Project Report on

DATA WAREHOUSING WITH DYNAMIC DATA STREAM OF REAL TIME STOCK MARKET DATA

Submitted by:

PIYUSH KESHARI

MTECH AIML

REG.NO. : AP24122040004

AML 508

DATA WAREHOUSING AND PATTERN MINING

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

April 2025

Table of Contents

1. List of figures.
2. List of tables
3. Abstract
4. Introduction
    1.1. Background
    1.2. Problem Statement
    1.3. Objectives
    1.4. Scope
    1.5. Report Structure
5. Literature survey
    2.1. Real-time Data Streaming Technologies
    2.2. Data Warehousing for Time-Series Data
    2.3. In-Memory Data Stores for Buffering
    2.4. Stock Market Prediction Models (ARIMA)
    2.5. Real-time Visualization Dashboards
6. Working methodology
    3.1. Data Flow Overview
    3.2. Data Acquisition: Socket Streaming
    3.3. Data Storage in Redis (Temporary Buffer)
    3.4. Data Transformation: From Ticks/Seconds to Aggregated Intervals
    3.5. Data Storage in Postgresql (Data Warehouse)
    3.6. Real-time Prediction Model (ARIMA)
    3.7. Dashboarding (Streamlit)
7. Hardware and software requirements
    4.1. Hardware Requirements
    4.2. Software Requirements
8. Implementation of the process
    5.1. Setting up the Environment
    5.2. Socket Client Implementation
    5.3. Redis Integration
    5.4. Data Aggregation and Transformation Script
    5.5. Postgresql Schema and Data Loading
    5.6. ARIMA Model Training and Prediction Service
    5.7. Streamlit Dashboard Development
9. Outcome of the process
    6.1. Real-time Data Pipeline
    6.2. Persistent Data Warehouse

6.3. Predictive Insights
6.4. Interactive Visualization
10. Different Evaluation of the process
7.1. System Performance (Latency, Throughput)
7.2. Data Integrity and Consistency
7.3. Prediction Accuracy (ARIMA Model)
7.4. Scalability Assessment
7.5. User Experience (Dashboard Usability)
11. Conclusion
12. Future scope
13. References

List of figures

1. Figure 1: System Architecture Diagram showing components]
2. [Figure 2: Detailed Data Flow Diagram showing data formats at each stage]
3. Figure 3: Redis Data Structure]
4. [Figure 4: PostgreSQL Table Schema for OHLC Data]
5. [Figure 5: ARIMA Model Prediction Flow]
6. [Figure 6: Streamlit Dashboard Screenshot - NIFTY50 View showing historical data based on custom selection date range]
7. [Figure 7: Streamlit Dashboard Screenshot - USDINR View with historical data]
8. [Figure8 : Example Candlestick Chart highlighted for USDINR with Indicators Moving Average[MA]]
9. [Figure 9: Example Candlestick Chart highlighted for NIFTY 50 with Indicators Moving Average[MA]]

List of tables

[Table 1: Hardware Requirements]

[Table 2: Software Requirements and Libraries]

[Table 3: NIFTY 50 index minute data stored in postgreSQL database]

[Table 4: USDINR minute data stored in postgreSQL database]

Abstract

The stock market generates vast amounts of high-velocity data, demanding efficient processing and analysis for timely decision-making. This project implements a data warehousing solution designed to handle dynamic, real-time stock market data streams. Data is acquired via socket streaming, initially buffered in a high-speed Redis server to handle ingestion bursts. A transformation process aggregates tick or second-level data into standardised OHLC (Open, High, Low, Close) intervals (e.g., one minute). This structured data is then persistently stored in a PostgreSQL database, serving as the data warehouse. An ARIMA (Autoregressive Integrated Moving Average) model is employed to predict the next minute's OHLC data based on historical patterns stored in PostgreSQL. Finally, a real-time, interactive dashboard built with Streamlit visualises the candlestick charts for selected instruments (NIFTY50, USDINR), overlays the predicted values, and displays potential buy/sell indicators derived from the predictions. The system provides a robust framework for ingesting, storing, processing, predicting, and visualizing real-time financial data streams.

1. Introduction

1.1. Background

The financial markets, particularly stock markets, are characterized by their high volatility and the continuous generation of data at immense speed (velocity) and volume. Real-time analysis of this data is crucial for traders, analysts, and financial institutions to make informed decisions, manage risk, and identify potential opportunities. Traditional batch-processing data warehouses struggle to cope with the demands of such dynamic data streams.

1.2. Problem Statement

Handling real-time stock market data presents several challenges:

- **High Velocity:** Data arrives continuously at high speeds (ticks per second).
- **Volume:** Even aggregated data (e.g., minute-level) accumulates rapidly.
- **Real-time Processing:** Data needs to be processed, stored, and analysed with minimal latency.
- **Prediction:** Generating timely predictions based on the latest data requires efficient modelling and integration.

- **Visualization:** Presenting complex time-series data and predictions in an intuitive, interactive manner is essential for end-users.

1.3. Objectives

The primary objectives of this project are:

1. To design and implement a pipeline for ingesting real-time stock market data via sockets.
2. To utilize Redis as an efficient in-memory buffer for incoming data streams.
3. To transform raw, high-frequency data into structured OHLC format at desired intervals (e.g., 1 minute, 5 minutes, 1 day).
4. To establish a Postgresql database as a persistent data warehouse for historical and real-time aggregated stock data.
5. To develop and integrate an ARIMA model for predicting next-minute OHLC values.
6. To create an interactive Streamlit dashboard for visualising real-time and historical data, predictions, and basic trading indicators for selected instruments (NIFTY50, USDINR).

1.4. Scope

This project focuses on:

- Data acquisition from a simulated or live socket feed for specific instruments (NIFTY50 index, USDINR currency pair).
- Processing and storage pipeline using Redis and PostgreSQL.
- Implementation of an ARIMA model for short-term (next minute) OHLC prediction.
- Development of a Streamlit dashboard for visualization, including candlestick charts, date selection, prediction overlay, and simple buy/sell indicators.
- The project does not include building a full-fledged trading execution system or implementing highly complex, computationally intensive prediction models beyond ARIMA.

1.5. Report Structure

This report details the project methodology, implementation, and outcomes. Section 2 reviews relevant literature. Section 3 describes the working methodology and system architecture. Section 4 lists hardware and software requirements. Section 5 details the implementation steps. Section 6 presents the outcomes. Section 7 discusses evaluation methods. Section 8 concludes the report, and Section 9 suggests future enhancements. Section 10 provides references.

2. Literature survey

2.1. Real-time Data Streaming Technologies

Handling continuous data streams requires specialised technologies. While this project uses direct socket programming for ingestion, industry standards often involve message queues and stream processing frameworks like Apache Kafka (for durable, high-throughput messaging), Apache Flink, and Apache Spark Streaming (for complex event processing and transformations on streams) [1, 2]. Sockets provide a fundamental, low-level mechanism for real-time communication suitable for direct connections.

2.2. Data Warehousing for Time-Series Data

Traditional data warehouses are often optimised for relational data and batch updates. Time-series data, like stock market prices, has unique characteristics (time-ordered, append-only, specific query patterns like time-based aggregation). Postgresql, especially with extensions like Timescaledb, is increasingly used for time-series workloads, offering optimisations for storage, indexing, and querying time-stamped data while retaining SQL capabilities [3].

2.3. In-Memory Data Stores for Buffering

High-velocity data streams can overwhelm downstream processing or storage systems. In-memory databases like Redis are frequently used as buffers or caches due to their extremely low latency read/write operations [4]. They can absorb ingestion spikes, decouple data producers from consumers, and facilitate rapid intermediate processing steps before data is committed to slower, persistent storage. Redis's data structures (like Lists or Sorted Sets) are suitable for temporarily holding ordered stream data.

2.4. Stock Market Prediction Models (ARIMA)

Predicting stock market movements is a complex challenge. Various statistical and machine learning models are employed. ARIMA (Autoregressive Integrated Moving Average) is a classical statistical model widely used for time-series forecasting [5]. It models the temporal dependencies in the data. While more complex models like LSTMs (Long Short-Term Memory) or GARCH exist [6], ARIMA provides a good baseline for short-term forecasting, is relatively simpler to implement, and computationally less intensive for real-time prediction updates, making it suitable for this project's scope.

