Lambda Architecture Report

Matteo Paparo

January 22, 2026

Abstract

The following report consists of the implementation of a **Lambda Architecture** and its adaptation to the management of financial data. The data in question concerns *cryptocurrencies*, characterized by high availability and very sharp price fluctuations in short timeframes. The architecture in question will be composed of three main layers: The **Batch Layer**, characterized by high latency and complete data processing; the **Speed Layer**, which will handle real-time data processing with very low latency; and finally, the **Serving Layer**, which will be responsible for providing the views of the Batch Layer. The goal is to perform analyses on the collected data, and to increase their veracity, a statistical model has been implemented to perform a preliminary *cleaning* phase, using an outlier removal filter based on the **3-Sigma Rule (Z-Score)**. Finally, to visualize these analyses, a **Dashboard** has been implemented, allowing data to be viewed in a simple and intuitive way.

Contents

Chapter 1

Configuration

For the realization of the project, **Docker** was used as the working environment, allowing the creation of isolated containers for each component of the Lambda Architecture. In this way, it was possible to manage dependencies and configurations simply and efficiently.

The docker compose used for the architecture implementation is composed of three macro-components:

- The **Hadoop** ecosystem for large-scale data storage management and processing.
- **Apache Kafka** as a message broker for decoupling producer and consumers, ensuring message persistence and replay.
- **Apache Spark** as a batch processing engine for calculating complex metrics (OHLC, RSI, Bollinger Bands) and for training the anomaly detection model.
- **Cassandra** database for real-time data management and fast storage in the Speed Layer.
- **Application Services**:
  - **Kafka Producer:** Data generator, which uses APIs (*Coinbase* and *CoinGecko*) and WebSockets (*Binance*) to collect cryptocurrency data. They allow for a continuous flow of data with fluctuations ranging from a minimum of 10 to peaks of 80 data points per second. Under optimal conditions, an average of about 35 data points per second was recorded.
  - **Dashboard** for visualizing collected data and performed analyses.

The resources allocated to Docker Desktop are 12GB of RAM, 8 CPUs, and 2GB of Swap, to ensure smooth operation of the implemented Lambda architecture.
The project was developed on a MacBook Air with an M3 chip, 16GB of RAM, and 216GB of storage space. CPU limits were introduced to prevent *resource starvation* situations, ensuring that no container can monopolize computational resources to the detriment of others, and to simulate a production environment with shared resources.

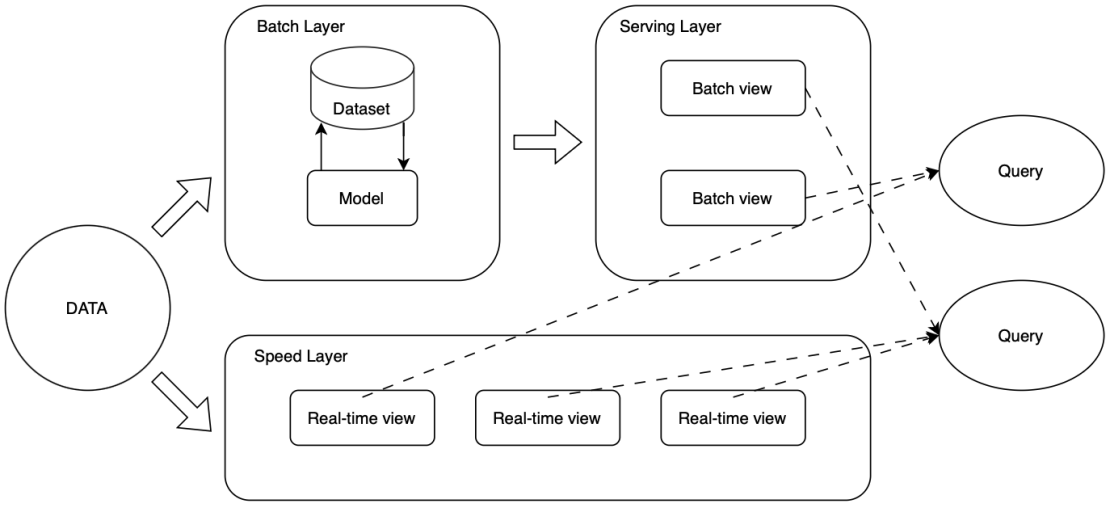
1.1 Workflow

From the start of the docker compose, the Lambda architecture workflow follows these main steps:

1. An initial calibration phase is performed, meaning that for the first 5 minutes, data is collected on HDFS without any processing, in order to have an initial dataset on which to create the data cleaning model.

2. Once the calibration phase is finished, the producer starts sending data to the Speed Layer as well, which will perform filtering using the newly created statistical model.

3. The filtered data will then be saved to HDFS and Cassandra respectively and made available for analysis and visualization via the Dashboard.

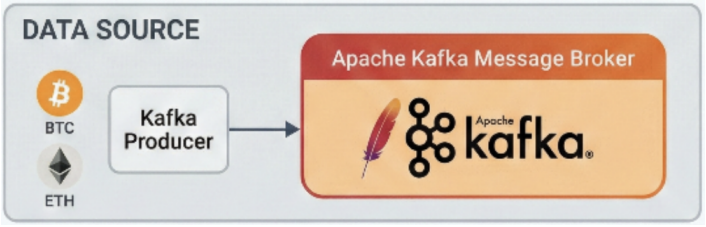


Chapter 2

Data Source

The component responsible for data generation is the **Kafka Producer**, which uses various APIs and WebSockets to collect cryptocurrency data, which is subsequently sent to the **Kafka Broker**, acting as an intermediary between producers and consumers.

It organizes files into topics and distributes the same data simultaneously to both layers through the use of dedicated consumers (in this case, one for each layer), thus allowing for increased system **scalability**.
It maintains messages for a configurable period (default is 7 days) and allows for data replay if necessary, always guaranteeing message delivery, making the system more **robust** and **reliable**.



