

Course 6 - ETL and Data Pipelines with Shell, Airflow and Kafka

Data Processing Techniques

Week 1

Module 1: - ETL and ETL Processes

1) Course Intro video

Will Master in: -



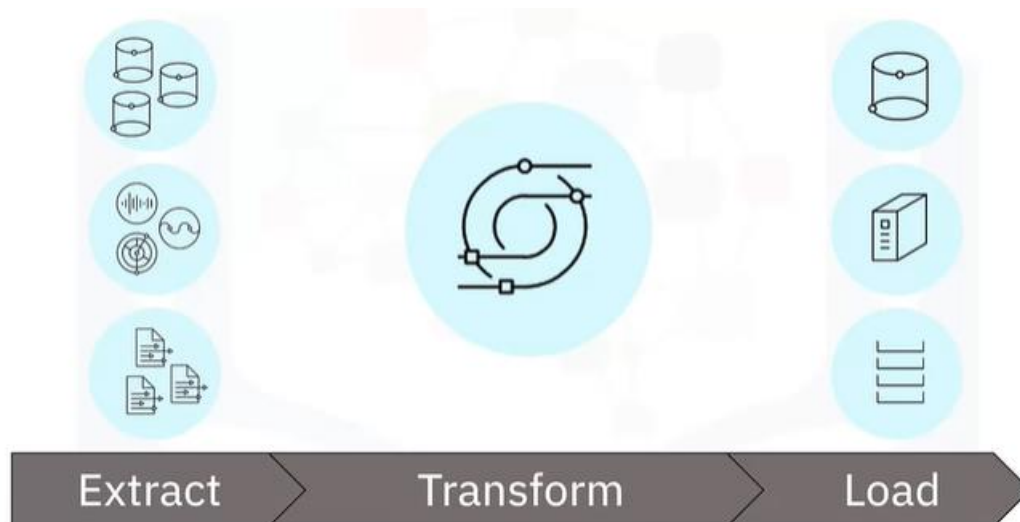
Will hone and apply: -



2) ETL Fundamentals

What is an ETL process?

- ETL stands for Extract, Transform, and Load.
- ETL is an automated data pipeline engineering methodology, whereby data is acquired and prepared for subsequent use in an analytics environment, such as a data warehouse or data mart.
- ETL refers to the process of curating data from multiple sources, conforming it to a unified data format or structure, and then loading the transformed data into its new environment.
- **Extraction:** - Process of extracting data from a source
- **Transformation:** - Transforming data into the format for the output.
- **Load:** - Loading data into a database, data warehouse or other storage



What is Extraction?

- Configuring access to data and reading it into an application:
 - Web scraping
 - Connecting programmatically via APIs
- The data may be static or something online

What is Data Transformation?

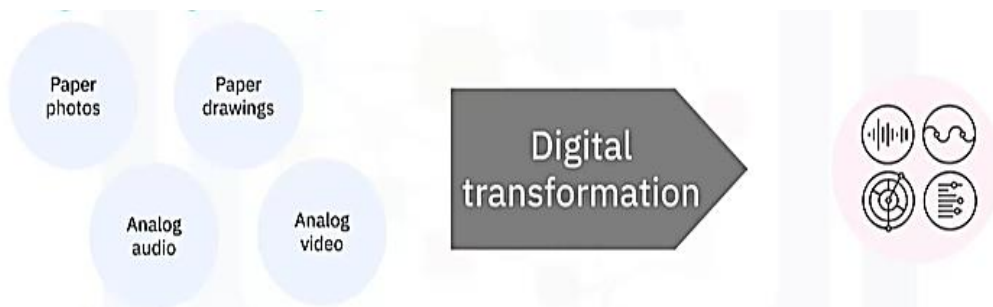
- Processing data
- Conforming to target systems and use cases
- Cleaning
- Filtering
- Joining
- Feature engineering
- Formatting and data typing

What is Data Loading?

- Moving data into a new environment.
- Examples: a database, data warehouse, or data mart
- Making the data readily available for analytics, dashboards, reports

Use cases for ETL pipelines –

- Digitizing analog media



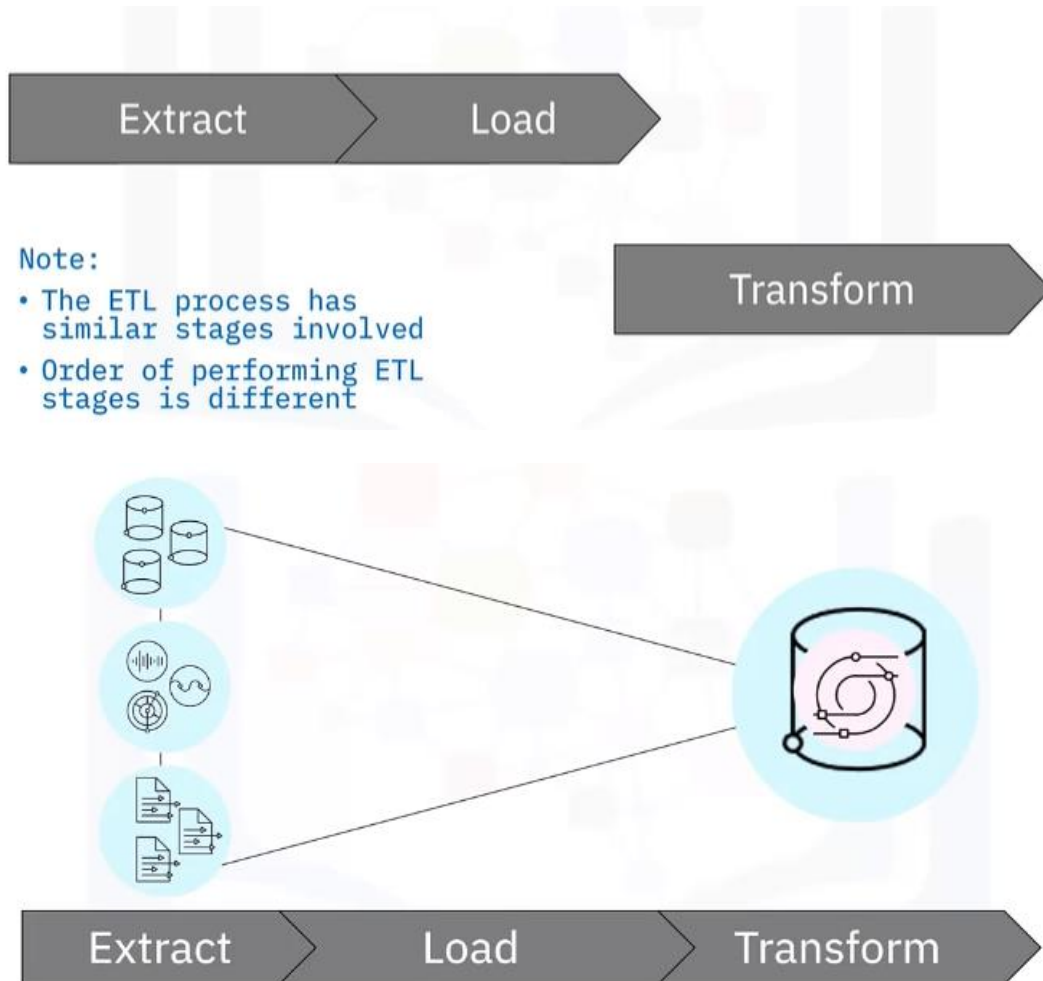
- Moving data from OLTP systems to OLAP systems
- Dashboards
- Machine Learning

Summary: -

- ETL stands for Extract, Transform, and Load.
- Extraction means reading data from one or more sources.
- Transformation means wrangling data to meet destination requirement.
- Loading means writing the data to its destination environment.
- ETL is used for curating data and making it accessible to end users.

3/ ETL Basics

What is an ETL process?



What is an Extraction process?

E = Extract: Extracting data from sources

L = Load: Loading data as-is into destination system

T = Transform: Transforming data on demand

ELT use cases –

Cases include:

- Demanding scalability requirements of Big Data
- Streaming analytics
- Integration of highly distributed data sources
- Multiple data products from the same sources

ELT is an emerging trend –

- Big Data → Cloud computing
- ELT separates the data pipeline from the processing
- More flexibility
- No information loss

Summary: -

- ELT processes are used for cases where flexibility, speed, and scalability are important.
- Cloud-based analytics platforms are ideally suited for handling Big Data and ELT processes.
- ELT is an emerging trend because cloud platforms are enabling it.

4) Comparing ETL to ELT

Different between ETL and ELT –

When and where the transformations happen:

- Transformations for ETL happen within the data pipeline.
- Transformations for ELT happen in the destination environment.

Flexibility:

- ETL is rigid – pipelines are engineered to user specifications
- ELT is flexible – end users build their own transformations

Support for Big Data:

- Organizations use ETL for relational data, on-premise – scalability is difficult.
- ELT solves scalability problems, handling both structured and unstructured Big Data in the cloud.

Time-to-insight:

- ETL workflows take time to specify and develop
- ELT supports self-service, interactive analytics in real time.

The evolution of ETL to ELT –

- Increasing demand for access to raw data.



- In ELT, the staging area fits the description of a data lake.
- Staging areas – private ETL landing zones
- Self-serve data platforms are the new “staging area”.

The shift from ETL to ELT –

- ETL still has its place for many applications



- ELT addresses key pain points:
 - Lengthy time-to-insight
 - Challenges imposed by Big Data
 - Demand for access to siloed information

Summary: -

- ETL and ELT differences include place of transformation, flexibility, Big Data support, and time-to-insight.
- Increasing demand for access to raw data drives the evolution from ETL to ELT
- ETL still has its place
- ELT enables ad-hoc, self-serve analytics

5) Data Extraction Techniques

Examples of raw data sources –



Techniques for extracting data –

Data extraction techniques include:

- OCR(Optical character recognition) – used to interpret and digitize text scanned from paper documents so it can be stored as a computer-readable file.
- ADC(Analog-to-Digital Converters) sampling, CCD(Charged-Coupled Device) sampling – ADCs used to digitize analog audio recordings and signals and, CCDs that capture and digitize image Opinions, questionnaires, and vital statistical data obtained through polling and census method
- Mail, phone, or in-person surveys and polls
- Cookies, user logs and other methods used for tracking human or system behaviour.

More techniques include:

- Web scraping – used to crawl web pages in search of text, images, tables, and hyperlinks
- APIs – readily available for extracting data from all sorts of online data repositories and feeds, such as government bureaus of statistics, libraries, weather networks, online shopping, and social networks.

- Database querying – SQL languages for querying relational databases, and NoSQL for querying document, key-value, graph or other non-structured data repositories.
- Edge computing – devices such as video cameras that have built-in processing that can extract features from raw data
- Biomedical devices – such as microfluidic arrays that can extract DNA sequences

Use cases –

- Integrating disparate structured data sources via APIs
- Capturing events via APIs and recording them in history
- Monitoring or surveillance with edge computing devices
- Data migration (direct to storage) for further processing
- Diagnosing health problems with medical devices

Summary: -

- Examples of raw data sources include archived media and web pages
- Data extraction often involves advanced technology
- Database querying, web scraping, and APIs are techniques for extracting data
- Medical devices extract biometric data for diagnostic purposes

6) Introduction to Data Transformation

Data transformation techniques –

Data transformations can involve various operations, such as:

- **Data typing** – involves casting data to appropriate types, such as integer, float, string, object, and category.
- **Data structuring** – includes converting one data format to another, such as JSON, XML, or CSV to database tables.
- **Anonymizing, encrypting** – to help ensure privacy and security

Other type of transformations include:

- **Cleaning** – duplicate records, missing value
- **Normalizing** – converting data to common units
- **Filtering, sorting, aggregating, binning** – for accessing the right data at a suitable level of details and in sensible order.
- **Joining, merging, disparate data sources**

Schema-on-write vs. schema-on-read –

Schema-on-write is conventional ETL approach:

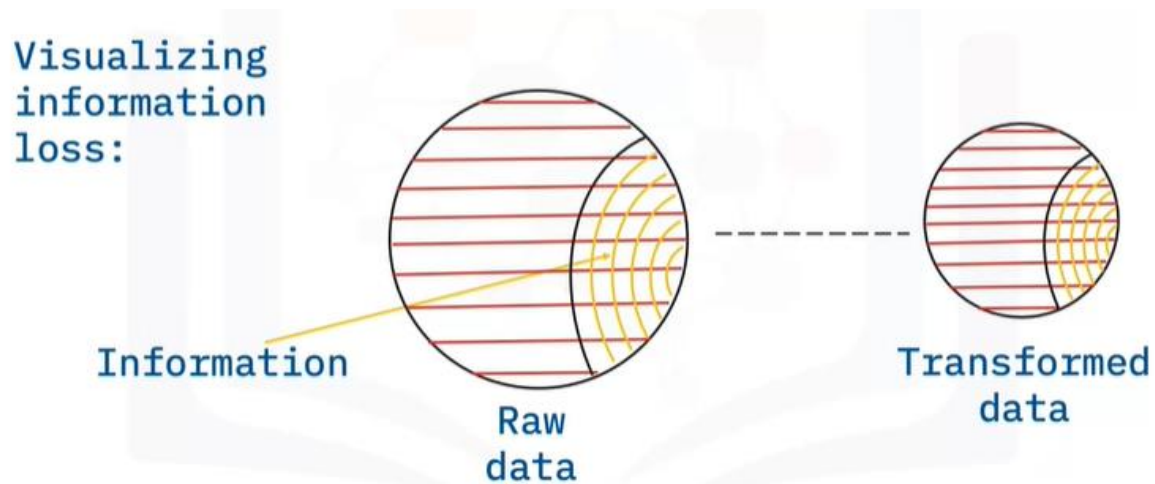
- Consistency and efficiency
- Limited versatility

Schema-on-read applies to the modern ELT approach:

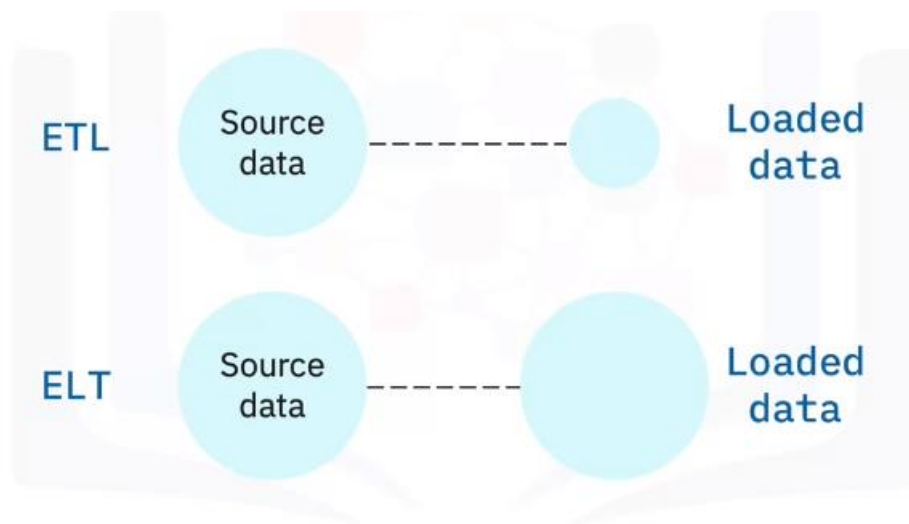
- Versatility
- Enhanced storage flexibility = more data

Information loss in transformation –

- Whether intentional or accidental, there are many ways in which information can be 'lost in transformation'.
- We can visualize this loss as follows.



- Raw data is normally much bigger than transformed data. Since data usually contains noise and redundancy, we can illustrate the 'information content' of data as a proper subset of the data. Correspondingly, we can see that shrinking the 'data volume' can also mean shrinking the 'information content'.
- Either way, for ETL processes, any lost information may or may not be recoverable, whereas with ELT, all the original information content is left intact because the data is simply copied over as-is.



Examples of ways information can be lost in transformation processes include:

- **Lossy data compression** – For example, converting floating point values to integers, reducing bitrates on audio or video.
- **Filtering** – For example, filtering is usually a temporary selection of a subset of data, but when it is permanent, information can easily be discarded.
- **Aggregation** – For example, average yearly sales vs. daily or monthly average sales.
- **Edge computing devices** – For example, false negatives in surveillance devices designed to only stream alarm signals, not the raw data.

