure), while -1 means heavy sell pressure. Large absolute imbalance could predict price moves (and thus volatility). We will use features like the average imbalance over the interval and perhaps the imbalance at the end of interval. We might also include an indicator if the imbalance ever crosses a certain threshold, to capture abrupt dominance of one side.

- **Depth (Volume) Features:** "Depth" refers to the total volume available in the book. If our data contains multiple levels (say top 10 bid and ask prices, as hinted by some references), we can compute:

  - **Total Bid Volume:** sum of sizes of orders on the bid side (for top 10 levels, or however many provided).
  - **Total Ask Volume:** sum of sizes on the ask side.
  - These give a sense of liquidity. We might use total volume $(bid + ask)$ as a measure of overall liquidity, and the **volume imbalance** $=$ $(bid\_vol - ask\_vol)$ as a broader measure of pressure.

  In a study, LOB depth was found to be very informative for volatility forecasting - more so than the slope of the order book. Depth essentially captures how much interest (in shares/contracts) is sitting in the book; low depth can mean the price will move more for a given trade, increasing volatility. We will include features like average total depth during the interval and minimum depth observed (a sudden drop in liquidity could lead to higher volatility). If depth is consistently high, perhaps volatility will be lower as the order book can absorb trades.

- **Bid/Ask Spread of Second Level:** If available, features like how far the second-best bid is from the best bid (and similarly for asks) can indicate *order book slope* - a steep slope (big difference between best and second-best) might mean thin book, whereas small difference (many orders close in price) means a thick book. However, as noted, prior research suggests the absolute depth is more predictive than slope. We might still include one feature for slope: e.g., difference between best and fifth-best price on each side, to quantify steepness.

- **Volatility of Order Book Quotes:** This means how much do the bid and ask quotes themselves fluctuate within the interval. We can measure the standard deviation of the best bid price and the best ask price in that $10 - minute$ window. Large quote oscillations could be a precursor to larger price swings (volatility).

- **Trade Volume and Count (if available):** Although not explicitly mentioned, if we had a companion trade data (number of trades, volume traded in that interval), those could be strong features (e.g., high trading volume often accompanies high volatility). The problem statement focuses on LOB snapshots, so we may not have raw trade info, but large changes in order book volumes can act as a proxy. If an interval had many shares executed, likely the order book sizes will reflect orders being taken out.

Feature Name	**Description**	**Calculation (for interval)**
Mid-Price Return	Log return of mid-price from start to end of interval (captures overall price change).	$\log(\text{mid}_{\text{end}}/\text{mid}_{\text{start}})$
Volatility (current)	Std. dev or RMS of 1-sec mid-price returns within the interval (a volatility proxy for that interval).	$\sqrt{\frac{1}{N}\sum(\Delta \log \text{mid})^2}$
Bid-Ask Spread	Average difference between best ask and best bid. Also track max/min spread.	$\frac{1}{N}\sum(\text{ask}_1 - \text{bid}_1)$
Relative Spread	Spread as a fraction of mid-price (dimensionless).	$\frac{\text{ask}_1-\text{bid}_1}{\text{mid}}$ (averaged)
Order Imbalance	Difference in top-level buy vs sell volume.	$\frac{\text{bid\_size}_1-\text{ask\_size}_1}{\text{bid\_size}_1+\text{ask\_size}_1}$
Total Depth	Total volume on both sides (liquidity) available at top k levels (e.g., $k = 10$).	$\text{depth}_{\text{bid},k} + \text{depth}_{\text{ask},k}$
Volume Imbalance	Difference between total bid and ask depth.	$\text{depth}_{\text{bid},k} - \text{depth}_{\text{ask},k}$
Microprice (WAP)	Volume-weighted best price (anticipates trade price).	$\frac{\text{bid}_1\cdot\text{ask\_size}_1+\text{ask}_1\cdot\text{bid\_size}_1}{\text{bid\_size}_1+\text{ask\_size}_1}$
Microprice Return	Log return of microprice from start to end (or last change).	$\log(\text{microprice}_{\text{end}}/\text{microprice}_{\text{start}})$
Quote Std (Bid/Ask)	Volatility of the best bid and ask quotes themselves.	Std. deviation of bid_1 or ask_1 over interval
#Price Updates	Count of times the best bid or ask price changed.	E.g., #bid quote changes, #ask quote changes
Trend Indicator	Direction of price movement at end vs beginning.	$\text{sign}(\text{mid}_{\text{end}} - \text{mid}_{\text{start}})$ or categorical (up, down, no change)
Previous Volatility	[Cross-interval] Realized volatility in the previous interval (volatility tends to cluster).	Same as volatility metric but for prior interval

