Université Claude Bernard Lyon 1

Data Processing and Analytics (DPA)

Analyzing Stock Market Values

Authors: Lovato Edoardo, Markhovski Julia, Piccoli Leonardo Arduino

Academic Year 2025–2026

Contents

1 Introduction

This report presents a real-time analysis of stock market values using a streaming-based pipeline composed of Kafka and Apache Spark Structured Streaming. The goal is to simulate the continuous flow of market information and to compute several financial parameters within fixed time windows.

The analysis is divided into six tasks, each corresponding to a financial indicator commonly used in market analysis. These tasks include **variance** computation, **price evolution** analysis, measuring frequency of **trades**, **volume** estimation, design of **stop-loss** and **take-profit** mechanisms, and finally the building and monitoring of a virtual investment **wallet**.

Project Structure Note

All tasks are provided in separate Jupyter notebooks because they each produce console output and cannot be run together in a single notebook.

2 Development Environment

2.1 Dataset

The dataset used in this project is provided in a CSV file named `stock.csv`. It is a synthetic dataset written according to the OHLC (Open High Low Close) standard, a common format that describes the movement of price of a stock over time. Every line correspond to the variation of the cost of a single stock within a specific time window.
The schema of the dataset is defined as:
`{true_timestamp, open_price, high_price, low_price, close_price, n_action, name}` where

- `true_timestamp`: is considered the starting instant of the time window;
- `open_price`: price of a single share of the stock at the beginning of the time window;
- `high_price`: maximum price reached within the window;
- `low_price`: minimum price reached within the window;
- `close_price`: the price of a single share of the stock at the closing of the time window;
- `n_actions`: total number of shares traded during that interval;
- `name`: unique identifier of the stock.

Additional considerations regarding these values will be discussed later.

2.2 Kafka Streaming Producer

The analysis is conducted through a streaming architecture where Kafka serves as the message producer and Spark Structured Streaming performs the real-time computations.
Each row in the CSV is ingested by Kafka, treated as a independent and atomic transaction event (even though it originally represented an entire time window) and returned as an output of a streaming. Spark then groups the rows into sliding windows of fixed duration.
The Kafka producer script ingests one line every 0.5 seconds, extracting only the second column as the invariant `price` and the last as the stock's `name`. It also attaches a synthetic `timestamp` corresponding to the actual moment of ingestion on the user's machine.

About the given code A mismatch has been found between the specification and the given Kafka producer implementation: the assignment states that the price is in column 2 (index 1), while the Kafka producer reads column 3 (index 2), line 26. The value at index 2 is still a valid price, so this part of the code has not been modified to align with the specification.

3 Methodology and Assumptions

Although the dataset provided for the assignment is synthetic rather than collected from an actual stock exchange, the analytical framework reproduces the logic and constraints of real market systems.

3.1 Timestamps and Definition of Day

Since the real timestamps from the original file are substituted by the Kafka producer with synthetic ones at ingestion time, corresponding to the actual time of the stream generation, it is necessary to clarify how time is interpreted within this simulated environment.

The project operates on five-minute windows, which are treated as the equivalent of a single trading "day" within this simulation environment, accordingly "1 hour" is scaled down to 12.5 seconds to follow the proportional reduction.
Moreover a watermark of 10 minutes is applied to ensure the stability of a "daily" window (or less for smaller window sizes).

Shorter daily time windows have been show not to return any meaningful results. This is due to the fact that the database do not represent a real streaming but it is just an toy example to work on, as such, it lacks the completeness to perform analyses over realistic 24-hours periods.
A key adjustment made was setting the Kafka producer timer interval to 0.01 seconds, instead of the default 0.5 seconds, to generate a higher frequency of events and simulate a more active market stream.

Under these assumptions the synthetic time generated by the producer will not influence the result of the queries.

3.2 Transactions

A transaction contains only the `name` of the stock and its `price` (not counting for the `timestamp` which is ignored), but the Kafka producer code has been modified to also return the sixth column of the dataset, `n_shares`, representing the number of shares exchanged in a single transaction, a data that will be needed for the subsequent queries.

Furthermore the `price` variable returned in the streaming is the only value of the stock considered for that transaction (as specified in the assignment), this decision will also influence the later queries.
In fact in the original dataset the price was time-variant given that it was analyzed in a time period (initial, maximum, minimum and final value); in contrast this system considers the only price value taken as constant inside that transaction.

Therefore, all the other metrics can be derived from these three fields:

- the **variance** of a stock is computed from the sequence of its `price` values inside the "daily" window;
- the **evolution** of a stock is measured by the difference between the maximum and minimum `price` in the window;
- the **most traded** stock is the one with the highest sum of `n_shares` over a window;
- the **trading volume** is defined as the average `price` observed for the stock multiplied by the sum of `n_shares`, within a window.

4 Task 1: Stocks with the Smallest Daily Variance

4.1 Definition

The first analysis focuses on identifying the stocks that exhibit the smallest price variance within each simulated trading day. Variance is a statistical measure of volatility, capturing how much the price of a stock fluctuates around its mean value.

In the context of this project, the variance is computed over all price values associated with a stock within the 'daily' window generated by the streaming pipeline. Since each transaction carries a single constant price value the formula is calculated using the classical statistical definition:

$$\mathrm{Var}(P, S) = \frac{1}{n}\sum_{i=1}^{n}(P_{S,i} - \bar{P_S})^2$$

where $P_{S,i}$ is the `price` recorded in the i^{th} transaction, $\bar{P_S}$ is the mean `price` of the stock in the entire window, and n is the total number of transactions for stock S.

