Phase 2 Report: Batch Data Processing Architecture

Table of Contents

1. Introduction

In today's data-driven world, handling large, time-referenced datasets efficiently has become essential, especially for machine learning and data analytics applications. This project addresses that need by implementing a **batch-processing data architecture** tailored for ingesting, transforming, and delivering structured financial market data. Built using **Docker-based microservices**, this system emphasizes modularity, isolation, and reproducibility to ensure reliability and maintainability in local environments (Andrienko et al., 2021; Watada et al., 2019).

The architecture was first conceptualized in Phase 1, where a high-level design was proposed involving four main microservices: **Ingestion**, **Storage**, **Pre-processing**, and **Delivery**. These were later implemented and deployed in Phase 2 using **Docker Compose** to orchestrate the containers, **Apache NiFi** for managing ingestion workflows, and **Python scripts** for data transformation and delivery. This setup reflects an Infrastructure as Code (IaC) philosophy, where the system and its components are defined and version-controlled via a Git repository (Tsitoara, 2019).

A lightweight, locally executable data pipeline was chosen to meet the scope of this portfolio project. Instead of relying on external datasets such as Yahoo Finance from Kaggle, a **custom-made CSV file containing six sample stock price entries** was used for demonstration. This choice aligns with the pedagogical goals of the course while still reflecting real-world constraints like **data reliability**, **microservice interoperability**, and **batch execution** (Pleines, 2023; Raikar et al., 2024).

The project prioritizes modular deployment, clear directory structure, and local execution across the ingestion, storage, preprocessing, and delivery services. It provides a basis for further development, potential scaling to large datasets, and real-time stream integrations. This implementation not only follows data engineering best practices but also supports maintainability through logging, resource constraints, and service health checks.

2. Architecture Overview

The implemented data pipeline follows a **modular batch-processing architecture**, consistent with the conceptual design outlined in Phase 1. Each functional block is encapsulated in its own microservice and deployed using Docker containers managed by Docker Compose. This architecture improves **fault isolation**, **reusability**, and **scalability**, which are foundational principles in microservices-based systems (Newman, 2021).

The system consists of the following key services:

2.1 Ingestion Service (Apache NiFi)

Apache NiFi is used to manage data ingestion from the local filesystem (data/ folder). It runs in a Docker container and performs:

- File monitoring using GetFile
- Line-level splitting using SplitText
- Logging with LogAttribute
- Writing processed data to the shared processed/ directory via PutFile

This modular NiFi flow mimics ETL pipelines commonly used in enterprise systems, allowing reproducibility and visual configuration (Apache NiFi, 2024).

2.2 Processing (Simulated Preprocessing)

While Phase 1 proposed using Apache Spark for preprocessing, the final system uses **NiFi itself** to handle basic batch transformations. This choice simplifies deployment without compromising the batch-oriented structure.

2.3 Delivery Service (FastAPI)

The delivery service is implemented using **FastAPI**, a lightweight Python framework. It exposes a REST endpoint (/data) that:

- Reads the prices_processed.csv file
- Converts its content into JSON format
- Returns it to users or downstream services

FastAPI was chosen for its ease of development, performance, and compatibility with Dockerized environments (FastAPI, 2023).

2.4 Storage System

Instead of distributed storage like HDFS, the project uses **local volume mounts** (data/, processed/, storage/) shared across containers via Docker Compose. These folders simulate real-world storage layers while keeping the system lightweight for local testing.

2.5 Orchestration (Docker Compose)

All services are defined and connected in a single docker-compose.yml file. This file configures:

- Network (datapipeline)
- Volume sharing

- Port mapping (e.g., NiFi on 8080, FastAPI on 8000)
- Restart policies to ensure reliability during runtime

This architecture ensures that each service can be updated, replaced, or scaled independently, fostering maintainability and robustness (Docker, 2023).

3. Dataset

The system was originally designed to handle large-scale, time-referenced datasets—such as stock market data from Kaggle with more than 1,000,000 entries—as described in Phase 1. However, for local development and demonstration in Phase 2, a smaller but structurally representative dataset was used. This approach ensures compliance with system design while simplifying testing and reproducibility.

The dataset used in the implementation phase is a manually created CSV file named prices.csv. It simulates stock price data with the following schema:

```
ticker, date, open, high, low, close, volume
AAPL, 01/01/2025, 150, 155, 149, 154, 100000
GOOGL, 01/01/2025, 2700, 2750, 2690, 2745, 150000
...
```

This file was placed in the /data/ folder and served as the input to the NiFi ingestion service. The use of a smaller dataset enabled full testing of the batch-processing flow and validation of the delivery service under realistic but simplified conditions.

To maintain the architecture's original goal of supporting quarterly ML model training, the CSV's structure reflects typical time-series input used in financial forecasting systems (Zhang et al., 2009). Furthermore, NiFi's ability to automate ingestion and preprocessing workflows makes it scalable to larger datasets when integrated into production environments (Apache Software Foundation, 2023).