Summary: -

- Data transformation is generally about formatting data to suit the needs of the intended application.
- Common transformation techniques include typing, structuring, normalizing, aggregating, and cleaning.
- Schema-on-write is the conventional approach used in ETL pipelines, and Schema-on-read relates to the modern ELT approach.
- Ways of losing information in transformation processes include filtering, aggregation, using edge computing devices, and lossy data compression.

7) Data Loading Techniques

Data Loading techniques –

- **Full** – You can load an initial history into a database
- **Incremental** – Applied to insert new data or to update already loaded data.
- **Scheduled** – You can schedule data loading to occur on a periodic basis
- **On-demand** – You can load it as required, on demand
- **Batch and stream** – Data can be loaded in batches , or it can be streaming continuously to its destination
- **Push and pull** – The data can be either pushed to a server or pushed to clients by a server.
- **Parallel and serial** – Data is usually loaded serially, but it can also be loaded in parallel.

Full vs. incremental loading –

Full loading –

- Start tracking transactions in a new data warehouse
- Used for porting over transaction history

Incremental loading –

- Data is appended to, not overwritten
- Used for accumulating transaction history
- Depending on the volume and velocity of data, can be batch loaded or stream loaded

Scheduled vs. on-demand loading –

Scheduled loading –

- Periodic loading, like daily transactions to database
- Windows Task Scheduler, cron

On-demand loading – triggered by

- Measures such as data size
- Event detection, like motion, sound, or temperature change
- User requests, like video or music streaming, web pages

Batch vs. stream loading –

Batch loading –

- Periodic updates using windows of data

Stream loading –

- Continuous updates as data arrives

Micro-batch loading –

- Short time windows used to access older data

Push vs. pull technology –

Client-server model –

Pull – Requests for data originate from the client

For example: RSS feeds, email

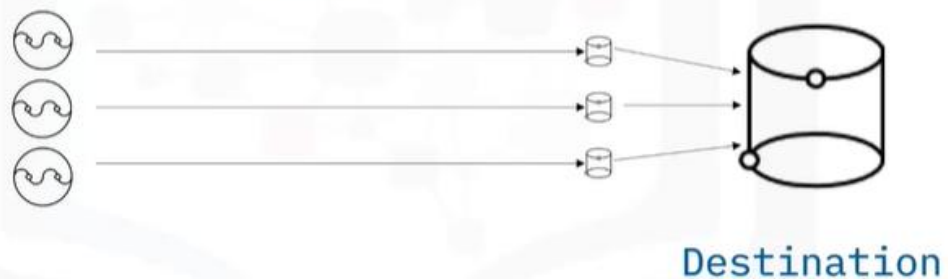
Push – Server pushes data to clients

For example: push notifications, instant messaging

Parallel loading –

Multiple data streams – to boost loading efficiency

Multiple data streams



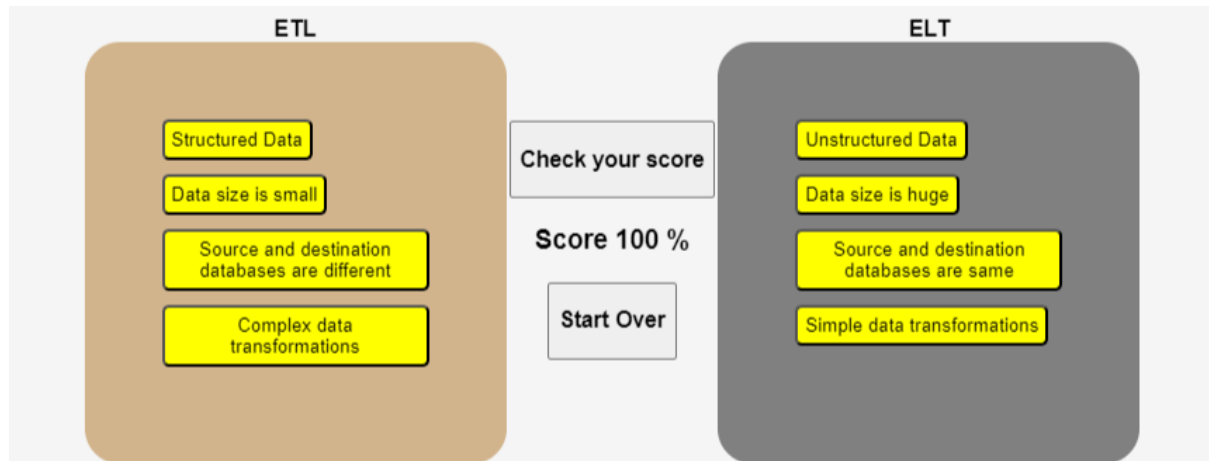
File partitioning – Splitting a single file into smaller chunks, the chunks can be loaded simultaneously.



Summary: -

- Some data loading techniques are scheduled, on-demand, and incremental.
- Data can be loaded in batches, or it can be streamed continuously into its destination.
- Servers can push data to subscribers as it becomes available.
- Clients can initiate pull requests for data from servers.
- You can employ parallel loading to boost loading efficiency of large volumes of data.

Interactivity: Tell the Difference between ETL and ELT



Summary & Highlights:-

- ETL stands for Extract, Transform, and Load
- Loading means writing the data to its destination environment
- Cloud platforms are enabling ELT to become an emerging trend
- The key differences between ETL and ELT include the place of transformation, flexibility, Big Data support, and time-to-insight
- There is an increasing demand for access to raw data that drives the evolution from ETL, which is still used, to ELT, which enables ad-hoc, self-serve analytics
- Data extraction often involves advanced technology including database querying, web scraping, and APIs
- Data transformation, such as typing, structuring, normalizing, aggregating, and cleaning, is about formatting data to suit the application
- Information can be lost in transformation processes through filtering and aggregation
- Data loading techniques include scheduled, on-demand, and incremental
- Data can be loaded in batches or streamed continuously

Week 2

ETL and Data Pipelines: Tools and Techniques

Module 1: - ETL using Shell Scripts

Linux Commands and Shell Scripting

A shell is a powerful user interface for Unix-like operating systems. It can interpret commands and run other programs. It also enables access to files, utilities, and applications, and is an interactive scripting language. Additionally, you can use a shell to automate tasks. Linux shell commands are used for navigating and working with files and directories. You can also use them for file compression and archiving.

ETL Techniques

ETL stands for Extract, Transform, and Load, and refers to the process of curating data from multiple sources, conforming it to a unified data format or structure, and loading the transformed data into its new environment.

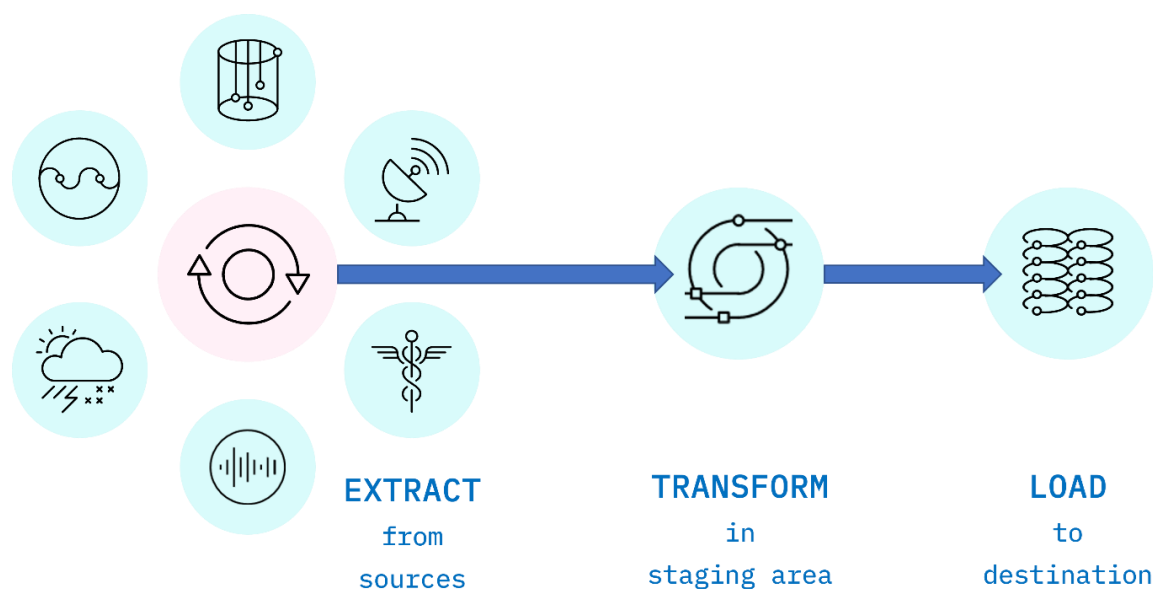


Fig. 1. ETL is an acronym used to describe the main processes behind a data pipeline design methodology that stands for Extract-Transform-Load. Data is extracted from disparate sources to an intermediate staging area where it is integrated and prepared for loading into a destination such as a data warehouse.

Extract

Data extraction is the first stage of the ETL process, where data is acquired from various source systems. The data may be completely raw, such as sensor data from IoT devices, or perhaps it is unstructured data from scanned medical documents or company emails. It may be streaming data coming from a social media network or near real-time stock market buy/sell transactions, or it may come from existing enterprise databases and data warehouses.

Transform

The transformation stage is where rules and processes are applied to the data to prepare it for loading into the target system. This is normally done in an intermediate working environment called a “staging area.” Here, the data are cleaned to ensure reliability and conformed to ensure compatibility with the target system.

Many other transformations may be applied, including:

- Cleaning: fixing any errors or missing values
- Filtering: selecting only what is needed
- Joining: merging disparate data sources
- Normalizing: converting data to common units
- Data Structuring: converting one data format to another, such as JSON, XML, or CSV to database tables
- Feature Engineering: creating KPIs for dashboards or machine learning
- Anonymizing and Encrypting: ensuring privacy and security
- Sorting: ordering the data to improve search performance
- Aggregating: summarizing granular data

Load

The load phase is all about writing the transformed data to a target system. The system can be as simple as a comma-separated file, which is essentially just a table of data like an Excel spreadsheet. The target can also be a database, which may be part of a much more elaborate system, such as a data warehouse, a data mart, data lake, or some other unified, centralized data store forming the basis for analysis, modeling, and data-driven decision making by business analysts, managers, executives, data scientists, and users at all levels of the enterprise.

In most cases, as data is being loaded into a database, the constraints defined by its schema must be satisfied for the workflow to run successfully. The schema, a set of rules called integrity constraints, includes rules such as uniqueness, [referential integrity](#), and mandatory fields. Thus such requirements imposed on the loading phase help ensure overall data quality.

ETL Workflows as Data Pipelines

Generally, an ETL workflow is a well thought out process that is carefully engineered to meet technical and end-user requirements.

Traditionally, the overall accuracy of the ETL workflow has been a more important requirement than speed, although efficiency is usually an important factor in minimizing resource costs. To boost efficiency, data is fed through a *data pipeline* in smaller packets (see Figure 2). While one packet is being extracted, an earlier packet is being transformed, and another is being loaded. In this way, data can keep moving through the workflow without interruption. Any remaining bottlenecks within the pipeline can often be handled by parallelizing slower tasks.

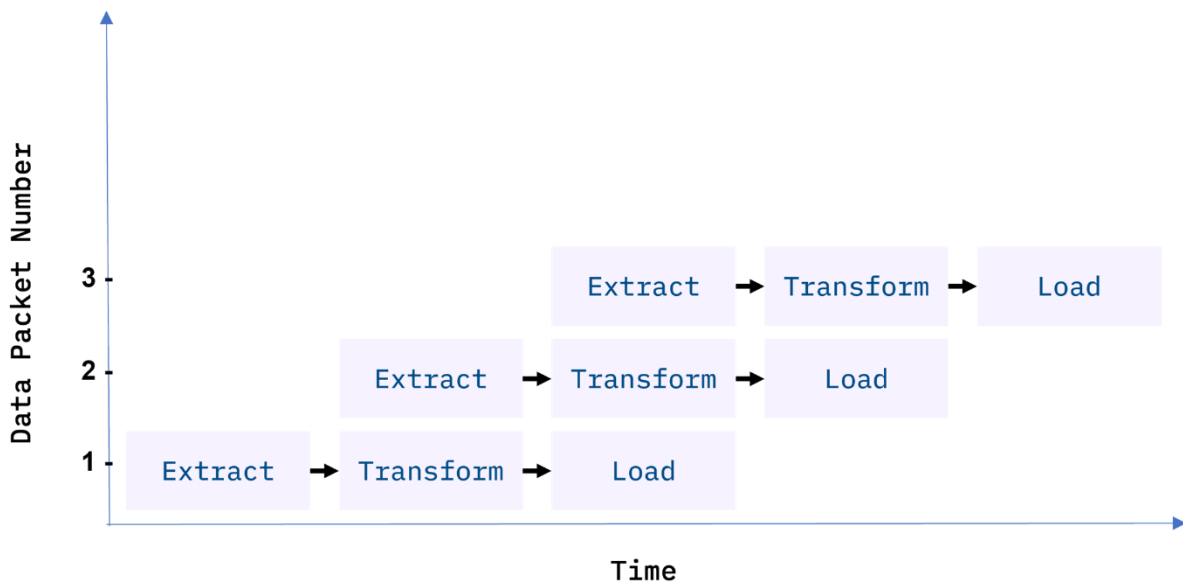


Fig 2. Data packets being fed in sequence, or “piped” through the ETL data pipeline. Ideally, by the time the third packet is ingested, all three ETL processes are running simultaneously on different packets.

With conventional ETL pipelines, data is processed in *batches*, usually on a repeating schedule that ranges from hours to days apart. For example, records accumulating in an Online Transaction Processing System (OLTP) can be moved as a daily batch process to one or more Online Analytics Processing (OLAP) systems where subsequent analysis of large volumes of historical data is carried out.

Batch processing intervals need not be periodic and can be triggered by events, such as

- when the source data reaches a certain size, or
- when an event of interest occurs and is detected by a system, such as an intruder alert, or
- on-demand, with web apps such as music or video streaming services

Staging Areas

ETL pipelines are frequently used to integrate data from disparate and usually siloed systems within the enterprise. These systems can be from different vendors, locations, and divisions of the company, which can add significant operational complexity. As an example, (see Figure 3) a cost accounting OLAP system might retrieve data from distinct OLTP systems utilized by the separate payroll, sales, and purchasing departments.

ETL pipelines are frequently used to integrate data from disparate and usually *siloed* systems within the enterprise. These systems can be from different vendors, locations, and divisions of the company, which can add significant operational complexity. As an example, (see Figure 3) a cost accounting OLAP system might retrieve data from distinct OLTP systems utilized by the separate payroll, sales, and purchasing departments.

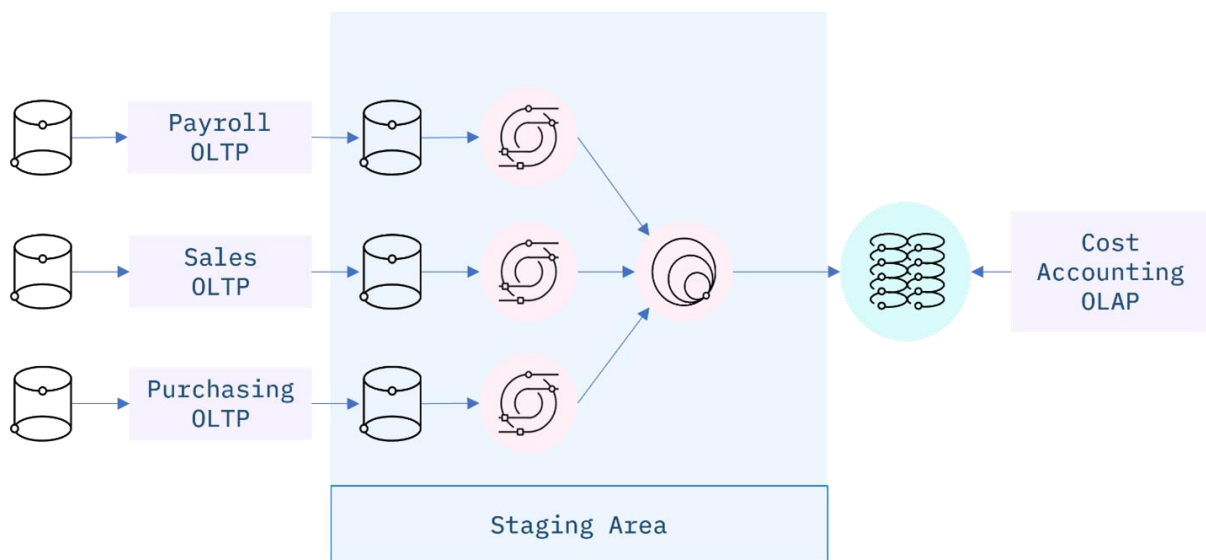


Fig 3. An ETL data integration pipeline concept for a Cost Accounting OLAP, fed by disparate OLTP systems within the enterprise. The staging area is used in this example to manage change detection of new or modified data from the source systems, data updates, and any transformations required to conform and integrate the data prior to loading to the OLAP.

