Real-Time Finance: Batch NBP Rates + Crypto Trade Stream

Adam Kaniasty, Igor Kołodziej

January 2026

Contents

1 Project goal and scope

The primary goal of the project is to build a complete batch-and-streaming Big Data pipeline that combines daily FX rates published by the National Bank of Poland (NBP) with a live market data stream from the Binance exchange (final configuration: the BTCUSDT pair). The system should support both bulk loading and historical analysis (batch layer) and the creation of aggregated facts for fast reads in the serving layer.

The project uses a classic Big Data stack: Apache NiFi orchestrates data flows and integrates external APIs, Apache Kafka serves as an optional intermediate buffer, Apache HDFS is the main storage layer (raw/curated/ aggregated), Apache Spark performs ETL and batch analytics, Apache Hive acts as a catalog and SQL layer over Parquet data, and Apache HBase is the serving database. The entire stack runs in a Docker/Docker Compose environment.

Functionally, the system implements multiple processing levels. The curated layer produces two structured tables: `fx_daily` (daily NBP rates) and `trades_agg_hourly` (hourly crypto aggregates). On top of these, Spark computes daily returns, daily and monthly aggregated views, and a 63-day rolling correlation between BTC and the USD/PLN exchange rate. The most important facts (e.g., closing prices, volumes, and, with longer history, returns and correlations) are published to the `finance:facts_daily` table in HBase, enabling fast reads by symbol and date.

Repeatability and ease of reproduction were key design requirements. All steps – from HDFS directory initialization, through Spark jobs, Hive schema setup, to loading facts into HBase – are wrapped in shell scripts. An additional end-to-end script (`tests/service/test-full-pipeline.sh`) runs the entire pipeline for a given date and provides a simple way to demonstrate the solution.

2 Data sources and their characteristics

Below we summarize key parameters of the data sources used in the project. The counts are shown for the reference run on `2026-01-05` (for the crypto stream, the counts depend on the length of the NiFi collection window).

Source	Type	Raw layer (HDFS)	Format	Ingestion frequency	Attributes (selected)	Row count (sample)
Binance WebSocket (aggTrade) for BTCUSDT	streaming	`/data/finance/raw/crypto-trades/date=YYYY-MM-DD/hour=HH/*.csv`	JSON → CSV	continuous stream	`symbol`, `price`, `quantity`, `event_time`	`2026-01-05`: 24 CSV files, 781 event records (`aggTrade`)
NBP API (Table A – average rates)	batch (daily)	`/data/finance/raw/nbp/date=YYYY-MM-DD/nbp_rate_{CODE}_{YYYYMMDD}.json`	JSON	once per day (business days)	`code`, `currency`, `mid` (+ rate date)	`2026-01-05`: 32 files/records for currencies (count depends on table/day)

For the same data sample, the processed layers (curated/aggregated) can be verified with simple SQL queries against Hive tables (e.g., `SELECT ... LIMIT 5`; when needed, `COUNT(*)`, although it is more expensive). Logs and sample outputs from end-to-end runs are stored under `artifacts/evidence_DATA_TIMESTAMP/`, generated by `./tests/service/capture-evidence.sh`. Reference run example: `artifacts/evidence_2026-01-05_20260106_100653/` (e.g., `pipeline.log`, `hive_checks.log`, `hbase_checks.log`).

To unambiguously verify data for a given date in Hive, we use date filters (the column `‘date‘` is a Hive keyword and must be escaped). Example checks for the reference run `2026-01-05`:

```
SELECT count(*) AS fx_daily_rows FROM finance.fx_daily WHERE fx_date='2026-01-05'; -- 32
SELECT count(*) AS trades_rows FROM finance.trades_agg_hourly WHERE `date`='2026-01-05'; -- 2
SELECT count(*) AS crypto_daily_rows FROM finance.crypto_daily WHERE `date`='2026-01-05'; -- 1
SELECT * FROM finance.corr_btc_usdpln_63d WHERE `date`='2026-01-05'; -- NULL (no 63-day history)
```

Note that in a demo environment the crypto stream is typically collected only for a short window within a single day, so history-dependent columns (`ret_1d`, 63-day correlation) may remain NULL. This behavior is expected and consistent with the metric definitions.

2.1 Streaming data – Binance

Our primary streaming source is the public Binance `aggTrade` transaction stream for the BTCUSDT pair. Unlike the 24-hour ticker stream, `aggTrade` carries actual trade events (price + quantity + trade time), which makes batch aggregates (volume, trade count, OHLC) semantically correct.

