Ministry of Higher Education and Scientific Research
University of Mohamed Boudiaf, Msila

Faculty of Mathematics and Computer Science
Department of Computer Science

Mater 1

Big Data Analytics

Dr. N.CHALABI

Novembre 2025

Ingestion and Data Pipeline Architectures

Collecting data is only the first step toward turning raw information into actionable insight.

Modern organizations whether in e-commerce, finance, or healthcare must continuously move massive amounts of data from many different sources into systems where it can be **processed, analyzed, and stored.**

This chapter introduces **data ingestion and data pipelines**:
the critical pathways that ensure data flows **smoothly, reliably, and efficiently** from its origin to its destination.

Why Data Ingestion Matters

Organizations generate enormous amounts of data every second.

Platforms like **Netflix** (millions of viewing events/minute) or **Uber** (real-time driver/passenger tracking) **rely on fast, reliable data movement**. The same applies to banks, e-commerce sites, hospitals, and social networks.

What Is Data Ingestion

Data ingestion = bringing data from multiple sources into a central system
(databases, files, sensors, apps, streams) so it can be **stored, processed, and analyzed**.

Without ingestion, there is **no analytics, no dashboards, no machine learning.**

It is the **first and foundational step** of any Big Data pipeline.

What Is a Data Pipeline

Once data is ingested, it cannot stay **raw or unstructured**.

It must be **cleaned, transformed, and prepared** before it can be stored or analyzed.

A **data pipeline** is an organized sequence of steps that moves data from its **source** to its **destination**, applying the necessary processing along the way.

types of a Data Pipeline

1.Ingestion

Collecting data from source systems.

2.Cleaning

Removing errors, duplicates, and irrelevant entries.

3.Transformation

Converting data into a usable format

(e.g., normalizing values, converting time formats, enriching data).

4.Storage

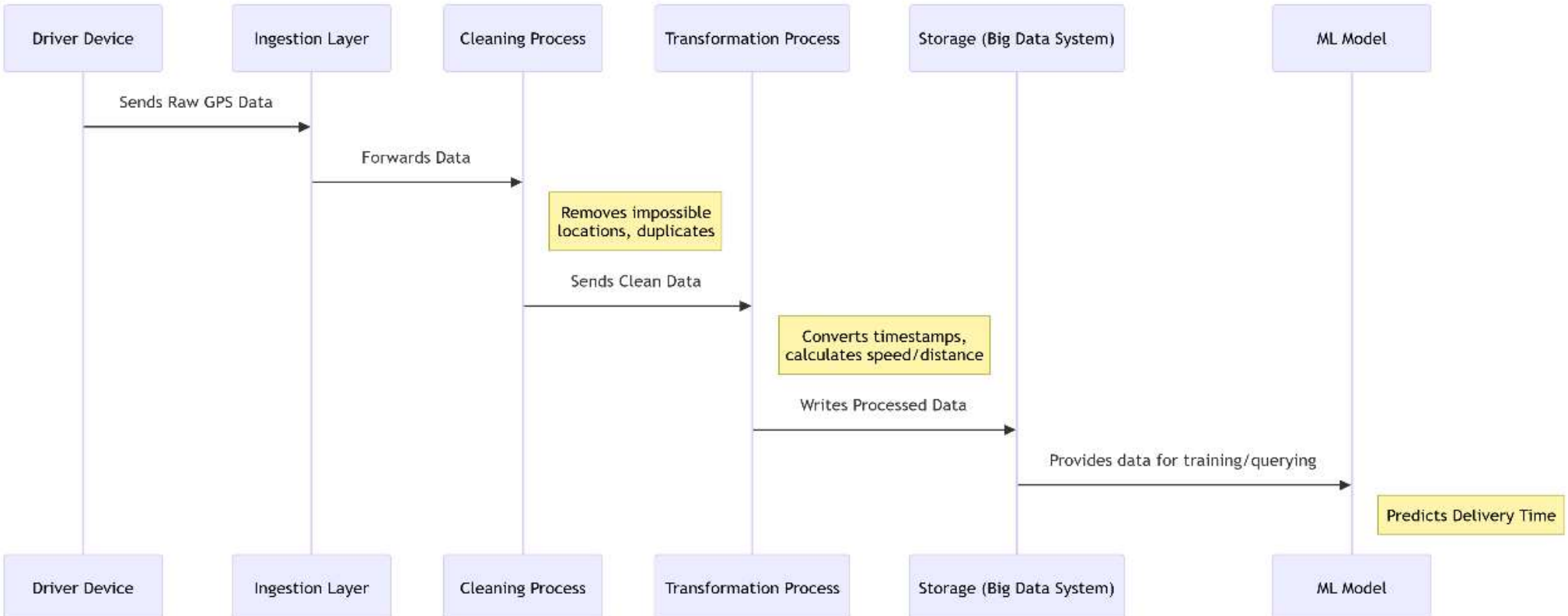
Sending processed data to a target system

such as HDFS, NoSQL databases, or a Data Lake.