ETL Workflows as DAGs

ETL workflows can involve considerable complexity. By breaking down the details of the workflow into individual tasks and dependencies between those tasks, one can gain better control over that complexity. Workflow orchestration tools such as Apache Airflow do just that.

Airflow represents your workflow as a directed acyclic graph (DAG). A simple example of an Airflow DAG is illustrated in Figure 4. Airflow tasks can be expressed using predefined templates, called operators. Popular operators include Bash operators, for running Bash code, and Python operators for running Python code, which makes them extremely versatile for deploying ETL pipelines and many other kinds of workflows into production.

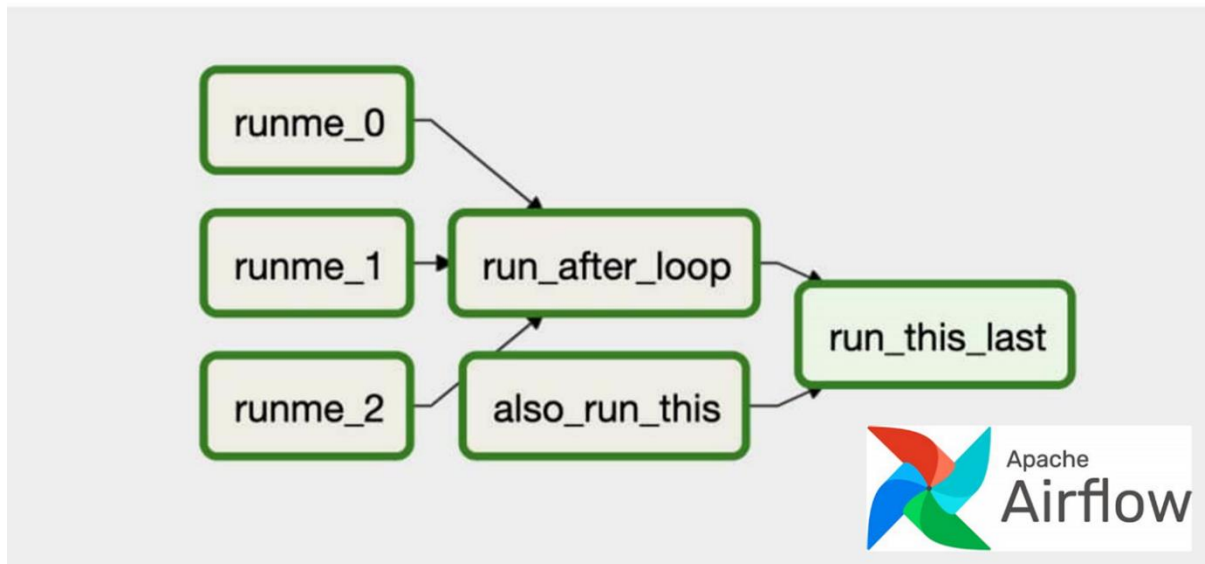


Fig 4. An Apache Airflow DAG representing a workflow. The green boxes represent individual tasks, while the arrows show dependencies between tasks. The three tasks on the left, 'runme_j' are jobs that run simultaneously along with the 'also_run_this' task. Once the 'runme_j' tasks complete, the 'run_after_loop' task starts. Finally, 'run_this_last' engages once all tasks have finished successfully.

Popular ETL tools

There are many ETL tools available today. Modern enterprise grade ETL tools will typically include the following features:

- Automation: Fully automated pipelines
- Ease of use: ETL rule recommendations
- Drag-and-drop interface: “o-code” rules and data flows
- Transformation support: Assistance with complex calculations
- Security and Compliance: Data encryption and HIPAA, GDPR compliance

Some well-known ETL tools are listed below, along with some of their key features. Both commercial and open-source tools are included in the list.

talend

Talend Open Studio

- Supports big data, data warehousing, and profiling
- Includes collaboration, monitoring, and scheduling
- Drag-and-drop GUI for ETL pipeline creation
- Automatically generates Java code

- Integrates with many data warehouses
- Open-source



AWS Glue

- ETL service that simplifies data prep for analytics
- Suggests schemas for storing your data
- Create ETL jobs from the AWS Console



IBM InfoSphere DataStage

- A data integration tool for designing, developing, and running ETL and ELT jobs
- The data integration component of IBM InfoSphere Information Server
- Drag-and-drop graphical interface
- Uses parallel processing and enterprise connectivity in a highly scalable platform



Alteryx

- Self-service data analytics platform
- Drag-and-drop accessibility to ETL tools
- No SQL or coding required to create pipelines



Apache Airflow and Python

- Versatile “configuration” as code data pipeline platform
- Open-sourced by Airbnb

- Programmatically author, schedule, and monitor workflows
- Scales to Big Data
- Integrates with cloud platforms



The Pandas Python library

- Versatile and popular open-source programming tool
- Based on data frames – table-like structures
- Great for ETL, data exploration, and prototyping
- Doesn't readily scale to Big Data

1/ ETL using Shell Scripting

Temperature reporting scenario –

Task:

Report temperature statistics for a remote location.

- Hourly average, min. and max. temperature
- Remote temperature sensor
- Update every minute

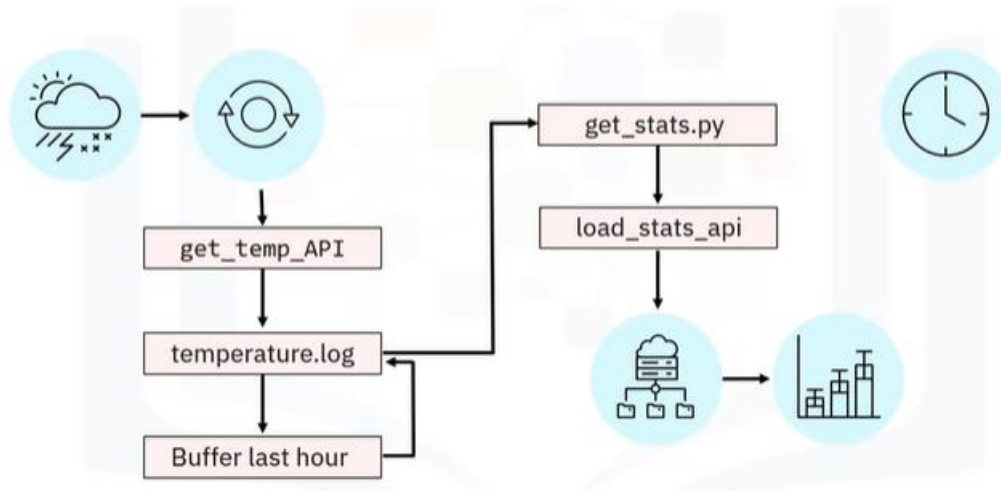
Given:

- 'get_temp_api' – read temperature from remote sensor.
- 'load_stats_api' – load stats to dashboard repo.

Temperature reporting workflow –

- Starting with the weather station and its data interface.
- The extraction step involves obtaining a current temperature reading from the sensor using the supplied 'get_temp' API.
- You can append the reading to a log file, say 'temperature.log'. Since you will only need to keep the most recent hour of readings, buffer the last 60 readings, and then just overwrite the log file with the buffered readings.

- Next, call a program, for example, a python script called 'get_stats.py' which calculates the temperature stats from the 60-minute log, and loads the resulting stats into the reporting system using the load_stats API.
- The stats can then be used to display a live chart showing the hourly min, max and average temperatures.
- You will also want to schedule your workflow to run every minute.



Creating an ETL shell script –

```

$ touch Temperature_ETL.sh
$ gedit Temperature_ETL.sh

```

```

#!/bin/bash
# Extract reading with get_temp_API
# Append reading to temperature.log
# Buffer last hour of readings
# Call get_stats.py to aggregate the readings
# Load the stats using load_stats_api

```

ETL script: Extract and buffer –

```
$ touch temperature.log
```

```
#!/bin/bash
# Extract reading with get_temp_API
# Append reading to temperature.log
get_temp_api >> temperature.log

# Buffer last hour of readings
tail -60 temperature.log > temperature.log
```

ETL script: Transform temperatures –

get_stats.py

- Reads temperature from log file
- Calculates temperature stats
- Writes temperature stats to file
- Input/Output filenames specified as command line arguments

```
# Call get_stats.py to aggregate the readings
python3 get_stats.py temperature.log temp_stats.csv
```

Load the transformed data –

```
# Load the stats using load_stats_api
load_stats_api temp_stats.csv
```

Set permissions –

```
$ chmod +x Temperature_ETL.sh
```

Schedule your ETL job –

- Open crontab editor:

```
$ crontab -e
```

- Enter schedule:

```
1 * * * * path/Temperature_ETL.sh
```

- Close and save
- Your job is now scheduled and running in production.

Summary: -

- ETL pipelines can be created with Bash scripts
- ETL jobs can be run on a schedule using cron

Summary & Highlights:-

- ETL pipelines are created with Bash scripts
- ETL jobs can be run on a schedule using cron

Module 2: - An Introduction to Data Pipelines

1) Introduction to Data Pipelines

What is a pipeline?

- Series of connected processes
- Output of one process is input of the next
- For example, take a box, pass it to your neighbour, and so on, until the box arrives at the end of the line
- Mass production – parts pass along conveyor belts between manufacturing stages.

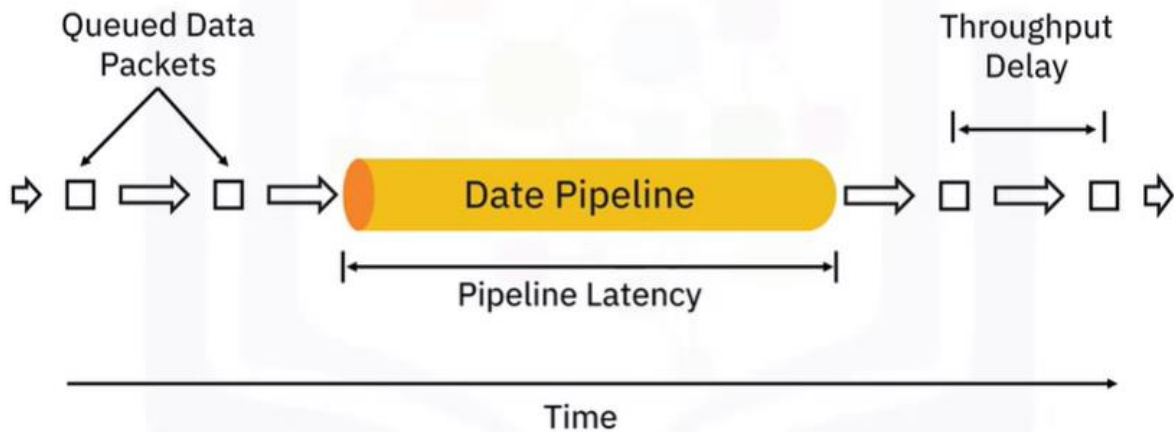
What is a data pipeline?



- Purpose – to move data from one place or form to another
- Any system which extracts, transforms, and loads data
- Includes low-level hardware architectures
- Software-driven processes – commands, programs and threads
- Bash 'pipe' command can connect such processes together

Packet flow through a pipeline –

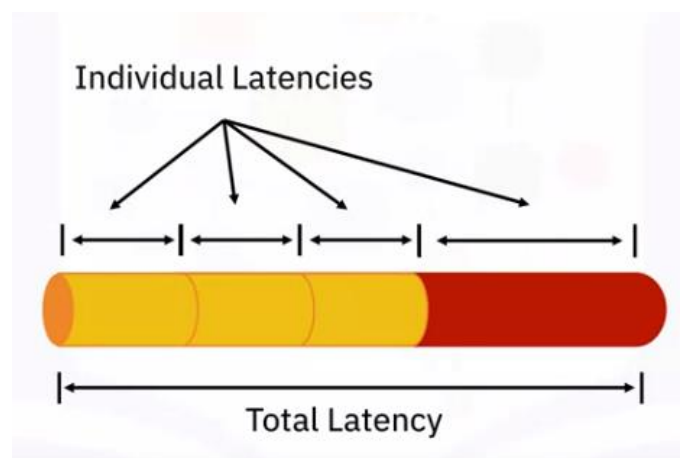
- Packets can range in size from a single record, or event, to large collections of data. Here, we have data packets queued for ingestion to the pipeline.
- The length of the data pipeline represents the time it takes for a single packet to traverse the pipeline.
- The arrows between packets represent the throughput delays, or the times between successive packet arrivals.



Data pipeline performance: Two key performance considerations

1. Latency –

- The first is latency, which is the total time it takes for a single packet of data to pass through the pipeline.
- Equivalently, latency is the sum of the individual times spent during each processing stage within the pipeline.
- Thus, overall latency is limited by the slowest process in the pipeline.
- For example, no matter how fast your internet service is, the time it takes to load a web page will be decided by the server's speed.



2. Throughput –

- The second performance consideration is called throughput. \
- It refers to:
 - How much data can be fed through the pipeline per unit of time.
 - Processing larger packets per unit of time increases throughput.

Use cases –

Applications of data pipelines

- Backing up files
- Integrating disparate raw data sources into a data lake

- Moving transactional records to a data warehouse
- Streaming data from IoT devices to dashboards
- Preparing raw data for machine learning development or production
- Messaging systems such as email, SMS, video meetings

Summary: -

- The purpose of a data pipeline is to move data from one place, or form, to another.
- We can visualize data flowing through a pipeline as a series of data packets flowing in and out, one by one.
- Latency and throughput are key design considerations for data pipelines.
- Use cases for data pipelines are many and range from simple copy-and-paste-like data backups to online video meetings.

2/ Key Data Pipeline Processes

Data pipeline processes –

Stages of data pipeline processes:

Data Extraction – Extraction of data from one or more data sources.

Data Ingestion – Ingestion of the extracted data into the pipeline.

Transformation stages – Optional data transformation stages within the pipeline

Loading – Loading of the data into a destination facility

Scheduling or triggering –

Monitoring – Monitoring the entire workflow

Maintenance and Optimizations – Required to keep the pipeline up and running smoothly.

Pipeline monitoring considerations –

Some key monitoring considerations include:

- **Latency**, or the time it takes for data packets to flow through the pipeline.
- **Throughput** demand, the volume of data passing through the pipeline over time.
- **Errors and failures**, caused by factors such as network overloading, and failures at the source or destination systems.

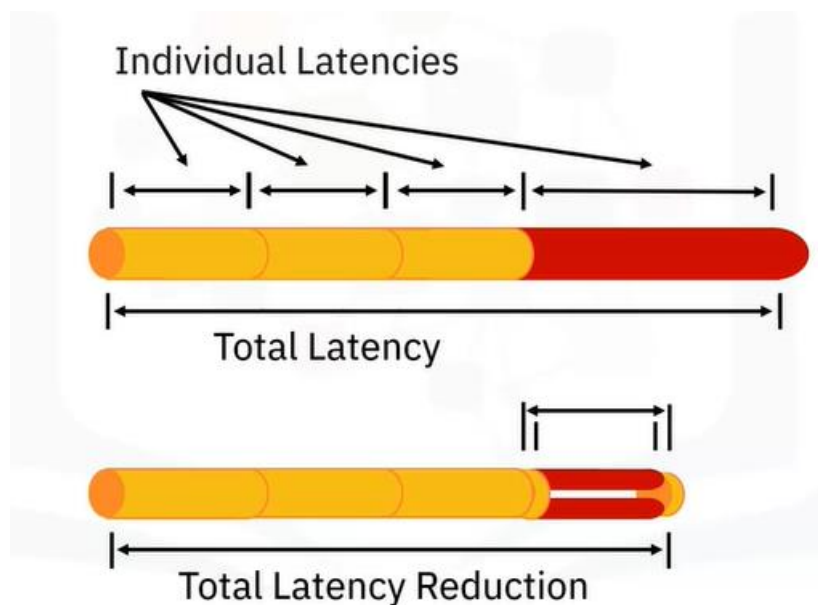
- **Utilization rate**, or how fully the pipeline's resources are being utilized, which affects cost.
- Lastly, the pipeline should also have a system for **logging events and alerting administrators** when failures occur.