2.1 Docker Configuration

Specifically, the Data Source architecture is composed of:

- **Kafka**: Distributed message broker that constitutes the core of the messaging system. It manages data ingestion from the Producer, temporary message persistence, and their distribution to the Consumers of the two layers (Speed and Batch). In the project, a single broker was configured with a dedicated topic (`crypto-prices`) for the cryptocurrency price stream.
- **Kafka Producer**: Responsible for generating and sending data to the Kafka broker. It uses public APIs (*Coinbase* and *CoinGecko*) and WebSockets (*Binance*) to collect real-time cryptocurrency data. The collected data includes only price information.
- **Zookeeper**: Distributed coordination service used by Kafka for cluster configuration management, leader broker election, node status monitoring via heartbeat, and synchronization of metadata related to topics and partitions.

Chapter 3

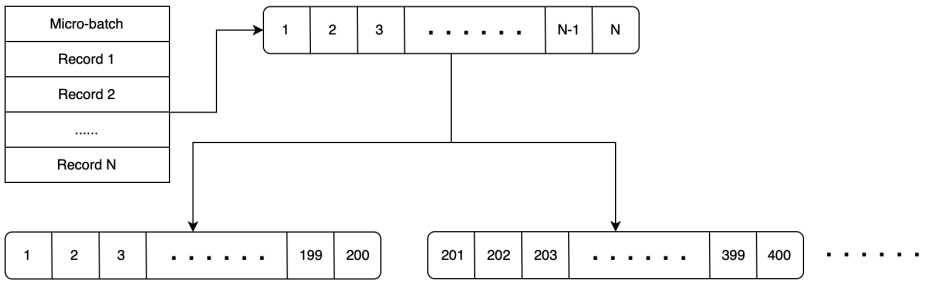
Batch Layer

The *Batch Layer* is one of the three main layers of the Lambda Architecture, responsible for storage and comprehensive data processing, allowing for in-depth analysis of historical data. In this section, the main components of the Batch Layer, the technologies used, and the implementation methods will be described.

The Batch Layer was implemented using **Apache Hadoop** as the main storage framework and **Apache Spark** for data processing. Hadoop allows for managing large amounts of data in a distributed manner, while Spark provides a fast, in-memory processing engine for data analysis.

To adapt the Batch Layer to cryptocurrency market analysis, its structure was reorganized to enable frequent analysis on the entire set of processed data. This remodulation focuses primarily on data management and involves the introduction of micro-batches, i.e., dimensional or temporal partitions, generated according to two alternative criteria: reaching 5000 records or, in the absence of this threshold, forced creation of the micro-batch after 180 seconds, including all records collected up to that moment. This approach allows for rapid and updated analyses on all available data, avoiding the need to reprocess the entire clean dataset stored on HDFS.
The analyses performed include the calculation of basic statistical metrics, such as **minimum**, **maximum**, **average**, and **volatility**, as well as technical indicators widely used in financial analysis, including **RSI** (Relative Strength Index), **Bollinger Bands**, and **Momentum**. Finally, to provide an indication of the quantity of analyzed information, the **total number of clean records** considered is also reported. The use of such indicators is fundamental in technical analysis, as it allows for evaluating market trends and supporting more informed investment decisions.

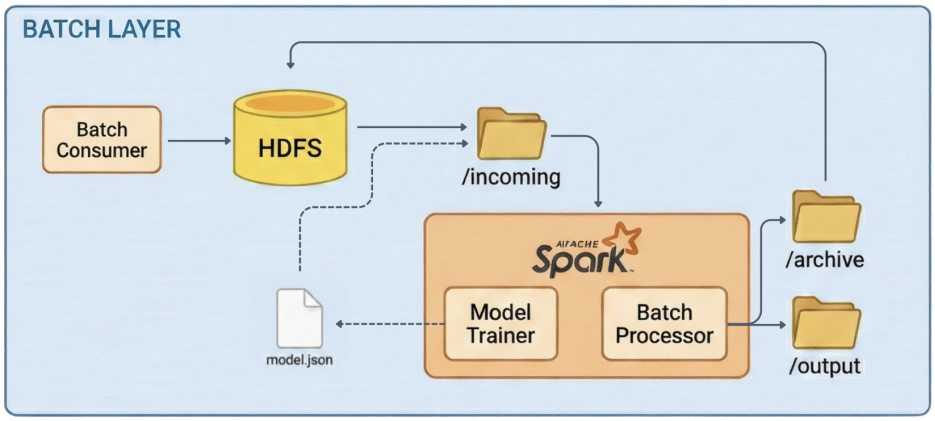
Regarding the cleaning phase, it is not performed using a static approach—meaning the entire micro-batch is filtered directly with information contained in the model—but rather, filtering occurs gradually. More precisely, the micro-batch is sorted chronologically and divided into **chunks** of 200 records. Subsequently, each chunk is filtered, and the clean data is used to update the model, thus allowing it to adapt to market trends and support market fluctuations.



Since the number of records contained in each micro-batch can vary due to the time constraint, the number of chunks is not always the same, implying that the last one may have fewer records. Even in this case, the chunk is filtered, and the clean data is used to update the model.

3.1 Workflow

Therefore, the workflow of this layer is as follows: (1) The **Batch Consumer** directs data arriving from the **Kafka Broker** and writes it to **HDFS** as `.jsonl` files in the `/iot-data/incoming` folder. (2) The Spark job `model_trainer.py` reads data **exclusively** from the `/incoming` folder (current time window), calculates the mean and standard deviation for each sensor, and saves the model to `/models/model.json`. This ensures that the model always reflects the most recent market conditions. (3) The Spark job `batch_processor.py` sequentially loads the model and filters anomalies from the raw data. It calculates metrics and saves the aggregated results in `/iot-output/spark/date=YYYY-MM-DD`. It archives filtered data in `/iot-data/archive` and subsequently deletes data from the `/incoming` folder to free up space for new data, preventing the processing of already processed data.



3.2 Docker Configuration

Specifically, the Batch Layer architecture is composed of:

- **Namenode**: The "master" node of the **HDFS** (Hadoop Distributed File System). It does not store real data but manages metadata.
- **Datanode**: The "worker" node that actually stores the data. It manages local storage and responds to data read and write requests.
- **Spark Scheduler**: The orchestrator that manages the execution times of Spark jobs:
  - `model_trainer.py`: Executed only once, 5 minutes after the architecture starts; it calculates mean and standard deviation per sensor from recent data and saves `model.json`.
  - `batch_processor.py`: Executed every 2 minutes or upon completion of each micro-batch. It processes data in `/incoming`, calculates OHLC metrics and technical indicators, and archives processed data.