5.Analysis

Making data available for querying, reporting, or machine learning.

Example of a Pipeline (Uber Eats)



ETL vs. ELT: Two Approaches to Data Pipelines

Should data be transformed **before** or **after** storage❓

This leads to two major paradigms: **ETL** and **ELT**.

ETL – Extract → Transform → Load

- **Extract**
  Pull data from sources (databases, files, APIs).
- **Transform**
  Clean, enrich, and reformat **before** loading.
- **Load**
  Store the processed data into the target system
  (Data Warehouse, NoSQL, HDFS).

ELT – Extract → Load → Transform

- **Extract**
  Pull data from sources.
- **Load**
  Store **raw** data immediately in a Data Lake or cloud storage.
- **Transform**
  Clean and reformat **after loading**, using engines like Spark or Flink.

Why Two Approaches

ETL (Traditional)

✓ Used when storage was expensive

✓ Data Warehouses required clean, structured data before loading

ELT (Modern Big Data)

✓ Enabled by cheap storage (HDFS, S3, Data Lakes)

✓ Transformations done later with scalable engines (Spark, Flink)

Key Differences and Trade-offs of ETL /ELT

Aspect	ETL	ELT
When transformation happens	Before loading	After loading
Storage cost	Lower (only clean data stored)	Higher (raw + processed kept)
Flexibility	Less — pipeline must be redesigned if source schema changes	More — raw data preserved, transformations can adapt later
Latency	Slower — must wait for transformation before load	Faster access to raw data
Best suited for	Stable, structured sources	Diverse, evolving, unstructured sources

1. ETL / ELT Tools: Sqoop and NiFi

Two important open-source tools from the Apache ecosystem are **Sqoop** and **NiFi**. Each one was designed to solve a different type of ingestion challenge, and together they give us a good overview of how ETL/ELT pipelines are built in real systems.

1. Apache Sqoop

- **Definition**: Sqoop (short for *SQL-to-Hadoop*) is a tool designed to transfer large amounts of data efficiently between **relational databases** (like MySQL, Oracle, or PostgreSQL) and the **Hadoop ecosystem** (HDFS, Hive, HBase).
- **Origins of Sqoop:** Sqoop was created to bridge the old world of relational databases with newer Big Data systems. Before Hadoop and NoSQL, most enterprise data was locked in traditional RDBMS systems. Sqoop fills that gap.

It became a top-level Apache project in March 2012.

As of 2021, Sqoop has been moved to the Apache Attic, meaning it is no longer under active development.

How Sqoop Works

- **User Issues a Command**

  Specify what to copy (database/table) and the destination

  → HDFS, Hive, HBase, etc.

- **Sqoop Analyzes the Table**

  Reads metadata (e.g., primary keys) to plan the import.

- **Data Splitting**

  Divides the table into **parallel chunks** instead of copying it as one big block.

