Architecture

Nikola Stojanovic Mert Baykal

December 2025

1 Architectural Design

This section describes the architecture of the Cryptocurrency Market Data Collection System from three complementary viewpoints: conceptual architecture, execution architecture and implementation architecture. Together, these views show how the system combines several architectural styles: pipe-and-filter for the ETL pipeline, a layered web architecture for the dashboard, and separate worker components for analysis and machine learning.

1.1 Conceptual Architecture

At the conceptual level, the system is organized into four main areas:

1. **External Data Source**
   The Binance public REST API is the only external provider of market data. It exposes endpoints for:
   - Exchange metadata (`/api/v3/exchangeInfo`) for all trading pairs.
   - 24h statistics (`/api/v3/ticker/24hr`) used as a liquidity proxy.
   - Historical klines (`/api/v3/klines`) for daily OHLCV candles.
2. **Pipeline + Filter System (ETL)**
   The core of the backend is a pipe-and-filter style data pipeline:
   - **Filter 1 – Symbol discovery and selection**
     Fetches metadata and 24h statistics from Binance, filters out delisted markets and unstable quote currencies, aggregates by base asset and selects the top N cryptocurrencies by liquidity.
   - **Filter 2 – Missing data detection**
     For each selected symbol, compares the existing records in the database with the desired historical window and creates tasks representing missing date ranges.
   - **Filter 3 – Data collection and storage**
     For each task, downloads all missing daily OHLCV candles, normalizes the data, fills any missing calendar days, and upserts the result into the database.

Conceptually, the filters are connected in sequence: the output of Filter 1 is the input of Filter 2, and the output of Filter 2 (tasks) is the input of Filter 3.

3. **Local Data Store (SQLite Database)**
   The pipeline persists its results in a local relational database. The main logical entities are:

   - **Cryptocurrencies / Symbols**: metadata for each tracked trading pair (symbol, base asset, quote asset, active flag).
   - **Daily Market Data**: normalized daily OHLCV candles and derived 24h metrics per symbol and date.
   - **Analysis Results**: additional tables that can store technical indicators and LSTM-based price predictions.

   The database is the single source of truth for the web dashboard and analysis components.

4. **Web Application and ML/Analysis Layer**
   On top of the stored data, the system provides:

   - A **Flask-based dashboard** which reads from the SQLite database and exposes HTML pages and JSON endpoints showing:
     - Overall statistics (number of symbols, data points, last update).
     - A ranked list of top cryptocurrencies by average or maximum volume.
     - Complete symbol list and metadata.
   - A **technical analysis component** which computes indicators (e.g., moving averages, volatility) from the historical OHLCV series and stores them back into the database or exposes them through the web interface.
   - An **LSTM price prediction component** which reads sequences of daily prices from the database, runs an LSTM model to predict future prices, and stores predictions as additional time series that can be visualized by the dashboard.

Overall, the conceptual architecture is a hybrid of several styles: a pipe-and-filter backend for data ingestion, a layered architecture (DATA SOURCE $\rightarrow$ PIPELINE $\rightarrow$ DATABASE $\rightarrow$ WEB/ML), and analytic components (technical indicators and LSTM) built on top of the shared data store.

In the final deployment, these logical components can be hosted as separate services: a web UI service (Flask dashboard), a data-ingestion service (ETL pipeline) and an analysis service (technical indicators and LSTM predictions). Each service can be containerized (e.g. Docker) and deployed independently, which supports a distributed, microservice-style architecture on top of the core pipe-and-filter and layered design.

1.2 Execution Architecture

The execution view describes the concrete runtime elements (processes, external services, and communication channels) when the system is running.

1. **Web Client (User Browser)**
   The user interacts with the system through a standard web browser. The browser sends HTTP requests to the Flask application and receives HTML pages, CSS, JavaScript and JSON data. All user interaction (e.g., viewing the dashboard, navigating to the symbols list) happens in the browser.

2. **Application Server / Container**
   On the host machine (a student laptop or a server) there is an application environment that runs multiple Python processes:

   - **Flask Web Application**
     Handles incoming HTTP requests from the browser, renders templates, and exposes JSON endpoints for charts and statistics. It communicates with the database using standard SQL queries.
   - **ETL Batch Worker (Pipeline + Filters 1–3)**
     A separate process (invoked via `python -m src.main` or a scheduled job) that executes the three filters sequentially. It:
     - Calls the Binance REST API to download metadata and klines.
     - Normalizes and stores data in the SQLite database.

     .
   - **Analysis Worker Process**
     Optionally, another process computes technical indicators and performs LSTM inference. It:
     - Reads historical OHLCV data from SQLite.
     - Computes indicators (e.g. moving averages, RSI) and/or runs the LSTM model.
     - Writes indicator values and predicted prices into dedicated tables.