On the NiFi side, Binance data arrives as JSON through a WebSocket processor and is validated and normalized into a strict record schema. Depending on the variant, records are either grouped with `MergeRecord` and written directly to HDFS, or buffered in Kafka via `PublishKafkaRecord` and `ConsumeKafkaRecord`. In the raw layer, HDFS stores CSV files under `/data/finance/raw/crypto-trades/date=<YYYY-MM-DD>/hour=<HH>/...`, partitioned by date and hour. Each file contains a series of records (CSV with header) with fields such as `symbol`, `price`, `quantity`, and `event_time`.

The stream is potentially continuous, but in lab conditions the pipeline runs in time windows of several to a dozen minutes. To reduce resource usage, the final configuration ingests only BTCUSDT. After ETL, the `trades_agg_hourly` table retains one aggregated observation per (symbol, date, hour), which serves as the base for downstream analysis.

2.2 Batch data – NBP

The second core data source is the public NBP API, which provides daily exchange rate tables (the project uses Table A – average rates). NiFi periodically calls the HTTP endpoint for the latest rates table and then splits it into individual records for each currency.

Each NBP record has a simple JSON structure: `currency` (currency name), `code` (three-letter code, e.g., USD, EUR), and `mid` (average rate versus PLN). NiFi also adds a date attribute based on API response metadata. Files are written to HDFS under `/data/finance/raw/nbp/date=<YYYY-MM-DD>/` as individual JSON files per currency and day, e.g., `nbp_rate_USD_20251204.json`.

The NBP API is updated once per day (business days), and a single historical query is limited to 93 days. The architecture plan considered windowed historical downloads ($\leq$ 93 days), but the current implementation focuses on daily ingestion of the latest tables and their persistence in HDFS. A typical table contains several dozen currencies, so a single date produces dozens of JSON files in the raw layer; after processing, `fx_daily` contains one observation per (date, currency code).

2.3 Optional sources (Kraken, Stooq)

In the original project plan we considered extending the system with additional sources: a transaction stream from Kraken (as a fallback in case of Binance connectivity issues) and historical stock index quotes from Stooq (e.g., WIG, WIG20) in CSV/ZIP format. Due to time constraints and to refine the primary FX–crypto path, we focused on the two key sources (Binance + NBP) and left Kraken/Stooq integration as a potential future enhancement, to be listed in the "Future work" section.

3 System architecture and tools

The architecture is based on a clear layering model that separates responsibilities for ingest, durable storage, batch processing/ETL, SQL access, and publication of aggregated facts to a NoSQL serving layer. The core is an HDFS cluster with three logical data tiers: raw, curated, and aggregated. Each tier has a fixed directory structure described in the documentation and initialization scripts (`hdfs/create-directories.sh`).

Ingest is handled by Apache NiFi, which connects to external sources (Binance, NBP), performs basic validation and transformation, and writes records to HDFS. In the extended variant, data passes through Apache Kafka as an intermediate buffer that improves resilience to connection issues. In both cases, the result is raw CSV/JSON files under `/data/finance/raw/crypto-trades/...` and `/data/finance/raw/nbp/...`.