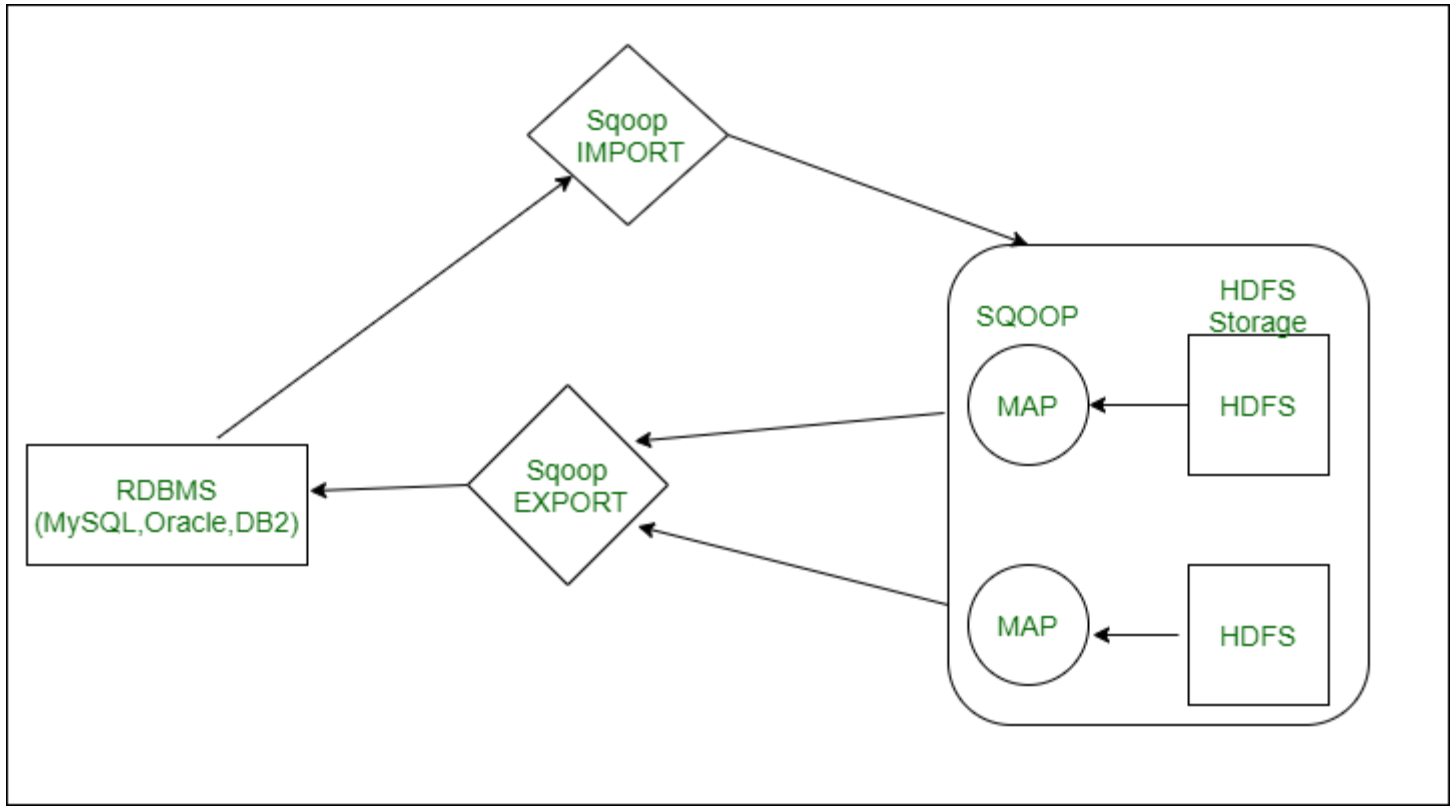
- **Parallel Transfer**

  Each chunk is imported simultaneously into Hadoop.

  Results appear as **part-files**.

- **Optional Export**

  Sqoop can also work in reverse—sending processed data **back to relational databases**.



Sqoop Architecture

Key Features & Capabilities

Feature	Description
Parallel Import/Export	Sqoop splits data into chunks and processes them in parallel, making large transfers faster.
Connectors for many relational databases	Supports MySQL, Oracle, SQL Server, PostgreSQL, and others using JDBC.
Support for full and incremental loads	You can import all records, or just new/changed ones since last import.
Integration with Hive/HBase	Sqoop can import directly into Hive tables or HBase, or export processed data from Hadoop back into RDBMS.
Security & Compression	Supports Kerberos-based authentication in secured environments; supports compressed output.

Use Case (Classroom Example)

Imagine a university's student records are kept in MySQL. At night, you want to import all updated records into Hadoop for analytics (e.g. analyzing patterns of grades, attendance). With Sqoop:

- You configure a **sqoop import** command pointing to the student table.
- Sqoop divides the table into parallel splits, connects to MySQL, and imports chunks.
- The imported files are stored in HDFS under /user/analytics/students/.
- Over time, you can schedule incremental imports (only rows updated since the last import) rather than reimporting the entire table.

2. Apache NiFi

- **Definition**: NiFi is a tool for **real-time data ingestion and flow management**. It allows data to be collected, transformed, and routed across systems using an intuitive **drag-and-drop interface**.
- **Why It Exists**: Modern organizations deal not only with databases, but also with **streaming data** from sensors, APIs, mobile apps, logs, and IoT devices. They need a flexible tool to manage these flows in real time.

THE CORE CONCEPTS OF APACHE NIFI

Apache NiFi works with several key building blocks.
The most important ones are:

1. **DFM (Data Flow Manager)**
2. **FlowFile**
3. **Bulletin**
4. **Processor**
5. **Connection**
6. **Controller Service**
7. **Flow Controller**
8. **Process Group**

These components work together to move, manage, and process data.

1. DFM

In simple words, a DFM is the user who has different permissions and complete management of Apache NiFi.

2. FlowFile:

A **FlowFile** is the main object that moves through a NiFi flow.

It contains:

- **Attributes** → information about the file (UUID, filename, path, size...)
- **Content** → the actual data (text, CSV, images, JSON, etc.)

Every FlowFile has a unique ID so NiFi can track it.

3. Bulletin

Bulletins act as **alerts or messages** from NiFi components.

They show:

- Rolling statistics
- Component status
- Severity levels (Debug, Info, Warning, Error)

Helps users quickly understand issues or performance problems.

4. Processor

A **Processor** performs the real work in a NiFi flow.

It can:

- Read data from files or databases
- Transform or clean data
- Run Python scripts
- Execute SQL queries

Directly interacts with FlowFile attributes and content.