  Additionally, every 5 minutes, it checks if a new sensor has been added to HDFS. If so, it re-runs the `model_trainer.py` job to include the new sensor.
- **Spark Master**: Coordinator of the Spark cluster. It manages resources and distributes tasks to worker nodes. It receives Spark jobs and decides which Worker will execute which task. It monitors the status of workers.
- **Spark Worker**[1]: The node that executes tasks assigned by the Master. It handles data processing (filter, map, aggregate) and, once done, sends results to the Master.
- **Batch Consumer**: Responsible for consuming data processed by the Batch Layer and writing it to the final storage system. To best utilize HDFS write capabilities, data is accumulated before being written, specifically every 5000 messages or every 180 seconds, whichever occurs first.

[1] In the project, 3 Spark Worker nodes were used, one for each sensor.

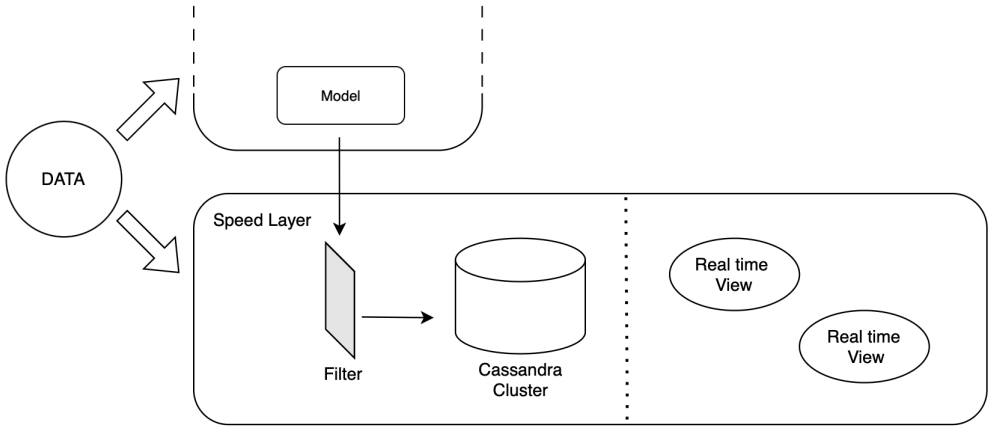
Chapter 4

Speed Layer

The second main layer composing the Lambda Architecture is the *Speed Layer*, responsible for real-time data processing with very low latency. This layer is designed to manage data requiring rapid responses and frequent updates, allowing for timely information on incoming data.

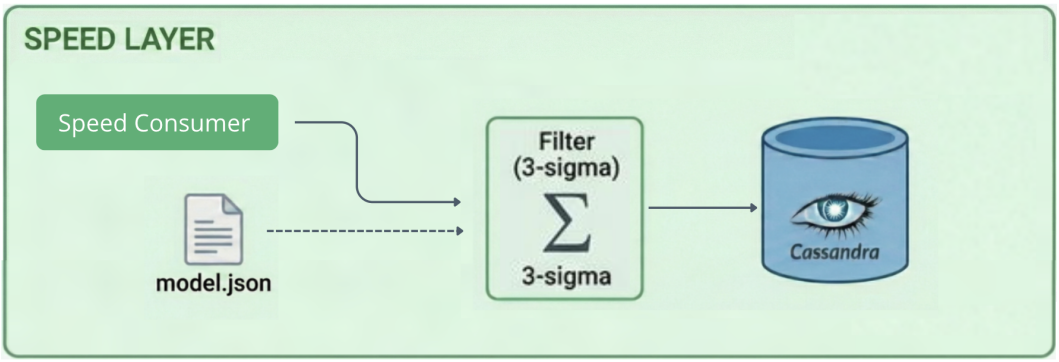
The **Speed Layer** was implemented using **Apache Cassandra** as the NoSQL database for data management. Cassandra is designed to handle enormous amounts of data across multiple servers, ensuring high availability and scalability. In the case of the Speed Layer, data is first filtered using, initially, the same model contained in `/model.json` to ensure consistency between the two layers. Subsequently, filtered data is saved in a Cassandra table specifically created to store real-time data.

Compared to the Batch Layer, the approach with the model is different, as it is updated in memory with every new valid data point received; this way, the model continuously adapts to the most recent market conditions. An **adaptive EMA** (Exponential Moving Average) filter is used, so instead of using a fixed average, the filter updates the mean and variance with every single message received, allowing for instant adaptation to market trends. If a new sensor, not yet reported in the model, is detected during execution, it is inserted into a buffer (maximum capacity of 1000 records) to wait for the model update on HDFS.



4.1 Workflow

The workflow of this layer is as follows: (1) The **Speed Consumer** waits for the `model.json` model to be available on HDFS. (2) It subsequently initializes the EMA structure in memory and directs data arriving from the Kafka Broker, applying the filter for data cleaning. (3) Filtered data is saved in the `sensor_data` table of Cassandra, and the model is updated with each received data point. (4) Every 10 seconds, it checks if the HDFS model has been updated; if so, it integrates any new sensors. Sensors already present are not modified. (5) The data is made available for visualization via the Dashboard.



4.2 Docker Configuration

Specifically, the Speed Layer architecture is composed of two nodes:

- **Cassandra-seed**: The main node of the Cassandra cluster. When other nodes start, they contact the Seed to discover the cluster topology. Besides acting as a coordinator, it is a fully operational node: it stores data in the `sensor_data` table and responds to Dashboard queries.
- **Speed Consumer**: Responsible for consuming data from the Kafka Broker; for each message, it applies the adaptive EMA filter. If the data is valid, it writes it to Cassandra and updates the EMA model in memory[1].

[1] Uses Kafka's `latest` offset to automatically skip calibration data

Chapter 5

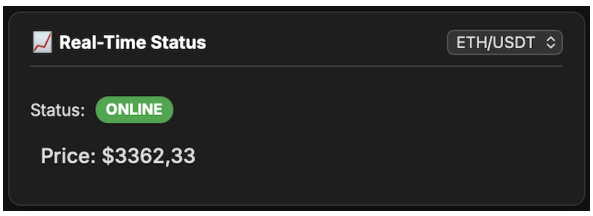
Serving Layer & Dashboard

The final component of the Lambda Architecture is the **Serving Layer**, which acts as an interface between the data processed by the two main layers (Batch and Speed) and the end user. The Serving Layer is responsible for providing data and analyses efficiently and accessibly, allowing users to visualize and interact with the collected data.