4.2 Implementation and Setup

The streaming application was developed using Spark Structured Streaming to consume stock price data from a Kafka topic. Data is ingested as JSON strings and parsed using a pre-defined schema to extract fields such as stock `(name)`, `(price)`, `(number of shares)`, and `(timestamp)`. The timestamp string is converted into Spark's `timestamp` type to enable time windowed aggregations.

Additionally, a 'day' in this simulation corresponds to 5 minutes of real time. This adjustment provides a more granular and timely analysis window.

Choice of Window Type This task uses **tumbling windows**, which are fixed-size, non-overlapping, and contiguous time intervals. Each transaction belongs to exactly one window, simplifying aggregation without overlapping data. Tumbling windows are suitable here to distinctly separate simulated 'daily' periods where variance and averages are computed clearly per window.
Sliding windows (which allow overlapping intervals) could provide more continuous updates but add complexity and duplicate counts. Given the project's aim to show stable daily variance snapshots, tumbling windows effectively balance simplicity and meaningful analytics.

Choice of Output Mode The streaming query uses the `complete` output mode, which outputs the entire updated aggregation result to the console at every trigger. This mode was chosen because it enables tracking the full set of computed aggregates (variances, averages) across all time windows and stocks continuously.
While `complete` mode can be more resource-intensive than alternatives, it provides a comprehensive view of streaming data evolution that is valuable for analysis and debugging. Other modes such as `append` or `update` would output only new or changed rows, which may omit valuable historical context needed for deep insight over the simulated 'days'.

4.3 Interpretation

A low variance indicates stable and predictable behavior, suggesting that the stock experiences minimal price oscillations during the observed window, while a high variance implies more volatile price activity. These results provide a baseline for later tasks such as designing risk management systems or comparing volatility profiles across different trading days.

The presence of zero variance for some stocks in shorter windows reflects cases where only a single transaction was recorded during the time window, resulting in no observed price variation.

Why Variance Can Exceed One Variance, defined as the average squared deviation from the mean, can naturally exceed one if the price values fluctuate widely within the window. This is mathematically valid and expected, especially for stocks with high volatility or low price levels where relative changes become more pronounced. It does not indicate an error but describes the true dispersion of the data.

4.4 Results

Table 1 shows the results for the ten stocks with the smallest daily variance at the end of the first simulated trading day (Batch 81). Combining variance and average price allows insight into both price stability and scale level.

Stock Name	**Daily Price Variance**	**Average Daily Price**
DWDP	0.205	71.370
AES [finance:AES Corporation]	0.746	11.902
NAVI	1.930	14.490
BHF	2.185	58.766
F [finance:Ford Motor Company]	2.646	13.648
HST	3.269	18.991
PHM	3.495	19.283
PBCT	3.613	15.950
NWS	3.823	14.229
NWSA	3.858	14.149

Table 1: Results for the first 10 stocks in Batch 81, representing the end of the first simulated trading day.

5 Task 2: Stocks with the Highest Hourly and Daily Growth

5.1 Definition

The second task evaluates the growth of stock `price` values both hourly and daily. Since the simulated day lasts five minutes, the hourly growth is computed by considering subdivisions within that period.

The absolute growth is defined as:

$$\Delta P_S = P_{\text{S, last}} - P_{\text{S, first}}$$

where $P_{\text{S, first}}$ is the first price value of the stock S recorded in the defined time window, while $P_{\text{S, last}}$ is the final price value observed.

The relative or percentage growth is:

$$\%\Delta P_S = \frac{P_{\text{S, last}} - P_{\text{S, first}}}{P_{\text{S, first}}} \times 100.$$

The percentage is generally preferred because it better accounts for the price scale. For example, a 5-euro change is more significant for a stock priced at 10 euros than for one priced at 500 euros.

5.2 Implementation and Setup

The streaming pipeline uses Spark Structured Streaming to consume Kafka stock price data and parse JSON events into structured columns: stock `(name)`, first price in window `(price_before)`, and last price in window `(price_after)` along with window timestamps.
For convenience, not only the gain but the prices before and after are displayed so that one can verify the result.

For hourly growth (Task 2.1), the window interval corresponds to 12.5 seconds (simulated hour), fitting four such windows into the 5-minute simulated trading day. This enables intraday hourly growth detection within the streaming data.

For daily growth (Task 2.2), tumbling windows are applied over the full 5-minute simulated trading day interval, providing distinct, non-overlapping daily aggregation windows.

Choice of Window Type Tumbling windows are used for both hourly and daily computations. These fixed-size, non-overlapping intervals assign each event to exactly one window, simplifying the aggregation of first and last prices and avoiding ambiguity or duplication that sliding or session windows might introduce.

Choice of Output Mode The streaming queries for both hourly and daily growth run with `complete` output mode. This choice ensures the full updated aggregation result for all stocks and windows is output at every micro-batch trigger, supporting comprehensive continuous monitoring. `Update` mode is unsuitable because it outputs only those rows that change per trigger, which risks omitting relevant historic or unchanged aggregation data needed for full visibility in these time-windowed growth analyses.

5.3 Interpretation

A positive ΔP_S indicates the stock price increased during the window, while a negative value indicates a decrease. The percentage growth normalizes this change relative to the starting price, allowing meaningful comparison across stocks with different price scales.

The joint analysis of absolute and relative growth highlights stocks with significant upward trends regardless of initial price. For example, CHRW demonstrates a larger percentage gain despite a smaller absolute increase compared to others.

5.4 Results for Hourly Growth (Task 2.1)

Table 2 lists the ten stocks with the highest absolute *hourly* gain from the first simulated hour, along with their corresponding percentage gains.