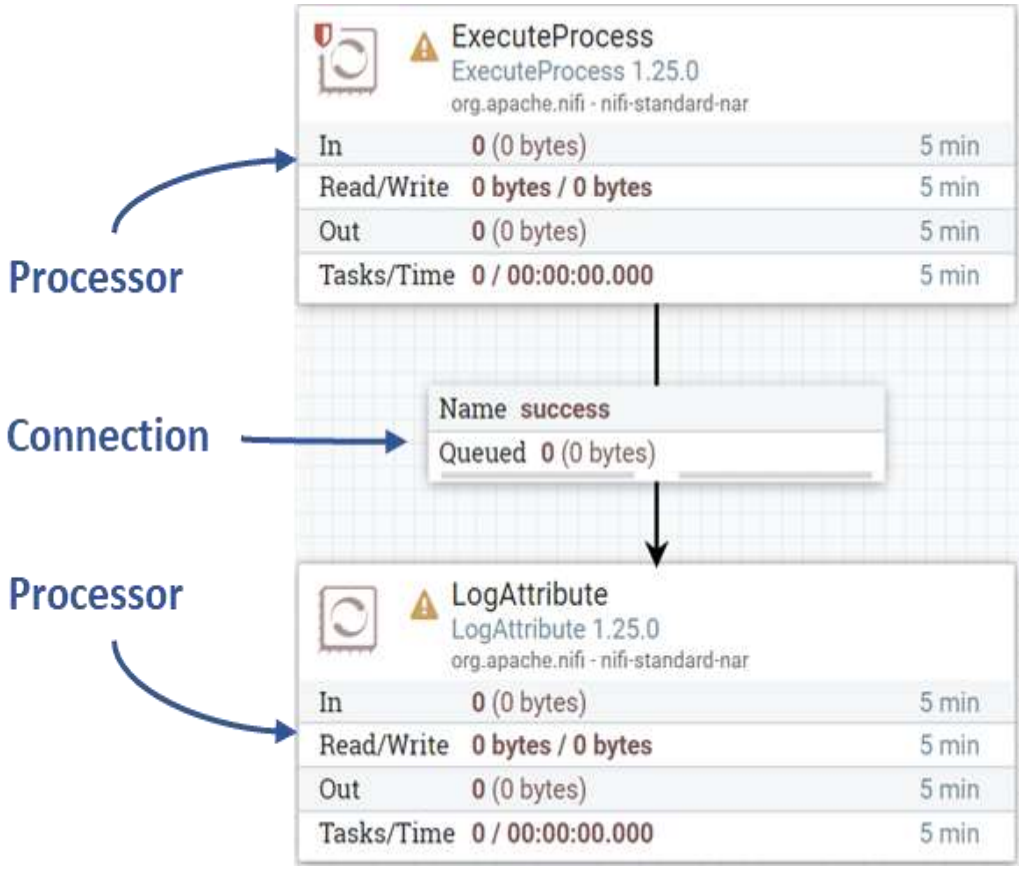
5. Connection

A **Connection** links two processors together.

It contains a **queue** that holds FlowFiles temporarily.

- This allows:
  - Smooth data movement
  - Processors to work at different speeds

FlowFiles stay in the queue until the next processor is ready.



6. Controller Service

A **Controller Service** provides reusable services for processors.

Starts automatically when NiFi starts.

Examples:

- **DBCPConnectionPool** → connects NiFi to databases
- **CSVRecordSetWriter** → writes FlowFile data as CSV

Once created, many processors can use the same service.

7. Flow Controller

- The **Flow Controller** manages system resources.
- Controls all threads used by processors.
- Adjusts the number of threads automatically depending on load.