Load balanced pipelines –

- Just-in-time data packet relays
- No upstream data flow bottlenecks
- Uniform packet throughput for each stage
- Such a pipeline is called as “load balanced”

Handling unbalanced loads –

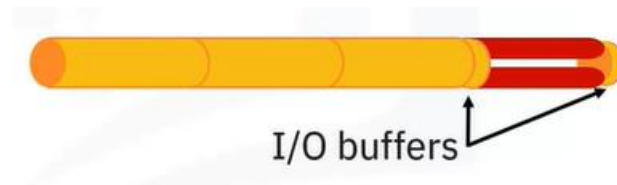
- Pipelines typically contain bottlenecks
- Slower stages may be parallelized to speed up throughput
- Processes can be replicated on multiple CPUs / cores / threads
- Data packets are then distributed across these channels
- Such pipelines are called dynamic or non-linear



Suppose your pipeline has a bottleneck in one of its stages, such as the longer red section here, which has more latency than the other stages in the pipeline. If it's possible to parallelize that stage, for example by splitting the data into two concurrent stages, like this, then you can reduce this stage's latency. There will be a little overhead in managing the parallelization and recombination of the output back into the pipeline, but the overall latency will be reduced.

Stage synchronization –

- I/O buffers can help synchronize stages
- Holding area for data between processing stages
- Buffers regulate the flow of data, may improve throughput
- I/O buffers used to distribute loads on parallelized stages



Summary: -

- In addition to extraction, transformation, and loading, data pipeline processes include scheduling, triggering, monitoring, maintenance, and optimization.
- Pipeline monitoring considerations include tracking latency, throughput, resource utilization, and failures,
- Unbalanced or varying loads can be mitigated by introducing parallelization and I/O buffers at bottlenecks.

3/ Batch Versus Streaming Data Pipeline Use Cases

Batch data pipelines –

- Operate on batches of data
- Usually run periodically – hours, days, weeks apart
- Can be initiated based on data size or other triggers
- When latest data is not needed
- Typical choice when accuracy is critical.

Note – Streaming alternatives are emerging

Streaming data pipelines –

- Ingest data packets in rapid succession
- For real-time results
- Records/events processed as they happen
- Event streams can be loaded to storage
- Users publish/subscribe to event streams

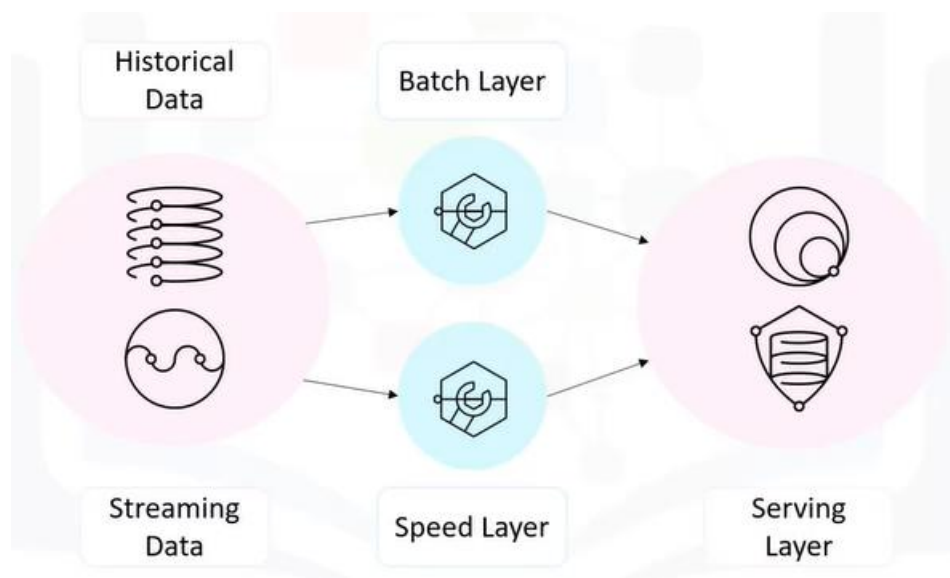
Micro-batch data pipelines –

- Tiny micro-batches and faster processing simulate real-time processing
- Smaller batches improve load balancing, lower latency
- When short windows of data are required.

Batch vs. stream requirements –

- Trade-off between accuracy and latency requirements
- Data cleaning improves quality, but increases latency
- Lowering latency increases potential for errors

Lambda Architecture –



A Lambda architecture is a hybrid architecture, designed for handling Big Data. Lambda architectures combine batch and streaming data pipeline methods. Historical data is delivered in batches to the batch layer, and real-time data is streamed to a speed layer. These two layers are then integrated in the serving layer.

- Data stream fills in “latency gap”
- Used when data window is needed but speed is critical
- Drawback is logical complexity
- Lambda architecture = accuracy and speed

Batch Versus Streaming Data Pipeline Use Cases: –

Batch Data Pipeline Use Cases –



Streaming Data Pipeline Use Cases –

- Watching movies, listening to music or podcasts
- Social media feeds, sentiment analysis
- Fraud detection
- User behaviour, advertising



Summary: -

- Batch pipelines extract and operate on batches of data.
- Batch processing is used when accuracy is critical or there is no need for the most recent data.
- Streaming data pipelines ingest data packets one-by-one in rapid succession.
- Streaming pipelines are used when the most current data is needed.

- Micro-batch processing can be used to simulate real-time data streaming.
- Lambda architecture can be used in cases where access to earlier data is required but speed is also important.

4) Data Pipeline Tools and Technologies

Feature of modern pipeline tools –

Typical enterprise grade technologies:

- **Automation:** Fully automated pipelines
- **Ease of use:** ETL rule recommendations
- **Drag-and-drop interface:** “no-code” rules and data flows
- **Transformation support:** Assistance with complex calculations
- **Security and compliance:** Data encryption and compliance with HIPAA and GDPR

Open-source data pipeline tools: –

The Pandas Python Library –

- Versatile and popular programming tool
- Based on data frames – table-like structures
- Great for ET, data exploration, and prototyping
- Does not readily scale to Big Data
- Libraries with similar APIS: Vaex, Dask, and Spark help with scaling up
- Consider SQL-Like alternatives such as PostgreSQL for Big Data applications



Apache Airflow and Python –

- Versatile “configuration” as code data pipeline platform
- Open-sourced by Airbnb
- Programmatically author, schedule, and monitor workflows
- Scales to Big Data
- Integrates with cloud platforms



Talend Open Studio –

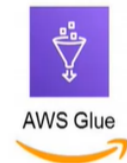
- Supports big data, data warehousing and profiling
- Includes collaboration, monitoring, and scheduling
- Drag-and-drop GUI allows you to create ETL pipelines
- Automatically generates Java code
- Integrates with many data warehouses.



Enterprise data pipeline tools: –

AWS Glue –

- ETL service that simplifies data prep for analytics
- Suggests schemas for storing your data
- Create ETL jobs from the AWS Console



Panoply –

- An ELT-specific platform
- No-code data integration
- SQL-based view creation
- Shifts emphasis from data pipeline development to data analytics
- Integrates with dashboard and BI tools such as Tableau and PowerBI



Alteryx –

- Self-service data analytics platform
- Drag-and-drop accessibility to ETL tools
- No SQL or coding required to create pipelines



IBM InfoSphere DataStage –

- A data integration tool for designing, developing, and running ETL and ELT jobs
- The data integration component of IBM InfoSphere Information Server
- Drag-and-drop graphical interface
- Uses parallel processing and enterprise connectivity in a highly scalable platform



Streaming data pipeline tools

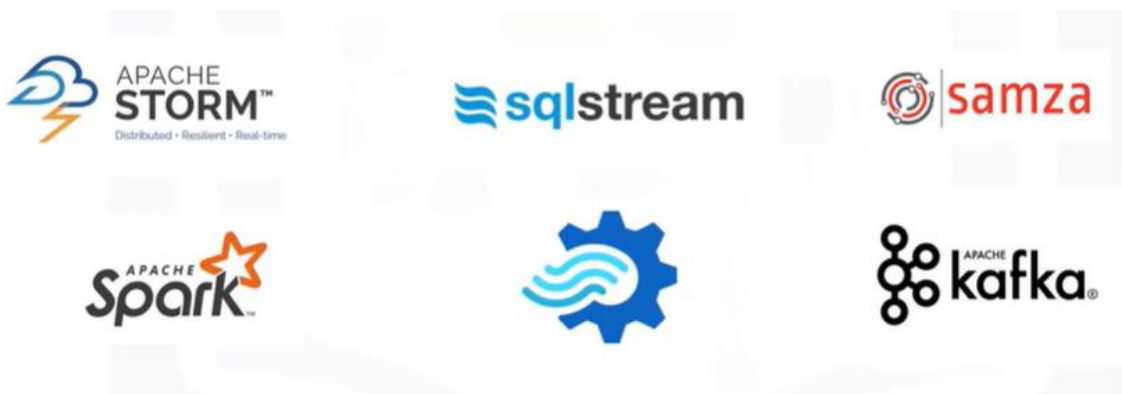
IBM Streams –

- Build real-time analytical applications using SPL, plus Java, Python, or C++
- Combine data in motion and at rest to deliver intelligence in real time
- Achieve end-to-end processing with sub-millisecond latency
- Includes IBM Streams Flows, a drag-and-drop interface for building workflows



More streaming data pipeline tools –

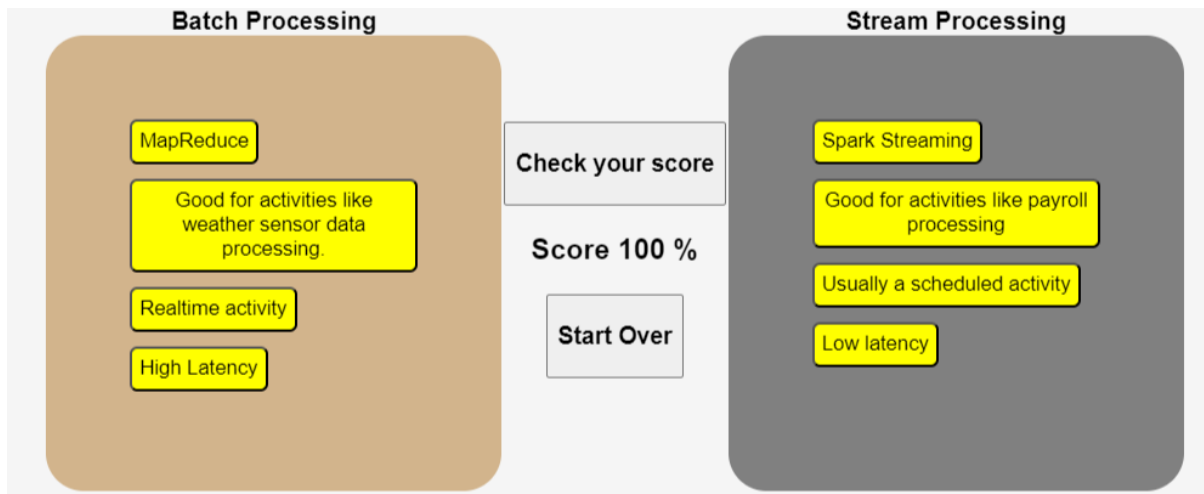
Stream-processing technologies include:



Summary: -

- Modern enterprise-grade data pipeline tools include technologies such as transformation support, drag-and-drop GUIs, and security and compliance features.
- Pandas, Vaex, and Dask are useful open-source Python libraries for prototyping and building data pipelines.
- Apache Airflow and Talend Open Studio allow you to programmatically author, schedule, and monitor Big Data workflows
- Panoply is specific to ELT pipelines
- Alteryx and IBM InfoSphere DataStage can handle both ETL and ELT workflows.
- Stream-processing technologies include Apache Kafka, IBM Streams, SQLStream, and Apache Spark.

Interactivity: Difference between Batch Processing and Stream



Summary & Highlights:-

- Data pipelines move data from one place, or form, to another
- Data flows through pipelines as a series of data packets
- Latency and throughput are key design considerations for data pipelines
- Data pipeline processes include scheduling or triggering, monitoring, maintenance, and optimization
- Parallelization and I/O buffers can help mitigate bottlenecks
- Batch pipelines extract and operate on batches of data
- Batch processing applies when accuracy is critical, or the most recent data isn't required
- Streaming data pipelines ingest data packets one-by-one in rapid succession
- Streaming pipelines apply when the most current data is needed
- Examples of streaming data pipelines use cases, such as social media feeds, fraud detection, and real-time product pricing
- Modern data pipeline technologies include schema and transformation support, drag-and-drop GUIs, and security features
- Stream-processing technologies include Apache Kafka, IBM Streams, and SQLStream

Week 3

Building Data Pipeline using Airflow

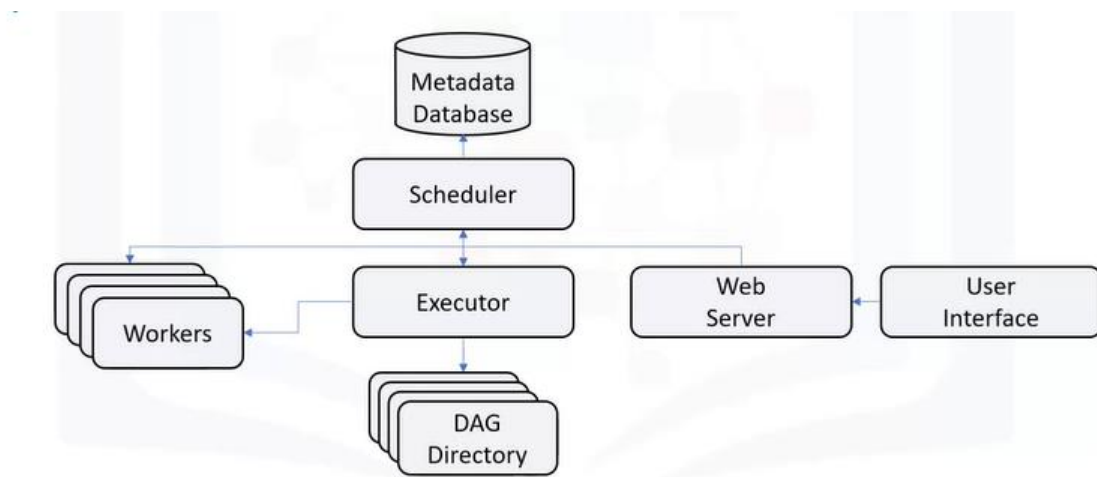
Module 1: - Using Apache Airflow to build Data Pipelines

1) Apache Airflow Overview

What is Apache Airflow?

- Great open-source workflow orchestration tool
- A platform that lets you build and run workflows
- A workflow is represented as a DAG
- Note: Airflow is not a data streaming solution

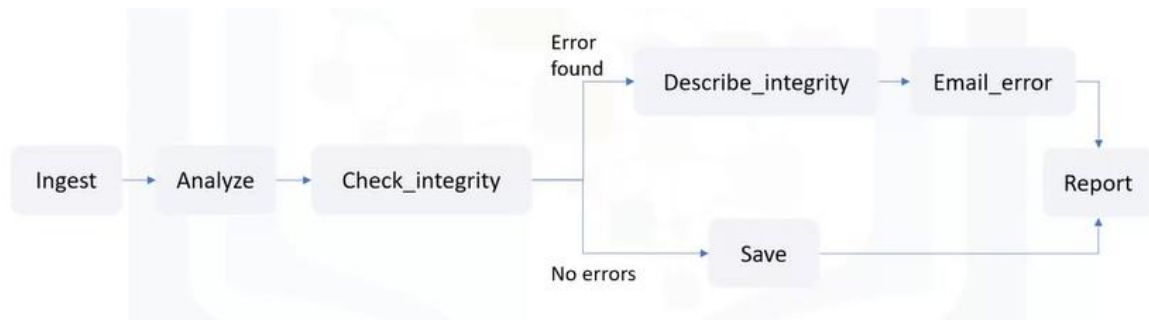
Simplified view of Airflow's architecture –



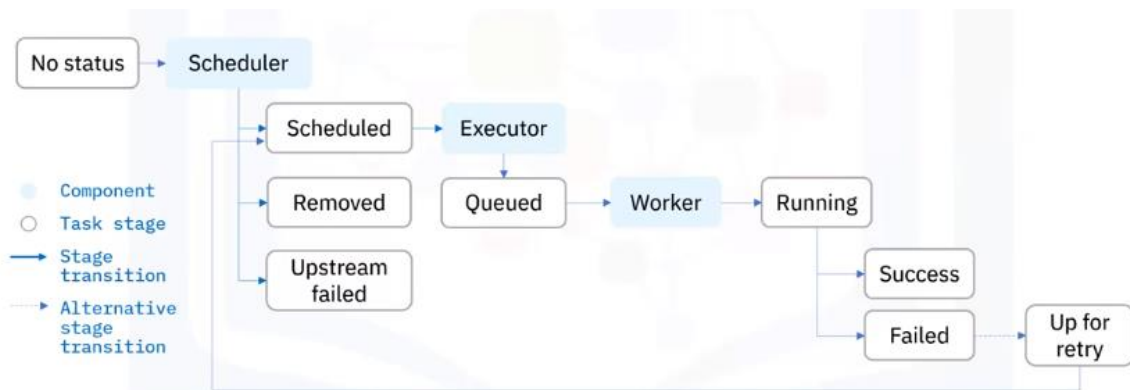
- Airflow comes with a built-in Scheduler, which handles the triggering of all scheduled workflows. The Scheduler is responsible for submitting individual tasks from each scheduled workflow to the Executor.
- The Executor handles the running of these tasks by assigning them to Workers, which then run the tasks.
- The Web Server , serves Airflow's powerful interactive User Interface. From this UI, you can inspect, trigger and debug any of your DAGs and their individual tasks.
- The DAG Directory contains all of your DAG files, ready to be accessed by the Scheduler, the Executor, and each of its employed Workers.
- Finally, Airflow hosts a Metadata Database, which is used by the Scheduler, Executor, and the Web Server to store the state of each DAG and its tasks

Sample DAG illustrating the leading of different branches –

- In this example DAG, the tasks include data ingestion, data analysis, saving the data, generating reports, and triggering other systems, such as reporting any errors by email.