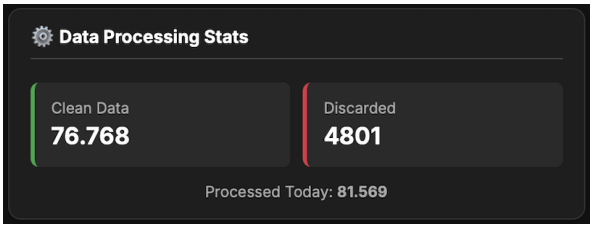
The Serving Layer was implemented using **Flask**, a lightweight and flexible web framework for Python. Flask allows for creating web applications simply and quickly, making it ideal for building user interfaces for data visualization.

The Dashboard is composed of several elements to show not only the architecture's operation but also its performance. Specifically, we have:

Widget "Real-Time Status"
Shows if sensors are active and the last price received. From the dropdown menu at the top right, it is possible to modify the cryptocurrency title to monitor.

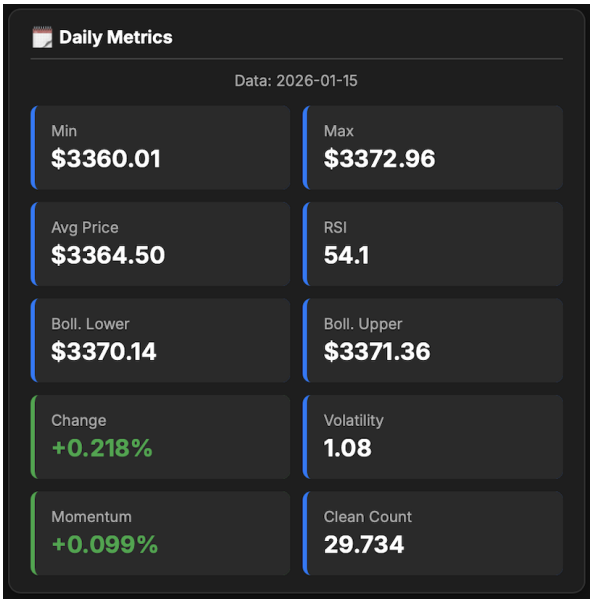


Data Processing Stats
Shows the number of data points received up to the last processed batch, highlighting the number of total data, clean data, and recorded anomalies.



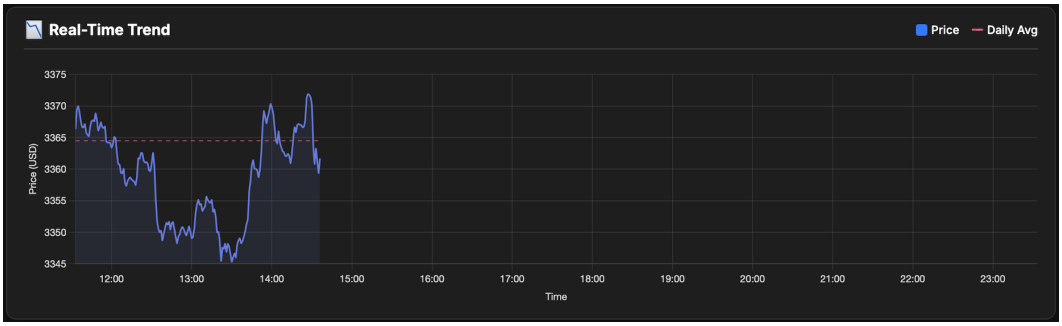
Daily Metrics

Shows the daily metrics calculated by the Batch Layer; these are general metrics, updated with every processed micro-batch. The data shown varies based on the selected title.



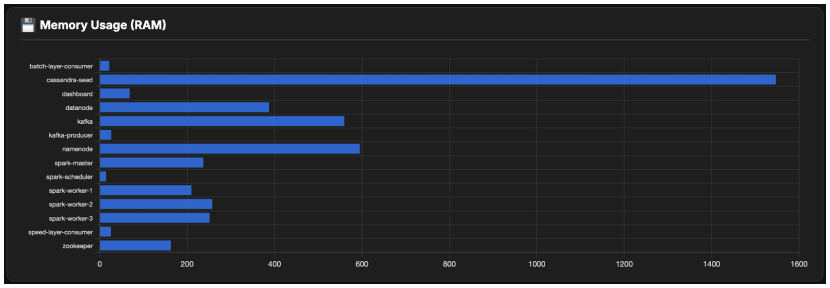
Real Time Trend

Shows the real-time trend of cryptocurrencies, updating the graph every minute with new data received from the Speed Layer. The graph reports the last 12 hours of data; upon receiving data in subsequent hours, the graph will remove the oldest data to make room for new data, creating a fluid and continuous movement.



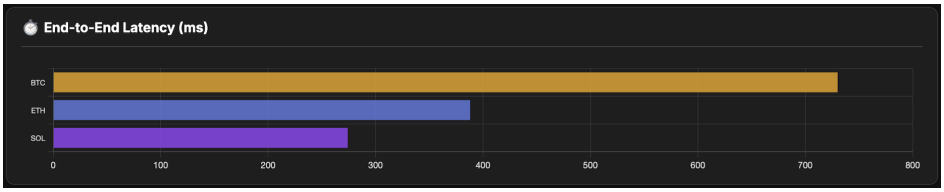
Memory Usage

Shows memory usage of Cassandra and HDFS, to monitor the health status of the storage system.