8. Process Group

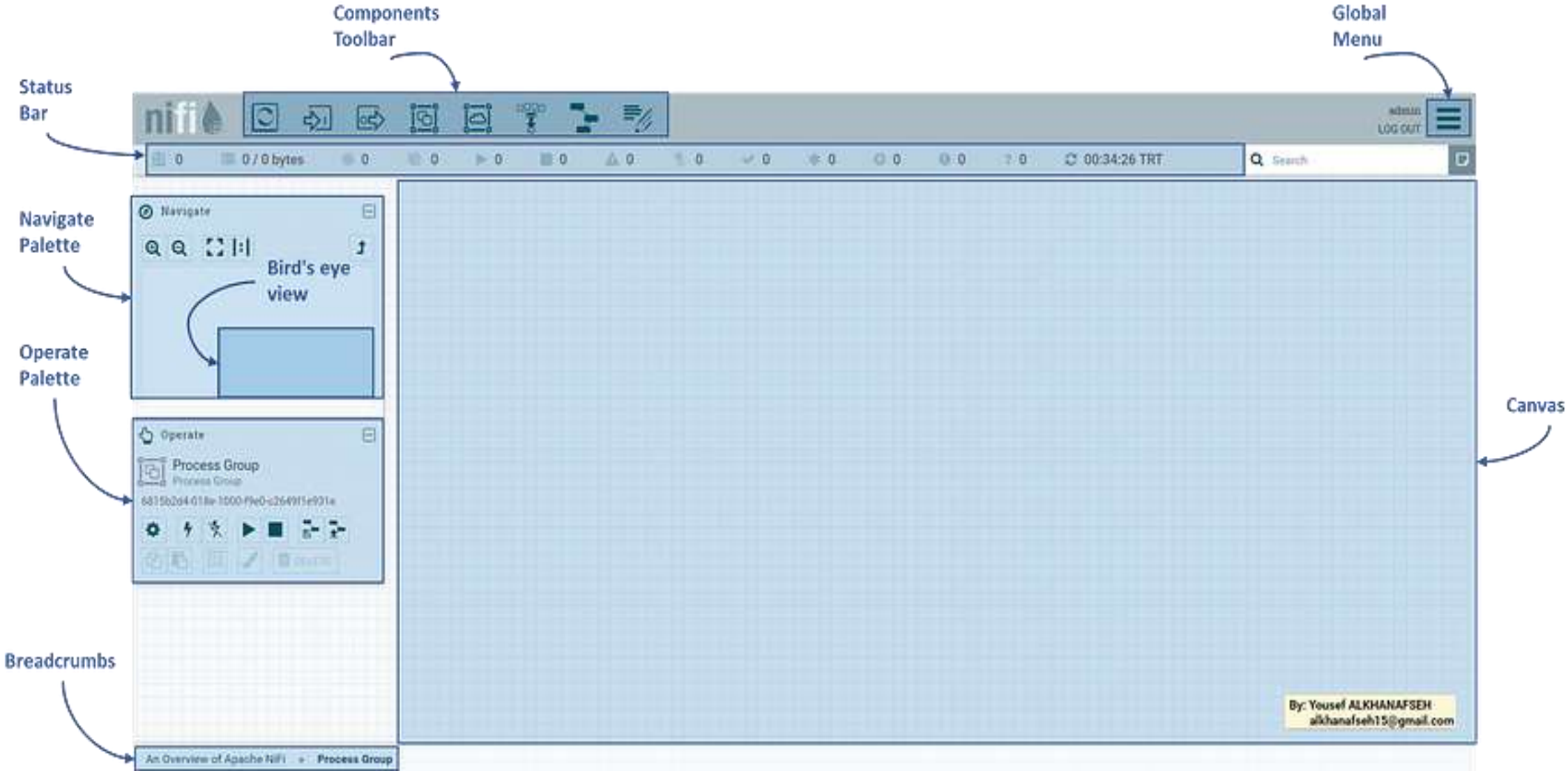
A **Process Group** organizes a set of processors and connections.

Helps group related tasks into one unit (like a sub-flow).

Uses:

- **Input Ports** to bring data in
- **Output Ports** to send data out

Makes large data flows easier to design and manage.


Components
Toolbar
Global
Menu
Status
Bar
Navigate
Palette
Bird's eye
view
Operate
Palette
Canvas
Breadcrumbs
Navigate
Operate
Process Group
An Overview of Apache NiFi
Process Group
By: Yousef ALKHANAFSEH
alkhanafseh15@gmail.com

Main Uses:

- Collecting logs from servers.
- Ingesting data from APIs or IoT devices.
- Routing data into Kafka, Hadoop, or cloud storage.

Example: A hospital wants to process **real-time patient monitoring data** from wearable devices. NiFi collects sensor readings, removes errors, adds metadata (patient ID, timestamp), and routes them to Apache Kafka for streaming analysis.

2. Streaming (Kafka, Flume, Kinesis)

What is Data Streaming 

In the context of Big Data, **streaming** refers to the continuous flow of data from various sources to a destination system, where it can be stored, processed, and analyzed in real time. Unlike batch processing, where data is collected and processed in bulk at scheduled intervals, streaming ensures that data is captured and made available almost instantly after it is generated.

Why Streaming Matters in Big Data

Modern organizations cannot rely solely on batch methods, which may introduce delays of hours or even days before insights are available. Many scenarios require **real-time analysis and action**, such as:

- Fraud detection in banking systems.
- Monitoring server logs for immediate troubleshooting.
- Providing personalized recommendations while a user is browsing a platform.
- Updating dashboards for operational monitoring in real time.

For this reason, **streaming has become a central component of Big Data ecosystems.**

Streaming Tools in the Big Data Ecosystem

Several frameworks have been developed to handle real-time data ingestion and processing. Among the most widely used are:

- **Apache Kafka**
- **Apache Flume**
- **Amazon Kinesis**

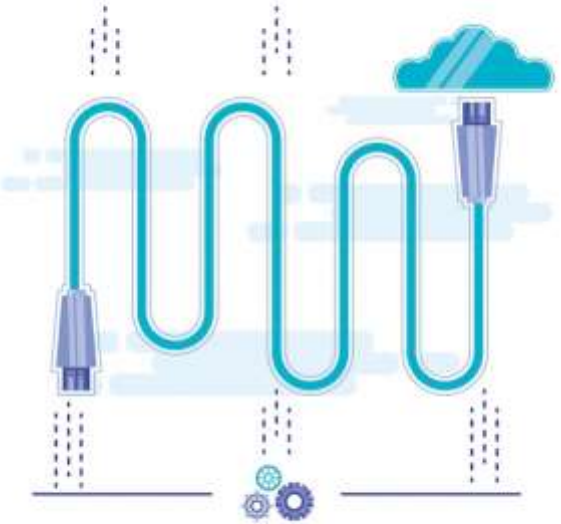
1. Apache Kafka

Apache Kafka is a distributed, high-throughput, **real-time,** low-latency data streaming platform. It's built to transport large volumes of data in real-time between systems, without needing to develop hundreds of intricate integrations.

Rather than integrating each system with each other system, you connect everything to Kafka and leave the data movement to Kafka, as a high-speed, fault-tolerant message bus.

Originally developed at **LinkedIn** and now under the maintenance of the Apache Software Foundation, Kafka is relied on by industry leaders such as Netflix, Uber, Walmart, and LinkedIn to process real-time data ingestion, streaming analytics, and event processing.

Whether for user behavior tracking, log aggregation, fraud detection, or fueling recommendation engines Kafka scales with ease, provides millisecond-level performance, and guarantees data never goes missing.



Understanding Apache Kafka

Imagine This Scenario

Think of an **airport announcement system**:

- Planes land and depart → each event needs to be communicated.
- There's a **loudspeaker system (the topic)** where announcements are broadcast.
- Some people (passengers, taxi drivers, baggage handlers) want to hear **different types of announcements** (arrivals, departures, gate changes).
- Everyone doesn't listen at the same time some might arrive late, but they can still check the **board history** (Kafka keeps messages for a while).

Kafka works in a very similar way:

- **Events** = things happening in the real world (user clicks, payment completed, sensor reading, etc.).
- **Producers** = the ones who send the events (like the control tower or staff making announcements).
- **Topics** = the channels where events are grouped (arrivals, departures, payments, orders, etc.).
- **Consumers** = applications that “tune in” to topics to get updates (like passengers, staff, or taxis listening to the loudspeaker).
- **Kafka Cluster** = the set of servers storing and forwarding events so nobody misses them.

Step-by-Step: How Kafka Operates

- **Producing Events**
  - An application (like an e-commerce site) **sends an event** to Kafka. Example: *"User123 purchased a phone."*
  - This event is written into a **topic** (say, orders).
- **Storing Events**
  - Kafka **stores the event** in that topic.
  - Events inside topics are split into **partitions** so many servers can share the load.
  - Each event is kept in order within its partition.
- **Consuming Events**
  - Other applications subscribe to the topic.
  - For example:
    - The **billing system** reads the event to generate an invoice.
    - The **inventory system** reads it to reduce stock.
    - The **shipping system** reads it to prepare delivery.
  - All these systems get the same event, independently, without interfering with each other.
- **Retention & Replay**
  - Unlike a normal queue that deletes messages once read, Kafka **keeps events for a set time** (e.g., 7 days).
  - If a new consumer starts later, it can "rewind" and read past events.