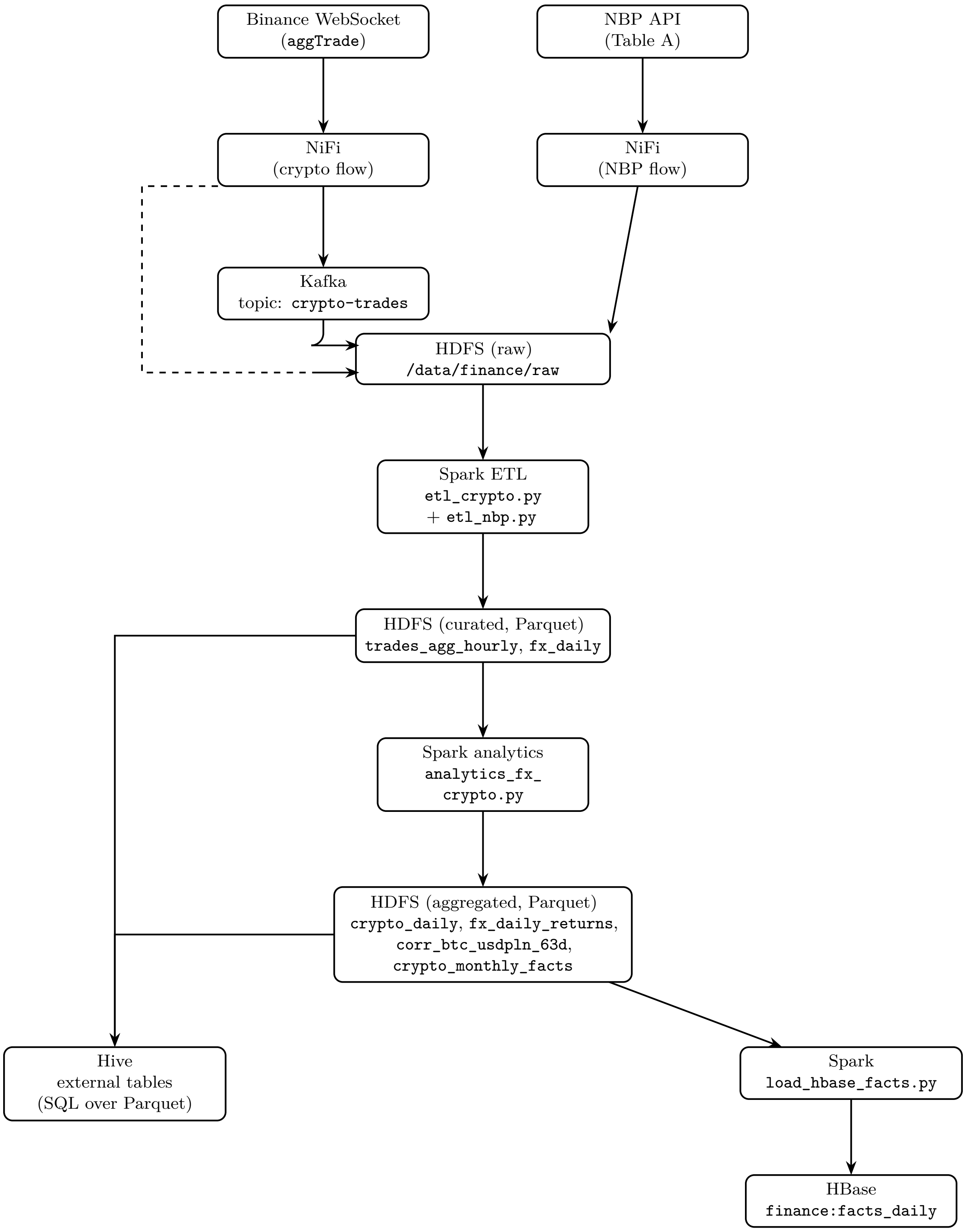

Figure 1: Architecture and data flow (with/without Kafka variants).

The raw layer is processed by Apache Spark, invoked in batch via `spark/run-etl.sh` and `spark/run-analytics.sh`. The first script builds the curated layer: normalization, deduplication (for NBP), partitioning (`year/month` for NBP and `date/hour` for crypto), and writing Parquet tables (`fx_daily`, `trades_agg_hourly`). The second Spark stage performs analytics – computing returns, correlations, and monthly metrics – and writes results under `/data/finance/aggregated/...`. An additional script `spark/run-check-analytics.sh` performs business-level sanity checks (returns range, positive prices, correlation range).

Apache Hive serves as a catalog layer over Parquet data in HDFS. The scripts `hive/init-tables.sh` and `hive/init-views.sh` create external tables in the `finance` database mapping:

- the curated layer: `fx_daily`, `trades_agg_hourly`,
- the aggregated layer: `fx_daily_returns`, `crypto_daily`; `corr_btc_usdpln_63d` and `crypto_monthly_facts`.

This makes both ETL outputs and analytical results accessible via SQL, facilitating manual exploration and ad-hoc queries.

The serving layer is implemented in Apache HBase. The `finance:facts_daily` table, initialized by `hbase/init-hbase.sh`, stores selected daily facts in a format optimized for fast key-based reads (symbol + date). Loading data from the aggregated layer into HBase is performed by a Spark job (`spark/load\_hbase\_facts.py`) invoked via `hbase/load-facts.sh`. Each full pipeline run ends with a control scan of the HBase table, automated in `tests/service/test-full-pipeline.sh`.

The entire stack runs under Docker/Docker Compose, allowing the configuration to be reproduced on different machines without manual installation of each component. Key services (HDFS, Hive, Spark, NiFi, Kafka, HBase) are defined as separate containers in `docker-compose.yml`. Hive serves as a metadata and SQL layer over Parquet data in HDFS and runs in Hive-on-MR (local MapReduce), which supports aggregate queries (e.g., `SELECT COUNT(*)`, `GROUP BY`) for demonstrational data sizes. Regardless of this, all ETL and batch analytics are implemented in Spark as intended by the architecture.