2.5. Real-time Visualisation Dashboards

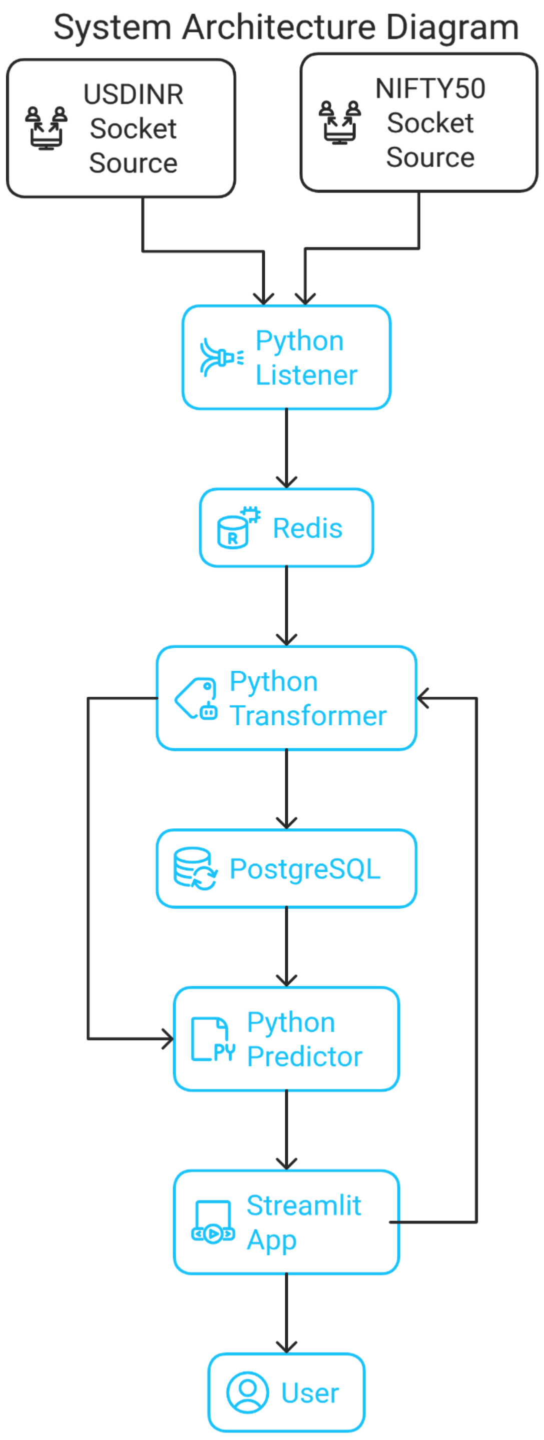
Effectively communicating insights from real-time data requires interactive dashboards. Tools like Streamlit [7], Plotly Dash, or Grafana allow developers to build web-based applications quickly for visualising data. Streamlit, specifically, enables rapid development of data-centric applications using Python, making it ideal for integrating visualisation directly with data processing and modelling scripts.

3. Working methodology

3.1. Data Flow Overview

The system follows a linear data flow pattern designed for real-time processing:

1. **Acquisition:** Real-time tick data for NIFTY50 and USDINR arrives via a network socket connection.
2. **Buffering:** A Python script listens to the socket and pushes the raw data immediately into a Redis instance.
3. **Transformation:** A separate Python process periodically (e.g., every few seconds or at minute boundaries) fetches data from Redis. It aggregates the raw ticks into OHLC bars for a defined interval (e.g., 1 minute).
4. **Warehousing:** The calculated OHLC bars are inserted into a structured table within the Postgresql database.
5. **Prediction:** A prediction service periodically queries the recent OHLC data from Postgresql, trains/updates an ARIMA model, and predicts the next minute's OHLC values.
6. **Visualisation:** The Streamlit application queries historical and the latest OHLC data from Postgresql, fetches the latest prediction, and displays interactive candlestick charts, predictions, and Moving average indicators based on user selection (instrument, date range).



[Figure 1: System Architecture Diagram showing components: Socket Source -> Python Listener -> Redis -> Python data Transformer -> Postgresql -> Python Predictor -> Streamlit App -> User]

3.2. Data Acquisition: Socket Streaming

- A Python client establishes a connection to a data provider's socket server (this could be a broker's API feed or a simulated feed).
- The client continuously listens for incoming data packets.
- Data is assumed to arrive in a structured format (e.g., JSON) containing a timestamp, instrument name (symbol), and price (Last Traded Price - LTP).
- Minimal processing occurs here; the primary goal is fast reception and forwarding to the buffer.

3.3. Data Storage in Redis (Temporary Buffer)

- **Purpose:** Acts as a high-speed, short-term buffer to decouple the ingestion rate from the database writing rate and facilitate aggregation.
- **Mechanism:** The socket listener script pushes received data (e.g., timestamp, symbol, price) into Redis.
- **Data Structure:** A Redis List or Stream could be used for each instrument (e.g., nifty50_ticks, usdinr_ticks). Each entry contains the raw tick data. Using Lists allows easy appending (RPUSH) and retrieval of ranges (LRANGE).
- **Volatility:** Data in Redis is considered transient and may be capped in size or time to manage memory usage.

