

## **Phase 3: Finalization & Presentation**

### **Functional Requirements**

The following functional requirements were defined based on the system goals and user expectations for the batch-processing data architecture:

#### **1. Data Ingestion:**

- The system must be able to ingest historical stock price data from local files (CSV format) located in the /data folder.
- Apache NiFi should automate ingestion and move data through a processing pipeline.

#### **2. Data Splitting and Logging:**

- The ingested data must be split line by line using SplitText processor.
- Each split record should be logged using the LogAttribute processor for verification.

#### **3. Data Delivery and Storage:**

- Successfully processed records should be moved to a designated processed directory via the PutFile processor.
- A FastAPI service should be able to serve processed data through an HTTP endpoint.

#### **4. Monitoring and Observability:**

- Each processing stage should be traceable within NiFi UI.
- System logs should be stored in Docker logs and accessible for debugging.

#### **5. Automation and Orchestration:**

- All processing tasks should be run automatically upon file detection in the source directory.
- The system should handle repeatable batch operations in scheduled intervals.

### **Non-Functional Requirements**

#### **1. Scalability:**

- The architecture should support scaling up to handle larger input files by adding more CPU/RAM to the Docker containers or adding more processors in NiFi.

## 2. **Reliability:**

- Data must not be lost during processing. NiFi's built-in retry and queuing mechanisms ensure reliability.

## 3. **Maintainability:**

- The pipeline must be easy to update and reconfigure using NiFi's drag-and-drop interface and versioned templates.
- Codebase should be modular and well-documented.

## 4. **Portability:**

- The entire solution is containerized with Docker Compose to ensure platform-independent deployment.

## 5. **Security:**

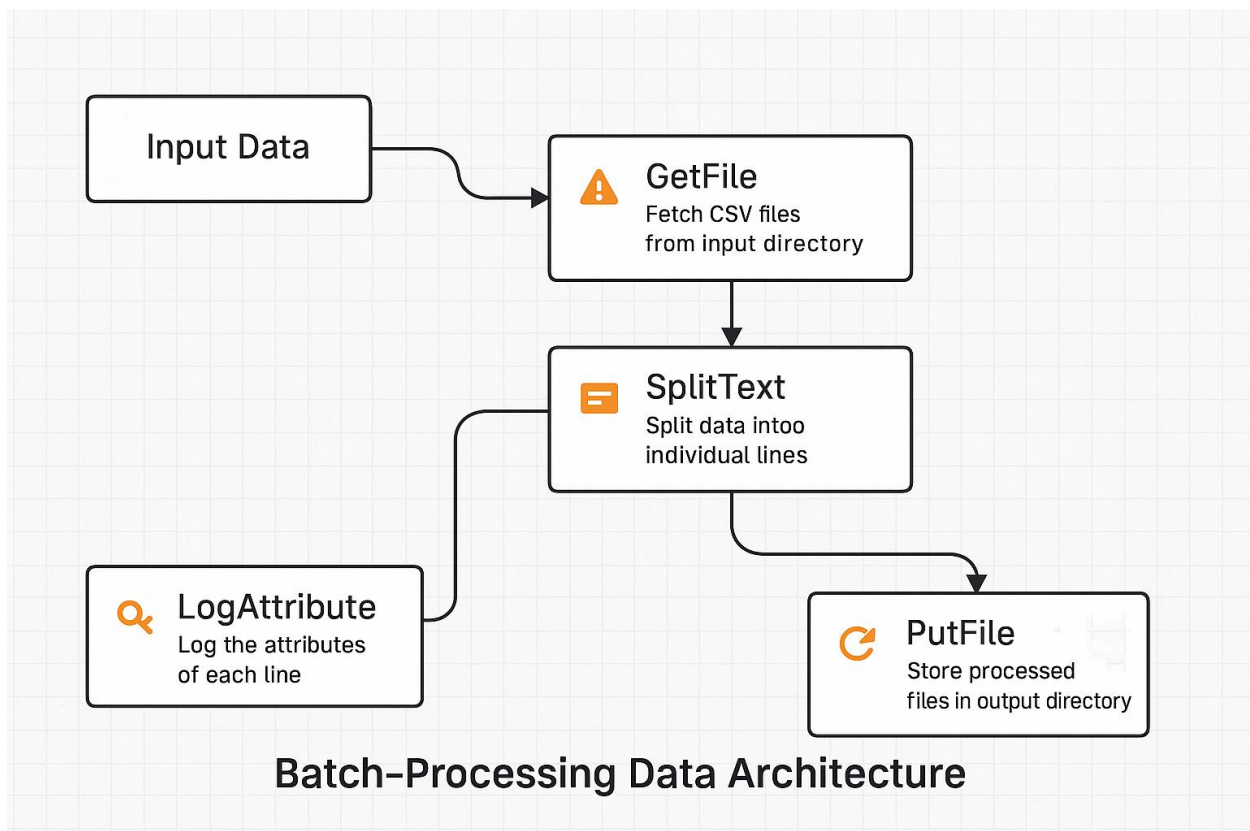
- Since the scope of the project is local batch processing, no external APIs are called and no user data is exposed, minimizing security concerns.

## 6. **Performance:**

- The solution must process data in a reasonable time (less than a minute for files with up to 10,000 records on a standard laptop).

## **Implementation Overview**

The implementation of the batch-processing data architecture was carried out using containerized services managed by Docker Compose. The core workflow leverages **Apache NiFi** for batch processing and **FastAPI** for data delivery.