Name	**Price Before**	**Price After**	**Hourly Gain**	**% Gain**
ILMN	137.070	155.000	17.930	13.081
CHRW	51.560	64.090	12.530	24.302
GOOG [finance:Alphabet Inc.]	533.000	545.320	12.320	2.311
SLG	99.650	110.630	10.980	11.019
MTD	242.290	253.130	10.840	4.474
ROP	131.040	141.320	10.280	7.845
MMM	136.250	145.300	9.050	6.642
TRIP	89.510	98.360	8.850	9.887
MCO	78.520	86.150	7.630	9.717
SRE	98.260	105.250	6.990	7.114

Table 2: Top-10 stocks by hourly absolute gain with corresponding percentage gain, extracted from Batch 1 representing the first window of the simulated trading hour.

5.5 Results for Daily Growth (Task 2.2)

Table 3 reports the ten stocks with the highest absolute *daily* gain, ordered by ΔP_S, alongside corresponding percentage gain.

Name	**Price Before**	**Price After**	**Daily Gain**	**% Gain**
PRGO	86.930	195.000	108.070	124.318
AZO [finance:AutoZone, Inc.]	692.940	761.910	68.970	9.953
AYI	207.760	251.780	44.020	21.188
MCK	146.660	183.930	37.270	25.413
WLTW	126.822	158.200	31.378	24.742
AAP	140.290	169.900	29.610	21.106
SIG	82.290	111.810	29.520	35.873
AMGN	126.320	154.950	28.630	22.665
ABC	77.640	105.807	28.167	36.279
REGN	388.800	415.820	27.020	6.950

Table 3: Top-10 stocks by daily absolute gain with corresponding percentage gain, extracted from Batch 40 representing the final window of the simulated trading day.

These rankings highlight stocks displaying the strongest upward movements during the respective windows. While ordered by absolute gain, the percentage gain provides essential complementary context: for instance, PRGO shows both high raw and relative increases, while AZO displays large absolute changes that correspond to modest percentage variation because of its already high initial price.

6 Task 3: Top-10 Most Traded Symbols Daily

6.1 Definition

The most traded stocks are identified by counting the total number of shares exchanged aggregated for every stock and ranking the 10 highest results within the window. The **total shares** for a stock is calculated as:

$$\text{totShares}(S) = \sum_{i=1}^{n} N_{S,i}.$$

where $N_{S,i}$ is the number of shares for stock S on the i^{th} transaction inside a given time window.

This task differs from Task 4 by focusing solely on the total number of shares exchanged per stock, without weighting by price, thereby capturing pure trading activity volume but not the monetary value involved.

6.2 Implementation and Setup

The streaming system ingests stock trade data from a Kafka topic using Spark Structured Streaming. JSON-encoded messages are parsed using a predefined schema extracting fields including the stock `(name)`, trade `(price)`, number of shares traded `(n_shares)`, and `(timestamp)`. The timestamp strings are converted into Spark timestamp types to enable time-based aggregations.

Total traded shares per stock are aggregated over fixed tumbling windows corresponding to a full simulated trading day (5 minutes). This approach provides clear daily snapshots of total trading volume per symbol.

Choice of Window Type Tumbling windows are used to segment the streaming data into non-overlapping, fixed-size intervals aligned to the simulated trading day duration. This ensures that all trades for a given day aggregate into one distinct window, simplifying the computation and interpretation of total daily shares traded.

Choice of Output Mode The streaming query uses the `complete` output mode, which outputs the full updated ranking of all stocks' total shares exchanged at every trigger. This mode ensures the entire top-10 list is consistently available for review, reflecting the latest aggregated volumes without omissions. Other modes like `append` or `update` would be insufficient here since the full ranking needs to be updated continuously and partial outputs could omit important data.

6.3 Interpretation

The total shares traded for each stock is a direct measure of trading activity and market interest. High trading volumes often correlate with liquidity and investor attention, highlighting which stocks dominate daily market participation in the simulated environment.

This metric is valuable for understanding market dynamics and supports subsequent analyses such as liquidity profiling, volatility assessment, and portfolio construction based on trading popularity.

6.4 Results

Table 4 presents the top-10 most traded stocks ranked by total shares exchanged during the simulation's daily window. Leading the list are BAC and AAPL [finance:Apple Inc.], indicating their dominant market liquidity and investor engagement. Other notable stocks include MU, AMD, GE, and Microsoft [finance:Microsoft Corporation], reflecting active trading in major technology and industrial sectors.

Name	**Total Shares Exchanged**
BAC	4,277,320,472
AAPL [finance:Apple Inc.]	3,475,999,553
MU	1,558,183,821
AMD	1,553,999,369
GE	1,427,502,364
CSCO	1,367,728,246
INTC	1,316,641,902
FCX	1,194,862,274
MSFT [finance:Microsoft Corporation]	1,120,064,818
FB	1,066,323,492

Table 4: Top-10 most traded stocks by total shares exchanged, extracted from Batch 301 representing the final window of the simulated trading day.

7 Task 4: Top-5 Stocks by Volume

7.1 Definition

In financial markets, **volume** represents the total number of shares traded during a given time interval, scaled by the average price of that stock over the same interval. The volume for a stock S is defined as:

$$\text{Volume}(S) = \left(\sum_{i=1}^{n} N_{S,i} \right) \cdot \bar{P_S} = \text{totShares}(S) \cdot \bar{P_S}$$

where $N_{S,i}$ is the number of shares traded for stock S in the i^{th} transaction, totShares(S) is the total number of exchanged shares as defined in Task 3, and $\bar{P_S}$ is the average price of stock S in the window considered.

Unlike Task 3, which aggregates only share counts, Task 4 accounts for both the number of shares and their average price within the window, providing a monetary-weighted view of trading volume reflecting total capital flow.

7.2 Implementation and Setup

The streaming application reads trade events from the Kafka topic using Spark Structured Streaming. Each message is parsed into structured columns: stock `name`, `price`, `n_shares`, and `timestamp`. The `timestamp` field is converted to Spark's `timestamp` type to enable time-based aggregations over the simulated trading day.