The lifecycle of an Apache Airflow task state –



- **No status:** This means that the task has not yet been queued for execution.
- **Scheduled:** The scheduler has determined that the task's dependencies are met and has scheduled it to run.
- **Removed:** For some reason, the task has vanished from the DAG since the run started.
- **Upstream failed:** An upstream task has failed.
- **Queued:** The task has been assigned to the Executor and is waiting for a worker to become available.
- **Running:** The task is being run by a worker.
- **Success:** The task finished running without errors.
- **Failed:** The task had an error during execution and failed to run, and
- **Up for retry:** The task failed but has retry attempts left and will be rescheduled. Ideally, a task should flow through the scheduler from 'no status', to 'scheduled', to 'queued', to 'running', and finally to 'success.'

Apache Airflow Features –

- **Pure Python** Create your workflows using standard Python. This allows you to maintain full flexibility when building your data pipelines.
- **Useful UI Monitor**, schedule, and manage your workflows via a sophisticated web app, offering you full insight into the status of your tasks.
- **Integration** Apache Airflow provides many plug-and-play integrations, such as IBM Cloudant, that are ready to execute your tasks.
- **Easy to Use** Anyone with Python knowledge can deploy a workflow. Airflow does not limit the scope of your pipelines.
- finally, **the open-source** feature. Whenever you want to share your improvement, you can do this by opening a pull request.

Apache Airflow Principles –

- **Scalable:** Airflow has a modular architecture and uses a message queue to orchestrate an arbitrary number of workers. It is ready to scale to infinity.
- **Dynamic:** Airflow pipelines are defined in Python, and allow dynamic pipeline generation. Thus, your pipelines can contain multiple simultaneous tasks.
- **Extensible:** You can easily define your own operators and extend libraries to suit your environment.
- **Lean:** Airflow pipelines are lean and explicit. Parameterization is built into its core using the powerful Jinja templating engine.

Apache Airflow Use Cases –

- Defining and organizing machine learning pipeline dependencies
- Increasing the visibility of batch processes and decoupling them
- Deploying as an enterprise scheduling tool
- Orchestrating SQL transformations in data warehouses.



Summary: -

- Apache Airflow is a platform to programmatically author, schedule, and monitor workflows.
- The five main features of Airflow are its use of Python, its intuitive and useful user interface, extensive plug-and-play integrations, ease of use, and the fact that it is open source.

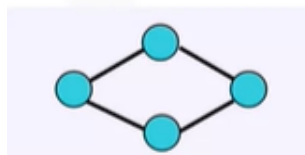
- The four principles of Apache Airflow are scalable, dynamic, extensible, and lean.
- And finally, defining and organizing machine learning pipeline dependencies with Apache Airflow is one of the common use cases.

2) Advantages of Using Data Pipelines as DAGs in Apache Airflow

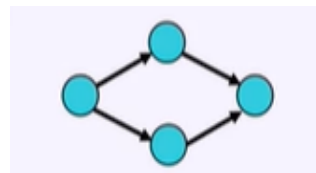
What is a DAG?

Directed Acyclic Graph

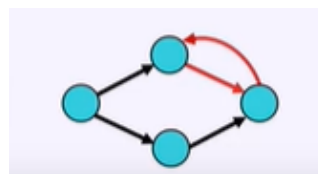
- **Graph** – nodes and edges



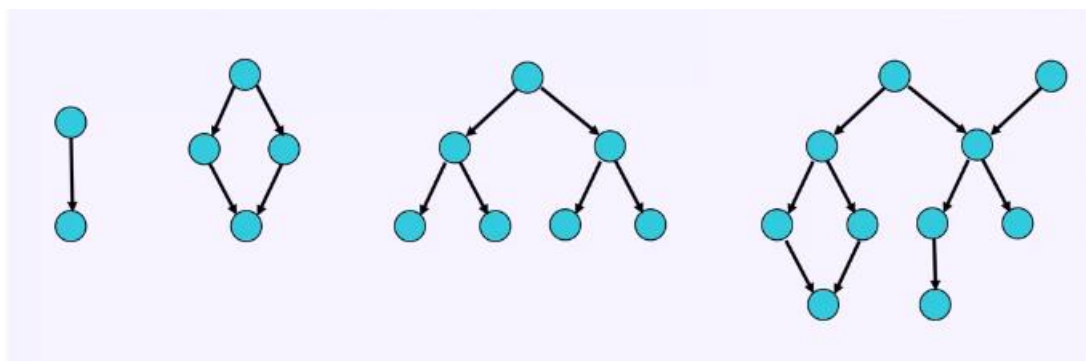
- **Directed graph** – each edge has a direction



- **Acyclic** – no loops (cycles)



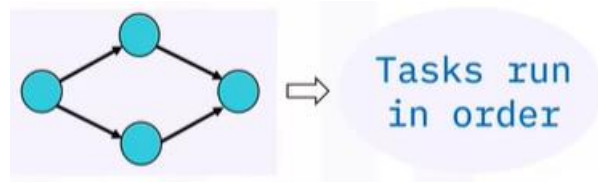
Examples of DAGs –



What is a DAG in Airflow?

DAG represent workflows:

- Nodes are tasks
- Edges are dependencies



Tasks and operators –

- Tasks are written in Python
- Tasks implement operators, for example, Python, SQL, or Bash operators
- Operators determine what each task does
- Sensor operators poll for a certain time or condition
- Other operators include email and HTTP request operators

DAG definition components –

Python script blocks:

- Library imports

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
import datetime as dt
```

- DAG arguments

```
default_args = {
    'owner': 'me',
    'start_date': dt.datetime(2021, 7, 28),
    'retries': 1,
    'retry_delay': dt.timedelta(minutes=5),
}
```


- DAG definition

```
dag = DAG('simple_example',
          description='A simple example DAG',
          default_args=default_args,
          schedule_interval=dt.timedelta(seconds=5),
          )
```

- Task definitions

```
task1 = BashOperator(
    task_id='print_hello',
    bash_command='echo \'Greetings. The date and time are \',
    dag=dag,
)

task2 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)
```

- Task pipeline

```
task1 >> task2
```

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
import datetime as dt

default_args = {
    'owner': 'me',
    'start_date': dt.datetime(2021, 7, 28),
    'retries': 1,
    'retry_delay': dt.timedelta(minutes=5),
}

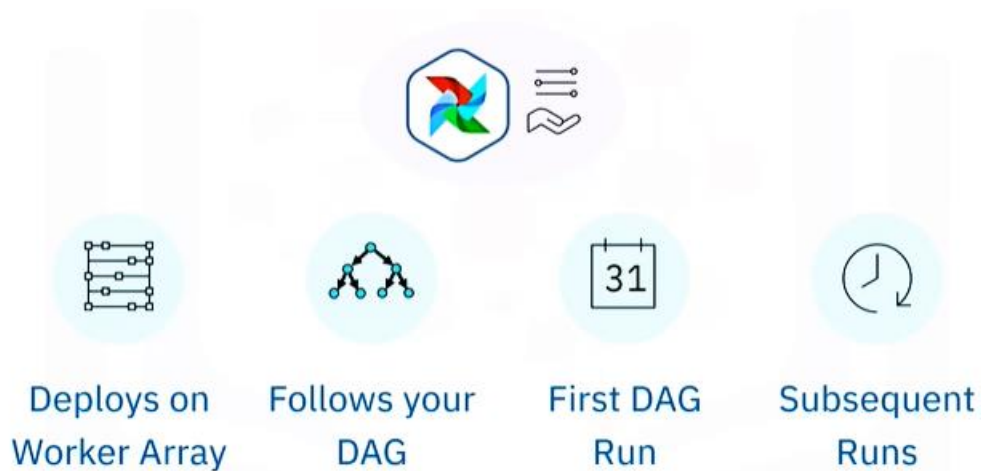
dag = DAG('simple_example',
          description='A simple example DAG',
          default_args=default_args,
          schedule_interval=dt.timedelta(seconds=5),
          )

task1 = BashOperator(
    task_id='print_hello',
    bash_command='echo \'Greetings. The date and time are \',
    dag=dag,
)

task2 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

task1 >> task2
```

Airflow Scheduler –



- Your new DAG has been created, but it hasn't yet been deployed.
- To that end, Airflow Scheduler is designed to run as a persistent service within an Airflow production environment.
- Apache Airflow Scheduler can be used to deploy your workflow on an array of workers.
- It follows the tasks and dependencies that you specified in your DAG.
- Once you start an Airflow Scheduler instance, your DAGs will start running based on the 'start date' you specified as code in each of your DAGs.
- After that, the Scheduler triggers each subsequent DAG run according to the schedule interval you specified.

Advantages of workflows as code –

- **Maintainable:** Developers can follow explicitly what has been specified, by reading the code.
- **Versionable:** Code revisions can easily be tracked by a version control system such as Git.
- **Collaborative:** Teams of developers can easily collaborate on both development and maintenance of the code for the entire workflow.
- **Testable:** Any revisions can be passed through unit tests to ensure the code still works as intended.

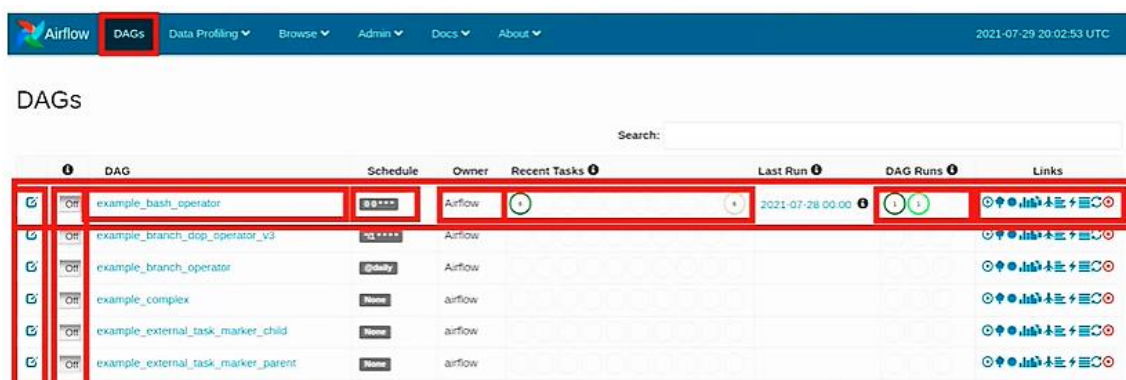
Summary: -

- In Apache Airflow, DAGs are workflows defined as Python code.
- Tasks, which are the nodes in your DAG, are created by implementing Airflow's built-in operators.

- Pipelines are specified as dependencies between tasks, which are the directed edges between nodes in your DAG.
- Airflow Scheduler schedules and deploys your DAGs.
- And finally, the key advantage of Apache Airflow's approach to representing data pipelines as DAGs is the fact that they are expressed as code. Accordingly, it makes your data pipelines more maintainable, testable, and collaborative.

3/ Apache Airflow UI

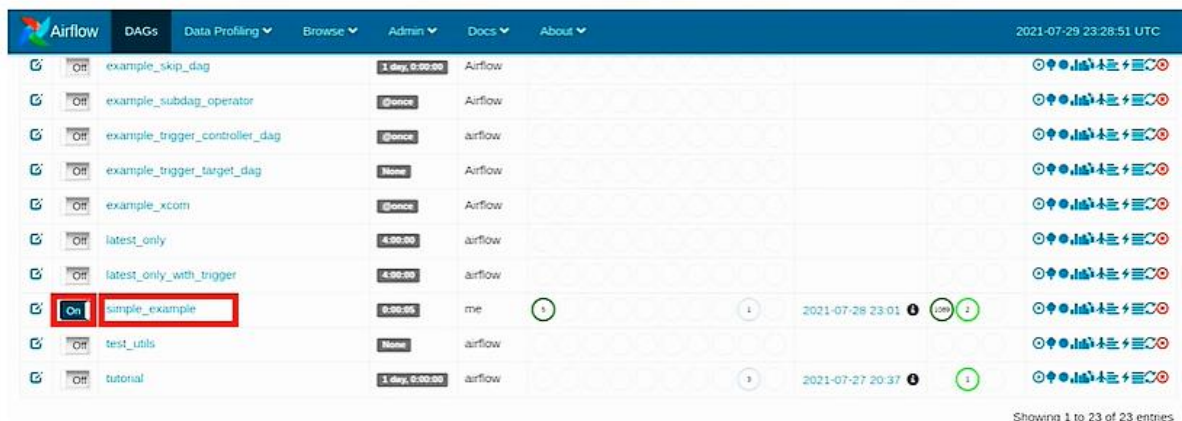
DAG View –



	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
<input type="checkbox"/>	example_bash_operator	<code>00 * * * *</code>	Airflow	example_bash_operator	2021-07-28 00:00	1	🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖
<input type="checkbox"/>	example_branch_dop_operator_v3	<code>00 * * * *</code>	Airflow	example_branch_dop_operator_v3			🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖
<input type="checkbox"/>	example_branch_operator	<code>00 * * * *</code>	Airflow	example_branch_operator			🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖
<input type="checkbox"/>	example_complex	<code>00 * * * *</code>	airflow	example_complex			🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖
<input type="checkbox"/>	example_external_task_marker_child	<code>00 * * * *</code>	airflow	example_external_task_marker_child			🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖
<input type="checkbox"/>	example_external_task_marker_parent	<code>00 * * * *</code>	airflow	example_external_task_marker_parent			🔍 📅 🔗 🔧 🔖 🔖 🔖 🔖

- It defaults to the 'DAGs View,' which is a table containing data about each DAG in your environment.
- Each row displays interactive information about a DAG in your environment, such as Each DAG's name;
- Its run schedule, in this case in the crontab format;
- The DAG's owner. This one is a built-in example from Airflow;
- The status of the tasks from the current or most recent DAG run;
- The status of all previous DAG runs;
- Plus, a collection of quick-links to drill down into more information related to the DAG.
- On the far left, you can drill down to access more interactive details, and in the next column you can toggle to pause a DAG.