## 1. Project Structure

The project is organized into the following key folders:

- /data: Contains the input CSV files (historical stock data) used by the pipeline.
- /ingestion: Includes the NiFi flow responsible for processing the data (via GetFile, SplitText, LogAttribute, PutFile).
- /delivery: Contains the FastAPI service that serves processed data from the processed directory.
- /processed: The output folder where processed files are stored after the NiFi pipeline.
- /storage: Reserved for any future expansion or intermediate data snapshots.
- /documentation: Contains explanations and architecture documentation.
- docker-compose.yml: Defines services and interconnects them using a custom bridge network.

## 2. Service Descriptions

### a. Apache NiFi

Apache NiFi runs as a service and performs the batch-processing pipeline with the following processors:

- GetFile: Reads new CSV files from the /data directory.
- SplitText: Splits the CSV into lines, each representing one record.
- LogAttribute: Logs metadata for debugging and traceability.
- PutFile: Writes processed data into the /processed directory.

The entire NiFi flow is designed using its drag-and-drop GUI and is reproducible.

## b. FastAPI

The FastAPI app is configured to serve the processed data by reading files from the /processed directory and offering endpoints like /files for listing or accessing contents. It is lightweight, fast, and easy to deploy inside Docker.

## 3. Docker-Based Deployment

All services are deployed using Docker Compose:

```
1  services:
2    nifi:
3      image: apache/nifi:1.23.2
4      ports:
5        - "8080:8080"
6      volumes:
7        - ./data:/data
8        - ./processed:/processed
9      networks:
10       - datapipeline
11
12    fastapi:
13      build: ./delivery
14      ports:
15        - "8000:8000"
16      volumes:
17        - ./processed:/app/processed
18      networks:
19        - datapipeline
20
21  networks:
22    datapipeline:
23      driver: bridge
24
```

This setup ensures consistency, easy startup with docker-compose up, and isolation between services.

#### 4. Execution Flow Summary

1. CSV data is placed in the /data folder.
2. NiFi detects the file and triggers the flow.
3. The file is split, logged, and written to the /processed directory.
4. FastAPI serves the output for any downstream system or user access.


#### Testing and Results

The batch-processing data system was tested end-to-end using a sample dataset of stock prices to validate its functionality, performance, and integration across services.

##### 1. Testing Setup

- **Input File:** A test CSV file prices.csv containing sample stock data was placed in the /data folder.
- **Docker Environment:** All services were launched using docker-compose up --build.
- **Monitoring Tool:** Apache NiFi's web UI (<http://localhost:8080/nifi>) was used to trace processing stages.
- **Delivery Test:** FastAPI was tested via browser and curl/Postman for API endpoints.

Step	Expected Result	Actual Result
Data Detection	NiFi detected the input file automatically	✓ Success
Splitting Records	CSV was split into individual lines	✓ Success
Logging Metadata	Each record logged by LogAttribute	✓ Success
Output Storage	Processed lines saved in /processed folder	✓ Success
API Access	FastAPI returned file list via /files endpoint	✓ Success

Step	Expected Result	Actual Result
System Stability	All services remained stable during run	 Success

### 3. Sample API Output

```
[
  "record_001.txt",
  "record_002.txt",
  "record_003.txt",
  ...
]
```

Each file contains one line of stock data, verifying that the system processed and exposed the records correctly.

### 4. Error Handling

- NiFi was configured to queue or re-route failed flows. During testing, no major failures were encountered.
- If an incorrect file is added, the system skips unreadable lines, logs the error, and continues processing.

### Lessons Learned

The development and deployment of this batch-processing data pipeline provided several technical and project management insights:

#### 1. Importance of Modular Architecture

Designing the system with clearly separated components—ingestion, preprocessing, and delivery—made debugging and updates easier. Using Docker Compose to manage isolated containers ensured a clean and reproducible environment.

#### 2. Hands-On with Apache NiFi

NiFi's intuitive interface made it easy to design and monitor data flows. However, small misconfigurations (e.g., incorrect directory paths or unconnected processor relationships) led to challenges that required careful examination of logs and flow queues.

### 3. Using FastAPI for Delivery

FastAPI proved to be a robust, minimalistic framework for delivering processed data. Setting up an endpoint to read from the processed folder was simple, and performance was excellent.

### 4. Workflow Debugging and Error Handling

It was essential to validate each processor and build the pipeline incrementally. NiFi's logging capabilities (e.g., LogAttribute) were invaluable for tracking progress. Handling unexpected behaviors (like empty output folders or container crashes) emphasized the importance of observability.

### 5. Data Flow Timing

Processing was triggered as expected when files were placed into the /data directory. However, a delay sometimes occurred in flow propagation due to buffer queues or misconfigured flow termination. Manual triggering and test files helped refine the flow logic.

### 6. Value of Version Control and Documentation

Keeping the code and configurations under Git version control made rollback and iteration easier. Maintaining a structured README and final report helped consolidate the learning journey and made the project portfolio-ready.

### References

- Apache NiFi Documentation. (2024). *Data Flow Management*. <https://nifi.apache.org/docs.html>
- FastAPI Documentation. (2024). *Modern Python Web Framework*. <https://fastapi.tiangolo.com>
- Docker Documentation. (2024). *Docker Compose and Containerization*. <https://docs.docker.com/compose/>
- Python Software Foundation. (2024). *The Python Language Reference*. <https://www.python.org/doc/>
- Marjanovic, B. (2023). *Price and Volume Data for All US Stocks & ETFs*. Kaggle. <https://www.kaggle.com/datasets/borismarjanovic/price-volume-data-for-all-us-stocks-etfs>
- Turnbull, J. (2022). *The Docker Book: Containerization is the New Virtualization*.