Table 1: Feature Engineering for LOB-Based Volatility Modeling

We will generate a rich set of features like the above for each interval. Some features are instantaneous (like end-of-interval imbalance), while others are summary statistics over the interval (like average spread, std of returns). We can also augment the feature set by splitting the $10 - minute$ interval into sub-periods to capture recent changes more finely. For example, one proven approach is to calculate features on overlapping sub-windows of the interval. For a $10 - minute$ ($600s$) interval, we could compute certain features on the last 2.5 minutes, last 5 minutes, last 7.5 minutes, etc., in addition to the full 10 minutes. This gives the model information about how conditions are evolving within the interval:

- For instance, **volatility in the last** 2 **minutes vs earlier:** if volatility is ramping up toward the end of the interval, the next interval might continue to be volatile.
- **Imbalance shift:** imbalance averaged over the last portion vs the earlier portion, indicating if buy/sell pressure is rising or fading.
- **Trend reversal:** perhaps the mid-price moved up in first half and down in second half (volatile behavior) vs consistently one direction.

Using overlapping windows, we can create features for multiple horizons ($0 - 600s, 150 - 600s, 300 - 600s, 450 - 600s$ as in one reference). This yielded hundreds of derived features in the cited approach. In our case, we should be mindful of not overloading the model with too many features given memory/time constraints, but a similar idea can be applied in moderation.

All features will need proper **preprocessing/normalization:**

- **Scaling:** Many raw features vary greatly in scale across stocks. For example, stock prices range widely (a penny stock vs a $\$1000$ stock) which affects spreads and volumes. It's important to normalize features to be dimensionless or on comparable scales. We already use ratios or log returns for many features (which are naturally scaled). For features like raw volume, we could take logarithm or normalize by a typical volume for that stock.
- **Stock-specific normalization:** One approach is to calculate per-stock means/std for features (from training data) and normalize each stock's features by its historical stats. However, in an online setting, we might not want to precompute over all data. Instead, we can normalize on the fly using running

statistics. Another simpler approach is to incorporate the stock ID as a feature (possibly one-hot encoded or as an embedding in certain models) so that the model can learn stock-specific offsets or scale differences.

- **Handling missing values:** After forward-filling missing seconds, our features should not encounter *NaNs* for price-related features. If a stock had zero depth on one side (e.g., no asks - which is rare in normal trading because market makers always quote, but could happen briefly), imbalance formula might have $0/0$; we would need to define that as 0 imbalance or skip that second. In practice, we can add a tiny epsilon in denominators to avoid division by zero. We will also fill any missing trade-related fields as 0 (e.g., if no trade occurred in the interval for volume features).

- **Outlier clipping:** High-frequency data can have glitches or extreme outliers (bad ticks). We should consider capping extreme feature values (winsorizing). For example, if an order imbalance is 0.999 (almost all on one side) consistently, or if a spread jumps abnormally due to a data error. Clipping to reasonable ranges can make training more stable.

By engineering these features, we transform each $10 - minute$ block of order book data into a fixed-length feature vector. This dramatically reduces data size (from potentially 600 rows of raw data to one consolidated row of features) and highlights the aspects most predictive of volatility. The focus on microstructural features (spreads, depths, imbalances) is motivated by the fact that order book conditions often lead price volatility. For example, a severely imbalanced book may precede a price jump, and a wide spread with low depth may precede large price moves due to lack of liquidity.

3 Online Learning Approach

With the features defined, we turn to the strategy for model training. We adopt an online learning approach, meaning the model will be updated incrementally as data streams in, rather than training in one big batch. This approach has several benefits for our scenario:

- **Memory Efficiency:** We never need to hold all 127 stock datasets in RAM at once. We can process one stock (or even one interval) at a time, generate features, update the model, and then discard that data before moving on. This keeps memory usage low, well within our RAM.

- **Adaptability:** Online learning naturally allows the model to adapt to new data. If the statistical properties of volatility change over time or differ across stocks, the model can gradually adjust as it sees more instances.
- **Speed:** Incremental updates can be faster than re-fitting a model from scratch for each new data point. Many online algorithms update in time proportional to the number of features, which is manageable. Our feature vector length might be on the order of tens or low-hundreds of features; updating a linear model or tree model with that many features is quite fast.