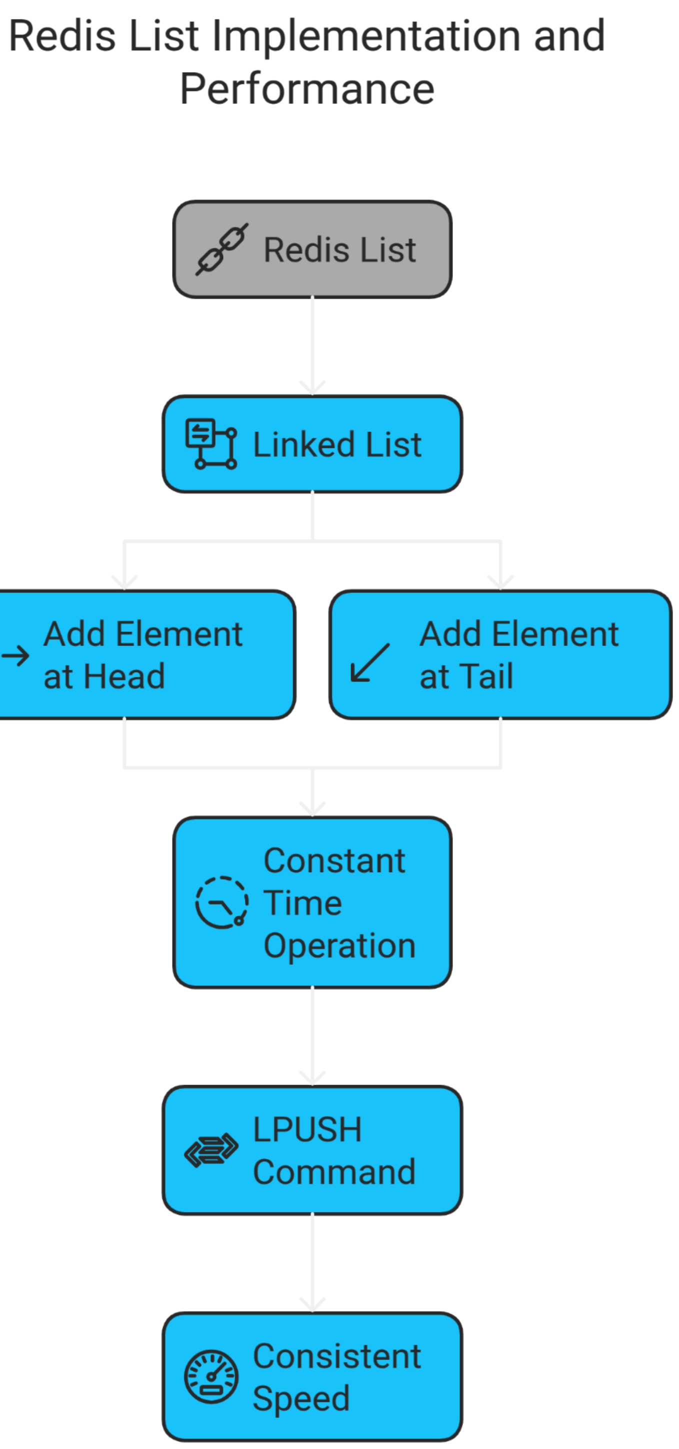


Figure 3: Redis Data Structure

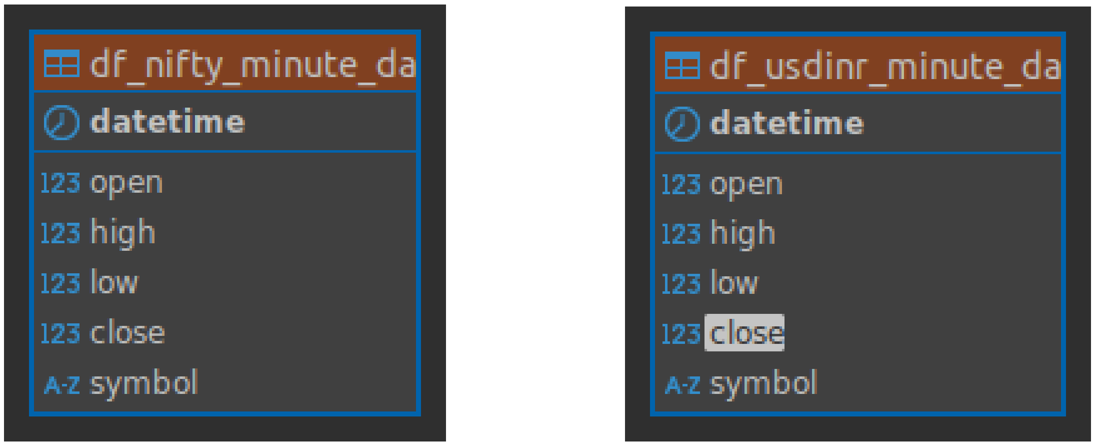
3.4. Data Transformation: From Ticks/Seconds to Aggregated Intervals

- **Purpose:** Convert high-frequency raw data into standardized OHLC bars suitable for analysis and charting.
- **Mechanism:** A dedicated Python script runs periodically (e.g., using a scheduler like APScheduler or triggered every minute).
- **Process:**
    1. Fetch relevant raw data points from Redis for the last interval (e.g., all ticks between hh:mm:00 and hh:mm:59).
    2. For each instrument (NIFTY50, USDINR):
        - Identify the **Open** price (first tick price in the interval).
        - Find the **High** price (maximum price during the interval).
        - Find the **Low** price (minimum price during the interval).
        - Identify the **Close** price (last tick price in the interval).
        - Record the timestamp corresponding to the start of the interval.
    3. Remove the processed data from Redis (e.g., using LTRIM) to prevent reprocessing and manage memory.
- **Dynamic Intervals:** While implemented here for 1-minute intervals (required for the ARIMA model), the *transformation logic* could be adapted to generate bars for dynamic or user-defined intervals (e.g., 5-min, 15-min) if needed, potentially storing multiple aggregations.

3.5. Data Storage in Postgresql (Data Warehouse)

- **Purpose:** Provides persistent, structured storage for the aggregated OHLC data, serving as the historical data warehouse.
- **Mechanism:** The transformation script connects to the PostgreSQL database and inserts the newly calculated OHLC bars.
- **Schema:** A table (e.g., ohlc_data) is created with columns like:
    - timestamp (TIMESTAMP, primary key or indexed)
    - symbol (VARCHAR, indexed)
    - open (NUMERIC)
    - high (NUMERIC)
    - low (NUMERIC)
    - close (NUMERIC)

    - (Optional) volume (BIGINT)
- **Benefits:** Allows efficient querying of historical data using SQL, supports indexing for fast lookups based on time and symbol, ensures data durability and consistency (ACID properties).


df_nifty_minute_da
datetime
123 open
123 high
123 low
123 close
A-Z symbol

df_usdinr_minute_da
datetime
123 open
123 high
123 low
123 close
A-Z symbol

[Figure 4: PostgreSQL Table Schema for OHLC Data

3.6. Real-time Prediction Model (ARIMA)

- **Purpose:** To forecast the next minute's OHLC values based on recent historical patterns.
- **Model:** ARIMA (p, d, q) - Autoregressive Integrated Moving Average.
    - *AR(p):* Dependency between an observation and 'p' lagged observations.
    - *I(d):* Differencing raw observations 'd' times to make the time series stationary.
    - *MA(q):* Dependency between an observation and a residual error from 'q' lagged observations.

- **Mechanism:**
    - A separate prediction service (Python script) runs periodically (e.g., every minute, after new OHLC data is expected).
    - It queries the last N (e.g., 100-200) minutes of OHLC data for a specific instrument from PostgreSQL.
    - It potentially checks for stationarity (e.g., using ADF test) and applies differencing (d) if needed.

    - It fits an ARIMA(p,d,q) model to the historical 'Close' prices (or separately for O, H, L, C). Determining optimal p, d, q might require offline analysis or adaptive methods (e.g., auto_arima).
    - It uses the fitted model to predict the value(s) for the next time step (next minute).
    - The prediction result (timestamp, symbol, predicted_open, predicted_high, predicted_low, predicted_close) can be stored temporarily (e.g., back in Redis or a dedicated prediction table in PostgreSQL) for quick access by the dashboard.
- **Buy/Sell Indicators:** Simple indicators are generated based on the prediction. Examples:
    - If predicted_close > current_close: Potential Buy Signal.
    - If predicted_close < current_close: Potential Sell Signal.
    - (More sophisticated: Compare predicted price to a moving average of historical prices).

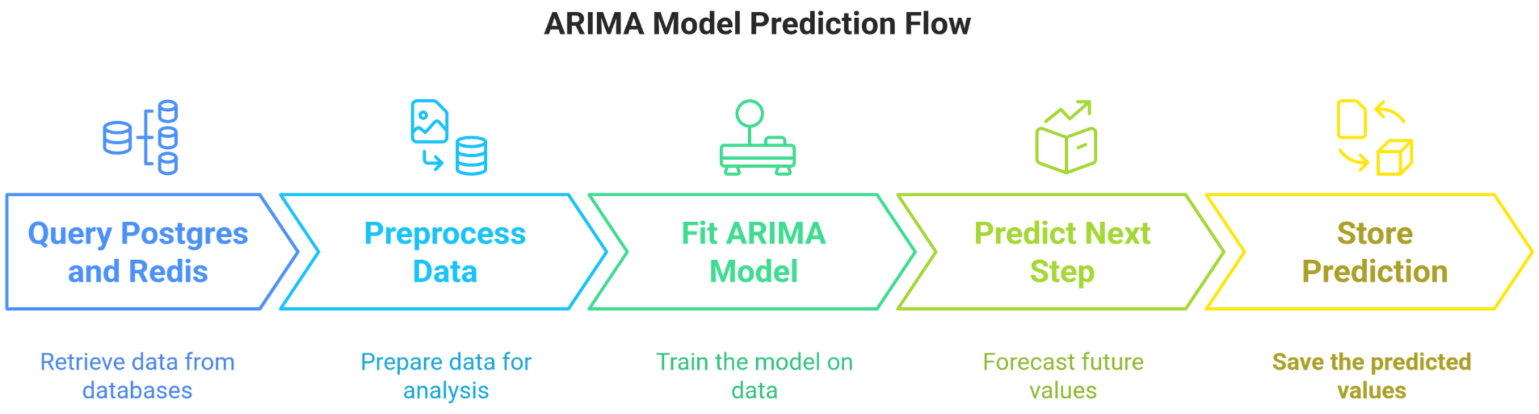
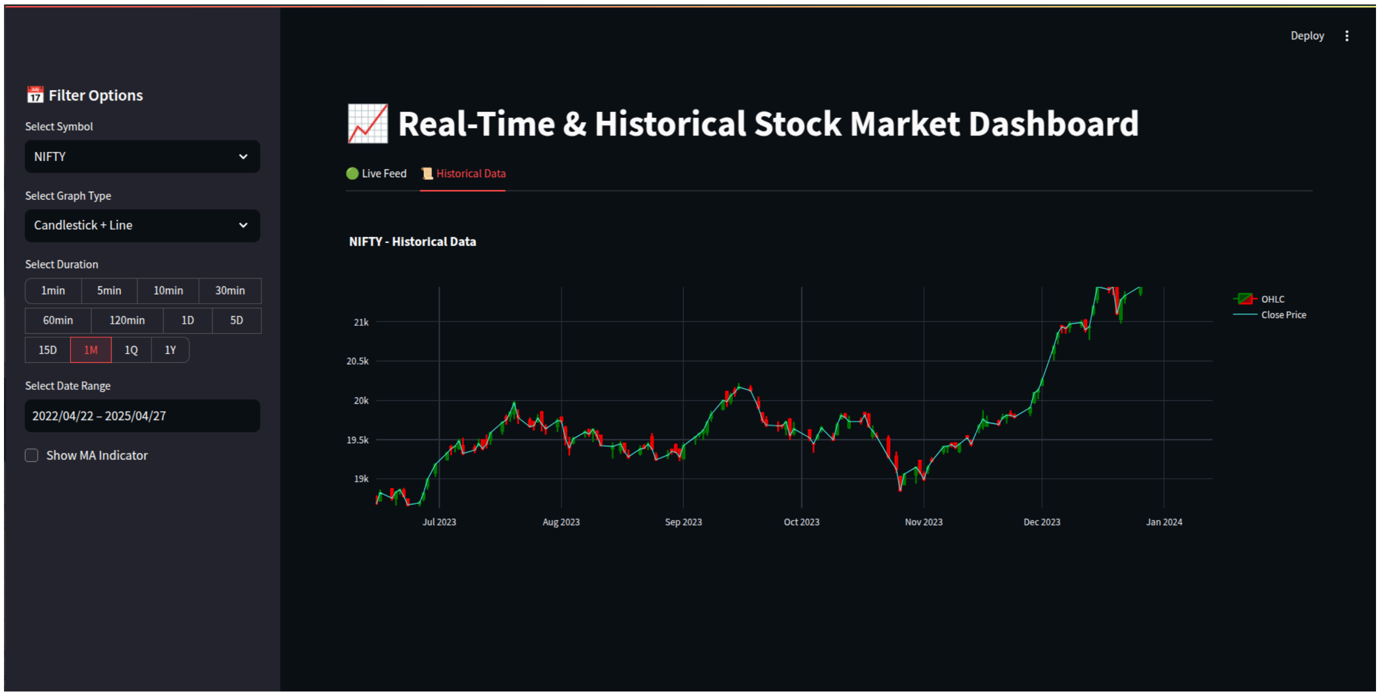