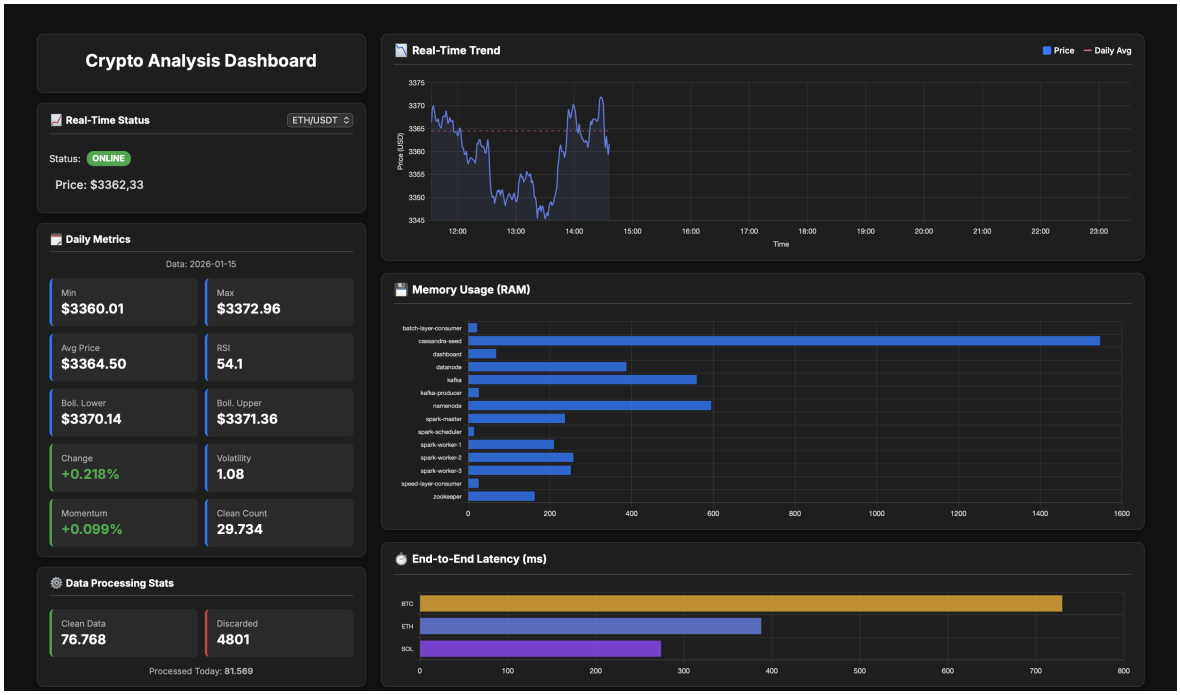


End-to-End Latency

Measures the total time taken to process a request, from data reception to result visualization in the Dashboard.



These are all the elements that make up the Dashboard, allowing for real-time monitoring of crypto analyses, as well as the health status of the Lambda architecture and the performance of the two main layers. The final result is visible in the following image:



Chapter 6

Results Analysis

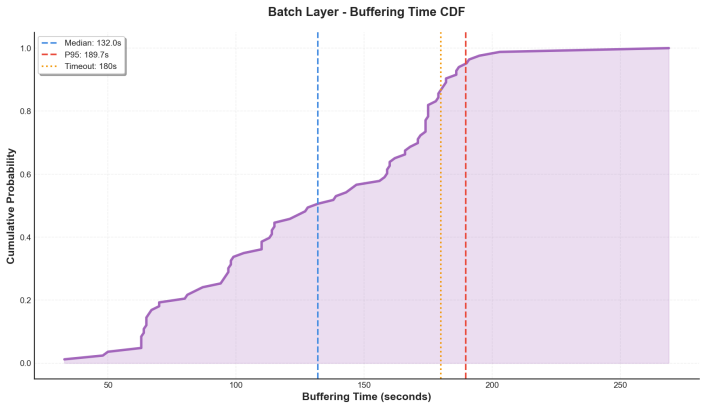
The implemented Lambda architecture underwent an intensive testing period of over 3 hours, during which detailed metrics were collected on all system layers. This analysis phase allowed for evaluating not only the individual performance of single components but also the effectiveness of isolation between layers and the system's ability to handle realistic workloads. Monitoring involved approximately 32,000 latency measurements for the Speed Layer, 84 batches processed by the Batch Layer, over 17,000 resource usage samples, and 820 measurements of Cassandra database performance.

6.1 Batch Layer Analysis

The Batch Layer represents the core of the Lambda architecture's offline processing, and its efficiency is fundamental for ensuring the quality of historical analyses. During the observation period, the system processed 84 batch files with an **average** size of 857.50 KB, demonstrating remarkable stability in data processing. The batch size oscillated between a **minimum** of 263.42 KB and a **maximum** of 945.93 KB, with an extremely contained coefficient of variation testifying to the regularity of the data flow generated by the Producer.

A particularly interesting aspect emerging from the analysis concerns the batch trigger mechanism. Statistical analysis of the data revealed that about 65% of batches are generated following the achievement of the target size (size-triggered), while the remaining 35% are due to the expiration of the 180-second timeout (time-triggered). This balance indicates that system throughput is high enough to regularly fill buffers before the timeout, but at the same time, the temporal mechanism ensures that data does not remain waiting indefinitely during periods of lower activity.

The average buffering time stands at 130.4 seconds, a value well below the preset maximum timeout. This figure confirms that the system operates under nominal regime conditions, processing data flows without generating critical accumulations or bottlenecks in queues.

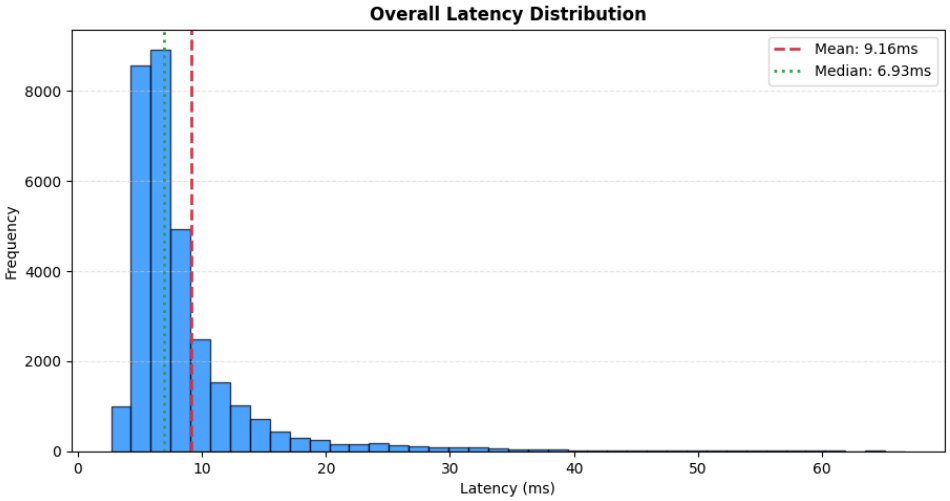


The cumulative distribution of buffering time shows almost linear growth in the central range, with the 50th percentile at 132 seconds and the 95th percentile at 193 seconds. This regularity confirms the absence of anomalous behaviors or systematic peaks due to system congestion. The only significant outlier observed, with a buffering time exceeding 250 seconds, represents less than 0.5% of total cases and can be attributed to transient phenomena during initialization phases or specific database compaction events.