For each stock and each tumbling window corresponding to a simulated day (5 minutes), the query computes:

- `tot_shares_exchanged` as the sum of `n_shares`,
- `average_price` as the mean of `price`,
- `total_volume` as `tot_shares_exchanged` × `average_price`.

The resulting dataset is then ordered by `total_volume` and the top-5 stocks per window are written to the console.

Choice of Window Type Tumbling windows over 5-minute intervals are used to represent discrete simulated trading days. Since each trade belongs to exactly one window, aggregation of total shares and average prices per day remains unambiguous and easy to interpret. Sliding windows were not chosen because overlapping windows would duplicate trades across multiple windows and make "daily" volume harder to interpret.

Choice of Output Mode For Task 4, the final ranking is obtained from a static snapshot of the aggregated data after the streaming execution for the simulated day is complete (using a batch query over the checkpointed data). This effectively corresponds to a *complete* view of the final window's aggregates rather than incremental `append` or `update` outputs. Using a complete snapshot ensures that the final top-5 ranking reflects all trades in the day without missing late-arriving data. In contrast, `update` mode would only output changed rows at each trigger and would be less suitable for reporting the final, global top-5 ranking.

7.3 Interpretation

Trading volume captures both how many shares have changed hands and at what price levels, providing an estimate of the total monetary flow associated with a stock in the considered period. High volume often signals increased market attention, higher liquidity, and potentially stronger reactions to news or other financial signals.

Therefore, the top-5 stocks by volume represent the most actively traded assets in value terms, rather than simply by count of shares, and thus highlight where the highest concentration of capital is moving in the simulated market.

7.4 Results

Table 5 reports the top-5 stocks ranked by total traded volume in the final window of the simulated trading day, together with their total shares exchanged and average prices.

Name	Total Shares Exchanged	Average Price	Total Volume
AAPL [finance:Apple Inc.]	486,763,543	111.229	5.41422E10
AMZN [finance:Amazon.com, Inc.]	47,072,436	766.150	3.60645E10
FB	291,645,331	90.824	2.64884E10
MSFT [finance:Microsoft Corporation]	360,125,562	49.563	1.78488E10
GOOGL [finance:Alphabet Inc.]	23,617,312	705.577	1.66638E10

Table 5: Top-5 stocks by total traded volume, extracted from Batch 72 representing the final window of the simulated trading day.

In Table 5, the ranking shows the stocks with the largest total traded volume during the final simulated day. The total volume combines the number of exchanged shares with their price level, providing a concise view of where most capital was concentrated. Notably, AAPL [finance:Apple Inc.] stands out with both a very high share count and large monetary volume, indicating strong liquidity and sustained market attention in the simulated data.

8 Task 5: Stop-Loss and Take-Profit System

8.1 Definition

For the fifth analysis, a risk management system based on **stop-loss** and **take-profit** thresholds was implemented, typical in automated trading.

It is assumed that an investor purchases **one single share** at the first observed price in the time window. This assumption aligns with the Spark Streaming data structure where the column `n_actions` represents total market shares exchanged rather than individual transactions.

The stop-loss and take-profit thresholds adopted are based on practical trading guidelines provided by IG Europe GmbH and IG Markets Limited [1]:

$$\text{Stop-loss} = -5\% \qquad \text{Take-profit} = +20\%$$

8.2 System Logic

The system monitors the price evolution of each stock in real time within the streaming window:

- If the price falls below 5% from the initial entry price, the position is closed to limit potential losses (stop-loss).
- If the price rises above 20% from the initial entry price, the position is closed to secure profits (take-profit).

Even after closing, subsequent prices are observed to evaluate if the exit decision prevented larger losses or prematurely missed further gains.

The stop-loss and take-profit logic is evaluated within each streaming window by comparing the first observed price (entry price) with subsequent prices. When the gain percentage crosses either threshold, an exit signal is generated. Even after the exit signal, later prices within the same window are still analyzed to assess whether the exit avoided further losses or prematurely limited additional profits.

8.3 Assumptions

The price evolution is evaluated in real time as each transaction arrives, aligned with Spark Streaming's sequential processing of market events. This approach assumes that price changes between consecutive transactions are monotonic, ignoring intra-interval fluctuations.

8.4 Implementation and Setup

The stop-loss/take-profit system processes real-time price events for each stock within a tumbling window corresponding to the simulated trading day (5 minutes). This windowing ensures clear segmentation of data into distinct trading days without overlap.
The streaming query employs the `complete` output mode, which outputs the full aggregation results for all stocks at each trigger. This mode guarantees comprehensive visibility of threshold breaches and system state, avoiding omission of interim or unchanged states which might occur using `update` or `append` modes.

Choice of Window Type The stop-loss and take-profit checks are applied on tumbling windows corresponding to the simulated trading day (5 minutes), grouping relevant price events distinctly.

Choice of Output Mode The streaming query operates in `complete` output mode to deliver full updated state at each trigger, which ensures thorough monitoring of all stock price progressions and their threshold-crossing events. This mode is preferred over `update` or `append` to avoid missing any intermediate state necessary for accurate risk assessments.

8.5 Interpretation

The stop-loss and take-profit analysis produces different possible scenarios reflecting various market behaviors and strategy outcomes.

An effective stop-loss situation occurs when the threshold prevents larger losses, as the price continues to decline after the position has been closed. Similarly, an effective take-profit scenario secures gains near a temporary peak before the price subsequently declines.
However, the system may also incur premature stop-losses, where positions are closed during a temporary drawdown only for prices to recover afterward, potentially missing gains. Premature

take-profits happen when positions close too early, failing to capitalize on a continued upward price movement.