Why Do We Use Kafka

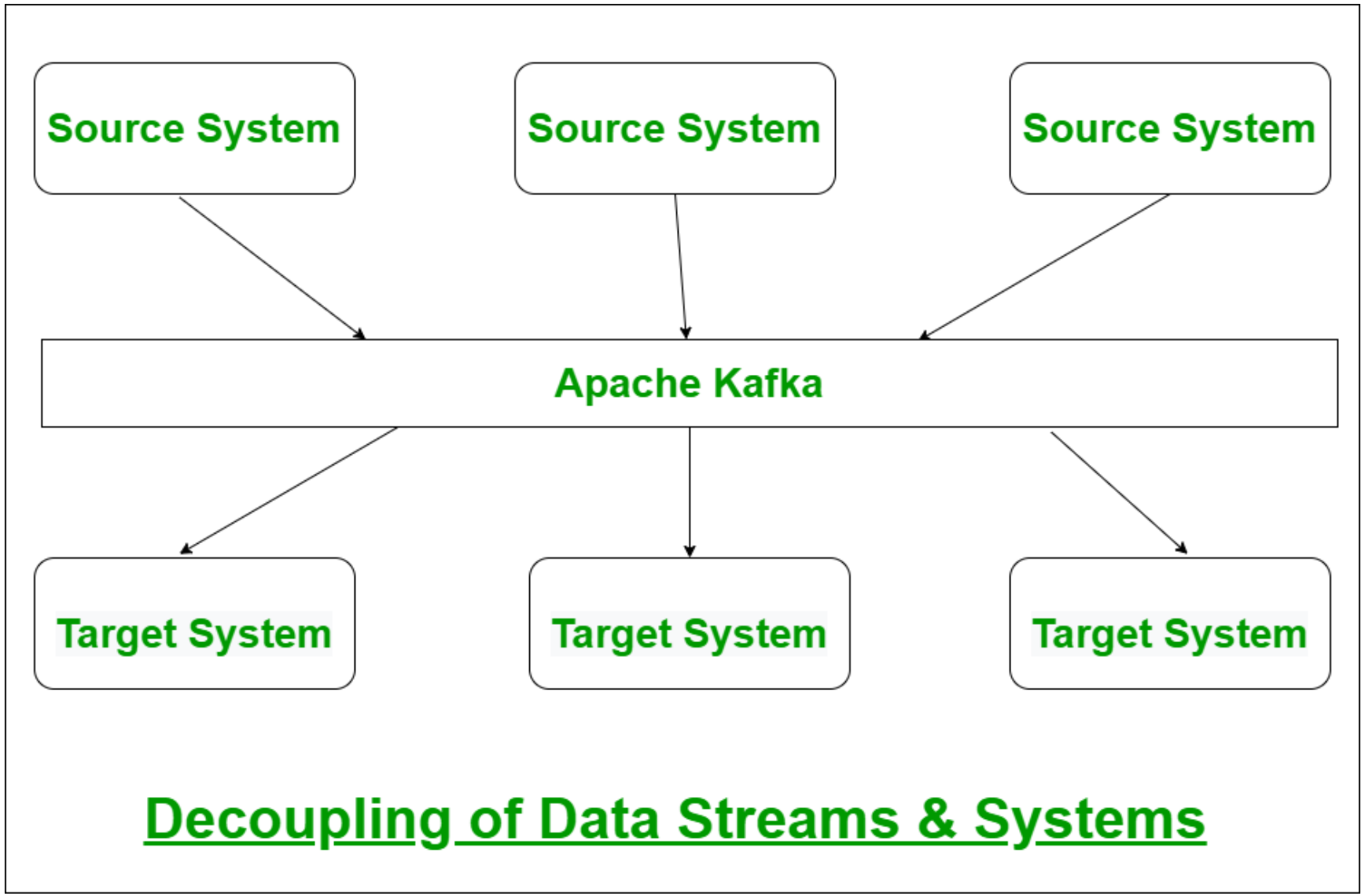
- **Real-Time Reactions**: Systems don’t need to wait for batch updates. They act instantly when events happen.
- **Scalability**: Can handle millions of events per second by distributing across partitions and brokers.
- **Reliability**: If one consumer fails, events are still stored; they can catch up later.
- **Flexibility**: Many different applications can consume the same event, each for its own purpose.

A Simple Example

E-commerce Order Flow with Kafka:

- A customer places an order → event produced into Kafka (orders topic).
- The event is picked up by:
  - Payment service → charges credit card.
  - Inventory service → updates stock.
  - Email service → sends confirmation.
  - Analytics service → tracks sales trends.

Without Kafka: each service would need to talk directly to every other service → messy, fragile connections.
With Kafka: services are decoupled; they just read from the same stream of events.



Decoupling of Data Streams & Systems

Kafka's Use Cases

Kafka is used in many domains because it supports real-time, large-scale event streaming with reliability. Some common examples:

- **Real-time streaming pipelines**

  Kafka can move millions or billions of events between systems (e.g. from a front-end application to analytics engines) without data loss or duplication.

- **Real-time applications**

  Applications can consume events as they happen to update user experiences in real time: e.g., live dashboards, dynamic recommendations, inventory updates.

- **Microservices communication**

  Kafka can serve as the event bus between microservices, enabling loosely coupled, event-driven architectures.

- **IoT and data pipelines**

  Kafka streams data from sensors and devices into analytics engines or storage systems.

- **Data lakes and real-time ingestion**

  Kafka is often used as the ingestion layer feeding data into data lakes or warehouses in real time.

Apache Flume

- In many organizations, servers and applications generate a massive number of **log files** every second. These logs contain valuable information such as user activity, system performance, or security events but they are often scattered across hundreds or thousands of machines. To analyze them effectively, all logs must be collected in one central location, usually in **Hadoop Distributed File System (HDFS)** or another big data platform.

Apache Flume was designed for exactly this task. It is a **distributed, reliable, and scalable system for collecting, aggregating, and transporting large amounts of log data**.