Figure 5: ARIMA Model Prediction Flow: Query Postgres -> Preprocess (Differencing) -> Fit ARIMA -> Predict Next Step -> Store Prediction

3.7. Dashboarding (Streamlit)

- **Purpose:** Provide an interactive user interface for visualising data and predictions.
- **Framework:** Streamlit (Python library).
- **Features:**
    1. **Instrument Selection:** Dropdown or radio buttons for users to choose between NIFTY50 and USDINR.

2. **Date Range Selection:** Calendar input for selecting start and end dates for historical data display.
3. **Candlestick Chart:** Displays OHLC data for the selected instrument and date range (using libraries like Plotly or Matplotlib integrated with Streamlit).
4. **Prediction Overlay:** Plots the predicted next-minute OHLC value(s) on the chart.
5. **Buy/Sell Indicators:** Displays the generated indicators (e.g., as text messages or markers on the chart).
6. **Real-time Update:** The dashboard periodically refreshes (e.g., every minute) to fetch the latest data from PostgreSQL and the latest prediction, updating the charts and indicators.


Filter Options

Select Symbol: NIFTY

Select Graph Type: Candlestick + Line

Select Duration: 1min | 5min | 10min | 30min | 60min | 120min | 1D | 5D | 15D | [✓ 1M] | 1Q | 1Y

Select Date Range: 2022/04/22 – 2025/04/27

Show MA Indicator

Figure 6: Streamlit Dashboard Screenshot - NIFTY50 View showing historical data based on custom selection date range