In highly volatile markets, frequent price reversals can trigger thresholds multiple times, causing the system to be sensitive to short-term fluctuations. Conversely, in trend-consistent situations, price movements follow a clear upward or downward trend, and the exit conditions behave predictably, reflecting the general market direction without unexpected triggers.
This characterization ensures a nuanced understanding of how stop-loss and take-profit thresholds interact with market dynamics and investor risk management objectives. [1]

8.6 Results

Event	**Entry Price**	**Current Price**	**Gain (%)**	**TP Hit**	**SL Hit**
1	70.336	70.336	0.000%	False	False
2	70.336	75.858	7.851%	False	False
3	70.336	73.358	4.297%	False	False
4	70.336	93.725	33.255%	True	False
5	70.336	102.350	45.518%	True	False
6	70.336	112.710	60.247%	True	False
7	70.336	133.600	89.947%	True	False
8	70.336	97.840	39.106%	True	False
9	70.336	117.760	67.427%	True	False
10	70.336	110.230	56.721%	True	False
11	70.336	115.920	64.811%	True	False
12	70.336	141.020	100.497%	True	False
13	70.336	163.890	133.012%	True	False
14	70.336	65.385	-7.039%	False	True
15	70.336	64.546	-8.232%	False	True

Table 6: Stop-loss and take-profit evaluation for the first 15 events of AAPL in the monitored five-minute window, that corresponds to a single simulated trading day.

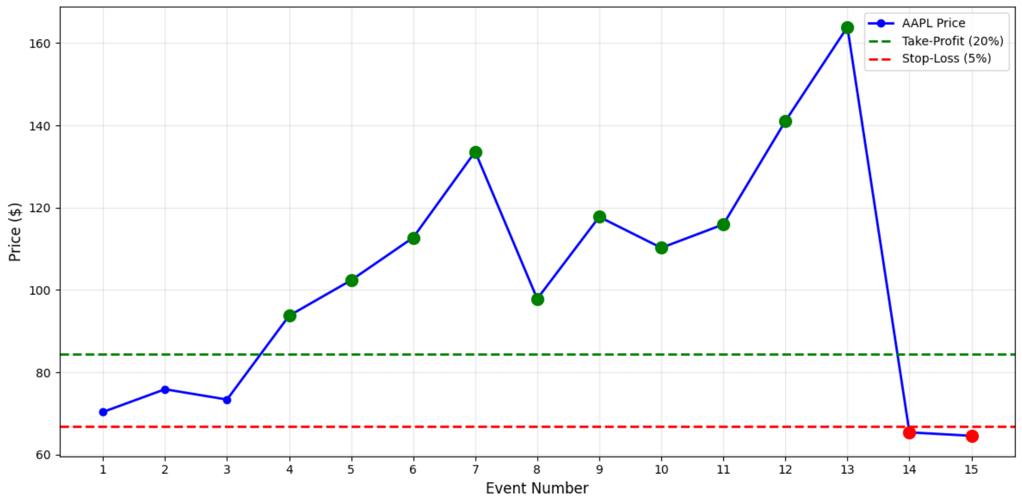


Figure 1: Real-time price evolution of AAPL showing stop-loss and take-profit thresholds: blue line (stock price), red dashed line (stop-loss), green dashed line (take-profit), and colored markers indicating threshold hits.

Table 6 and Figure 1 illustrate the behavior of the stop-loss and take-profit mechanism when applied to the first fifteen market events of AAPL within the monitored five-minute Spark Streaming window, which in the context of this project corresponds to a single simulated trading day. Although the full day may include more events, the analysis presented here focuses exclusively on the first fifteen observations.

The position is opened at the first observed price of the window, assuming the purchase of a single share. During the initial events (1–3), price fluctuations remain well within the stability band defined by the +20% take-profit and –5% stop-loss thresholds, and no exit condition is activated.

Beginning with event 4, the price increases sharply, consistently exceeding the take-profit boundary. Events 4 through 13 represent a sequence of strong upward movements in which the mechanism repeatedly identifies valid profit-taking points. These cases can be characterized as instances of premature take-profit behavior: although the mechanism secures gains at each threshold breach, the continuing price appreciation indicates that maintaining the position would have yielded substantially larger returns.

The trend reverses abruptly in events 14 and 15, where the price declines by –7.039% and –8.232% relative to the entry point. Both values surpass the stop-loss threshold, demonstrating an effective protective response. In these cases, the mechanism would have prevented the investor from experiencing deeper losses as the downward movement intensified.

Overall, the fifteen-event sequence captures two contrasting market regimes—an extended upward trend followed by a rapid downturn—and illustrates the corresponding reactions of the stop-loss and take-profit mechanism. The results reflect both its capacity to lock in gains during rising markets and its ability to limit losses when conditions deteriorate.

Moreover, the behavior confirms that the stateful Spark Streaming implementation correctly detects threshold breaches in real time and consistently preserves the investor's entry-state throughout the simulated trading day.

9 Task 6: Construction and Tracking of a Five-Stock Wallet

9.1 Definition

For the final analysis, a virtual investment wallet composed of five stocks is constructed. The system assumes that the investor purchases **one single share** of each selected stock at the first observed price in the time window. The value of the wallet at any moment is computed as the sum of the most recent `price` available for each of the five assets.

A single selection strategy is adopted for this task: choosing the **five most traded stocks**, as identified in Task 3. This choice is coherent with Spark Streaming, since highly traded stocks produce a large number of streaming events, guaranteeing continuous price updates. Moreover, liquid assets tend to exhibit fewer abrupt price jumps, resulting in a more stable and interpretable portfolio evolution.