The Batch Layer proved extremely efficient in terms of storage, mitigating NameNode overhead through an appropriate batching policy. The collected data excludes the presence of excessively fragmented files: the average size of 850 KB (minimum 250 KB) indeed ensures fluid metadata management without compromising Hadoop performance. Resource occupation on HDFS resulted predictable and free of anomalies, with a regular increase of 0.17 MB per minute and a total accumulation of 30 MB during monitoring.

6.2 Speed Layer Performance

The Speed Layer was designed to ensure near real-time processing of incoming data, with minimal end-to-end latency between data generation by the Producer and its availability in the Serving Layer (Cassandra).



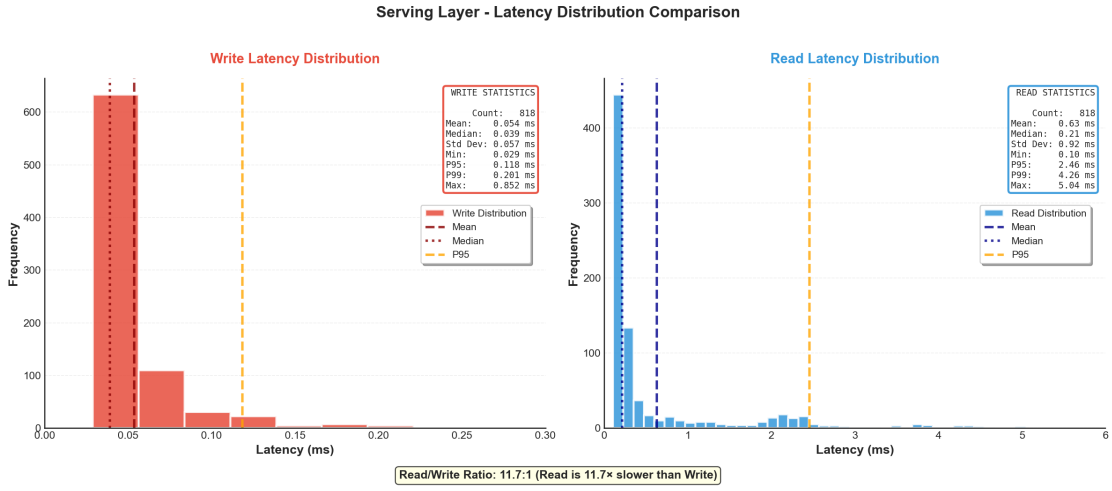
The observed average latency stands at 9.16 ms, a value that confirms the effectiveness of the realtime pipeline composed of Kafka, Speed Layer Consumer, and Cassandra. Analyzing the distribution, a characteristic leptokurtic shape emerges with a strong concentration of values in the 5-15 ms range, while the right tail of the distribution extends to maximum values of about 340 ms. The 95th percentile is positioned at 29 ms, while the 99th percentile reaches 44.54 ms, indicating that the vast majority of requests are served with latencies below 50 milliseconds.

Sensor analysis reveals substantially uniform behaviors, with small variations likely due to scheduling phenomena and resource contention. Sensor A1 shows a slightly higher average latency (10.2 ms) compared to B1 (8.8 ms) and C1 (8.5 ms), but the difference remains in the order of milliseconds and does not significantly impact the overall experience. The observed latency peaks, although present in all three sensors, manifest with a frequency of less than 1% of total measurements and can be traced back to transient JVM garbage collection events or Cassandra database compaction operations.
A fundamental aspect verified during tests concerns the isolation of the Speed Layer with respect to Batch Layer operations. The correlation between batch arrival times on HDFS and Speed Layer latency variations was calculated, obtaining a **correlation coefficient** of just **0.025**. This value, extremely close to zero, confirms that offline processing does not interfere with realtime performance, thus guaranteeing one of the cardinal principles of the Lambda architecture: separation of responsibilities and independence between the two data processing paths.

Temporal consistency of performance was maintained throughout the duration of the test, with no evidence of progressive latency degradation. This behavior suggests that the system is correctly sized for the sustained workload and that Kafka's buffering mechanisms and Cassandra's memory management function effectively.

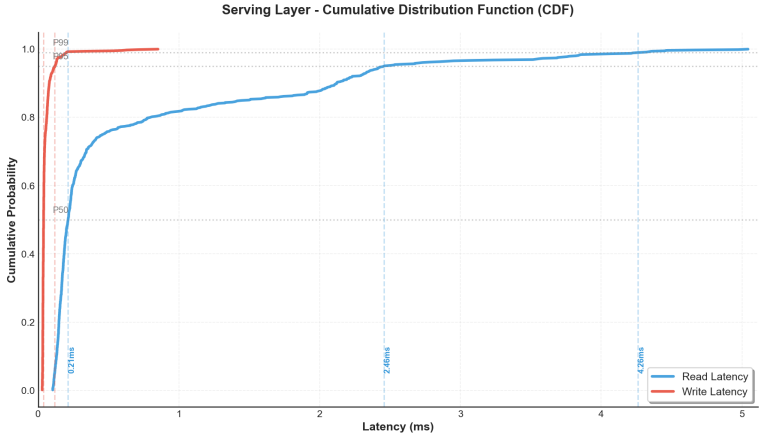
6.3 Serving Layer Efficiency

The Serving Layer represents the convergence point of the entire architecture, responsible for exposing both batch processing results and data processed in real-time by the Speed Layer. Its ability to respond with minimal latencies constitutes a determining factor for the overall usability of the system. During the test period, 818 measurements of read and write latencies were collected, providing a statistically significant picture of database performance.