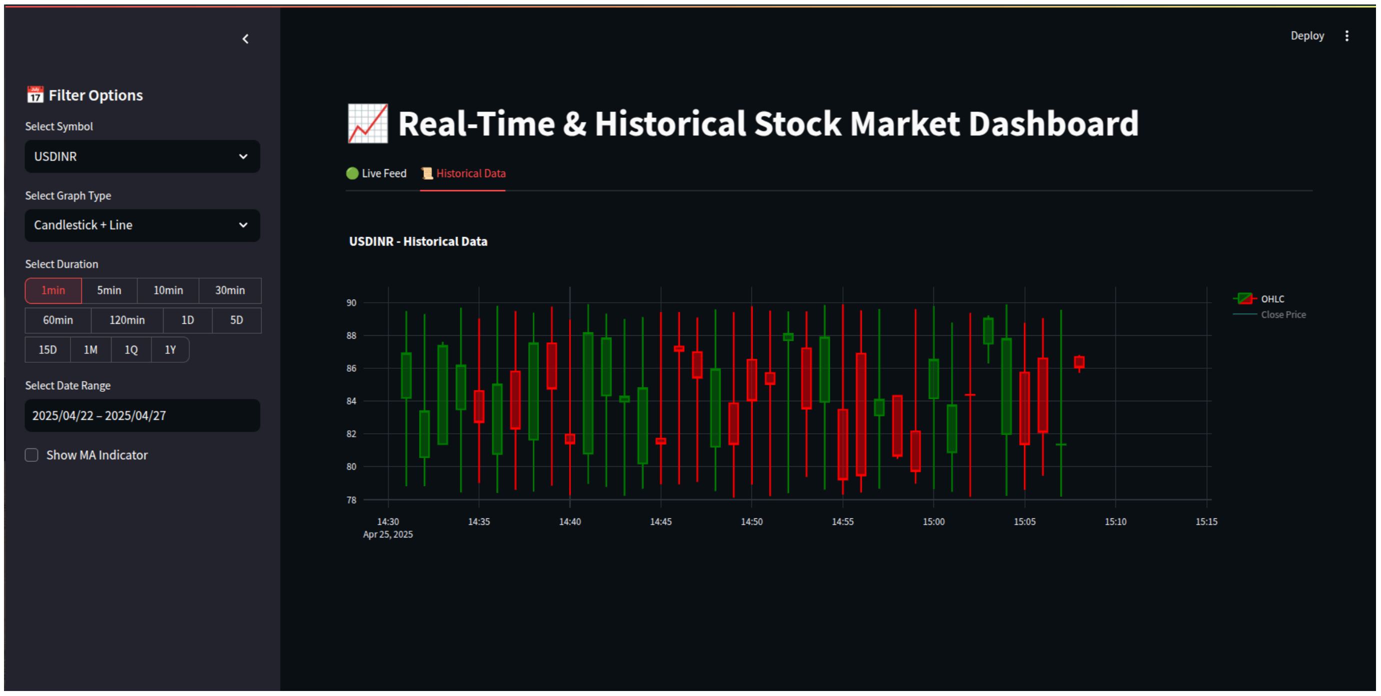


Figure 7: Streamlit Dashboard Screenshot - USDINR View with historical data

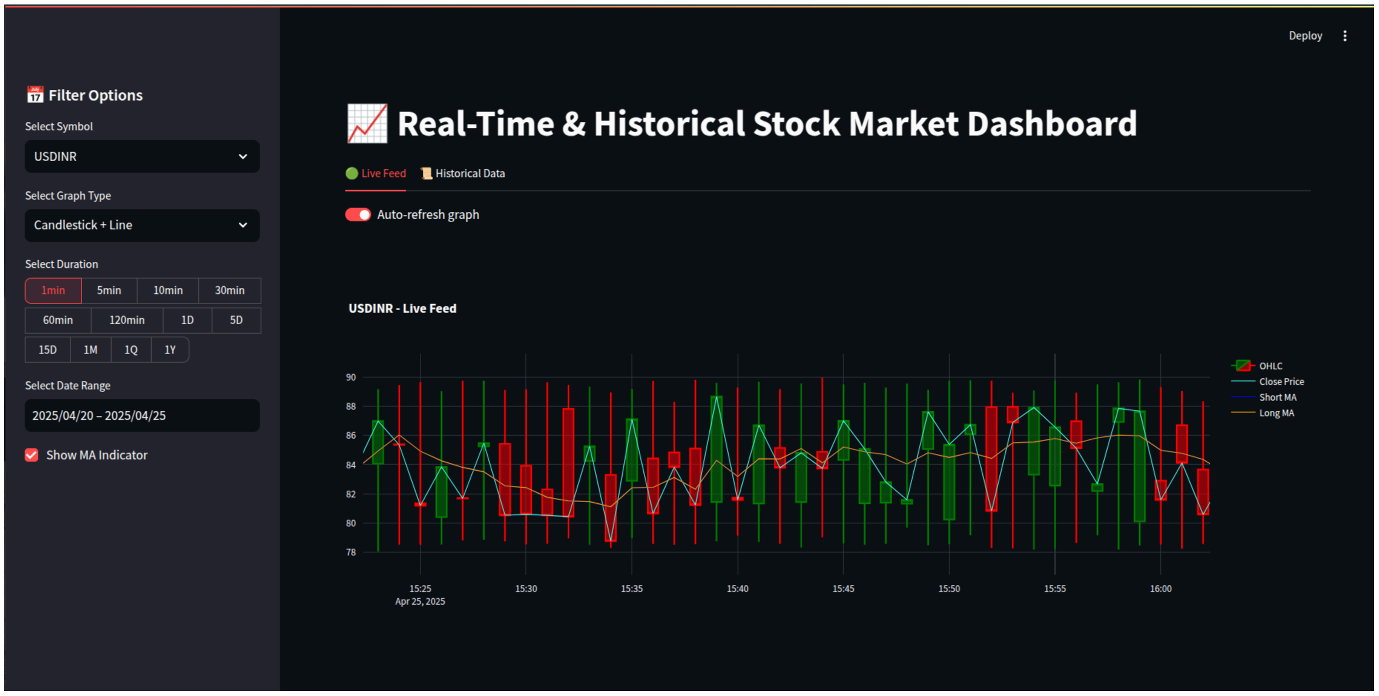
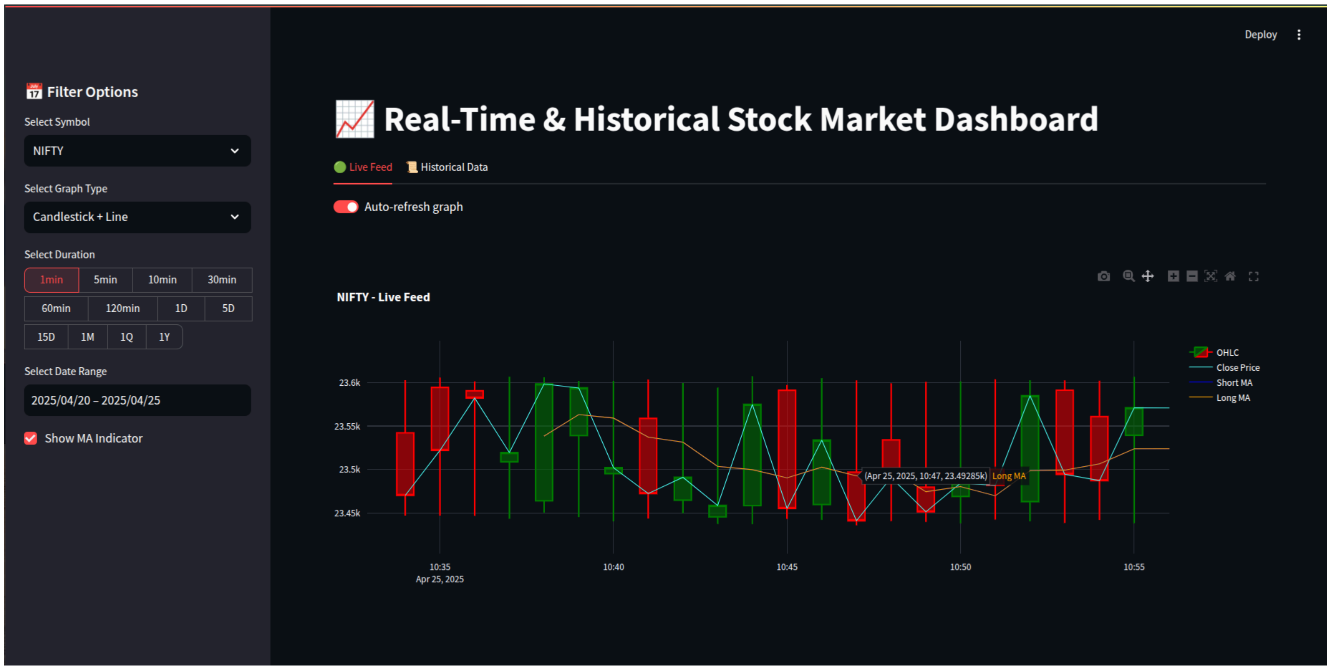


Figure8 : Example Candlestick Chart highlighted for USDINR with Indicators Moving Average[MA]



Real-Time & Historical Stock Market Dashboard

Select Symbol: NIFTY

Select Graph Type: Candlestick + Line

Select Duration: 1min

Select Date Range: 2025/04/20 – 2025/04/25

Show MA Indicator ✓

Auto-refresh graph ✓

Figure 9: Example Candlestick Chart highlighted for NIFTY 50 with Indicators Moving Average[MA]

4. Hardware and software requirements

4.1. Hardware Requirements

Table 1: Hardware Requirements

Component	**Specification**	**Rationale / Notes**
Processing Server	CPU: 4+ Cores (e.g., Intel Xeon, AMD Ryzen)	Required for running multiple Python processes concurrently (Listener, Transformer, Predictor, Streamlit).
	RAM: 8 GB+	ARIMA model fitting and data handling (Pandas) can be memory-intensive. More RAM allows larger datasets/models.

	Disk: 100 GB+ SSD	For Operating System, software installation, Python libraries, and application logs. SSD recommended for speed.
Redis Server	CPU: 2+ Cores	Redis is generally single-threaded for commands but benefits from cores for background tasks/IO.
	RAM: 4 GB+	Crucial. Size depends directly on tick frequency, number of instruments, and buffer duration. Monitor usage.
	Disk: 20 GB+ (SSD Recommended)	Primarily for OS and Redis persistence snapshots/AOF logs if configured.
PostgreSQL Server	CPU: 4+ Cores	Handles concurrent connections, complex queries, indexing, and background tasks (VACUUM).
	RAM: 8 GB+	Database performance heavily relies on caching data in RAM. More RAM improves query speed significantly.
	Disk: 500 GB+ SSD	Fast I/O is essential. Size depends on data retention period and number of instruments. SSD highly recommended.
Network	Low Latency, High Bandwidth	Critical for real-time data acquisition from the socket and communication between services.

Note: These are starting estimates. Actual requirements may vary significantly based on data volume, velocity, number of concurrent users, prediction model complexity, and data retention policies. Monitoring resource usage is crucial.

4.2. Software Requirements

Table 2: Software Requirements and Libraries

Category	**Software / Library**	**Version (Recommended)**	**Purpose**
Operating System	Linux	Ubuntu 20.04+ / CentOS 7+	Preferred for server deployment; stable and well-supported.
	macOS / Windows	Latest	Suitable for development environments.
Programming Lang.	Python	3.8+	Core language for all custom scripts.
Data Acquisition	socket (Python stdlib)	N/A	Low-level network communication for receiving data stream.
Buffering Client	redis-py	4.x+	Python client for interacting with the Redis server.
DB Client	psycopg2-binary or SQLAlchemy	Latest	Python client(s) for interacting with the PostgreSQL database.

Data Handling	pandas	1.4+	Data manipulation, time-series indexing, aggregation (OHLC).
	numpy	1.21+	Foundational package for numerical operations, used by Pandas.
Prediction Model	statsmodels	0.13+	Provides ARIMA model implementation and time-series analysis tools.
	pmdarima (Optional)	Latest	For automatic selection of ARIMA order (p, d, q).
Dashboarding	streamlit	1.10+	Framework for building the interactive web application/dashboard.
Visualization	plotly	5.5+	Library for creating interactive charts (especially candlestick).
	matplotlib (Alternative)	Latest	Alternative plotting library if Plotly is not used.
Scheduling	APScheduler (Optional)	3.x+	For scheduling periodic tasks (transformation, prediction) reliably.

In-Memory DB	Redis Server	6.x / 7.x	High-speed in-memory data store for buffering incoming ticks.
Relational DB	PostgreSQL Server	12+	Persistent storage for OHLC data warehouse.
	TimescaleDB (Optional Ext.)	Latest compatible	PostgreSQL extension optimized for time-series data (highly recommended).
Web Browser	Modern Browser	Chrome, Firefox, Edge	For accessing and interacting with the Streamlit dashboard.