Data Streaming Setup: We will iterate through each stock's CSV file one by one. For each stock file:

- Read the data in chunks (we could read the whole file if it's not huge, but assume some stocks could have a lot of intervals, so streaming is safer).
- If needed, sort by time (the data likely is pre-sorted by timestamp within each file).
- Identify the distinct $10 - minute$ intervals (time buckets). In the Optiver data, each row had a *time_id* indicating the interval. If not explicitly given, we might infer intervals by grouping timestamps into $10 - minute$ blocks.
- For each interval in this stock:
  - Extract the segment of data corresponding to that interval (all rows with that time_id or within that $10 - min$ window).
  - Forward-fill to a complete time series of length 600 (or appropriate length if window size differs) to ensure we account for every second.
  - Compute all the features for this interval.
  - Also, compute the realized volatility target for the next interval (if available). If we are at the last interval of a stock and no next interval, we might skip that one for training (since it has no target), or if doing same-interval prediction (not our assumption) we compute it directly. Assuming forecasting, the current interval's features correspond to next interval's volatility. Thus, we might want to actually compute realized volatility of the current interval and use it as target for the previous interval's features. In implementation, one can compute realized vol for every interval and then when training, use

    $(features\_interval\_i, target = volatility\_interval\_i)$ pair but note that those features actually came from interval i (which in forecasting scenario are features of previous interval). To avoid confusion: we can simply shift the data by one interval when forming training examples. For example, buffer the last interval's features until we have computed the next interval's volatility, then emit the training example.

  - Feed the features and target into the model for training (or evaluation as needed, see Evaluation section).

- After finishing all intervals of a stock, move to the next stock file and repeat.

Because each stock is processed sequentially, we should be cautious about stock-specific differences. A single model will see data from different stocks one after another. There is a potential issue of the model "forgetting" patterns from earlier stocks when it sees new stocks, if their characteristics differ a lot. To mitigate this, we can do a couple of things:

- **Shuffle training order:** Instead of training all intervals of stock A, then all of B, etc., we could interweave them. For example, process one interval of each stock in a round-robin fashion sorted by actual time if timestamps are comparable. However, syncing 127 stocks by time is complex (they might not cover the same dates). A simpler method is randomizing the order of intervals globally. But since data is on disk per stock, that would require loading all into a single list first which is memory heavy. So this might not be feasible.

- **Use stock ID as a feature:** Include a categorical feature for stock identifier. An online model like a linear model can handle a high-cardinality categorical by one-hot encoding or embedding (Vowpal Wabbit, for example, can naturally handle categorical features by hashing them into the feature space). This allows the model to learn stock-specific biases. For instance, some stocks might have persistently higher volatility; the model can learn a weight associated with that stock's ID to adjust predictions. It can also interact stock ID with other features if using nonlinear models or manually creating interaction features (e.g., "stock X typically has wider spreads").

- **Separate models per stock:** This is another approach - train 127 distinct models, one for each stock. This could potentially yield better accuracy per stock (as each model fine-tunes to that stock's patterns) but it's less scalable and loses the benefit of cross-learning (plus maintaining 127 models is cumbersome). Given our resources, a single model (with stock as feature) or a small number of clustered models is preferable.

We will prefer the single-model-with-stock-feature approach for scalability, unless empirical results suggest otherwise.

3.1 Model Selection for Online Learning

Not all machine learning models can be trained online easily. We will consider a few options that strike a balance between complexity and efficiency:

- **Online Linear Models:** A linear regression model (with possibly some polynomial or interaction features added manually) can be trained online using stochastic gradient descent. Frameworks:
  - **Scikit-learn:** SGDRegressor supports incremental training via *.partial_fit()*. We can use a squared loss (for MSE minimization) or even a Huber loss to be robust to outliers. We can also try *PassiveAggressiveRegressor*, which is an online algorithm that adjusts weights aggressively when it makes errors. These linear models are very fast and memory-friendly.
  - **Vowpal Wabbit (VW):** Vowpal Wabbit is a highly optimized library for online learning, particularly efficient for linear or logistic models with large feature sets. We could feed our data to VW in its text input format (including the stock ID as a namespace or feature) and let it run online training. VW can easily handle millions of examples on a laptop. It also has built-in options for handling interactions and learning rates. For example, we might use a command like: *vw -l 0.5 –loss_function squared* for a regression with squared loss, and stream in data.
  - **River:** River is a Python library specifically for streaming machine learning. It provides a *LinearRegression* (SGD-based) as well as many other models. An advantage of River is that it allows building a pipeline, e.g., include a feature scaler and the model in one, and it can track metrics as data flows. Using River's *linear_model.LinearRegression* or *SGDRegressor* equivalent, we can update model weights with *.learn_one(x,*

y) for each instance. River also has optimizers and regularization we can tune.