9.2 Assumptions

The evaluation relies on the actual sequence of observed market events. Since each streaming record corresponds to a single price update, the system assumes that price movements between consecutive events are monotonic: if the new price is higher, the price is considered to have increased directly; if it is lower, it is considered to have decreased directly. Any fluctuations that might have occurred between these two observations are necessarily ignored, as they are not observable in the incoming data. This behavior is fully aligned with Spark Streaming, which processes discrete events rather than continuous price trajectories.

9.3 Implementation and Setup

The real-time wallet computation is performed over tumbling windows of 30 seconds, each representing a “day” in the simulated market. This window type ensures that each event is uniquely assigned to one day, providing distinct portfolio snapshots without overlap or ambiguity.

The streaming query employs the `complete` output mode, which outputs the entire updated wallet value and constituent stock prices at each trigger. This mode is suitable here to maintain a comprehensive, consistent view over the portfolio evolution, avoiding information loss that can occur with incremental update or append modes in streaming aggregation.

9.4 Interpretation

The wallet evolution may present various behaviors depending on the individual stock trends within the portfolio. A stable upward evolution indicates overall positive momentum across all assets. Strong positive acceleration occurs when one or more stocks experience notable gains, driving rapid portfolio growth. Flat or weakly varying behavior reflects low-volatility or stable phases in the market. Progressive decline signals a downward trend across the stocks. Mild oscillations show short-term variations from liquid assets without extreme volatility. Mixed-trend behavior occurs when some stocks rise while others fall, partially balancing the wallet value.

These insights help evaluate portfolio stability and performance under streaming market conditions, demonstrating the impact of liquidity on volatility and risk.

9.5 Results

The evolution of the five-stock wallet, composed of `BAC`, `AAPL`, `MU`, `AMD`, and `GE`, is presented in Figure 2. Each 30-second streaming window represents a single "day" in this implementation.

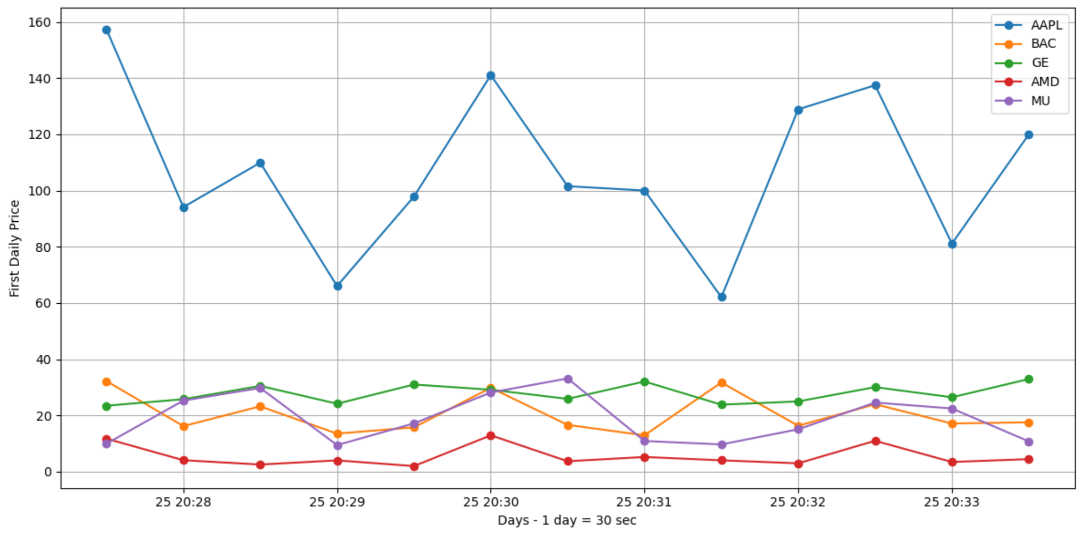


Figure 2: Live evolution of the five-stock wallet. Each 30-second window in the streaming data represents a "day". The figure illustrates how individual stock price variations contribute to overall wallet value.

From the the trends in Figure 2, it emerges that the portfolio shows mixed-trend behavior: some stocks experience gains while others show losses, partially stabilizing the wallet value.
Highly volatile stocks such as `AMD` and `AAPL` contribute significantly to short-term fluctuations. Relatively stable stocks, such as `GE`, reduce overall volatility and help moderate abrupt swings.

10 Conclusion

The analysis demonstrates the feasibility of building a real-time financial monitoring system using Kafka and Spark Structured Streaming. Although the dataset is synthetic, the methodology successfully reproduces core concepts from financial market analysis, including variance, price growth, trade frequency, trading volume, and basic risk-management strategies.
The examination of stop-loss and take-profit mechanisms shows how automated trading strategies can be implemented within a streaming architecture, while the construction of a five-stock portfolio highlights the role of diversification even under simplified market conditions.

Future developments could involve the use of more realistic or high-frequency datasets, the extension of trading logic with adaptive or data-driven thresholds, and the integration of machine-learning models to enable real-time forecasting and more sophisticated decision-making within the streaming pipeline.

References

[1] Charles Archer. Que sont les ordres take-profit et stop-loss et comment fonctionnent-ils ? G Europe GmbH and IG Markets Limited. Stratégies de trading.