4 Data ingestion, processing, and storage

4.1 NiFi + Kafka → HDFS (raw layer)

The raw layer is built exclusively by Apache NiFi and, in the extended variant, Apache Kafka. For the crypto stream, the main NiFi flow connects to the Binance WebSocket API (`aggTrade`), validates and transforms incoming JSON messages into a normalized record format (e.g., `ValidateRecord`, `UpdateRecord`), and then either groups records into minute batches (`MergeRecord`) and writes directly to HDFS (basic variant), or publishes them to a Kafka topic (`crypto-trades`), from which they are consumed and written to HDFS (buffered variant). The reference configuration uses the Kafka variant (backup `nifi/backups/crypto-flow-kafka-4.json`), which fulfills the buffering requirement. In both cases, raw crypto data lands under `/data/finance/raw/crypto-trades/date=<YYYY-MM-DD>/hour=<HH>/...` and is stored as CSV with headers.

For NBP, we use a batch flow: NiFi pulls the rates table (NBP API, Table A) via `InvokeHTTP`, extracts metadata and splits the rates array into individual elements (`EvaluateJsonPath` + `SplitJson`), maps fields (`code`, `currency`, `mid`), and writes each record as a JSON file to HDFS (`PutHDFS (NBP)`) under `/data/finance/raw/nbp/date=<YYYY-MM-DD>/` with names like `nbp_rate_<CODE>_<YYYYMMDD>.json`. Unlike the crypto stream, we do not use a Kafka buffer for NBP – the data is small and daily, so a direct HDFS write is sufficient.

NiFi flows are designed to be safe on re-runs: HDFS writes use "replace" mode, and pipeline idempotency is ensured by the ETL layer (partition overwrite; for NBP, additional deduplication by date+code). The choice between Kafka and non-Kafka variants affects only the crypto stream (backups `nifi/backups/crypto-flow.json` and `nifi/backups/crypto-flow-kafka-4.json`) and does not change the final HDFS data structure.

4.2 Spark ETL → HDFS (curated layer)

The curated layer transforms raw CSV/JSON files from HDFS into consistent Parquet tables (with deduplication for NBP) that can be used directly in analysis and exposed via Hive. The ETL process is implemented in Apache Spark and run via `./spark/run-etl.sh`, which invokes two jobs: `etl_nbp.py` (NBP) and `etl_crypto.py` (Binance).

For NBP, `etl_nbp.py` reads JSON files from `/data/finance/raw/nbp/date=<DATA>/nbp_rate_???_*.json` (matching any three-character currency code in the file name). The file name contains the rate date in `YYYYMMDD` format, extracted via regex and converted to DATE (`fx_date`). Key fields (`code`, `currency`, `mid`) are selected, rates are cast to DOUBLE, `load_ts` is added, and partitioning fields `year` and `month` are derived from the date. A second deduplication step by (date, currency code) prevents multiple loads of the same table. The result is written as Parquet to `/data/finance/curated/nbp/fx_daily` using "dynamic partition overwrite". Because we partition by (`year`, `month`), a single-date refresh overwrites the entire month partition; with longer history, ETL should process full months or extend partitioning by day.

Similarly, `etl_crypto.py` processes raw CSV files under `/data/finance/raw/crypto-trades/date=<DATA>/hour=<HOUR>/*.csv`. Spark reads data with headers, casts numeric fields (`price`, `quantity`) to DOUBLE, and converts `event_time` (milliseconds since epoch) into TIMESTAMP (`event_ts`). Records missing key fields are dropped, and `date` (calendar day), `hour`, and `hour_start_ts` (hour start) are derived from the timestamp. Aggregation at the (symbol, date, hour) level computes `price_avg`, approximate 95th percentile (`price_p95`), hourly high/low (`price_high`, `price_low`), base volume (`volume_base` = sum of quantity), quote volume (`volume_quote` = sum of `quantity * price`), trade count (`trade_count`, number of `aggTrade` events), and open/close prices (`price_open`, `price_close`) using `min_by`/`max_by` over event time. The earliest and latest timestamps within the hour (`ts_min`, `ts_max`) and `load_ts` are also recorded.

Aggregated data is written as Parquet to `/data/finance/curated/crypto/trades_agg_hourly`, partitioned by `date` (STRING) and `hour` (INT). As with NBP ETL, we use dynamic partition overwrite, so re-running ETL for a given date/hour refreshes data rather than duplicating it. The output is two stable, well-defined curated tables (`fx_daily`, `trades_agg_hourly`) that serve as the single source of truth for downstream analytics.

4.3 Hive over the curated layer

To enable SQL access to the curated layer and easier exploration from client tools (e.g., Beeline), Hive external tables are defined over the Parquet files in HDFS. Their definitions are in `hive/init_tables.sql`, and database/table initialization is automated by `./hive/init-tables.sh`.