5. Implementation of the process

5.1. Setting up the Environment

- Install Python and required libraries (pip install -r requirements.txt).
- Install and configure Redis server.
- Install and configure PostgreSQL server. Create a database and user for the project.

5.2. Socket Client Implementation

- Write a Python script (socket_listener.py) that:
  - Connects to the specified host/port of the data feed.
  - Enters a loop to receive data (socket.recv()).
  - Parses the received data (assuming JSON format).
  - Connects to Redis.
  - Pushes the parsed data (timestamp, symbol, price) into the appropriate Redis list (e.g., RPUSH nifty50_ticks '...').

    - Includes error handling for connection issues and data parsing errors.

5.3. Redis Integration

- Ensure the Redis server is running and accessible from the listener and transformer scripts.
- Use the redis-py library in Python scripts to connect and interact (RPUSH, LRANGE, LTRIM).

5.4. Data Aggregation and Transformation Script

- Write a Python script (transformer.py) that:
    - Connects to Redis and PostgreSQL.
    - Runs periodically (e.g., using time.sleep(60) in a loop, or integrated with APScheduler).
    - Calculates the start and end timestamps for the interval to process.
    - Fetches all ticks for each instrument from Redis within that interval (LRANGE).
    - Uses Pandas DataFrames for efficient calculation of Open, High, Low, Close from the fetched ticks.
    - Handles cases with no data in an interval.
    - Inserts the calculated OHLC data into the PostgreSQL ohlc_data table.
    - Removes the processed ticks from Redis (LTRIM).
    - Includes logging and error handling.
-

5.5. PostgreSQL Schema and Data Loading

- Connect to PostgreSQL using psql or a GUI tool.
- Execute SQL CREATE TABLE statement to define the ohlc_data table structure as specified in section 3.5.
- Create indexes on the timestamp and symbol columns for performance.
- The transformer.py script handles the ongoing data loading using INSERT statements via psycopg2 or an ORM like SQLAlchemy.

5.6. ARIMA Model Training and Prediction Service

- Write a Python script (predictor.py) that:
    - Connects to PostgreSQL.

    - Runs periodically (e.g., every minute).
    - Queries the last 1 month OHLC records for the target instrument (e.g., NIFTY50 'Close' price) and merges with the current transformed data.
    - Preprocesses data (checking stationarity, differencing if needed).
    - Fits an ARIMA model using statsmodels.tsa.arima.model.ARIMA. (Specify order (p,d,q) - these might be predetermined or found using pmdarima.auto_arima initially).
    - Generates a 1-step ahead forecast (model_fit.predict()).
    - (Optional) Repeat for O, H, L if predicting full OHLC.
    - Stores the prediction (e.g., in a simple Redis key like nifty50_prediction or a dedicated DB table).
    - Calculates simple buy/sell indicators based on the prediction. Store these alongside the prediction.

5.7. Streamlit Dashboard Development

- Write a Python script (dashboard.py).
- Import streamlit, pandas, plotly.graph_objects (for candlestick), psycopg2, redis.
- Set page title and layout (st.set_page_config).
- Add widgets for instrument selection (st.selectbox) and date range (st.date_input).
- Write functions to query PostgreSQL for historical OHLC data based on user selections.
- Write a function to fetch the latest prediction and indicators from Redis/PostgreSQL.
- Use plotly.graph_objects.Candlestick to create the chart.
- Add the predicted point(s) to the chart (e.g., using add_scatter).
- Display the buy/sell indicators (st.write or st.metric).
- Implement auto-refresh using st.experimental_rerun() within a loop with time.sleep(), or rely on user interaction for updates.
- Run the dashboard using streamlit run dashboard.py.

5.8. DataFlow Diagram

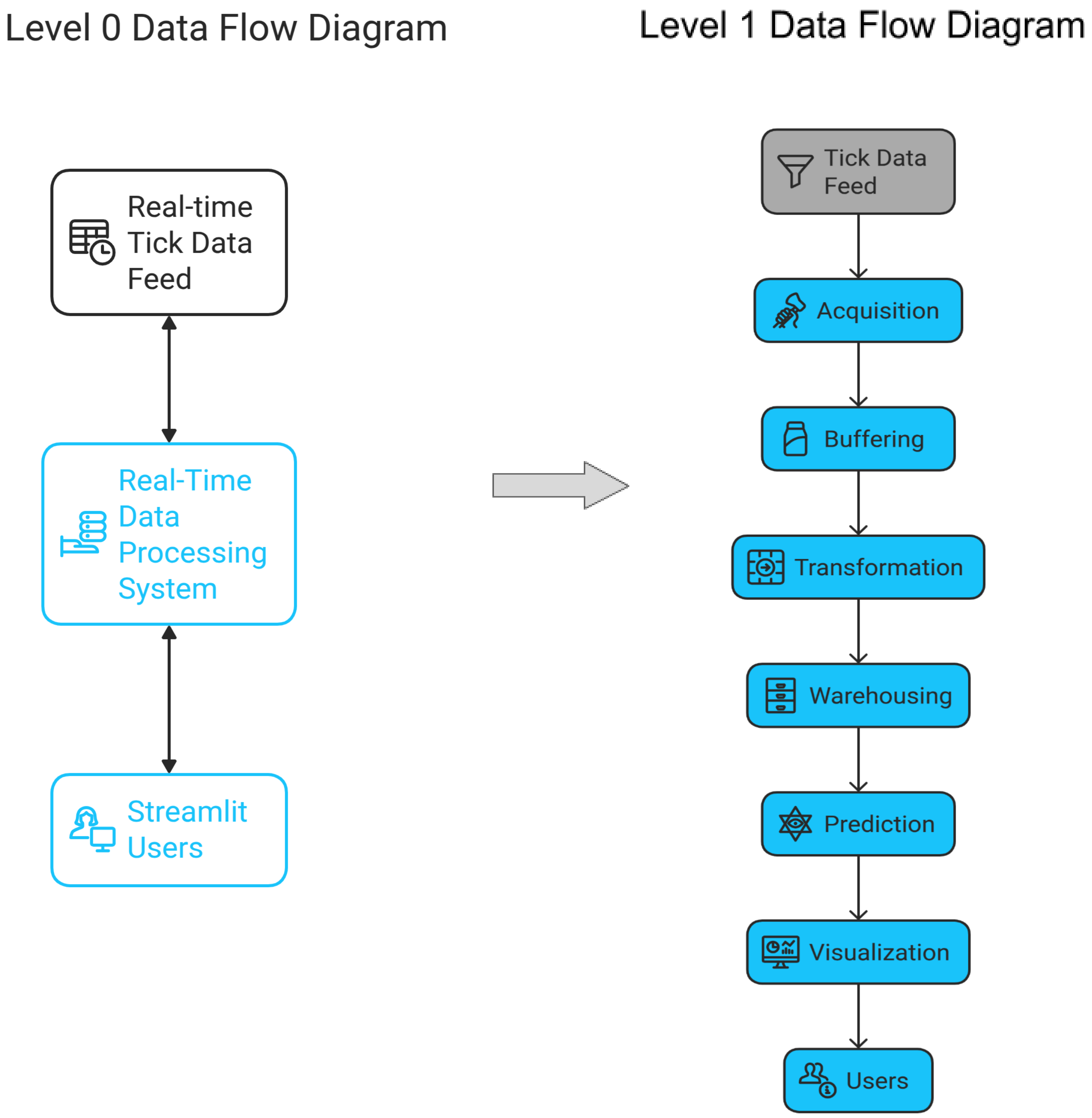


Figure8 : level 0 and level 1 data flow diagram (DFD)