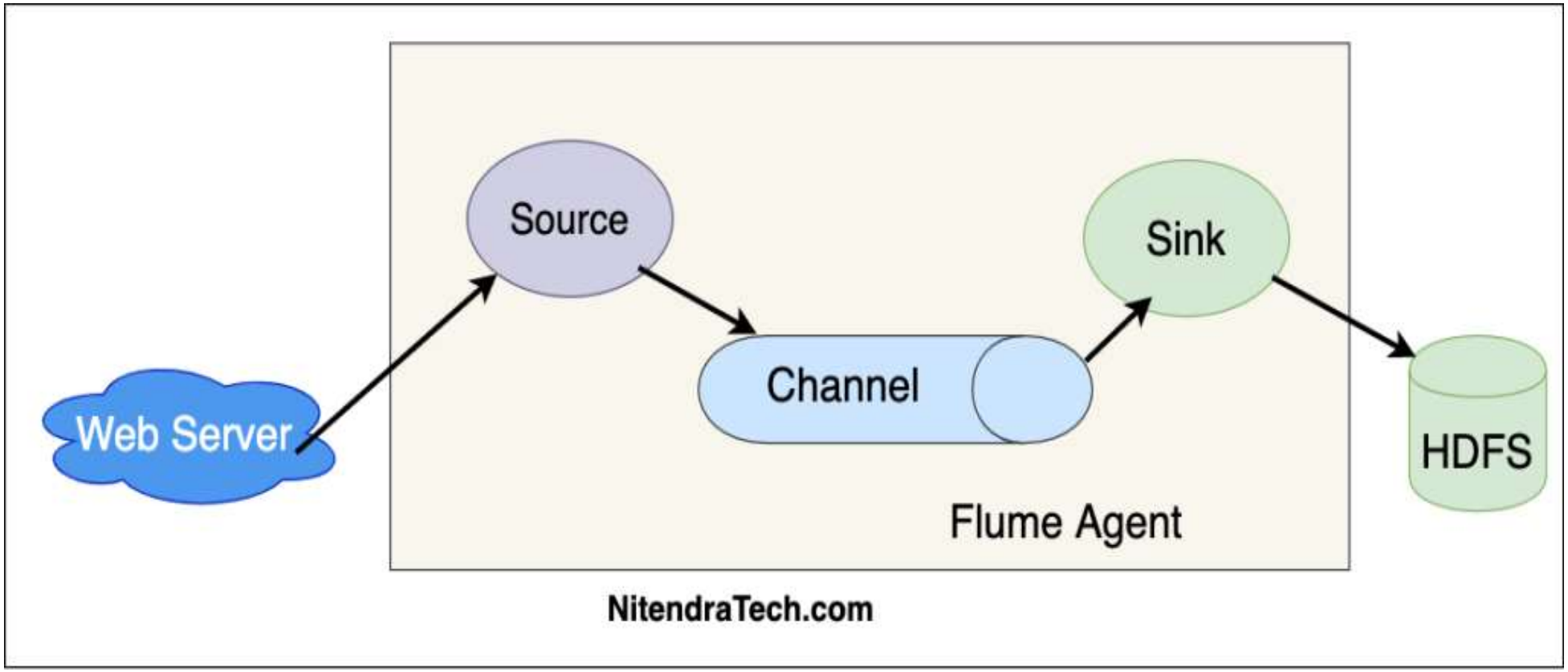
How Flume Works

The architecture of Flume is based on the concept of a **Flume agent**. Each agent is responsible for moving data from a source system to a destination system.

A Flume agent is composed of three main components:

- **Source** – This is where the data enters the agent.
  - Example: a web server generating access logs, a syslog service, or an application log.
- **Channel** – A temporary **storage buffer** that sits between the source and the sink.
  - Channels guarantee reliability: if the destination is slow or temporarily unavailable, the channel holds the data safely.
  - Channels can be memory-based (fast but volatile) or file-based (slower but durable).
- **Sink** – This is where the data exits the agent and is sent to its destination.
  - Common sinks include **HDFS**, **HBase**, or other centralized storage systems.

- The data flow in Flume is therefore:
  Source → Channel → Sink



Why Use Flume

- **Log Collection at Scale**: Collects logs from thousands of servers without manual effort.
- **Reliability**: Ensures no data loss thanks to the buffering mechanism of channels.
- **Scalability**: Multiple Flume agents can be chained or distributed to handle larger workloads.
- **Hadoop Integration**: Works seamlessly with HDFS and HBase, making it an ideal tool in Hadoop ecosystems.

Example Use Case

Imagine a large **news website** (e.g., CNN, BBC) that serves millions of visitors every day.

- Every visitor action (clicking, reading, watching videos) generates a log entry on a web server.
- Hundreds of servers are producing logs at the same time.
- A Flume agent on each server collects these logs (Source).
- The logs are temporarily stored in a channel (Channel).
- Finally, they are delivered into HDFS (Sink).

Once stored in HDFS, the logs can be analyzed with Hadoop, Hive, or Spark to answer questions like:

- Which articles are most popular?
- What regions drive the most traffic?
- At what times of day does traffic peak?

Flume vs. Kafka

It is important to understand how Flume differs from Kafka:

- **Flume**: Specialized for **log data ingestion into Hadoop**. It is simpler and well-suited when the main need is to collect server/application logs.
- **Kafka**: A more **general-purpose event streaming platform**, capable of handling logs, metrics, real-time messages, and integration with many systems beyond Hadoop.

Today, many organizations still use Flume for log collection, but Kafka has become the standard for broader real-time data streaming use cases.