Two core external tables are created in the `finance` database:

- `finance.fx_daily` pointing to `/data/finance/curated/nbp/fx_daily`, with schema matching the NBP ETL output: `fx_date` (DATE), `code` (STRING), `currency` (STRING), `mid` (DOUBLE), `load_ts` (TIMESTAMP), and partitions `year` (INT) and `month` (INT).
- `finance.trades_agg_hourly` pointing to `/data/finance/curated/crypto/trades_agg_hourly`. The schema includes `symbol` (STRING), timestamps (`hour_start_ts`, `ts_min`, `ts_max`), price fields (`price_open`, `price_close`, `price_high`, `price_low`, `price_avg`, `price_p95`), volume fields (`volume_base`, `volume_quote`, `trade_count`), `load_ts`, and partitions `date` (STRING) and `hour` (INT).

After creating the tables, `init-tables.sh` runs `MSCK REPAIR TABLE` for each one, which automatically discovers existing partition directories in HDFS and registers them in the Hive metastore. As a result, partitions created by Spark ETL become visible in Hive without manual registration. Correctness and data access are verified by simple queries such as `SELECT * FROM finance.fx_daily LIMIT 5` and `SELECT * FROM finance.trades_agg_hourly LIMIT 5`, executed as part of the evidence script `tests/service/capture-evidence.sh` (file `hive_checks.log`) or manually during verification. In practice, Hive acts as a catalog layer over curated data, while all expensive aggregation operations are delegated to Spark.

5 Data analysis and batch views (Spark + Hive)

5.1 Daily views – FX and crypto

After building the curated layer, the next step is to compute daily views that serve as the basis for further aggregation and correlation. These operations are implemented in a single Spark job (`spark/analytics_fx_crypto.py`) run via `./spark/run-analytics.sh`. The script performs four analytical steps; the first two produce the daily FX and crypto views.

For FX rates, we compute `fx_daily_returns`. Based on the curated `fx_daily` table (including `fx_date`, `code`, `mid`), we define a window ordered by date within each currency (`code`). We then compute the previous-day rate (`lag(mid)`) and the daily return `ret_1d = mid / lag(mid) - 1`. The result is written to `/data/finance/aggregated/daily/fx_daily_returns` as Parquet partitioned by `year` and `month` derived from `fx_date`. For the first day in each currency series, the return remains NULL.

For crypto data, an analogous daily view `crypto_daily` is computed from the hourly aggregates in `trades_agg_hourly`. The first step is to determine the daily close `close` for each (symbol, date) pair, defined as `price_close` from the last hourly observation on that day (by `ts_max`). Then, as for FX, we compute `ret_1d = close / lag(close) - 1` within each symbol. The `crypto_daily` view is written to `/data/finance/aggregated/daily/crypto_daily` and partitioned by `symbol` to facilitate downstream grouping.

5.2 FX–crypto correlations

Using daily returns for FX and crypto, we compute a 63-day rolling correlation between the BTC market and the USD/PLN exchange rate. From `crypto_daily` we select only `BTCUSDT`, and from `fx_daily_returns` we select the `USD` currency code. The series are joined by date (`date`/`fx_date`) to build daily return vectors `ret_btc` and `ret_usdpln` for common days.

The resulting dataset is processed with a 63-day rolling window (in practice: 63 consecutive observations, `rowsBetween(-62, 0)`). For each date, we compute sums and sum of squares for `x` and `y`, along with the sum of `x*y`, and then compute the standard Pearson correlation between `ret_btc` and `ret_usdpln`. The resulting view `corr_btc_usdpln_63d` contains the date and correlation value and is stored under `/data/finance/aggregated/daily/corr_btc_usdpln_63d`. For early days with fewer than two observations, the correlation remains NULL – a full 63-day horizon is needed for a stable interpretation.

5.3 Monthly views

The final aggregation level in Spark is the monthly view `crypto_monthly_facts`, computed from `crypto_daily`. For each (symbol, year, month), we compute basic metrics describing the instrument in that month: average daily return (`avg_ret_1d`), volatility measured as the standard deviation of daily returns (`vol_ret`), the number of days with available returns (`days_ret`), and the last available closing price in the month (`last_close`).

`crypto_monthly_facts` is written as Parquet to `/data/finance/aggregated/monthly/crypto_monthly_facts` and partitioned by `symbol`. This format simplifies filtering and joining with other sources (e.g., FX correlations) and is a natural input for the serving layer if we choose to publish monthly facts instead of daily.

5.4 Hive over the aggregated layer

As in the curated layer, a set of Hive external tables is defined over aggregated data under `/data/finance/aggregated/...` to allow SQL inspection. The definitions are in `hive/init_views.sql`, and the creation and basic verification are handled by `./hive/init-views.sh`.