- **Tree-Based Models:** Sometimes linear models may not capture all patterns (especially if relationships are nonlinear). For volatility, interactions like "if spread is high AND imbalance is high, then volatility goes up disproportionately" might exist. Online tree models can capture such interactions:
  - **Hoeffding Tree Regressor (in River):** This is an incremental decision tree that grows based on the Hoeffding bound, suitable for streaming data. Over time it can approximate a normal regression tree. It can handle numeric and categorical features (stock ID can be treated as categorical). A single tree might be a bit unstable, but River also offers ensemble methods.
  - **Adaptive Random Forest (ARF) in River:** This is an ensemble of Hoeffding Trees with mechanisms to adapt to concept drift. It tends to achieve accuracy closer to batch random forests. The trade-off is that it's more computationally intensive and uses more memory (multiple trees). However, we could start with a modest number of trees (e.g., 10 trees) and shallow depth to keep resource usage within our limits. ARF can improve accuracy if the relationship between features and volatility is complex.
  - **Incremental Gradient Boosting:** Traditional gradient boosting (XGBoost, LightGBM) doesn't inherently support true online learning, though XGBoost has a *warm_start* or iterative training mode but it still requires batch updates. There's an experimental approach to use River in combination with LightGBM via feeding mini-batches, but it's complicated. We likely will avoid relying on batch GBMs due to memory constraint and complexity.
- **Neural Networks:** In theory, a neural network could be trained online by updating weights for each new batch of data (this is stochastic gradient descent). Frameworks like TensorFlow or PyTorch can be used in an online loop. However, building a neural net might be overkill and slower on CPU for this task, given that simpler models often perform quite well for volatility forecasting (and given limited data per interval). A small neural net (say 1 hidden layer) could be tried if nonlinearity is needed, but we must be

careful with forgetting (we might need a very small learning rate or replay buffer to not forget earlier data).

  - scikit-learn's MLPRegressor unfortunately does not support *partial_fit* for regression (only for classification via MLPClassifier). So we'd have to implement our own loop in a deep learning library if we go this route.

We start with a regularized linear model (ridge regression via SGD) as a baseline. This model is fast and will give us a sense of performance. We can then consider a nonlinear upgrade if needed, such as an adaptive random forest, which many streaming ML practitioners find effective for structured data. The linear model can also be enhanced by manually creating some interaction features (for example: $spread \times imbalance, ordepth \times volatilityproxy$) if we suspect nonlinear effects but want to keep the model linear in parameters.

Online Update Procedure: For whichever model we choose, the update step in pseudo-code will look like:

```
model = initialize_model()

for stock in stocks:
    for interval in stock.intervals:  # sequentially or however we iterate
        x = features(interval)
        y = realized_vol_target(interval)  # next interval's vol if forecasting
        y_pred = model.predict(x)  # (optional: to evaluate pre-update prediction)
        model.partial_fit(x, y)    # update model with this new data point
```

If using River, it would be *model.learn_one(x, y)*. If using VW, we would prepare a line with features and pipe it to VW which updates internally.

This one-pass online training will effectively treat the data stream as i.i.d. for the learner, but as discussed, volatility data has a time component (autocorrelation). We are capturing some of that via features (like previous interval volatility) rather than via model state, since each data point is an independent example to the model (no recurrence in the model itself, unless we used a recurrent neural net which is beyond scope).

Performance Considerations:

- The model itself (say a linear model with 100 features) is tiny (a few hundred coefficients). Even an ARF with 10 trees of depth maybe $5-6$ will be in the low MBs. Processing one interval at a time means we only hold at most 600 rows of raw data in memory plus some overhead for features.
- Using vectorized NumPy/Pandas for feature calc will make it fast. The model updates (SGD or tree splits) are the main overhead but those are designed to be fast. Vowpal Wabbit, if used, is written in C++ and extremely fast for linear models (processing millions of instances per second in some cases). River's Python implementations are efficient (some parts in C) but might be slower than VW; still, given maybe on the order of tens of thousands of intervals total (127 stocks * maybe dozens/hundreds of intervals each), it's easily manageable.
- We might consider writing critical loops (like feature computation per interval) in NumPy or using tools like Numba for any custom loops to ensure speed.