Analysis of the distributions reveals distinctly differentiated behavior between the two types of operations. Writes show exceptional performance, with an average time of just 0.054 ms (54 microseconds) and an even more contained median at 0.039 ms. Value concentration results extremely high, with the 95th percentile standing at only 0.118 ms and the 99th percentile at 0.201 ms. Even in the most unfavorable cases, the observed maximum does not exceed 0.85 ms, confirming remarkable consistency in write performance.

Read operations, while inevitably slower, still maintain latencies in the sub-millisecond order for most cases. The average is positioned at 0.63 ms, while the median at 0.21 ms indicates a distribution strongly skewed to the right. This asymmetry is highlighted by the 95th percentile at 2.46 ms and the 99th percentile at 4.26 ms, with a recorded maximum of 5.04 ms. The occasional peaks observed can be traced back to physiological database phenomena, such as background compaction operations, cache misses during SSTable access, or moments of greater resource contention.



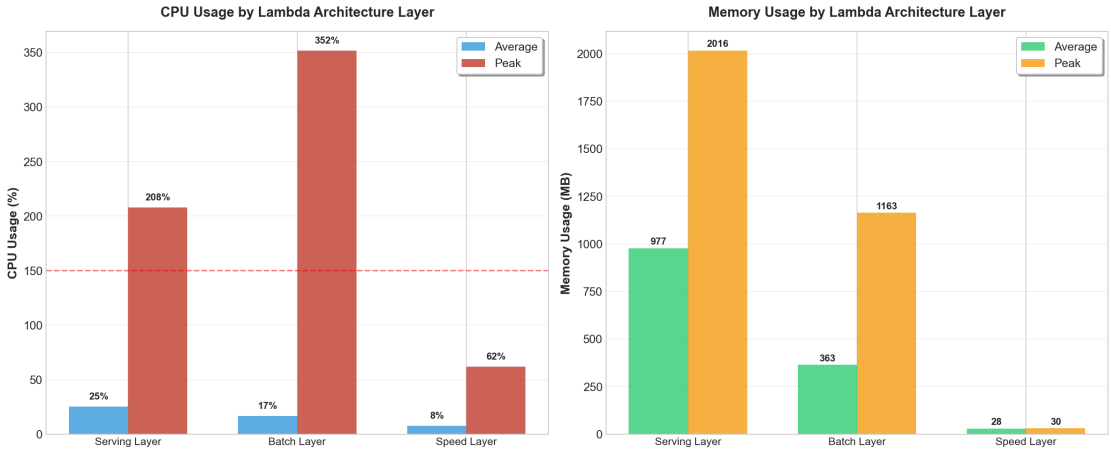
The CDF (Cumulative Distribution Function) graph allows for clearly visualizing the difference between the two curves. The write curve presents almost vertical growth within the first 0.1 ms, reaching 99% of cases within 0.2 ms. The read curve shows instead a more gradual growth: 50% of operations complete under 0.21 ms, 80% under 0.6 ms, and 95% under 2.5 ms.
The performance gap between reading and writing, quantifiable in a ratio of about

11.7:1, finds justification in Cassandra's internal architecture. The database adopts a *log-structured* approach: writes are rapidly recorded in the commit log and the memtable resident in memory, postponing disk persistence to subsequent asynchronous operations. Reads, conversely, may require access to multiple SSTables on storage, especially when data is not present in the memory cache.

Overall, Serving Layer performance proves amply adequate for the application context. Almost all write operations conclude in less than 200 microseconds, while reads remain under 2.5 ms in 95% of cases. These values ensure a fluid user experience in the Dashboard and effectively support real-time data visualization needs.

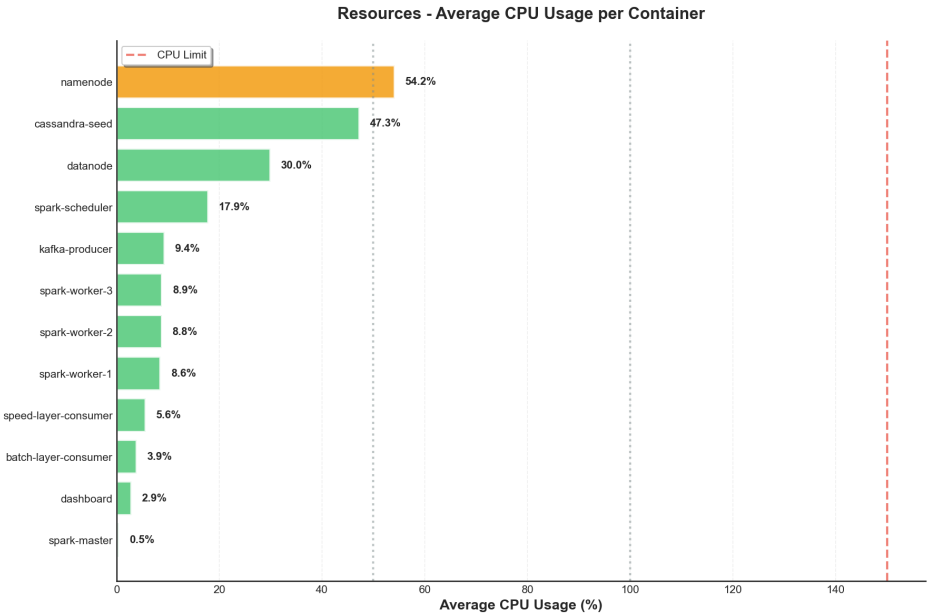
6.4 Resource Balancing

One of the most critical aspects of a Lambda architecture is the effective management of computational resources among the different layers. During the test, 12 Docker containers were monitored, collecting over 17,000 CPU and memory usage samples, allowing for detailed analysis of load balancing.