Four external tables are created in the `finance` database:

- `finance.fx_daily_returns` over `/data/finance/aggregated/daily/fx_daily_returns` (columns: `fx_date`, `code`, `mid`, `ret_1d` and partitions `year`, `month`),
- `finance.crypto_daily` over `/data/finance/aggregated/daily/crypto_daily` (columns: `date`, `ts_max`, `close`, `ret_1d` and partition `symbol`),

- `finance.corr_btc_usdpln_63d` over `/data/finance/aggregated/daily/corr_btc_usdpln_63d` (columns: `date`, `corr_btc_usdpln_63d`),
- `finance.crypto_monthly_facts` over `/data/finance/aggregated/monthly/crypto_monthly_facts` (columns: `year`, `month`, `avg_ret_1d`, `vol_ret`, `days_ret`, `last_close` and partition `symbol`).

After creation, `init-views.sh` runs `MSCK REPAIR TABLE` for partitioned tables, which registers existing partitions in the Hive metastore. It then runs simple verification queries (`SELECT * FROM ... LIMIT 5`) for each table to confirm that the analytical views are available and consistent with Spark outputs. This makes the analytical layer visible both to Spark and to SQL tools via Hive, while computation remains in Spark.

5.5 Sanity checks and tests

Because the project values not just running analytics but also validating its reasonableness, we implemented an additional sanity check script for the aggregated layer: `spark/check_analytics_sanity.py`, executed by `./spark/run-check-analytics.sh`. The script reads all four analytical views:

- `fx_daily_returns`
- `crypto_daily`
- `corr_btc_usdpln_63d`
- `crypto_monthly_facts`

and performs a set of simple but effective quality checks.

Among other things, it verifies that each view contains a non-zero number of rows (which in practice indicates that ETL and analytics ran correctly), that daily and monthly returns are not absurdly large (e.g., $|\text{ret\_1d}| > 10$), that closing prices are positive, and that volatility (`vol_ret`) is non-negative. For the correlation table it checks that values fall in the expected range [-1, 1] (with a small numerical margin). If any check fails, the script exits with an error and prints a corresponding message, enabling quick identification of data or logic issues. Sanity checks are included in the full end-to-end test (`tests/service/test-full-pipeline.sh`), so every pipeline run ends with an automatic validation of analytical outputs.

6 Serving layer – HBase

The final pipeline layer is the serving layer implemented in Apache HBase. Its purpose is to expose selected aggregated metrics in a structure optimized for fast key-based access (instrument symbol + date), which enables lightweight client applications, dashboards, or ad-hoc analysis without rerunning Spark jobs.

The central element is the `finance:facts_daily` table, created and reset by `hbase/init-hbase.sh`. The table is in the `finance` namespace, and the row key format is `salt|symbol|yyyyMMdd`, e.g., `2|BTCUSDT|20260105`. The `salt` prefix (digit 0–9) is computed as `abs(hash(symbol, date)) % 10` to spread keys and reduce potential hot regions. The second part is the Binance symbol (BTCUSDT), and the third is the date in `yyyyMMdd` format.

The table has four column families: `metrics`, `price`, `volume`, and `correlation`. In the current version we populate selected columns:

- in `price`, the daily close as `price:avg` (from `crypto_daily`),
- in `volume`, the daily base volume as `volume:sum` (sum of `quantity`) and trade count as `volume:count` (sum of `trade_count` from `trades_agg_hourly`),
- in `metrics`, the daily return as `metrics:ret_1d` for days where the return is defined (from the second observation onward),

- in `correlation`, the column `correlation:btc_usdpln_63d`, which can store the 63-day BTC correlation with USD/PLN for the corresponding dates.

Loading data into HBase is performed by a dedicated Spark job `spark/load_hbase_facts.py`, run via `./hbase/load-facts.sh`. The job reads aggregated views (daily crypto facts from `crypto_daily`, volumes from `trades_agg_hourly`, and, when sufficient history is available, correlations from `corr_btc_usdpln_63d`), joins them by symbol and date, and then builds row keys in the format described above. Spark generates HBase Shell `put` commands and writes them to `/data/finance/hbase/facts_daily_puts` in HDFS as a plain text file.

The `load-facts.sh` script first ensures the `finance:facts_daily` table exists (calling `init-hbase.sh`), then runs the Spark job to generate `put` commands, and finally streams them from HDFS into HBase Shell via `hdfs dfs -cat ... | hbase shell`. This avoids direct Spark-HBase API integration and relies on a simple text bridge. After loading, the script performs a control scan:

```
scan 'finance:facts_daily', {LIMIT => 10}
```

In a typical run for the reference date (e.g., `2026-01-05`), the scan shows a row for BTCUSDT with populated `price:avg`, `volume:sum`, `volume:count`, and (for days with defined returns) `metrics:ret_1d`. This confirms correct data flow from the raw layer (NiFi/HDFS), through Spark ETL and analytics, to the HBase serving layer. Additional scripts `hbase/verify-hbase.sh` and `hbase/test-hbase.sh` independently verify table structure, `put/get/scan` operations, and correctness across column families.