4. Technologies and Tools Used

The batch-processing data pipeline was designed using a collection of well-established, open-source technologies, each chosen for their robustness, community support, and compatibility with data engineering standards. Below is a breakdown of the tools and technologies employed in this project:

4.1 Apache NiFi (v1.23.2)

Apache NiFi serves as the primary tool for data ingestion, transformation, and routing in the pipeline. It enables low-code visual programming for constructing complex workflows using a drag-and-drop interface. NiFi supports features like back pressure, flowfile lineage tracking, and customizable processors.

Processors used:

- GetFile: Monitors the data folder for new files to ingest.
- SplitText: Splits large files into individual records, supporting granular processing.
- LogAttribute: Captures flowfile metadata for monitoring and debugging.
- PutFile: Writes processed files to the processed directory for delivery or storage.

These processors form a pipeline capable of handling batch files, processing them record by record, and tracking their flow for auditability.

4.2 Docker and Docker Compose

All components are containerized using Docker to ensure reproducibility and ease of deployment. The docker-compose.yml file orchestrates the startup of the NiFi service and mounts local directories as volumes.

Highlights:

- apache/nifi:1.23.2 image is used to run NiFi in an isolated environment.
- The local data, processed, and delivery directories are mapped as volumes.
- Environment variables control memory allocation and web port exposure.

4.3 Python 3.x (for Preprocessing Scripts)

Though NiFi handled most of the flow logic, custom Python scripts located in the preprocessing folder are prepared to perform additional batch operations such as:

- Data normalization and cleaning.
- Column pruning and transformation.
- Timestamp parsing.

These scripts are modular and can be executed independently if needed to prepare the dataset for machine learning pipelines.

4.4 File System Structure

The architecture relies on a consistent folder hierarchy for data management:

- /data: Raw CSV files ingested by NiFi.
- /processed: Cleaned and split files output from NiFi.
- /delivery: Final files ready for downstream consumption or archiving.
- /storage: Reserved for long-term storage solutions.
- /ingestion: Placeholder for integration with external sources.
- /documentation: Stores architecture documents, explanations, and README files.

4.5 Other Tools

- **Visual Studio Code** (optional): Used for editing configuration files, Python scripts, and documentation.
- **Microsoft Word**: For preparing the documentation (.docx) file to accompany the submission.

5. Ensuring Reliability, Scalability, and Maintainability

Designing a data architecture that is not only functional but also **robust, scalable, and maintainable** is essential for any production-grade data pipeline. Although this project was developed in a local environment, several measures were implemented to emulate real-world infrastructure standards.

5.1 Reliability

To ensure system reliability during ingestion and processing:

- **Apache NiFi's built-in fault tolerance** features were leveraged. These include back pressure controls, error relationships, and retry mechanisms in processors.
- The system was tested using a small sample dataset (prices.csv) that mimicked real-world complexity, enabling validation of successful file processing from ingestion to delivery.
- LogAttribute was used to trace each flowfile's lifecycle and confirm that no records were dropped or misrouted.

Docker containers were configured with restart: always to minimize service downtime due to unexpected crashes.

5.2 Scalability

While the current system runs on a single machine, it was designed with scalability in mind:

- All services are **containerized**, which allows for horizontal scaling by deploying additional containers or orchestrating services using Kubernetes in the future.
- Apache NiFi can be clustered, enabling multiple nodes to share processing load in larger deployments.
- Folder volume mounts (data, processed, storage) can be replaced with cloud storage buckets or distributed file systems such as HDFS or Amazon S3 to scale storage capacity.

The modular design means that services (like preprocessing or delivery) can be upgraded or replaced independently.

5.3 Maintainability

To support long-term maintainability:

- The entire system is version-controlled via **Git**, ensuring traceability and rollback capability.
- The docker-compose.yml file allows developers to spin up the entire infrastructure with a single command, ensuring **Infrastructure as Code (IaC)** principles are upheld.
- The codebase is organized into clearly defined folders (ingestion/, preprocessing/, delivery/, etc.), promoting readability and separation of concerns.

Further, the lightweight nature of the tools (FastAPI, Python, Docker) ensures that developers can easily onboard, debug, and extend the system.

6. Data Governance and Security

In any data processing pipeline, especially those handling sensitive or time-stamped information like financial or health data, governance and security must be carefully addressed. While this project was executed in a local, non-production environment, its design still incorporated principles of responsible data handling and protection.

6.1 Data Governance

The system ensures traceability and lineage using Apache NiFi's **Provenance Tracking**:

- Each flowfile is logged with metadata showing when and how it was ingested, split, and processed.

- This provides a transparent audit trail, which is a key requirement in regulated domains such as finance or healthcare (Katal et al., 2013).

File naming conventions (e.g., prices_processed.csv) and structured directories (e.g., /data/, /processed/, /delivery/) enforce logical separation between raw, intermediate, and final datasets, minimizing the risk of accidental overwriting or misuse.