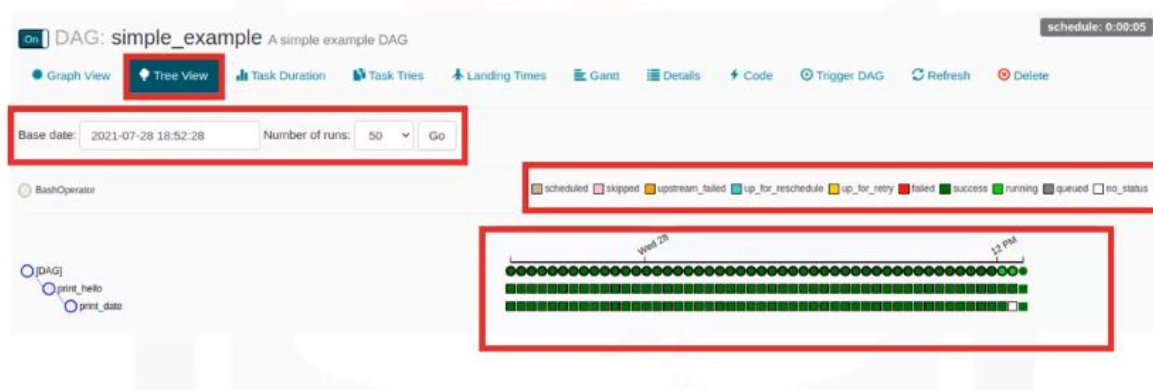
Visualize your DAG –



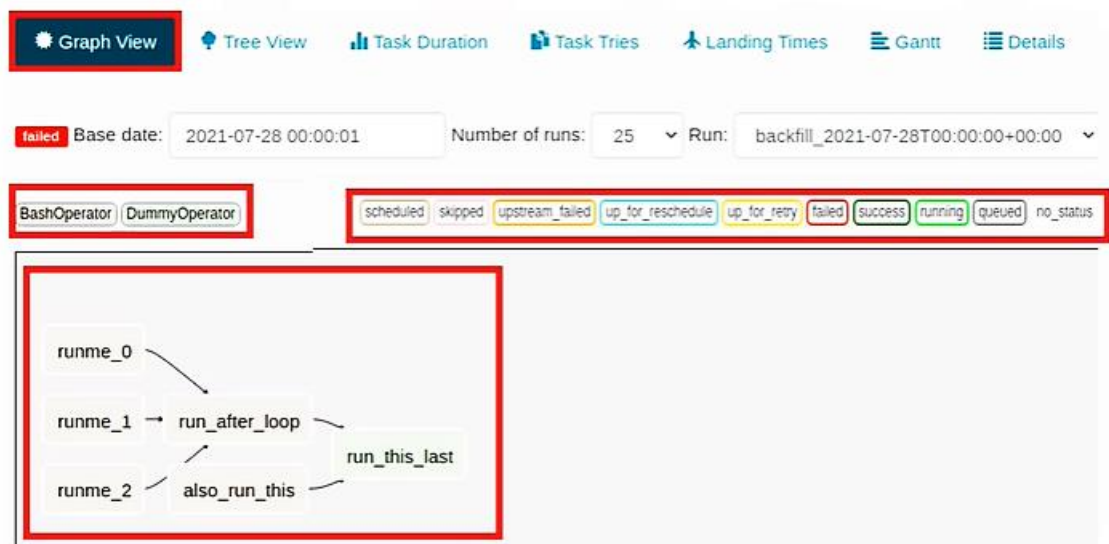
The screenshot shows the Airflow web interface with a list of DAGs. The 'simple_example' DAG is highlighted with a red box, showing its status as 'On' and its last run date as '2021-07-28 23:01'. The interface includes a top navigation bar with tabs for DAGs, Data Profiling, Browse, Admin, Docs, and About. The DAG list table has columns for DAG name, status, frequency, owner, last run date, and a set of action icons. The 'simple_example' DAG is the only one with a green 'On' status indicator.

DAG	Status	Frequency	Owner	Last Run	Actions
example_skip_dag	Off	1 day, 0:00:00	Airflow		
example_subdag_operator	Off	@once	Airflow		
example_trigger_controller_dag	Off	@once	airflow		
example_trigger_target_dag	Off	None	Airflow		
example_xcom	Off	@once	Airflow		
latest_only	Off	4:00:00	airflow		
latest_only_with_trigger	Off	4:00:00	airflow		
simple_example	On	0:00:05	me	2021-07-28 23:01	
test_utils	Off	None	airflow		
tutorial	Off	1 day, 0:00:00	airflow	2021-07-27 20:37	

- All of the DAGs here are currently not running.
- You can visualize DAGs in several ways, as follows. Start by clicking on the name of the DAG you want to visualize.
- This DAG, called 'simple example' happens to be running in production, as you can see here by the 'on' button.

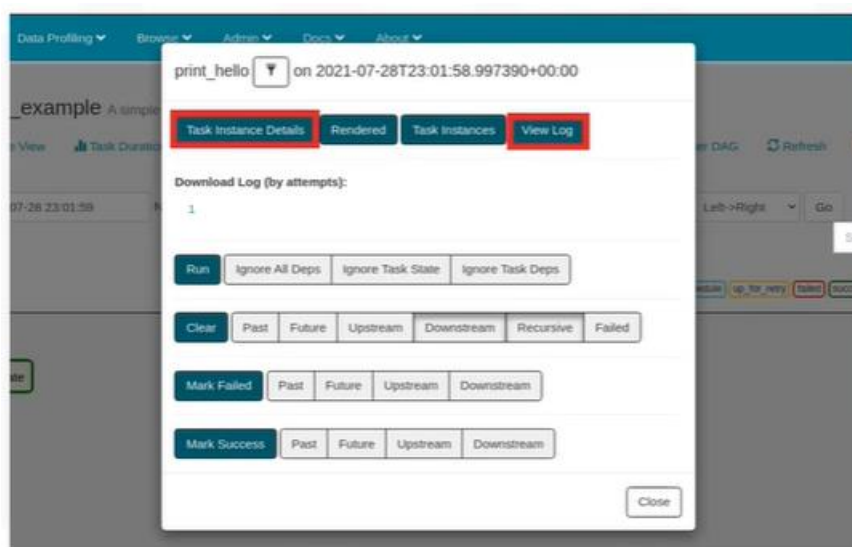


- Your DAG's 'Tree View' opens by default when you click on its name.
- It shows a timeline of the status of your DAG and its tasks for each run. Here, you can select the base-date and the number of runs to display.
- Each status is color-coded according to the legend, displayed here.
- You can also hover your mouse pointer over any task in the timeline to view more information about it.



- Here's another DAG, this time displayed in 'Graph View'.
- At the bottom of the screen, you can see your DAG's tasks and dependencies.
- Each task is color coded by its operator type, according to this legend .
- Here, you can filter your view by toggling these status-option buttons.

View context metadata for a task –



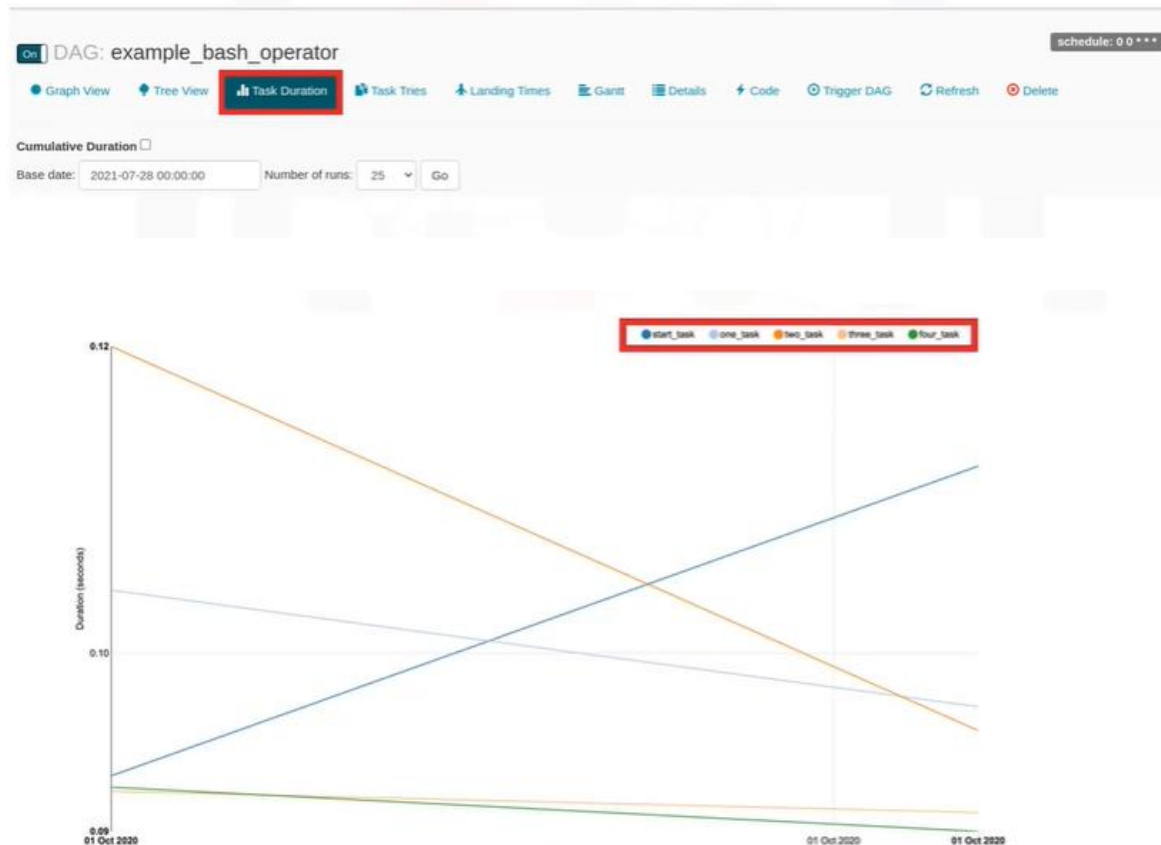
- The 'Task Instance Context Menu' can be accessed from any of these kinds of views, that is, views that display task instances for your DAG.
- This menu allows you, for example, to drill down on a selected task instance to view and edit many details about it, or to view the log file for your task.

Viewing the code for your DAG –

```
1 from airflow import DAG
2 from airflow.operators.bash_operator import BashOperator
3 import datetime as dt
4
5 default_args = {
6     'owner': 'me',
7     'start_date': dt.datetime(2021, 7, 28),
8     'retries': 1,
9     'retry_delay': dt.timedelta(minutes=5),
10 }
11
12 dag = DAG('simple_example',
13         description='A simple example DAG',
14         default_args=default_args,
15         schedule_interval=dt.timedelta(seconds=5),
16 )
17
18 task1 = BashOperator(
19     task_id='print_hello',
20     bash_command='echo \'Greetings. The date and time are \''
21     dag=dag,
22 )
23
24 task2 = BashOperator(
25     task_id='print_date',
26     bash_command='date',
27     dag=dag,
28 )
29
30 task1 >> task2
```

- By clicking on the Code button, you can also view the complete Python source code that defines your DAG.
- Here we see the building blocks of your DAG, for example, the library imports, and the individual task definitions, which invoke Bash Operators in this case.

View task duration timelines –



- By clicking on 'Task Duration', You can view a timeline chart of your DAG's task durations to see how they have been performing.
- Here you can toggle the tasks that you wish to highlight the last N runs for.

Summary: -

- Apache Airflow has a rich UI that simplifies working with data pipelines.
- You can visualize your DAG in several informative ways, including both graph and tree mode.
- You can also review the Python code that originally defined your DAG.
- You can analyze the duration of each task in your DAG over multiple runs.
- You can select context metadata for any task instance.

4) Build DAG Using Airflow

Airflow DAG definition script –

Python script blocks:

- Library imports
- DAG arguments
- DAG definition
- Task definitions
- Task pipeline

Build an Airflow pipeline –

```
$ simple_example_DAG.py
```

Print Greeting → Print Date & Time



Python script blocks:

- Python library imports

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
import datetime as dt
```

Python script containing:

- Python library imports
- DAG arguments

```
default_args = {
    'owner': 'me',
    'start_date': dt.datetime(2021, 7, 28),
    'retries': 1,
    'retry_delay': dt.timedelta(minutes=5),
}
```

Python script containing:

- Python library imports
- DAG arguments
- DAG definition

```
dag=DAG('simple_example',
        description='A simple example DAG',
        default_args=default_args,
        schedule_interval=dt.timedelta(seconds=5),
)
```

Python script containing:

- Python library imports
- DAG arguments
- DAG definition
- Task definitions
- Task pipeline


```

task1 = BashOperator(
    task_id='print_hello',
    bash_command='echo \'Greetings. The date and time are \'',
    dag=dag,
)

task2 = BashOperator(
    task_id='print_date',
    bash_command='date',
    dag=dag,
)

```

Python script
containing:

- Python library imports
- DAG arguments
- DAG definition
- Task definitions
- Task pipeline



Summary: -

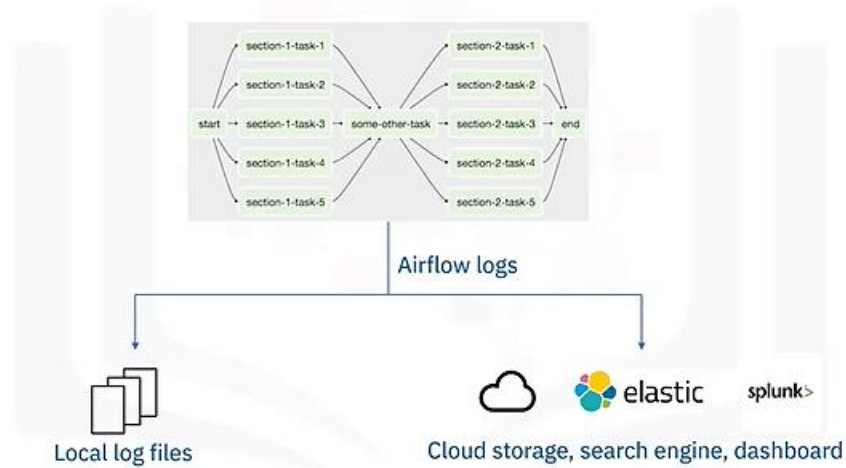
- An Airflow pipeline is a Python script that instantiates an Airflow DAG object.
- Key components of a DAG definition file are library imports, DAG arguments, DAG and task definitions, and the 'task pipeline' specification.
- You can specify a 'schedule interval' in your DAG definition if you want it to run repeatedly by setting the 'schedule interval' parameter.
- And finally, tasks are instantiated operators, imported from the Apache Airflow operators (apache.operators library)

5/ Airflow Monitoring and Logging

Logging –

- The logging capability is required for developers to monitor the status of tasks in DAG runs, and to diagnose and debug issues.
- By default, Airflow logs are saved to local file systems as log files. This makes it convenient for developers to quickly review the log files, especially in a development environment.

- For Airflow production deployments, the log files can be sent to cloud storage such as IBM cloud, AWS, or Azure for remote accessing. The log files can also be sent to search engines and dashboards for further retrieval and analysis.
- Airflow recommends using Elasticsearch and Splunk, which are two popular document database and search engines, to index, search, and analyze log files.



Airflow log files –

- By default, log files are organized by DAG IDs and Task IDs, and you can find a specific log file for a task execution using the following path convention:
logs/dag_id/task_id/execution_date/try_number.log
- For example, suppose you want to find the log of the first execution try of task 1 in a dummy DAG at a specific time. You can go to the folder
logs/dummy_dag/task1/execution_date/1.log
- And if you click the first log file, you can check a lot of useful information, such as: the running command, command results, task results, and so on.