11 Appendix: User Guide for Running the Notebook

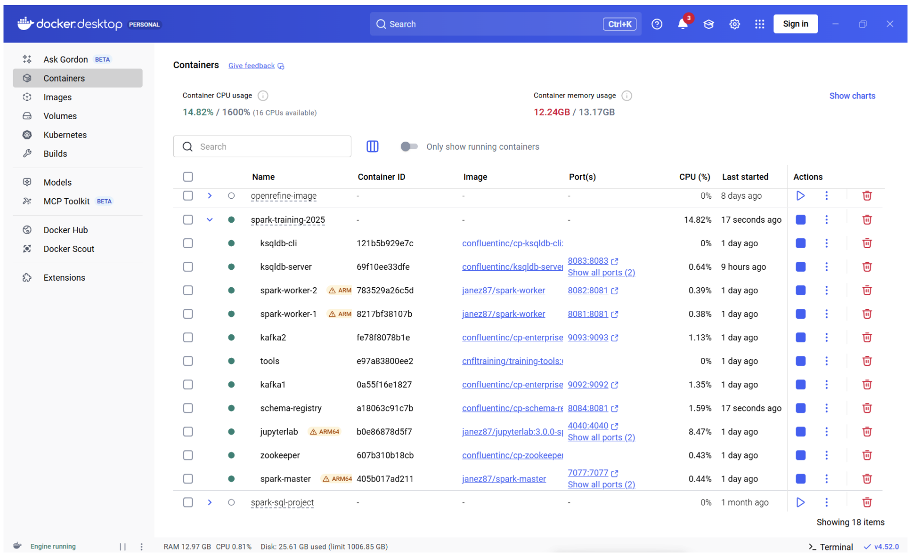
This description explains what must run, in which order, how to start/stop streaming jobs, and how to avoid all possible problems.
The project requires a fully configured environment with Docker, Spark, Kafka, Zookeeper, and Jupyter Notebook. All real-time processing tasks rely on a Kafka Producer continuously streaming stock data, and Spark Structured Streaming queries consuming that data.

Run Docker Desktop

Inside Docker, the project environment must be running.
Launch `spark-training-2` environment → This starts 11 containers, including:

- Kafka broker
- Zookeeper
- Spark master
- Spark workers
- Hadoop namenode
- Hadoop datanode
- Jupyter Notebook server
- Others required internally

All 11 containers must be green / running. If any container is stopped (especially kafka1 or zookeeper), Spark Streaming cannot read from Kafka, and the tasks will never output anything.



Containers

Container CPU usage
14.82% / 1600% (16 CPUs available)

Container memory usage
12.24GB / 13.17GB

Name	Container ID	Image	Port(s)	CPU (%)	Last started
openrefine-image	-	-	-	0%	8 days ago
spark-training-2025	-	-	-	14.82%	17 seconds ago
ksqldb-cli	121b5b929e7c	confluentinc/cp-ksqldb-cli:		0%	1 day ago
ksqldb-server	69f10ee33dfe	confluentinc/ksqldb-server	8083:8083 Show all ports (2)	0.64%	9 hours ago
spark-worker-2 ⚠ ARM	783529a26c5d	janez87/spark-worker	8082:8081	0.39%	1 day ago
spark-worker-1 ⚠ ARM	8217bf38107b	janez87/spark-worker	8081:8081	0.38%	1 day ago
kafka2	fe78f8078b1e	confluentinc/cp-enterprise	9093:9093	1.13%	1 day ago
tools	e97a83800ee2	cnfltraining/training-tools:		0%	1 day ago
kafka1	0a55f16e1827	confluentinc/cp-enterprise	9092:9092	1.35%	1 day ago
schema-registry	a18063c91c7b	confluentinc/cp-schema-re	8084:8081	1.59%	17 seconds ago
jupyterlab ⚠ ARM64	b0e86878d5f7	janez87/jupyterlab:3.0.0-s	4040:4040 Show all ports (2)	8.47%	1 day ago
zookeeper	607b310b18cb	confluentinc/cp-zookeeper		0.43%	1 day ago
spark-master ⚠ ARM64	405b017ad211	janez87/spark-master	7077:7077 Show all ports (2)	0.44%	1 day ago
spark-sql-project	-	-	-	0%	1 month ago

Showing 18 items

Engine running RAM 12.97 GB CPU 0.81% Disk: 25.61 GB used (limit 1006.85 GB) Terminal v4.52.0

Figure 3: Running Docker Containers.

Open Jupyter Notebook

Open any browser and type "localhost:8888". If Docker is running correctly, Jupyter Notebook opens.

Prepare and start the Kafka Producer

Inside Jupyter, there exist three producer notebooks, but the correct one is "Kafka-Producer-for-project.ipynb". Ignore the possibly two others. Before running it:

- Click Kernel → Interrupt Kernel
- Click Kernel → Restart Kernel Clear All Outputs
- Scroll to the last cell (in this case Cell 8)
- Click into the last cell
- Click top menu → Run → Run All Above Selected Cell
- Watch Cell 7 to confirm stock data is being produced

This starts the producer. You must see streaming JSON messages appear in cell 7. If nothing appears, the producer is not running → restart kernel again. The Kafka Producer must continue running the entire time while you run your Spark tasks.

Place Input Files

The notebook must be executed in an environment configured for Spark and Kafka. The CSV file stock.csv and the Kafka producer script should be placed in the same directory as the notebook. The user should begin by starting the Kafka and Zookeeper services, followed by running the producer script to initiate data ingestion. Do not modify the first cells of the notebook as they contain the configuration for connecting Spark to Kafka and parsing the incoming stream.

- Cell 1: Imports
- Cell 2: SparkSession configuration
- Cell 3: Schema
- Cell 4: Kafka stream reader

These must remain unchanged for every task.