7 Functional tests

The project emphasizes test coverage for all key pipeline components (NiFi, Kafka, HDFS, Spark, Hive, HBase) via shell scripts. This allows running both individual layer tests and full end-to-end tests, with repeatable results documented in logs.

7.1 Ingest layer tests – NiFi + HDFS

These tests verify that NiFi flows correctly write Binance and NBP data into the appropriate HDFS directories. Specifically:

- `tests/service/test-crypto-trades.sh DATE [HOUR]` checks for the presence and contents of crypto CSV files in `/data/finance/raw/crypto-trades/date=DATE/hour=HOUR/`. If `HOUR` is not provided, the script automatically selects the latest available hour for the day. It prints the path, file list, and file contents (limited to the first lines). Example run:

```
./tests/service/test-crypto-trades.sh 2026-01-05 12
```

Output snippet:

```
Reading CSV files from HDFS:
Path: /data/finance/raw/crypto-trades/date=2026-01-05/hour=12/

Files in directory:
-rw-r--r-- 3 nifi supergroup 1812 ... /crypto-trades/date=2026-01-05/hour=12/05-12-44.csv
...
```

which confirms that NiFi correctly writes CSV files to the expected location and format.

- `tests/service/test-nbp-rates.sh DATE` similarly checks `/data/finance/raw/nbp/date=DATE/` and lists JSON files with NBP rates. Example run:

```
./tests/service/test-nbp-rates.sh 2026-01-05
```

Output snippet:

```
Reading NBP exchange rate files from HDFS:
Path: /data/finance/raw/nbp/date=2026-01-05/
```

```
Files in directory:
-rw-r--r-- 3 nifi supergroup 59 ... /nbp_rate_USD_20260105.json
...
```

which confirms that NBP data is available in HDFS as individual JSON files for each currency.

7.2 Kafka tests

The script `tests/service/test-kafka.sh` runs integration tests for Kafka:

- creating and listing temporary topics,
- producing a few messages and consuming them,
- validating message contents,
- checking broker reachability from the host (port 9092).

Example output snippet:

```
Kafka Comprehensive Tests
...
checkmarkPASS: Create topic: test-topic-...
checkmarkPASS: Produce messages to topic
checkmarkPASS: Consume messages (found 4 messages)
checkmarkPASS: Message content verification
checkmarkPASS: External connectivity (port 9092 accessible)
...
Passed: 8
Failed: 0
All tests passed!
```

which confirms correct Kafka configuration and integration with the test container.

7.3 Spark ETL and analytics tests

Spark ETL and analytics are tested on two levels:

- via the full pipeline test (see Section 7.5), which runs `./spark/run-etl.sh`, `./spark/run-analytics.sh`, and `./spark/run-check-analytics.sh`,
- via direct execution of analytics sanity checks:

```
./spark/run-check-analytics.sh
```

The script validates counts and value ranges across the four analytical views:

- `fx_daily_returns`
- `crypto_daily`
- `corr_btc_usdpln_63d`
- `crypto_monthly_facts`

Summary log snippet:

```
Running analytics sanity checks...
[check] fx_daily_returns rows: 131
[check] crypto_daily rows: 1
[check] corr_btc_usdpln_63d rows: 1
[check] crypto_monthly_facts rows: 1
[check] All analytics sanity checks passed.
Analytics sanity checks completed successfully.
```

which confirms that analytical views are non-empty and satisfy basic quality constraints (e.g., no extreme returns, positive prices, and correlations within expected bounds when available).

7.4 Hive tests

Hive tests are integrated into the initialization scripts:

- `./hive/init-tables.sh`:
  - creates the `finance` database and external tables `fx_daily` and `trades_agg_hourly`,
  - runs `MSCK REPAIR TABLE` for both tables,
  - verifies table existence with `SHOW TABLES` and runs simple `SELECT ... LIMIT 5` queries.
- `./hive/init-views.sh`:
  - creates external tables over analytical views:
    `fx_daily_returns`, `crypto_daily`,
    `corr_btc_usdpln_63d`, `crypto_monthly_facts`,
  - runs `MSCK REPAIR TABLE` for partitioned tables,
  - executes control `SELECT ... LIMIT 5` queries for each table in the `finance` database.