Finally, after training online, we can **persist the model** (save the weights) so that it can be used for inference on new data (e.g., tomorrow's intervals). With a library like River, we can pickle the model object. With VW, we can save the model to a file after training (*-f model_filename*).

4 Evaluation Protocol

Evaluating a volatility prediction model requires careful consideration of both accuracy and the practical usefulness of predictions. We will use multiple metrics to assess performance, each capturing a different aspect:

- **Mean Squared Error (MSE):** This measures the average squared difference between predicted volatility and actual realized volatility. It's a standard regression metric. We might also take the square root to get RMSE which is in the same units as volatility. Lower MSE/RMSE is better. MSE heavily penalizes large errors (which could be important if we occasionally miss a volatility spike by a wide margin).

- QLIKE (Quasi-Likelihood) Loss: QLIKE is a specialized loss function often used in volatility forecasting evaluation. It is defined for each forecast-observation pair as:

$$\text{QLIKE}(\hat{\sigma}, \sigma) = \frac{\sigma^2}{\hat{\sigma}^2} - \ln \frac{\sigma^2}{\hat{\sigma}^2} - 1$$

  where $\hat{\sigma}$ is the predicted volatility and σ is the realized volatility (so σ^2 and $\hat{\sigma}^2$ are the variances). If the prediction equals actual ($\hat{\sigma} = \sigma$), QLIKE = 0 (ideal). QLIKE strongly penalizes under-prediction of volatility: if actual σ is much higher than forecast $\hat{\sigma}$, the $\sigma^2/\hat{\sigma}^2$ term blows up, yielding a large loss. It also penalizes over-prediction, but somewhat less severely for the same relative error. This asymmetry is useful because in risk management, underestimating volatility is more problematic than overestimating. We will compute the average QLIKE across all predicted intervals. Many volatility forecasting studies consider QLIKE alongside MSE as complementary metrics

- **Directional Accuracy / Classification Metrics:** Although volatility is a continuous variable, we might be interested in some qualitative accuracy. For example, we can measure accuracy in predicting volatility regimes – perhaps classify each interval as "high volatility" vs "low volatility" based on whether it's above or below some threshold (such as the median or a fixed value). Then we can see what fraction of intervals we correctly classified the regime (this would be a standard accuracy metric). Another possibility is to measure sign accuracy of change: did we correctly predict whether volatility will increase or decrease compared to the previous interval? This can be framed as a binary classification of volatility change direction. These classification-oriented evaluations are secondary, but they give insight into whether the model captures the trend in volatility, not just the level.

- **Mean Absolute Error (MAE):** MAE is the average of absolute errors. It's more interpretable in terms of actual volatility points. Sometimes an MAE or MAPE (Mean Absolute Percentage Error) is reported to gauge typical error magnitude. Volatility values might range, say, from 0 to some number (depending on asset; realized vol of a stock over 10min could be, for example, 0.002 as a standard deviation in log returns – it's not a large number as an absolute value). MAPE would require non-zero vols (if there are zero-vol intervals, MAPE could be problematic).

- **R-squared or Explained Variance:** We can compute an R^2 by comparing our predictions to a naive model. For instance, a constant model that always predicts the mean volatility, or a naive last interval model that predicts next volatility equals last realized volatility. $R^2 = 1 - \frac{\sum(\hat{\sigma}-\sigma)^2}{\sum(\sigma_{\text{naive}}-\sigma)^2}$. This tells us how much variance in σ we explain relative to a baseline. It's common in academic papers to report R^2 for volatility forecasts. However, due to the low signal-to-noise in short-term volatility, R^2 might be quite low (it's hard to predict perfectly).

Given the online nature of training, we will also do online evaluation (also called prequential evaluation). This means: as we process each new interval, we first generate a prediction (using the current model state before updating with the current data), then we observe the actual volatility and update the model. We record the error for that prediction. This simulates how the model would perform in a real-time setting, always predicting the next interval then learning from it. Using prequential evaluation ensures we're always evaluating on unseen data (at that moment in time) and it naturally averages performance over the whole dataset in a sequential manner.

Concretely:

- We initialize the model (maybe even with a short "burn-in" training on a small sample to get reasonable initial weights or we can start cold).
- For each new interval (features x, actual volatility y):
  - Compute $\hat{y} = \text{model.predict}(x)$ using the model before it sees y.
  - Record the error metrics comparing $\hat{y}$ to y.
  - Feed (x, y) to the model to update it (model learns from this new data).
- Continue this through all data.