Running a Task Notebook

Each task is provided in a separate notebook:

- task1.ipynb
- task2.ipynb

- task3.ipynb
- task4.ipynb
- task5.ipynb
- task6.ipynb

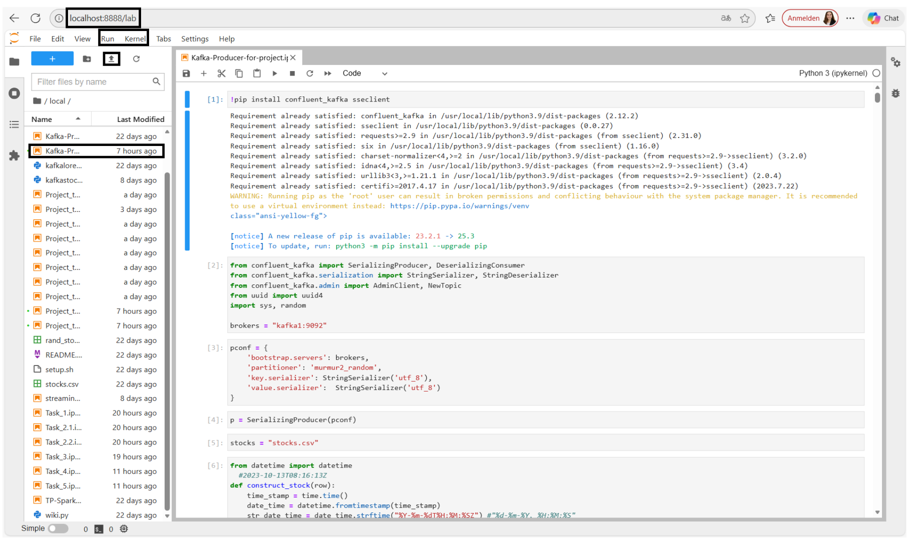
Since Spark streaming prints to the console, you cannot run multiple tasks simultaneously.
Steps to run a query for a task

- In Jupyter, click Upload Files
- Upload the notebook for the task (e.g., task3.ipynb)
- Open the notebook
- Click Run → Run All Cells

A streaming query starts and prints Batch 0, Batch 1, Batch 2… in the output
To stop a running task

- Click on the cell producing streaming output
- Click Kernel → Interrupt Kernel

Important: Merely stopping the cell is not enough if you want a fresh state.
For a clean run:
After interrupting the task, also stop the Kafka Producer. Restart and clear the Kafka producer notebook.



```
[1]: !pip install confluent_kafka sseclient

     Requirement already satisfied: confluent_kafka in /usr/local/lib/python3.9/dist-packages (2.12.2)
     Requirement already satisfied: sseclient in /usr/local/lib/python3.9/dist-packages (0.0.27)
     Requirement already satisfied: requests>=2.9 in /usr/local/lib/python3.9/dist-packages (from sseclient) (2.31.0)
     Requirement already satisfied: six in /usr/local/lib/python3.9/dist-packages (from sseclient) (1.16.0)
     Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.9/dist-packages (from requests>=2.9->sseclient) (3.2.0)
     Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests>=2.9->sseclient) (3.4)
     Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests>=2.9->sseclient) (2.0.4)
     Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests>=2.9->sseclient) (2023.7.22)
     WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended
     to use a virtual environment instead: https://pip.pypa.io/warnings/venv
     class="ansi-yellow-fg">

     [notice] A new release of pip is available: 23.2.1 -> 25.3
     [notice] To update, run: python3 -m pip install --upgrade pip

[2]: from confluent_kafka import SerializingProducer, DeserializingConsumer
     from confluent_kafka.serialization import StringSerializer, StringDeserializer
     from confluent_kafka.admin import AdminClient, NewTopic
     from uuid import uuid4
     import sys, random

     brokers = "kafka1:9092"

[3]: pconf = {
         'bootstrap.servers': brokers,
         'partitioner': 'murmur2_random',
         'key.serializer': StringSerializer('utf_8'),
         'value.serializer':  StringSerializer('utf_8')
     }

[4]: p = SerializingProducer(pconf)

[5]: stocks = "stocks.csv"

[6]: from datetime import datetime
       #2023-10-13T08:16:13Z
     def construct_stock(row):
         time_stamp = time.time()
         date_time = datetime.fromtimestamp(time_stamp)
         str_date_time = date_time.strftime("%Y-%m-%dT%H:%M:%SZ") #"%d-%m-%Y, %H:%M:%S"
```

Figure 4: Jupyter Notebook Starting Page - Important Buttons Highlighted.

Ensuring the Data Flows

Every task ultimately depends on the stream:

```
query
```

which must output:

```
Batch: 0
...
Batch: 1
...
Batch: 2
...
```

If batches are not appearing:

- Spark task will hang and show no results
- Task 6 will produce no wallet output
- Task 5 sliding windows will output empty windows

Therefore:

- Always confirm producer is running
- Always confirm `parsed_df` is receiving batches

Stopping a Task and Switching to Another

- Click the running output cell
- Kernel → Interrupt Kernel
- Wait until the cell stops
- Run the next notebook

You do not need to restart Docker and the producer (but it is safer).

Common symptoms and causes

Symptom	**Cause**
No batches appear	Kafka producer not running
Task 1/2/3 never prints	Window too large for your speed
daily_variance always 0	Window contains only 1 event
wallet is empty	Selected tickers do not appear
Spark hangs	Previous task still running
Cannot stop Spark	Need to "Interrupt Kernel"

Special Case: Task 6

Task 6 is the **only** task where the following two rules apply:

- **Do not run** the general streaming query

  ```
  query
  ```

  because Task 6 starts its *own* streaming query internally using `foreachBatch` and writing results to Parquet.

- **Do not execute all cells at once**. Execute every cell *except the last one.* The last cell reads historical Parquet snapshots and therefore must be run **only after the streaming query is stopped**.

The correct execution order for Task 6 is:

1. Run all cells *up to* the cell that starts the streaming query (the cell calling

   ```
   writeStream.foreachBatch(...).start()}).
   ```

2. Let the stream run for a while so wallet snapshots are generated.
3. Stop the streaming query manually in Jupyter (Kernel → Interrupt).
4. Only now run the final cell which loads and displays the wallet evolution.