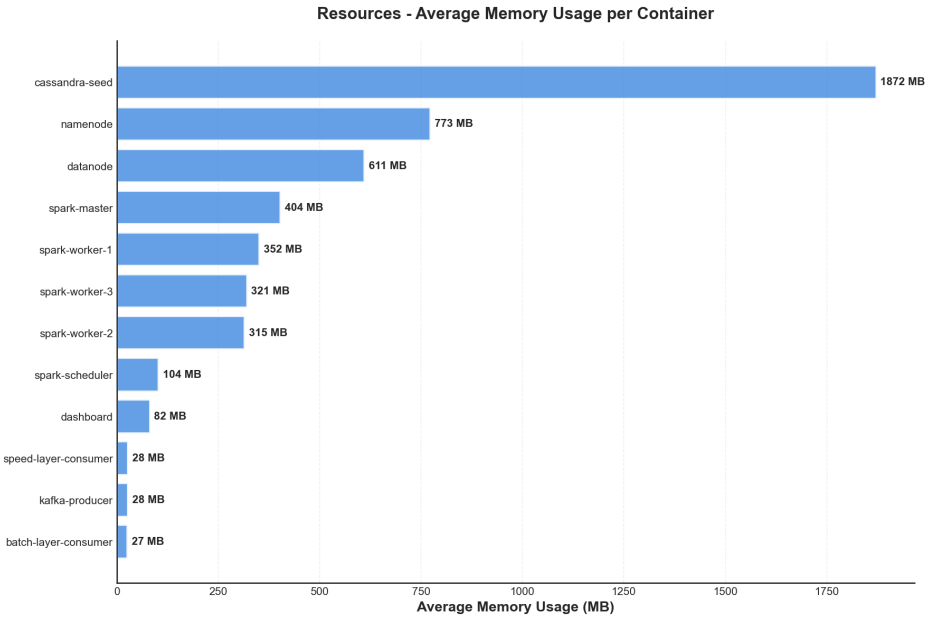
Aggregate analysis by layer reveals that the Batch Layer results as the most resource-intensive, with average CPU usage of 35% and peaks up to 170%.

This pattern is expected: Spark workers and Hadoop components (NameNode, DataNode) execute computationally intensive analysis operations when processing new batches. The Speed Layer shows a more contained profile, with average CPU of 15% and peaks at 60%, consistent with its nature of continuous but less intensive stream processing. The Serving Layer maintains stable usage around 40% average CPU, with peaks at 200% during Cassandra compaction operations.



The implementation of resource limits proved effective in preventing starvation situations. Spark workers, limited to 1.5 cores each, show peaks contained within 150%, while NameNode and DataNode, also limited to 1.5 cores, rarely exceed their thresholds. Cassandra, with a 2-core limit, occasionally reaches 200% during compaction phases, but this did not impact Speed Layer performance thanks to the effective isolation guaranteed by containerization.

From a memory perspective, Cassandra dominates with average usage of 680 MB, followed by the NameNode with 470 MB. Spark Workers maintain contained footprints around 300 MB each, while Kafka consumers use minimal resources (30-50 MB). This distribution confirms that memory allocation design (1GB for Cassandra heap, 512MB for HDFS) is appropriate for the system workload.



An interesting aspect emerging from the analysis is the correlation between CPU usage

and batch arrival: during Spark processing phases, a coordinated spike in CPU of the three workers and scheduler is observed, followed by a DataNode peak for writing results to HDFS. This pattern confirms the correct functioning of the batch pipeline and the absence of bottlenecks in distributed processing.

6.5 Overall Assessment

The overall testing results confirm the validity of the adopted architectural choices. The system proves capable of effectively managing simultaneous realtime and batch data processing, maintaining contained latencies and adequate throughput. Isolation between layers was empirically verified through the lack of correlation between Batch Layer activities and Speed Layer performance, thus ensuring that offline processing does not impact system responsiveness.

The observed stability in batch sizes and buffering times indicates that the Producer generates data regularly and that the system is correctly sized for the sustained load. Good management of micro-batches on HDFS and efficient management of computational resources confirm that the architecture is scalable and ready for higher workloads.

Excellent Speed Layer performance, with 95% of requests served in less than 30 milliseconds, demonstrates that the system is suitable for applications requiring near-instant feedback, such as anomaly detection in financial data. At the same time, the Batch Layer ensures the accuracy of historical analyses through complete data processing over large time windows.

Efficient resource usage, with average CPUs well below configured limits and appropriately allocated memory, suggests that the system could handle significantly higher workloads before encountering hardware limitations. This headroom represents an important safety margin to handle temporary traffic spikes or to accommodate future system growth.

Chapter 7

Conclusion

The present report documented the conception and implementation of a Lambda Architecture optimized for the financial domain, demonstrating how such a paradigm can effectively respond to the challenges posed by extreme cryptocurrency volatility. The success of the project lies not only in achieving performance targets but in validating a system capable of self-adapting and ensuring data integrity in every phase.

7.1 Synthesis of Project Value and Innovation

The implemented approach solved the classic trade-off between accuracy and timeliness. While the **Batch Layer** ensures a "single version of truth" through deep historical analysis and cleaning based on the 3-Sigma Rule, the **Speed Layer** introduces an element of adaptive intelligence. The use of the **EMA** (Exponential Moving Average) filter represents a critical strength: it allows the system not to be a simple static executor but to evolve in real-time following market trends. Performances recorded by the **Serving Layer** on Apache Cassandra — with write times in the order of microseconds — ensure that the end user interacts with an always-updated Dashboard, eliminating any perception of operational delay and making the monitoring experience fluid and professional.

7.2 Scientific Validation of Isolation

One of the most significant and convincing results of the entire work is the empirical proof of isolation between layers. The correlation coefficient close to zero (0.025) between Batch Layer load and Speed Layer latency is not just a statistic, but technical confirmation of the architecture's resilience. This ensures that, even during heavy calculation phases on large volumes of historical data, the system's reaction capacity to new price anomalies remains instant and undisturbed.

7.3 Scalability and Future Vision

Detailed analysis of computational resources revealed an extremely "healthy" architecture, oversized relative to the current load. The ample *headroom* detected on CPU and memory indicates that the system is ready to scale horizontally, welcoming a significantly larger number of financial assets without requiring infrastructural redesign.

Containerization via Docker further guarantees total portability, making the project ready for deployment in production cloud environments.

7.4 Final Considerations

In conclusion, the Lambda Architecture proved to be not only a valid theoretical choice but a superior practical solution for high-frequency financial monitoring. The clear separation between processing flows, although complex to implement, offers unparalleled advantages in terms of resilience and precision. The realized system stands as a solid and reliable basis for the development of advanced decision support tools in the financial Big Data market.