     The web application then reads these derived values to display them to the user.

3. **SQLite Database**
   The database file (`crypto.db`) resides on the same machine as the application server. Both the web application and the worker processes open connections to this file using the SQLite driver. All reads and writes (symbols, historical candles, indicators, predictions) are performed through SQL queries. In a simple setup, only one worker writes at a time and the web application mainly performs read operations.

4. **Binance REST API**
   The ETL worker and analysis components use HTTPS to call the Binance REST API endpoints. No direct calls from the browser to Binance are required; all external communication is mediated by the backend processes. This keeps API access under control and avoids exposing Binance endpoints directly to users.

From an execution perspective, the architecture is loosely coupled: the web client depends only on the Flask application; the ETL and analysis workers can run it; and all components share data through the SQLite file.

Although in the current setup all processes may run on a single machine, the same structure can be deployed in a distributed way: the Flask web application, the ETL batch worker and the analysis worker can be packaged as individual Docker containers and run as separate microservices. They communicate via HTTP (for external data) and via the shared database, which satisfies the requirement for a distributed, containerized architecture.

1.3 Implementation Architecture

The implementation view describes how the software is decomposed into modules, packages and files inside the codebase.

- **Configuration and Infrastructure**
  - `config.py`
    Central place for configuration constants such as: Binance base URL, API endpoints, default quote currency, HTTP timeouts, number of top symbols, and years of history to download.
  - `db.py`
    Functions for opening and closing SQLite connections, initializing the schema, and executing prepared SQL statements. Loads SQL definitions from external `.sql` files (e.g., table creation and insert/upsert queries).
  - `http_client.py`
    Wrapper around the `requests` library that encapsulates all HTTP calls to Binance (`exchangeInfo`, `ticker/24hr`, `klines`), including error handling, timeouts and pagination logic.
- **Pipeline and Filters**
  - Package `filters/` containing:
    - `filter1.py`: symbol discovery and liquidity-based selection.
    - `filter2.py`: missing-data detection and task generation.
    - `filter3.py`: kline download, normalization, gap filling and insertion into the database.

  - `main.py` (or `pipeline.py`)
    Orchestrates the end-to-end ETL pipeline by calling the filters in sequence. This script can be scheduled as a batch job.

- **Web Application**
  - `webapp/app.py`
    Flask application entry point, defines routes such as: `/` (dashboard) and `/symbols` (complete symbol list). Uses the `db` module to run queries against SQLite.
  - `templates/`
    Jinja2 templates (`dashboard.html`, `symbols.html`) which render HTML using data returned by the Flask routes.
  - `static/`
    Static assets such as `style.css`, JavaScript files, and images. These files implement the visual design of the dashboard and the responsive layout.

- **Analysis and Machine Learning**
  - `analysis/technical_indicators.py`
    Contains functions to compute technical indicators (e.g., moving averages, volatility metrics) based on OHLCV data retrieved from `daily_market_data`.
  - `analysis/lstm_model.py`
    Contains the code for preparing time series windows, defining the LSTM model, training it (offline) and performing inference to generate future price predictions. Predictions are written back to the SQLite database as additional rows or tables.

The implementation architecture follows a layered and modular structure:

- The *infrastructure layer* (`config`, `db`, `http_client`) is reused by both the ETL pipeline and the web application.
- The *pipeline layer* (`filters` and `main`) is responsible for ingesting and maintaining historical market data.
- The *presentation layer* (`webapp`, templates, static files) exposes dashboards and views to the user.
- The *analysis layer* (`analysis` package) builds on top of the stored data to compute indicators and LSTM predictions.

This modular decomposition makes it easier to extend the system in future homework phases, for example by containerizing components or adding new analytical modules (e.g., on-chain metrics or sentiment analysis) without changing the core pipeline.

1.4 Applied Architectural Styles and Patterns

The project does not only mention architectural styles in abstract, but applies them concretely to the implemented system:

Pipe-and-filter. The ETL part of the system is implemented as a pipeline of three filters (`filter1.py`, `filter2.py`, `filter3.py`). Filter 1 discovers and ranks symbols using Binance metadata and 24h statistics; its output is a cleaned and ranked list of cryptocurrencies. Filter 2 receives this list, inspects the SQLite database and produces tasks that describe missing date ranges. Filter 3 consumes these tasks, calls the Binance kline endpoint, normalizes and fills the OHLCV data and writes the result back to the database. The data flows strictly in one direction through the filters, which is a direct application of the pipe-and-filter style.

Layered web architecture. The system follows a layered structure: *external data sources* (Binance REST API) $\rightarrow$ *ETL pipeline* $\rightarrow$ *SQLite data layer* $\rightarrow$ *web presentation and analysis layer*. The Flask web application never calls Binance directly; instead it depends only on the local database and on higher-level functions in the analysis layer. This separation of concerns between data acquisition, storage and presentation is how the layered architecture is applied.