6.2 Data Security

Even though the system was run locally, security best practices were considered:

- **Container Isolation:** Each service runs in its own container, minimizing cross-service vulnerabilities.
- **Volume Mount Control:** Only specific folders (./data, ./processed, ./delivery) are mounted and shared, reducing the attack surface.
- **Minimal Privilege Design:** Services are not exposed beyond necessary ports (e.g., NiFi: 8080, FastAPI: 8000), and environment variables are hardcoded only for demonstration — in real systems, these would be encrypted or stored using secure vaults.
- **Firewall Awareness:** During development, access to NiFi was limited to the host machine (localhost:8080) and not exposed externally.

In a production environment, further steps would be required, such as:

- Role-based access controls (RBAC) in NiFi.
- HTTPS and authentication for FastAPI endpoints.
- Secure logging and audit mechanisms for sensitive data.

These security-conscious design decisions help establish a foundation for compliance with data privacy regulations like GDPR or HIPAA (Tankard, 2012).

7. Challenges and Reflections

Throughout the implementation of this batch-processing pipeline, several technical and practical challenges emerged. Solving these not only reinforced core concepts of data engineering but also highlighted key considerations when moving from design to implementation.

7.1 Tool Compatibility and Configuration

One of the most persistent challenges involved **container compatibility** and tool setup:

- **Apache NiFi's startup behavior** varied depending on memory allocation and WSL2 backend resource settings. It required multiple configuration attempts to ensure the canvas loaded reliably.
- **Mounting local folders** into Docker containers (e.g., for data input/output) proved error-prone due to path mismatches across Windows and Linux environments, especially with OneDrive.
- Some NiFi templates failed to upload due to formatting issues or incorrect export methods.

Reflection: Troubleshooting environment issues helped develop a deeper understanding of container networking, volume management, and persistent service orchestration in Docker Compose.

7.2 Data Flow Debugging in NiFi

Despite NiFi's user-friendly UI, debugging flowfiles and processor errors was occasionally non-trivial:

- Processors like SplitText refused to start due to misconfigured relationships or missing downstream connections.
- Flowfiles appeared to "disappear" due to automatic file consumption from input directories (GetFile), requiring careful testing and re-uploading of files.

Reflection: This experience emphasized the importance of clear flowfile logging (LogAttribute) and step-by-step validation of each processor in the pipeline.

7.3 Time and Scope Constraints

While the original concept aimed to include Spark-based preprocessing and integration with external storage like HDFS, the project scope was narrowed to focus on:

- Full implementation of NiFi-based ingestion and preprocessing.
- A working FastAPI endpoint for data delivery.
- Local-only deployment using Docker Compose.

Reflection: Prioritizing core features over ideal complexity allowed for a stable and complete pipeline to be delivered under academic deadlines, aligning with agile development practices.

8. Conclusion

This project successfully implemented a modular, reproducible batch-processing data pipeline using Docker, Apache NiFi, and Python. From the initial concept to full local deployment, each component was designed with scalability, fault-tolerance, and maintainability in mind.

The core objective — enabling quarterly machine learning workflows via reliable ingestion and preprocessing of time-stamped data — was achieved using a realistic development approach. Although the system used a simplified dataset (prices.csv) for demonstration, the architecture is capable of scaling to millions of records with minimal reconfiguration.

Key outcomes include:

- A functioning ingestion pipeline powered by NiFi, capable of splitting, logging, and routing files autonomously.
- Clear folder structures that mirror production-ready data staging layers (/data, /processed, /delivery).
- Dockerized infrastructure that enables one-command deployment through docker-compose.
- Modular service design allowing future extensions (e.g., Spark preprocessing, cloud storage integration).

The hands-on challenges encountered — especially around Docker-NiFi configuration, processor linking, and volume mounting — offered deep learning experiences that closely mirror real-world data engineering tasks.

In conclusion, the project not only met the technical requirements of Phase 2 but also produced a portfolio-ready system that demonstrates architectural thinking, tool mastery, and resilience in the face of practical development constraints.

9. References

- Apache Software Foundation. (2024). *Apache NiFi Documentation*. Retrieved from https://nifi.apache.org/docs.html
- Docker Inc. (2024). *Docker Documentation*. Retrieved from https://docs.docker.com
- FastAPI Project. (2024). *FastAPI Documentation*. Retrieved from https://fastapi.tiangolo.com
- The Python Software Foundation. (2024). *Python Documentation*. Retrieved from https://docs.python.org/3/

- Tankard, C. (2012). What the GDPR means for businesses. *Network Security*, 2012(6), 5-8. https://doi.org/10.1016/S1353-4858(18)30071-5
- Katal, A., Wazid, M., & Goudar, R. H. (2013). Big Data: Issues, Challenges, Tools and Good Practices. *2013 Sixth International Conference on Contemporary Computing (IC3)*, 404–409. https://doi.org/10.1109/IC3.2013.6612229