6. Outcome of the process

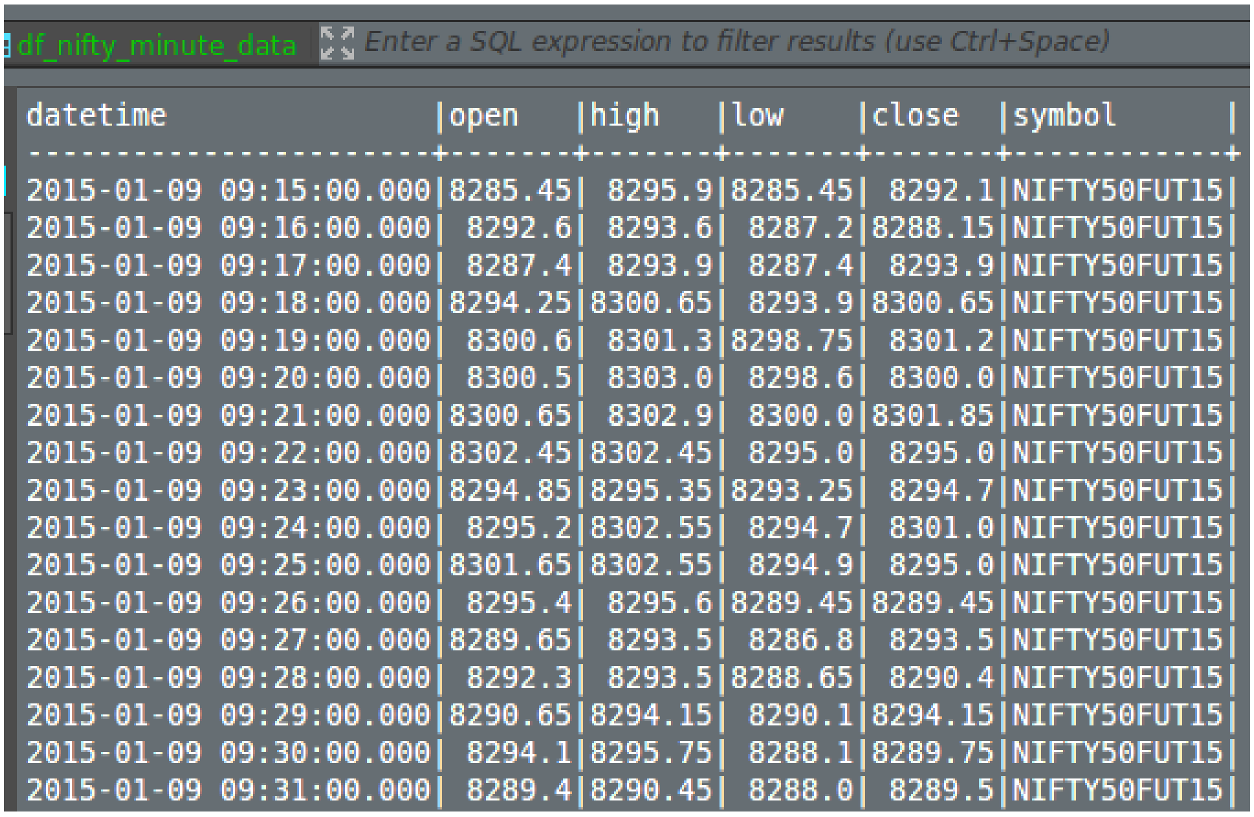
6.1. Real-time Data Pipeline

A functioning pipeline that successfully ingests data from a socket, buffers it in Redis, transforms it into OHLC format, and stores it persistently in PostgreSQL with measurable, low latency.

6.2. Persistent Data Warehouse

A PostgreSQL database populated with structured, historical OHLC data for NIFTY50 and USDINR, enabling analytical queries and serving as the foundation for the prediction model and dashboard.

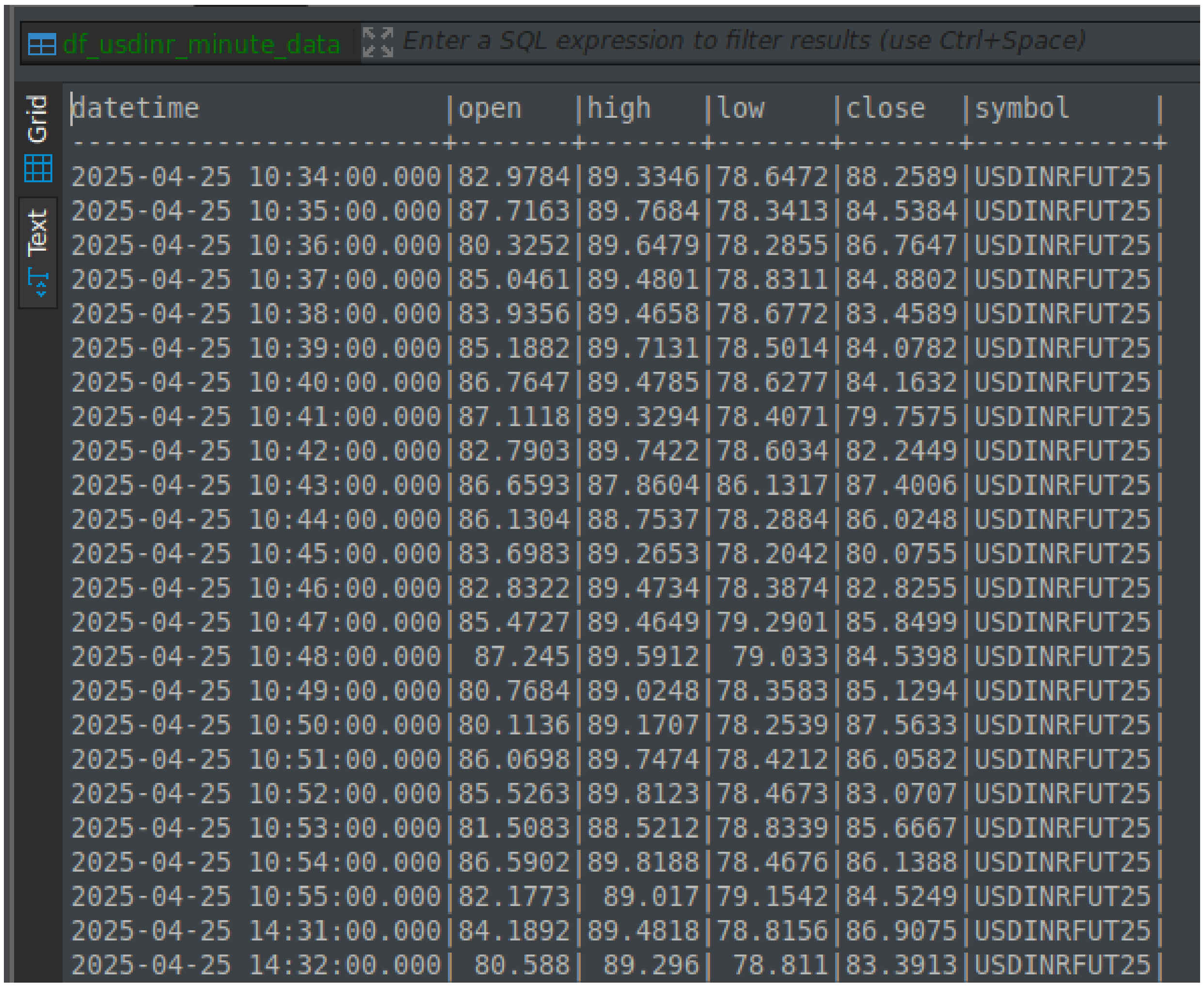
Table 3: NIFTY 50 index minute data stored in postgreSQL database



```
df_nifty_minute_data   Enter a SQL expression to filter results (use Ctrl+Space)

datetime               |open   |high   |low    |close  |symbol      |
-----------------------+-------+-------+-------+-------+------------+
2015-01-09 09:15:00.000|8285.45| 8295.9|8285.45| 8292.1|NIFTY50FUT15|
2015-01-09 09:16:00.000| 8292.6| 8293.6| 8287.2|8288.15|NIFTY50FUT15|
2015-01-09 09:17:00.000| 8287.4| 8293.9| 8287.4| 8293.9|NIFTY50FUT15|
2015-01-09 09:18:00.000|8294.25|8300.65| 8293.9|8300.65|NIFTY50FUT15|
2015-01-09 09:19:00.000| 8300.6| 8301.3|8298.75| 8301.2|NIFTY50FUT15|
2015-01-09 09:20:00.000| 8300.5| 8303.0| 8298.6| 8300.0|NIFTY50FUT15|
2015-01-09 09:21:00.000|8300.65| 8302.9| 8300.0|8301.85|NIFTY50FUT15|
2015-01-09 09:22:00.000|8302.45|8302.45| 8295.0| 8295.0|NIFTY50FUT15|
2015-01-09 09:23:00.000|8294.85|8295.35|8293.25| 8294.7|NIFTY50FUT15|
2015-01-09 09:24:00.000| 8295.2|8302.55| 8294.7| 8301.0|NIFTY50FUT15|
2015-01-09 09:25:00.000|8301.65|8302.55| 8294.9| 8295.0|NIFTY50FUT15|
2015-01-09 09:26:00.000| 8295.4| 8295.6|8289.45|8289.45|NIFTY50FUT15|
2015-01-09 09:27:00.000|8289.65| 8293.5| 8286.8| 8293.5|NIFTY50FUT15|
2015-01-09 09:28:00.000| 8292.3| 8293.5|8288.65| 8290.4|NIFTY50FUT15|
2015-01-09 09:29:00.000|8290.65|8294.15| 8290.1|8294.15|NIFTY50FUT15|
2015-01-09 09:30:00.000| 8294.1|8295.75| 8288.1|8289.75|NIFTY50FUT15|
2015-01-09 09:31:00.000| 8289.4|8290.45| 8288.0| 8289.5|NIFTY50FUT15|
```