Distributed / microservice-style processes. At runtime the system is split into separate processes: the Flask web server (user-facing UI), the ETL batch worker (pipeline with filters 1–3) and the analysis worker (technical indicators and LSTM prediction). Each process can be started independently and communicates only via HTTP (external Binance API) and the shared database file. This process-level decomposition corresponds to a microservice-style architecture at a small scale, even when the services run on the same machine.

Containerization. The Flask application, ETL worker and analysis worker can be packaged as separate Docker images that share a volume containing the SQLite file. In that setup, each container runs exactly one responsibility (web UI, data ingestion, analysis), and they can be deployed, scaled or updated independently. This is how containerization is applied on top of the logical and execution structures described in the previous sections.

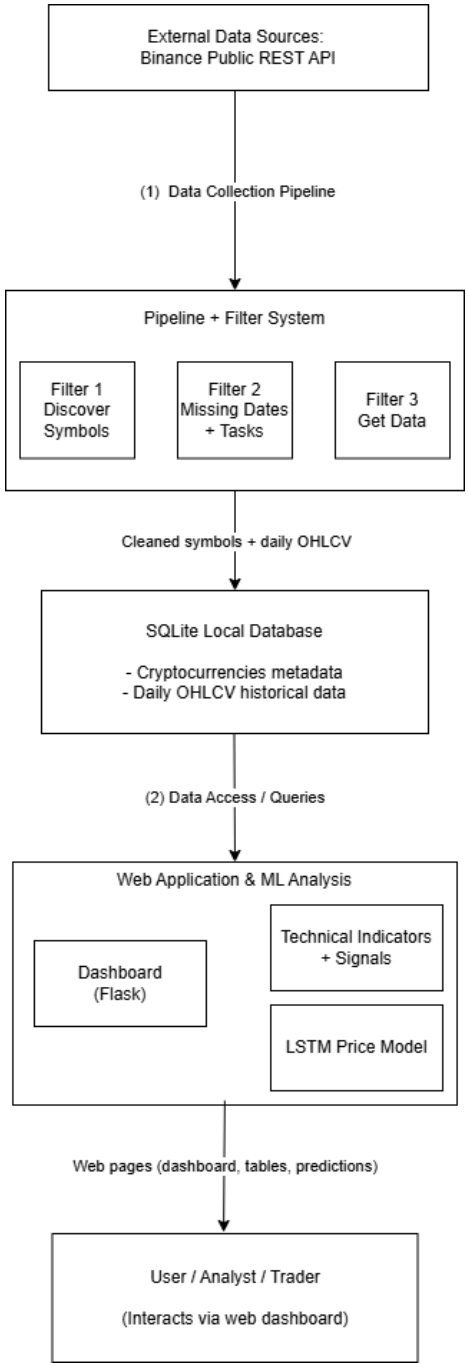

Figure 1: Conceptual architecture of the system

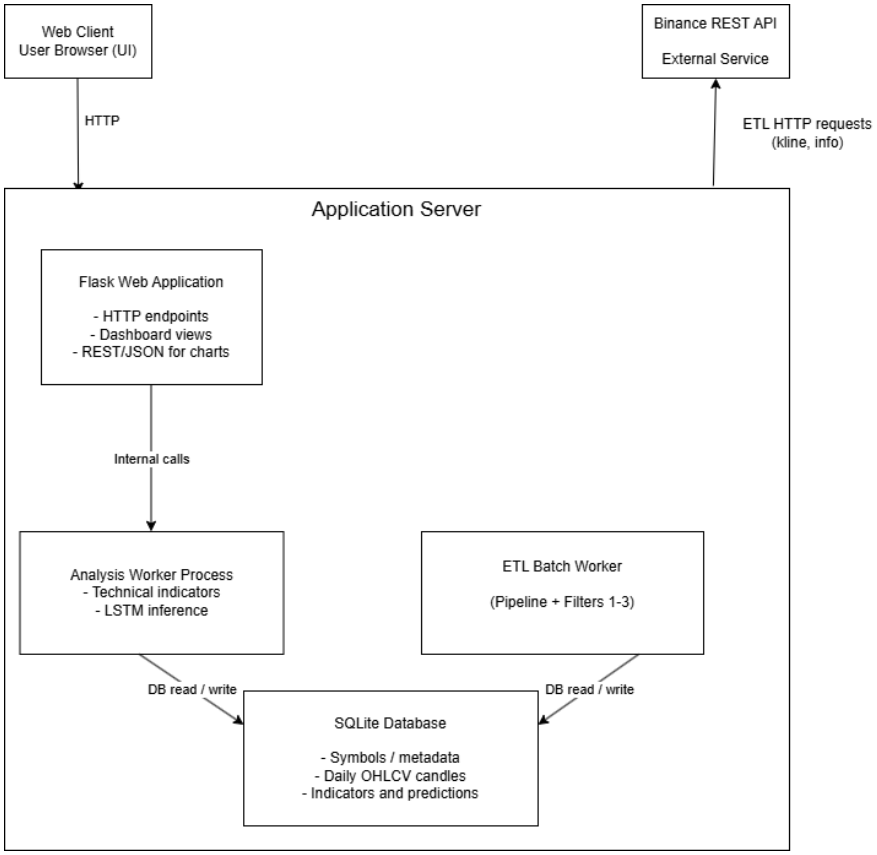

Figure 2: Execution architecture: runtime processes and communication

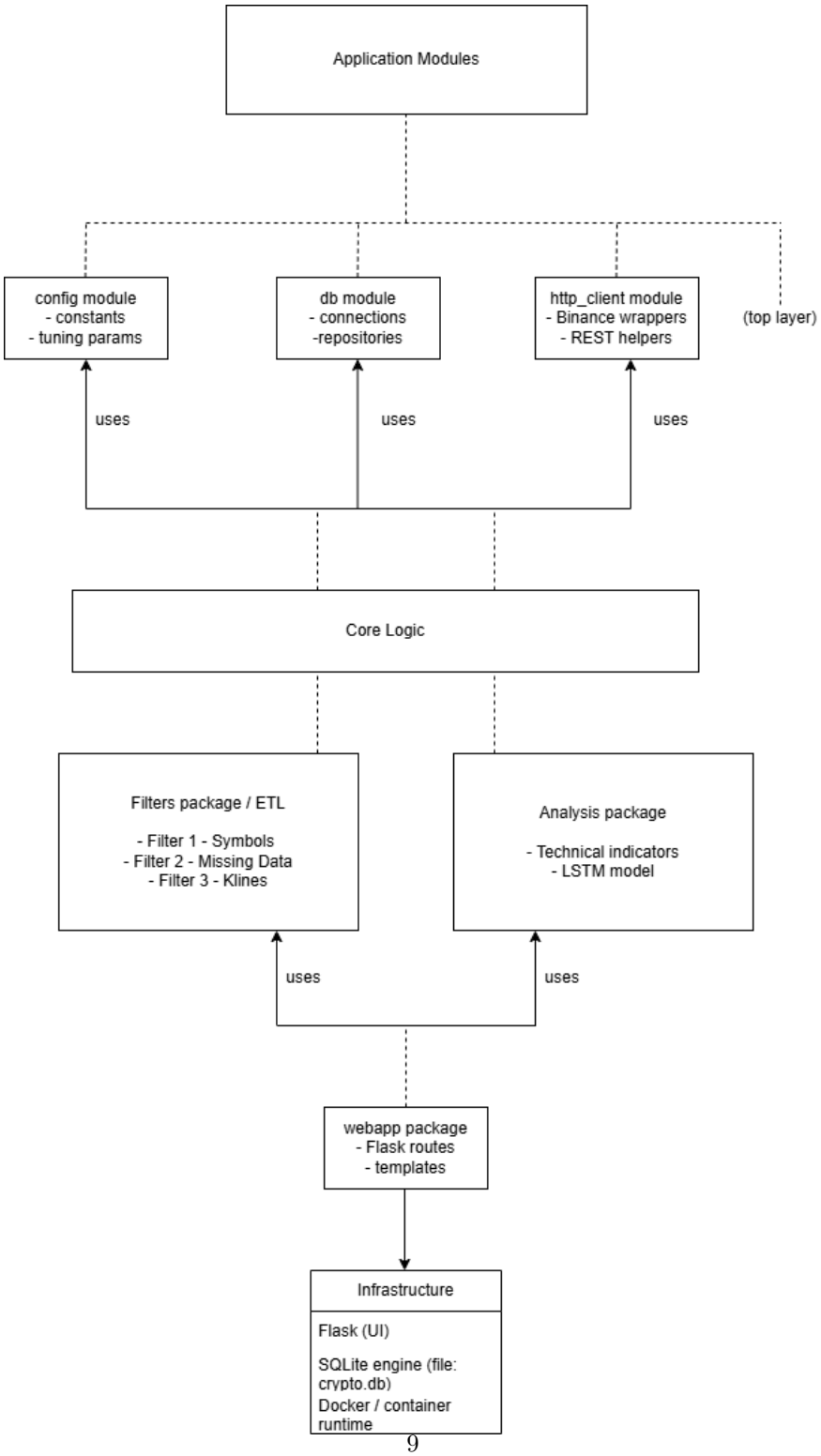

Figure 3: Implementation architecture: modules and packages