Running both scripts without errors and seeing the expected tables in `SHOW TABLES IN finance` is treated as evidence of correct Hive integration with curated and aggregated layers. Verification includes built-in data samples (`SELECT ... LIMIT 5`); in this lightweight stack, Hive-on-MR aggregates (e.g., `COUNT(*)`, `GROUP BY`) may work for demo-sized data but can be unreliable for heavier queries. Nevertheless, all ETL logic and business analytics remain in Spark.

7.5 HBase tests and end-to-end test

The HBase layer has dedicated tests:

- `hbase/verify-hbase.sh`:
  - checks HBase Shell availability,
  - verifies the `finance` namespace and the `finance:facts_daily` table,
  - checks the presence of all column families (`metrics`, `price`, `volume`, `correlation`),
  - performs a test write/read of a sample row and then deletes it.
- `hbase/test-hbase.sh`:
  - runs a series of write/read tests for multiple row keys,
  - verifies access to individual column families,
  - confirms table scanning capability.

The most important test is the full end-to-end pipeline test `tests/service/test-full-pipeline.sh YYYY-MM-DD`, which ties all layers together. For the reference date `2026-01-05`, it executes:

1. `./spark/run-etl.sh -date 2026-01-05` – build the curated layer,
2. `./spark/run-analytics.sh` – compute analytical views,
3. `./spark/run-check-analytics.sh` – sanity checks,
4. `./hive/init-tables.sh` and `./hive/init-views.sh` – initialize Hive tables,
5. `./hbase/load-facts.sh -date 2026-01-05` – publish facts to HBase.

Final log snippet from a sample run:

```
Step 3: Streaming commands from HDFS into HBase shell ...
...
Step 4: Verifying a few rows in HBase ...
scan 'finance:facts_daily', {LIMIT => 10}
ROW COLUMN+CELL
 2|BTCUSDT|20260105 column=price:avg, ... value=92840.54
 2|BTCUSDT|20260105 column=volume:count, ... value=781
 2|BTCUSDT|20260105 column=volume:sum, ... value=19.974790000000013
1 row(s)
...
Full pipeline completed successfully for 2026-01-05
```

which unambiguously confirms that raw data (NiFi/HDFS) was correctly processed by Spark ETL and analytics, registered in Hive, and finally stored in HBase in the expected form. This test is the primary evidence of system correctness and can be used as the key demonstration during project presentation.

8 Summary of the final solution

In summary, the project delivered a complete, multi-layer Big Data pipeline that meets the plan's goals: integrating batch NBP FX rates with continuous Binance data, persisting them in HDFS, processing them in Spark, exposing them via Hive, and publishing selected facts to HBase. A key strength of the solution is that every step – from HDFS directory setup, through ETL and analytics, to Hive and HBase initialization – is scripted, enabling reproducible runs and easy demonstration (`tests/service/test-full-pipeline.sh` runs the full pipeline for a chosen date).

The most important design choices include a consistent layer structure (raw/curated/aggregated/serving) with clear schema contracts and idempotent Spark ETL (dynamic partition overwrite; NBP deduplication). We also built analytical views aligned with business needs (returns, correlations, monthly metrics) and designed the HBase key and schema for fast access by symbol and date. An additional value is the sanity check layer, which automatically validates the reasonableness of analytical outputs and helps detect logic issues before results are published.

Several deliberate trade-offs were made. Due to the lack of a full YARN environment, Hive runs in local MR mode and is used primarily as a catalog and SQL layer over Parquet outputs (including simple aggregates for demo data sizes). All meaningful transformations and analytics are implemented in Spark, the primary processing engine. We also deferred integration with additional sources (Kraken, Stooq) to focus on a robust Binance + NBP pipeline. NiFi flow stability (e.g., behavior during WebSocket disconnects) was validated manually during configuration and testing.

It is also worth noting the final implementation decisions relative to the plan. The aggregated view/table names in code and in this report do not use the `v_` prefix (e.g., `fx_daily_returns`, `corr_btc_usdpln_63d`, `crypto_monthly_facts`), and the raw crypto CSV includes the full aggTrade payload plus a normalized subset of fields; the ETL layer uses only `symbol`, `price`, `quantity`, `event_time`. In HBase we ultimately populate `price:avg`, `volume:sum`, `volume:count`, `metrics:ret_1d`, and `correlation:btc_usdpln_63d`; the remaining metrics from the plan (e.g., `vol_21d`, additional price statistics) are a natural future extension.

Potential future work includes extending the analytical layer with additional metrics (e.g., 21-day volatility `vol_21d`, other correlation windows, risk indicators), adding automated scheduling for ETL and analytics (e.g., cron or a dedicated orchestrator), integrating with a visualization layer (dashboard over HBase), and expanding data sources to additional exchanges or equity indices. The current version is a solid foundation for these enhancements while meeting course requirements and demonstrating a full end-to-end Big Data data flow.