Table 4: USDINR minute data stored in postgreSQL database



```
datetime               |open   |high   |low    |close  |symbol     |
-----------------------+-------+-------+-------+-------+-----------+
2025-04-25 10:34:00.000|82.9784|89.3346|78.6472|88.2589|USDINRFUT25|
2025-04-25 10:35:00.000|87.7163|89.7684|78.3413|84.5384|USDINRFUT25|
2025-04-25 10:36:00.000|80.3252|89.6479|78.2855|86.7647|USDINRFUT25|
2025-04-25 10:37:00.000|85.0461|89.4801|78.8311|84.8802|USDINRFUT25|
2025-04-25 10:38:00.000|83.9356|89.4658|78.6772|83.4589|USDINRFUT25|
2025-04-25 10:39:00.000|85.1882|89.7131|78.5014|84.0782|USDINRFUT25|
2025-04-25 10:40:00.000|86.7647|89.4785|78.6277|84.1632|USDINRFUT25|
2025-04-25 10:41:00.000|87.1118|89.3294|78.4071|79.7575|USDINRFUT25|
2025-04-25 10:42:00.000|82.7903|89.7422|78.6034|82.2449|USDINRFUT25|
2025-04-25 10:43:00.000|86.6593|87.8604|86.1317|87.4006|USDINRFUT25|
2025-04-25 10:44:00.000|86.1304|88.7537|78.2884|86.0248|USDINRFUT25|
2025-04-25 10:45:00.000|83.6983|89.2653|78.2042|80.0755|USDINRFUT25|
2025-04-25 10:46:00.000|82.8322|89.4734|78.3874|82.8255|USDINRFUT25|
2025-04-25 10:47:00.000|85.4727|89.4649|79.2901|85.8499|USDINRFUT25|
2025-04-25 10:48:00.000| 87.245|89.5912| 79.033|84.5398|USDINRFUT25|
2025-04-25 10:49:00.000|80.7684|89.0248|78.3583|85.1294|USDINRFUT25|
2025-04-25 10:50:00.000|80.1136|89.1707|78.2539|87.5633|USDINRFUT25|
2025-04-25 10:51:00.000|86.0698|89.7474|78.4212|86.0582|USDINRFUT25|
2025-04-25 10:52:00.000|85.5263|89.8123|78.4673|83.0707|USDINRFUT25|
2025-04-25 10:53:00.000|81.5083|88.5212|78.8339|85.6667|USDINRFUT25|
2025-04-25 10:54:00.000|86.5902|89.8188|78.4676|86.1388|USDINRFUT25|
2025-04-25 10:55:00.000|82.1773| 89.017|79.1542|84.5249|USDINRFUT25|
2025-04-25 14:31:00.000|84.1892|89.4818|78.8156|86.9075|USDINRFUT25|
2025-04-25 14:32:00.000| 80.588| 89.296| 78.811|83.3913|USDINRFUT25|
```

6.3. Predictive Insights

The system generates real-time, next-minute OHLC predictions using the ARIMA model. Basic buy/sell indicators derived from these predictions provide timely, albeit simple, decision support signals.

6.4. Interactive Visualization

A user-friendly web dashboard built with Streamlit allows users to:

- Select NIFTY50 or USDINR.
- Choose custom date ranges.
- View real-time updating candlestick charts.
- See the next-minute prediction overlaid on the chart.
- Observe the calculated buy/sell indicators.

7. Different Evaluation of the process

7.1. System Performance (Latency, Throughput)

- **Ingestion Latency:** Time taken from data arrival at the socket to storage in Redis (should be milliseconds).
- **Processing Latency:** Time taken from data arrival to storage in PostgreSQL (includes buffering and transformation time; target < 1 minute).
- **End-to-End Latency:** Time from data arrival to display on the dashboard (includes prediction and query time).
- **Throughput:** Maximum rate of ticks per second the system can handle without significant delays or data loss. Measured by load testing.

7.2. Data Integrity and Consistency

- Verify that no data points are lost between stages (socket -> Redis -> Postgres).
- Check for data correctness in the PostgreSQL OHLC bars (e.g., high >= open, high >= close, low <= open, low <= close).
- Ensure timestamps are handled correctly across components.

7.3. Prediction Accuracy (ARIMA Model)

- **Offline Evaluation:** Backtest the ARIMA model on historical data. Use metrics like:
  - Root Mean Squared Error (RMSE)
  - Mean Absolute Error (MAE)
  - Mean Absolute Percentage Error (MAPE)
  - Compare predicted values against actual next-minute values.
- **Online Monitoring:** Track the prediction error in real-time as new data arrives.

7.4. Scalability Assessment

- **Vertical Scalability:** Evaluate performance improvements when increasing resources (CPU, RAM) on individual servers (Redis, Postgres, Processing Node).
- **Horizontal Scalability:** Assess how the system could be modified to handle more instruments or higher data volumes (e.g., sharding Redis, using multiple

transformer instances, read replicas for Postgres). Identify potential bottlenecks (e.g., Redis memory, PostgreSQL write contention, ARIMA computation time).

7.5. User Experience (Dashboard Usability)

- **Responsiveness:** How quickly the dashboard loads and updates.
- **Intuitiveness:** Ease of use for selecting instruments, dates, and understanding the displayed information (charts, predictions, indicators).
- **Clarity:** How clearly the predictions and indicators are presented. Gather feedback from potential users.

8. Conclusion

This project successfully demonstrated the design and implementation of a real-time data warehousing system for stock market data. By integrating socket streaming, Redis buffering, PostgreSQL storage, ARIMA prediction, and Streamlit visualization, the system effectively addresses the challenges of handling high-velocity financial data streams. It provides a persistent data store for historical analysis, generates timely short-term predictions, and offers an interactive dashboard for visualizing market trends, forecasts, and basic trading signals like moving average (MA) for NIFTY50 and USDINR. The modular architecture allows for individual component evaluation and potential future upgrades. The outcomes highlight the feasibility of building robust, real-time analytical systems using readily available open-source technologies.

9. Future scope

- **Expand Instrument Coverage:** Add support for more stocks, indices, or asset classes.
- **Enhanced Prediction Models:** Implement more sophisticated models (e.g., LSTMs, Prophet, GARCH) potentially offering better accuracy, especially for longer forecast horizons. Compare model performances.
- **Advanced Indicators:** Incorporate more complex technical indicators (e.g., MACD, RSI, Bollinger Bands) based on historical data and predictions.
- **Cloud Deployment:** Deploy the system on a cloud platform (AWS, GCP, Azure) for better scalability, reliability, and manageability using services like EC2, RDS, ElastiCache, Kinesis/PubSub.

- **Backtesting Framework:** Integrate a framework to rigorously backtest the trading signals generated by the prediction model and indicators against historical data.
- **Error Handling and Monitoring:** Implement more robust error handling, logging, and system monitoring (e.g., using Prometheus and Grafana) across all components.
- **News Sentiment Analysis:** Integrate real-time news sentiment analysis as an additional input feature for the prediction model.
- **Distributed Stream Processing:** For very high volumes, replace Python scripts with a dedicated stream processing framework like Apache Flink or Spark Streaming.

10. References

[1] Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A Distributed Messaging System for Log Processing. *Proceedings of the NetDB Workshop*.
[2] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Kesk
[3] TimescaleDB Documentation. https://docs.timescale.com/
[4] Redis Documentation. https://redis.io/documentation
[5] Box, G. E. P., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time Series Analysis: Forecasting and Control*. John Wiley & Sons.
[6] Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2), 654-669.
[7] Streamlit Documentation. https://docs.streamlit.io/