At the end, we will have a series of prediction errors for each interval. We can then compute overall metrics:

- MSE = average of $(\hat{y} - y)^2$ over all predictions.
- QLIKE = average of the QLIKE formula over all predictions.
- Etc.

If we incorporate the stock ID as a feature, the model's first few predictions for a new stock (that it hasn't seen before) might be poor (since it doesn't know that stock's characteristics yet). This will show up in the error. This is acceptable in evaluation, but it's something to note (the model might have a "cold start" issue for new stocks). In our case, we do have 127 stocks, presumably all present in training, so in a real scenario we wouldn't be applying to totally new stocks; thus we could even omit the first few intervals of each stock from the evaluation to focus on steady-state performance (or include them to be realistic about adaptation).

We will also evaluate performance per stock and per time period:

- It's useful to see if the model is consistently good or bad for certain stocks. Perhaps it does well on very liquid stocks but struggles on illiquid ones (where volatility might be more sporadic). We can compute metrics for each stock's predictions.
- We can also see if performance drifts over time. If our data covers multiple days, we might see that the model improves after seeing more data, or that during certain market conditions (e.g., a crisis day) errors spike. Plotting the predictions vs actual over time could reveal such patterns.

Benchmarking: We should compare our model against simple benchmarks:

- Historical Mean Volatility: Use the average realized volatility of previous intervals (or previous day) as the prediction for the next. This is a very naive model but sets a baseline.
- Previous Interval's Volatility (Volatility Clustering): A common strong baseline is to predict that next interval's vol = last interval's vol. Volatility is known to cluster, so this could be hard to beat for very short horizons. Our model includes previous vol as a feature, effectively it should learn to mimic this baseline plus adjust for other signals. We can measure the MSE of this baseline to see how much improvement we get.
- Simple Regression on Spread or Volume: To illustrate the value of our complex feature set, we could evaluate a model that uses just one feature, say the end-of-interval spread or the trade volume, to predict vol. It will likely be worse than our full model, showing that combining multiple microstructure indicators is beneficial.

Metrics Interpretation: In volatility forecasting, it's common that R^2 values are low (like 0.1 or 0.2) because volatility is very noisy. A better gauge is whether our model's predictions, when ranked, correlate well with actual volatility (spearman rank correlation) or if it can identify the top X% volatile intervals accurately. If one of the goals is risk management, we might care about how often we correctly predict the highest volatility events. We can measure something like precision/recall for the top quartile of volatility. These are more specialized evaluations, but worth mentioning if the use-case is to catch extreme volatility.

All metrics will be calculated on the hold-out or test set. Since we are doing online evaluation on the training stream, one might argue we're evaluating on training data. However, because it's sequential and we always predict before seeing the true value, it's effectively a test on each new point. There is no data leak in prequential eval as long as we don't train on future data before predicting the present. To further validate, we could do a true backtest: e.g., train the model on the first N-1 intervals of each stock, then freeze it and predict on the Nth interval (the last 10 minutes of each stock on a given day) and measure error. Or train on first day, test on second day. If data is dated, a sensible split is train on earlier dates, test on later dates to simulate forward-in-time prediction.

5 Conclusion

In summary, the proposed training strategy processes each stock's LOB data sequentially, computing a rich set of features that capture the state and dynamics of the order book, and uses an online learning algorithm to predict 10-minute realized volatility. By leveraging microstructure insights (spreads, order imbalance, price movement) and updating the model incrementally, we create a system that can learn from thousands of intervals of data without exceeding local computational resources. The flexibility of the design allows adjusting the volatility window or incorporating additional features with minimal changes.

We have emphasized proper handling of the intricacies of high-frequency data (irregular timestamps and noisy behavior) and provided an evaluation framework that uses appropriate metrics for volatility forecasts (emphasizing not just average error but also the penalty for under-predicting volatility).

This strategy should yield a solid model that improves upon naive benchmarks and provides traders or risk managers with a reliable short-term volatility prediction. As a next step, one could further tune hyperparameters (learning rates, regularization strength, tree depth, etc.) on a validation set and test the model in a live simulation. Additionally, monitoring the model's performance over time would be important – if the market changes (e.g., a volatility regime shift), the online learning setup will help the model adapt, but recalibration or feature adjustments might be needed. Overall, this approach offers a practical and scalable solution to forecasting realized volatility from LOB data in an online fashion.