```
[2021-07-29 17:29:07,778] {subprocess.py:63} INFO - Running command: ['bash', '-c', 'sleep 1']
[2021-07-29 17:29:07,836] {subprocess.py:75} INFO - Output:
[2021-07-29 17:29:08,857] {subprocess.py:83} INFO - Command exited with return code 0
[2021-07-29 17:29:09,377] {taskinstance.py:1191} INFO - Marking task as SUCCESS. dag_id=dummy_dag, task_id=task1, execution_date=20210729T000000, start_date=20210729T172906
[2021-07-29 17:29:09,777] {taskinstance.py:1245} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2021-07-29 17:29:09,817] {local_task_job.py:151} INFO - Task exited with return code 0
```

Reviewing task events via UI –

- You can also quickly review the task events via UI provided by the Airflow webserver. You can search events with fields like DAG ID, Task ID, and Execution Date, and quickly get an overview of the specific DAGs and tasks you are looking for.

List Log

Search -

Add Filter ▾

x Dag Id

Starts with ▾

dummy_dag

x Task Id

Starts with ▾

task1

x Execution Date

Greater than ▾

2021-07-29 00:44:26+00:00

Search Q

⏪

Id	Dttm	Dag Id	Task Id	Event	Execution Date	Owner	Extra
613	2021-07-29, 17:32:23	dummy_dag	task1	success	2021-07-29, 00:51:00	Ramesh Sannareddy	
604	2021-07-29, 17:32:20	dummy_dag	task1	cli_task_run	2021-07-29, 00:51:00	default	{"host_name": "2155f2e70f0e", "full_command": "[/home/airflow/local/bin/airflow, 'celery', 'worker']"}
603	2021-07-29, 17:32:20	dummy_dag	task1	running	2021-07-29, 00:51:00	Ramesh Sannareddy	
602	2021-07-29, 17:32:19	dummy_dag	task1	success	2021-07-29, 00:50:00	Ramesh Sannareddy	
598	2021-07-29, 17:32:16	dummy_dag	task1	cli_task_run	2021-07-29, 00:51:00	default	{"host_name": "2155f2e70f0e", "full_command": "[/home/airflow/local/bin/airflow, 'celery', 'worker']"}
594	2021-07-29, 17:32:15	dummy_dag	task1	cli_task_run	2021-07-29, 00:50:00	default	{"host_name": "2155f2e70f0e", "full_command": "[/home/airflow/local/bin/airflow, 'celery', 'worker']"}

Monitoring metrics –

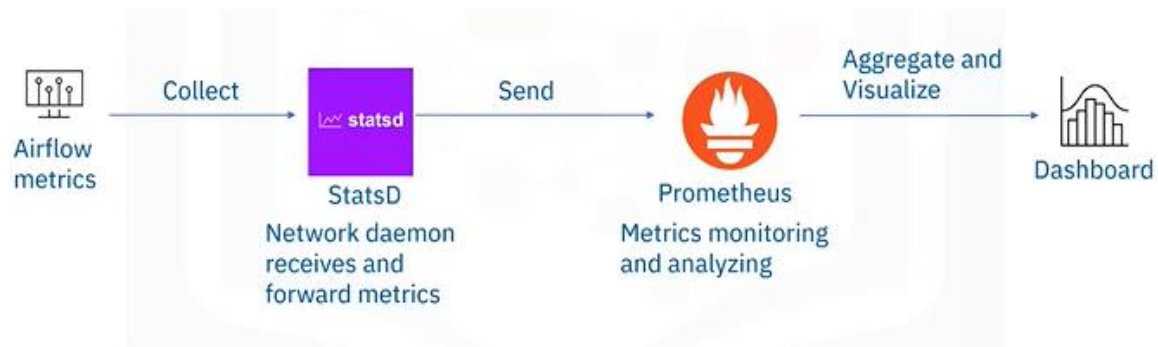
- Airflow produces three different types of metrics for checking and monitoring components' health.

These are:

- **Counters** – Metrics that always increase
 - Total count of task instances failures
 - Total count of task instances successes
- **Gauges** – Metrics that may fluctuate
 - Number of running tasks
 - DAG bag size, or number of DAGs in production
- **Timers** – Metrics related to time duration
 - Milliseconds to finish a task
 - Milliseconds to reach a success or failed state

Storing and analyzing metrics –

- Similar to logs, the metrics produced in Airflow production deployments should be sent and analyzed by dedicated repositories and tools.
- Airflow recommends using StatsD, which is a network daemon that can gather metrics from Airflow and send them to a dedicated metrics monitoring system.
- For metrics monitoring and analysis, Airflow recommends using Prometheus, which is a popular metrics monitoring and analysis system.
- Prometheus can also aggregate and visualize metrics in a dashboard for more interactive visual analytics.



Summary: -

- You can save Airflow logs into local file systems and send them to cloud storage, search engines, and log analyzers.
- Airflow recommends sending production deployment logs to be analyzed by Elasticsearch or Splunk.
- With Airflow's UI, you can view DAGs and task events easily.
- You also learned that The three types of Airflow metrics are counters, gauges, and timers.
- Airflow recommends that you send production deployment metrics for analysis by Prometheus via StatsD.

Summary & Highlights: -

- Apache Airflow is scalable, dynamic, extensible, and lean
- The five main features of Apache Airflow are pure Python, useful UI, integration, easy to use, and open source
- A common use case is that Apache Airflow defines and organizes machine learning pipeline dependencies
- Tasks are created with Airflow operators
- Pipelines are specified as dependencies between tasks
- Pipeline DAGs defined as code are more maintainable, testable, and collaborative
- Apache Airflow has a rich UI that simplifies working with data pipelines
- You can visualize your DAG in graph or tree mode
- Key components of a DAG definition file include DAG arguments, DAG and task definitions, and the task pipeline
- The 'schedule_interval' parameter specifies how often to re-run your DAG

- You can save Airflow logs into local file systems and send them to cloud storage, search engines, and log analyzers
- Airflow recommends sending production deployment logs to be analyzed by Elasticsearch or Splunk
- With Airflow's UI, you can view DAGs and task events
- The three types of Airflow metrics are counters, gauges, and timers
- Airflow recommends that production deployment metrics be sent to and analyzed by Prometheus via StatsD

Week 4

Building Streaming Pipelines using Kafka

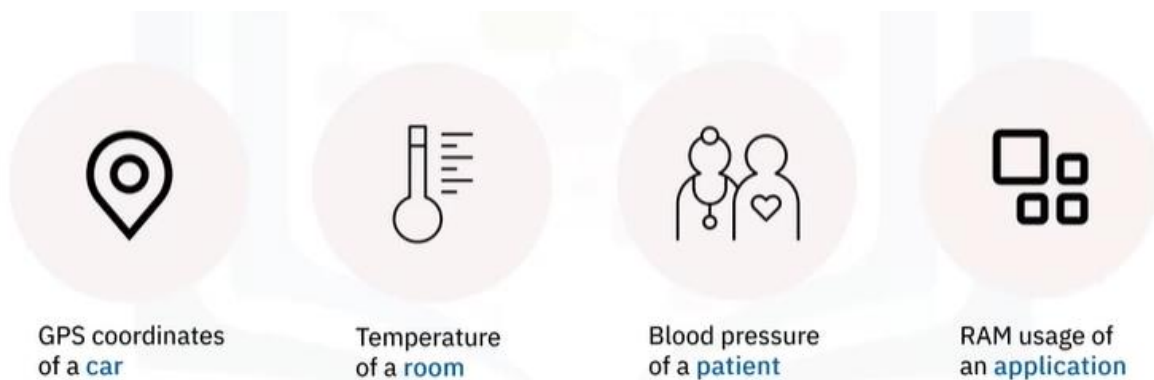
Module 1: - Using Apache Kafka to build Pipelines for Streaming Data

1/ Distributed Event Streaming Platform Components

What is an Event?

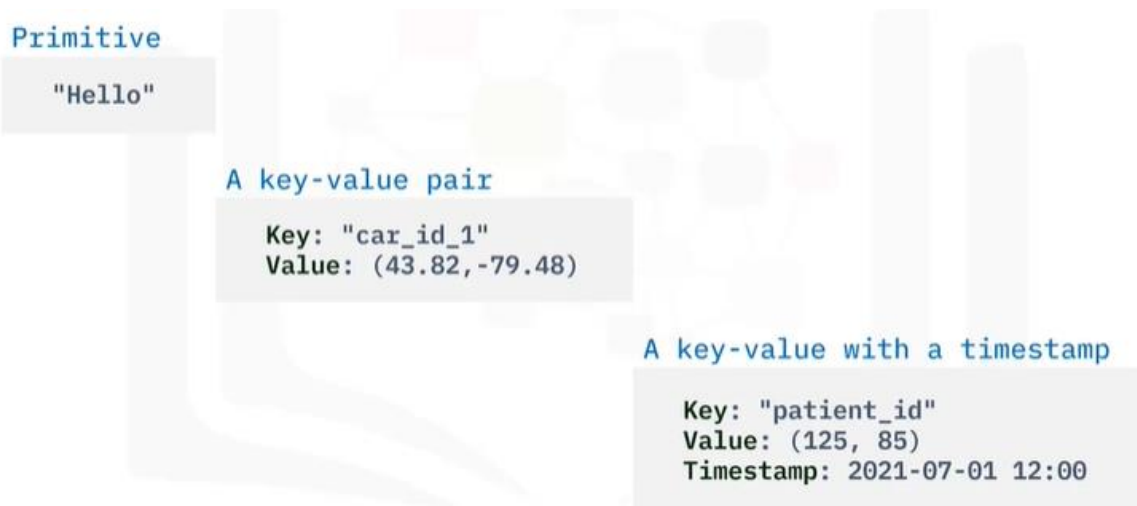
Events describe an entity's observable state updates over time.

Examples –



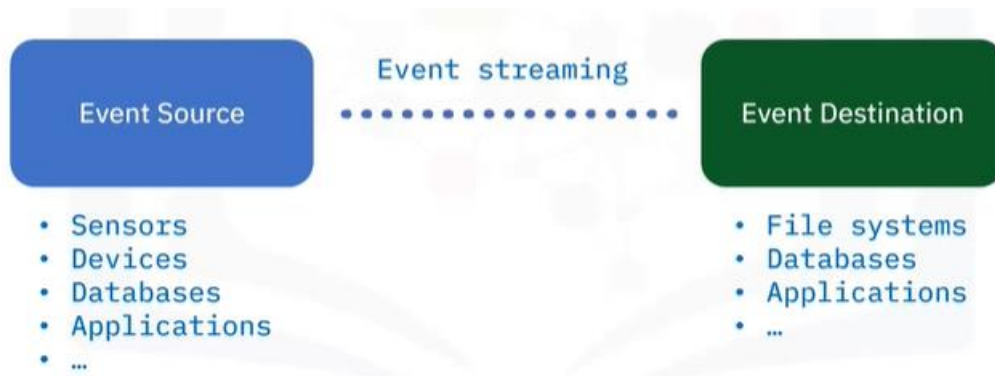
Common Event formats –

An Event, as a special type for data, has different formats.



One source to one destination –

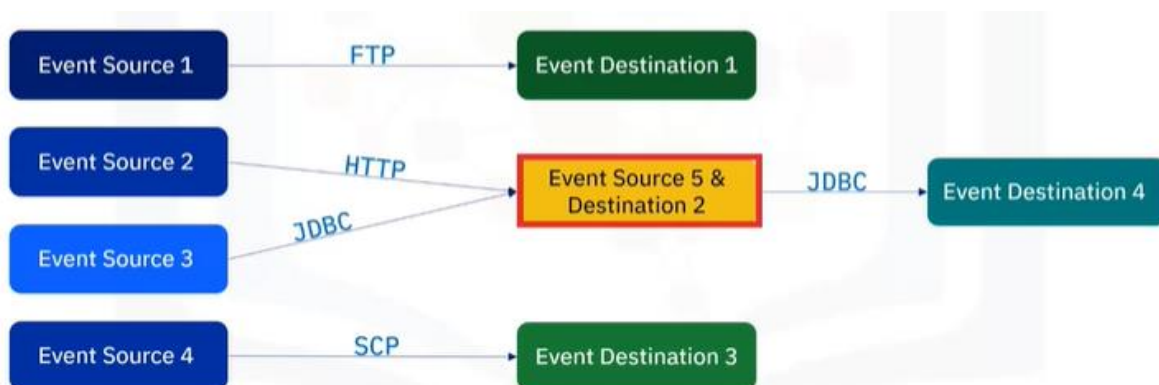
Event streaming from one event source to one destination.



Suppose we have one event source such as a group of sensors, a monitoring device, a database, or a running application. This event source may continuously generate a large event volume at a short time interval or nearly real-time. Those real-time events need to be properly transported to an event destination such as a file system, another external database, or an application. The continuous event transportation between an event source and an event destination is called event streaming.

Many sources to many destinations –

Event streaming from multiple event sources to multiple destinations.



In a real-world scenario, event streaming can be really complicated with multiple distributed event sources and destinations, as data transfer pipelines may be based on different communication protocols such as: **FTP**: File Transfer Protocol **HTTP**: Hypertext Transfer Protocol **JDBC**: Java Database Connectivity **SCP**: Secure copy. And so on. An event destination can also be an event source simultaneously. For example, one application could receive an event stream and process it, then transport the processed results as an event stream to another destination. To overcome such a challenge of handling

different event sources and destinations, we will need to employ the Event Stream Platform, or ESP.

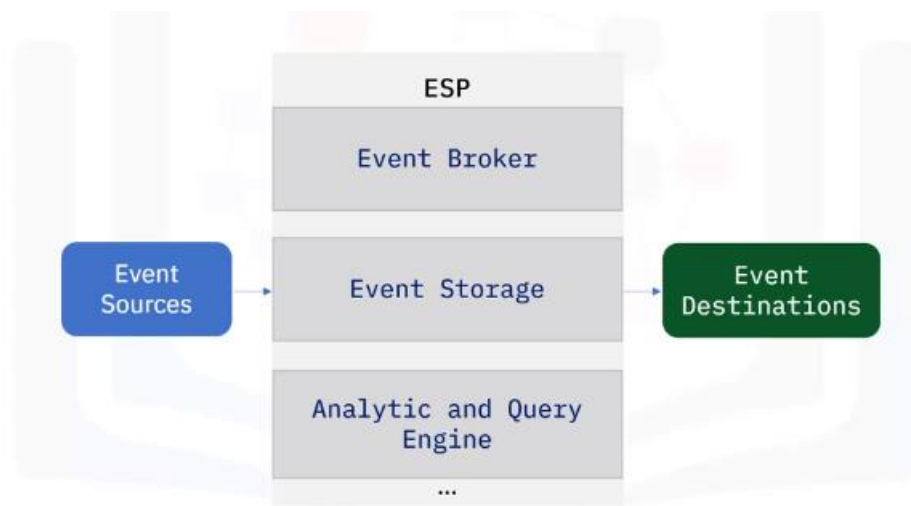
Event Streaming Platform (ESP) –

- An ESP acts as a middle layer among various event sources and destinations and provides a unified interface for handling event-based ETL.
- As such, all event sources only need to send events to an ESP instead of sending them to the individual event destination.
- On the other side, event destinations only need to subscribe to an ESP, and just consume the events sent from the ESP instead of the individual event source.



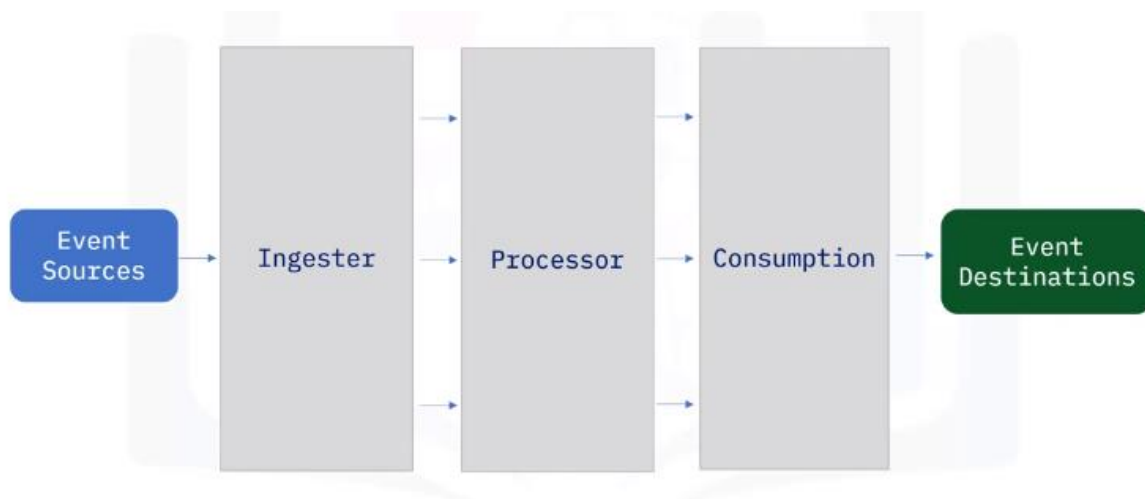
Common components of an ESP –

- The first and foremost component is the Event broker, which is designed to receive and consume events. It is the core component of an ESP.
- The second common component of an ESP is Event Storage, which is used for storing events being received from event sources. Accordingly, event destinations do not need to synchronize with event sources, and stored events can be retrieved at will.
- The third common component is the Analytic and Query Engine which is used for querying and analyzing the stored events.



Event broker –

- It normally contains three sub-components: ingester, processor, and consumption.
- The ingester is designed to efficiently receive events from various event sources.
- The processor performs operations on data such as serializing and deserializing; compressing and decompressing; encryption and decryption; and so on.
- The consumption component retrieves events from event storage and efficiently distributes them to subscribed event destinations.



Popular ESPs –

- There are many ESP solutions, including: Apache Kafka Amazon Kinesis Apache Flink Apache Spark Apache Storm and more such as Logstash in Elastic Stack, and so on.
- Each has its unique features and application scenarios.
- Among these ESPs, Apache Kafka is probably THE most popular one.



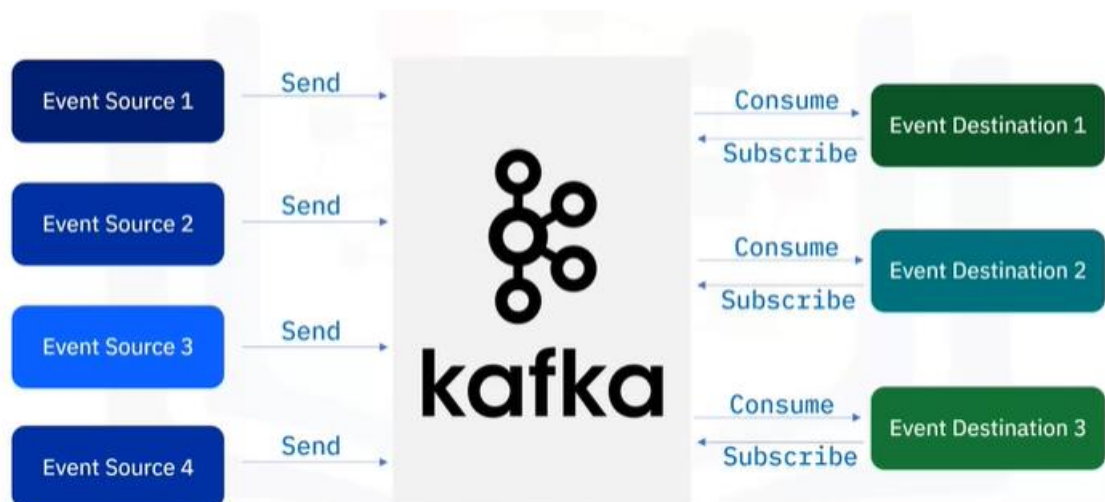
Summary: -

- An event stream represents entities' status updates over time. Common event formats include primitive data types, key-value, and key-value with a timestamp.
- An ESP is needed especially when there are multiple event sources and destinations.
- The main components of an ESP are Event broker, Event storage, Analytic and Query Engine.
- Apache Kafka is the most popular open source ESP.
- And finally, popular ESPs include Apache Kafka, Amazon Kinesis, Apache Flink, Apache Spark, Apache Storm, and more.

2/ Apache Kafka Overview

Apache Kafka –

Apache Kafka, an open source project, has probably become THE most popular event streaming platform.



Common use cases –

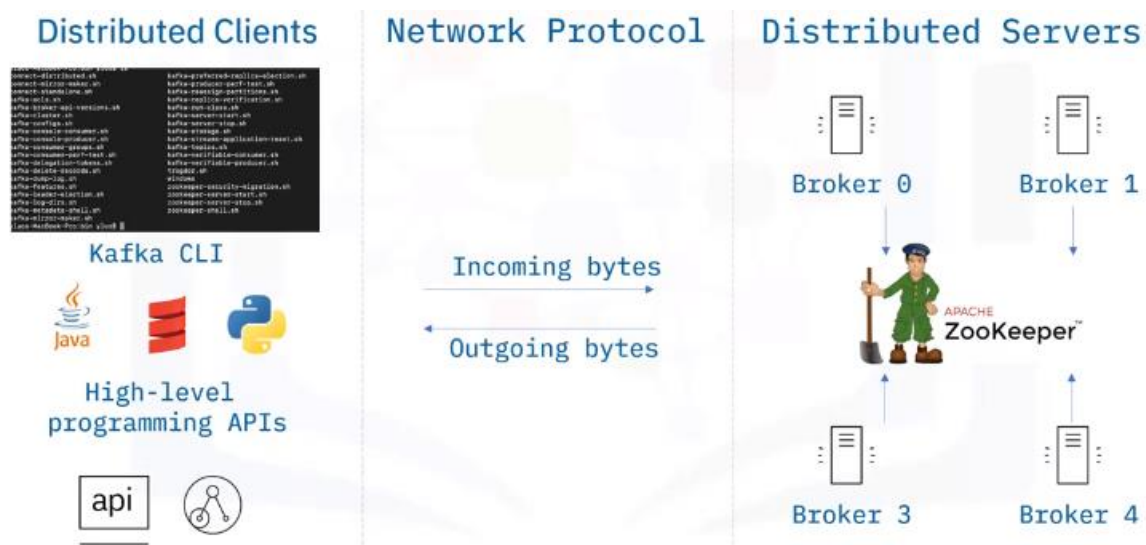
Kafka is a comprehensive platform and can be used in many application scenarios.



- Kafka was originally used to track user activities such as keyboard strokes, mouse clicks, page views, searches, gestures, screen time, and so on.
- But now it is also suitable for all kinds of metric-streaming such as sensor readings, GPS, as well as hardware and software monitoring.
- For enterprise applications and infrastructure with a huge number of logs, Kafka can be used to collect and integrate them into a centralized repository.
- For banks, insurance, or FinTech companies, Kafka is also widely used for payments and transactions.
- These scenarios are just the tip of the iceberg. You basically can use Kafka in scenarios when you want high throughput and reliable data transportation services among various event sources and destinations.
- All these events will be ingested in Kafka, and become available for subscriptions and consumption including: Further data storage and movement to other online or offline databases and backups.
- Real-time processing and analytics, including dashboard, machine learning, AI algorithms, and so on.
- Generating notifications such as email, text messages, and instant messages, or data governance and auditing to make sure sensitive data such as bank transactions are complying with regulations.

Kafka architecture –

- Kafka has a distributed client-server architecture.
- For the server side, Kafka is a cluster with many associated servers called brokers, acting as the event broker to receive, store, and distribute events. All those brokers are managed by another distributed system called ZooKeeper to ensure all brokers work in an efficient and collaborative way.
- Kafka uses a TCP (Transmission Control Protocol) based network communication protocol to exchange data between clients and servers.
- For the client side, Kafka provides different types of clients, such as Kafka CLI (Command Line Interface) which is a collection of shell scripts to communicate with the Kafka server, many high-level programming APIs such as Java, Scala, REST APIs, and some specific third-party clients made by the Kafka community. Accordingly, you could choose different clients based on your requirements.



Main features of Apache Kafka –

- **Distribution system** – which makes it highly scalable to handle high data throughput and concurrency.
- **Highly Scalable** – A Kafka cluster normally has multiple event brokers which can handle event streaming in parallel. As such, Kafka is very fast and highly scalable.
- **Highly reliable** – Kafka also divides event storage into multiple partitions and replications which makes it fault-tolerant and highly reliable.
- **Permanent persistency** – Kafka stores the events permanently. As such, event consumption could be done whenever suitable for consumers without a deadline.
- **Open-source** – Kafka is open source, meaning that you can use it for free, and even customize it based on your specific requirements.

Event streaming as a service –

- Even though Kafka is open source and well-documented, it is still challenging to configure and deploy a Kafka without professional assistance.
- Deploying a Kafka cluster requires extensive efforts for tuning infrastructure and consistently adjusting the configurations, especially for enterprise-level deployments.
- Fortunately, several commercial service providers offer an on-demand ESP-as-a-Service to meet your streaming requirements. Many of them are built on top of Kafka and provide added value for customers.
- Some well-known ESP providers include:
 - Confluent Cloud, which provides customers with fully managed Kafka services, either on-premises or on-cloud.
 - IBM Event Streams, which is also based on Kafka and provides many add-on services such as enterprise-grade security, disaster recovery, and 24-7 cluster monitoring.
 - Amazon Managed Streaming for Apache Kafka, which is also a fully managed service to facilitate the build and deployment of Kafka.



Confluent
Cloud



IBM Event
Streams



Amazon
MSK

Summary: -

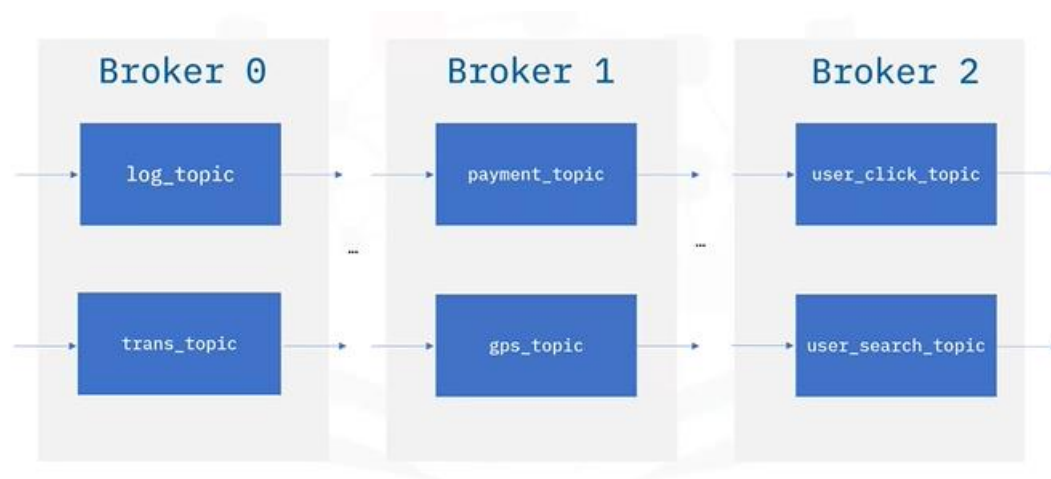
- Apache Kafka is one of the most popular open source ESPs.
- Common Kafka use cases include user-activity tracking, metrics and logs integrations, and financial transaction processing.
- Apache Kafka is a highly scalable and reliable platform that stores events permanently.
- And finally, popular Kafka-based ESP service providers include Confluent Cloud, IBM Event Streams, and Amazon Managed Streaming.

3) Building Event Streaming Pipelines using Kafka

Broker and Topic –

- A Kafka cluster contains one or many brokers. You may think of a Kafka broker as a dedicated server to receive, store, process, and distribute events.
- Brokers are synchronized and managed by another dedicated server called ZooKeeper.
- Brokers manage to save published events into topics and distribute the events to subscribed consumers.

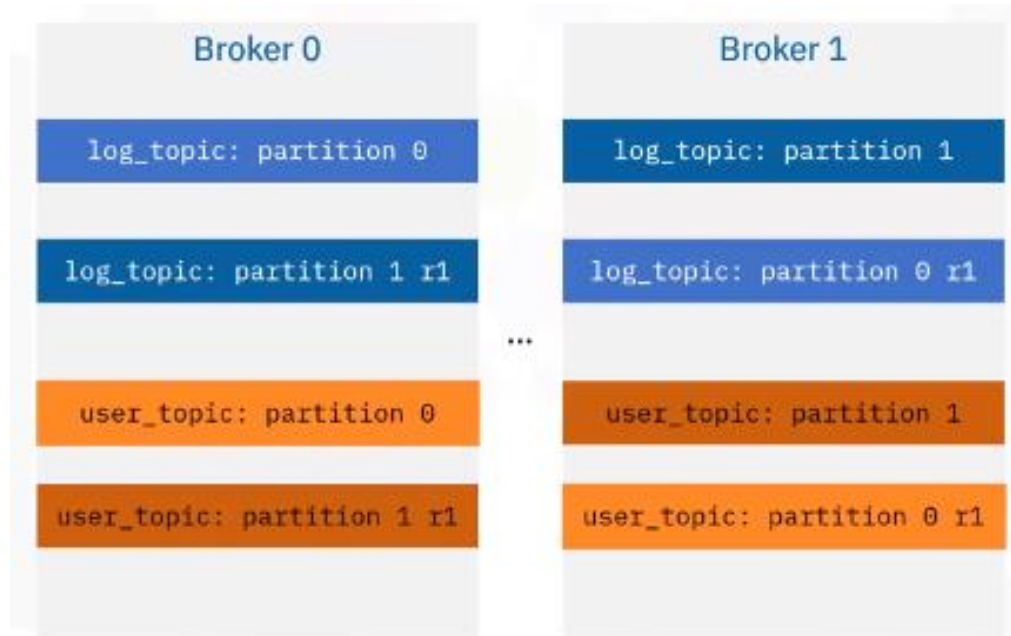
For example, here we have A log topic and a transaction topic in broker 0. A payment topic and a GPS topic in broker 1. and, a user click topic and user search topic in broker 2. Each broker contains one or many topics. You can think of a topic as a database to store specific types of events, such as logs, transactions, and metrics, for example.



Partitions and replications –

- Like many other distribution systems, Kafka implements the concepts of partitioning and replicating.
- It uses topic partitions and replications to increase fault-tolerance and throughput so that event publication and consumption can be done in parallel with multiple brokers.
- In addition, even if some brokers are down, Kafka clients are still able to work with the target topics replicated in other working brokers.

For example: A log topic has been separated into two partitions: 0, 1, and a user topic has been separated into two partitions: 0, 1. And each topic partition is duplicated into two replications and stored in different brokers.



Kafka topic CLI –

- The Kafka CLI, or command line interface client provides a collection of powerful script files for users to build an event streaming pipeline: The Kafka-topics script is the one you probably will be using often to manage topics in a Kafka cluster.
- It is straightforward.
- Let's have a look at some common usages:
 - This first one is to create a topic. Here we are trying to create a topic called 'log_topic' with two partitions and two replications.
 - One important note here is that many Kafka commands, like kaf-topics, require users to refer to a running Kafka cluster with a host and a port, such as a localhost with the port 9092.
 - After you have created some topics, you can check all created topics in the cluster using the 'list option.'
 - And, if you want to check more details of a topic, such as partitions and replications, you can use the 'describe option'.
 - And you can delete a topic using the 'delete option.'

Create a topic:

```
kafka-topics --bootstrap-server localhost:9092 --topic log_topic --create --partitions 2 --replication-factor 2
```

List topics:

```
kafka-topics --bootstrap-server localhost:9092 --list
```

Get topic details:

```
kafka-topics --bootstrap-server localhost:9092 --describe log_topic
```

Delete a topic:

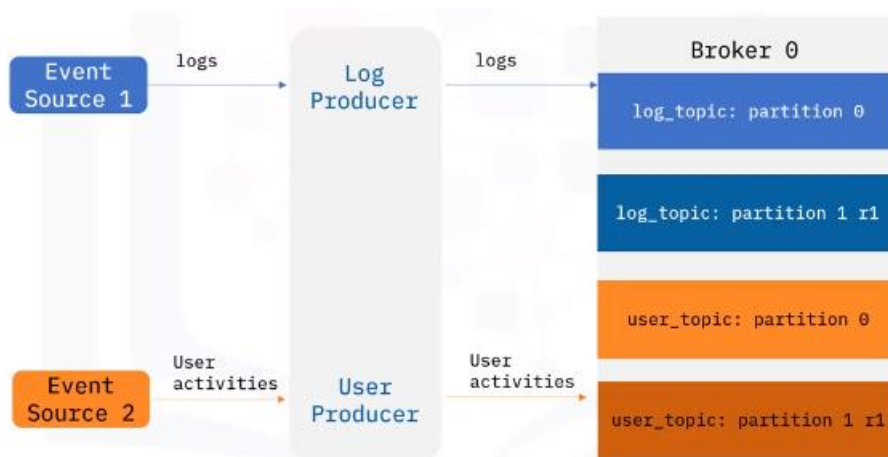
```
kafka-topics --bootstrap-server localhost:9092 --topic log_topic --delete
```

Kafka producer –

- Client applications that publish events to topic partition
- An event can be optionally associated with a key
- Events associated with the same key will be published to the same topic partition
- Events not associated with the key will be published to topic partitions in rotation

Kafka producer in action –

Suppose you have an event source 1, which generates various log entries, and an event source 2, which generates user-activity tracking records. Then, you can create a Kafka producer to publish log records to log topic partitions, and a user producer to publish user-activity events to user topic partitions, respectively. When you publish events in producers, you can choose to associate events with a key, for example, an application name or a user ID.



Kafka producer CLI –

- Kafka provides the Kafka producer CLI for users to manage producers.
- The most important aspect is starting a producer to write or publish events to a topic:
 - Here you start a producer and point it to the `log_topic`, then you can type some messages in the console to start publishing events.
 - For example, `log1`, `log2`, and `log3`.
 - You can provide keys to events to make sure the events with the same key will go to the same partition.
 - Here you are starting a producer to `user_topic`, with the `parse.key` option to be true, and you also specify the `key.separator` to be comma. Then you can write messages as follows: key: 'user1', value: 'login website'. Key: 'user1', value: 'click the top item'. And, key: 'user1', value: 'logout website'.

Start a producer to a topic, without keys:

```
kafka-console-producer --broker-list localhost:9092 --topic log_topic
> log1
> log2
> log3
```

Start a producer to a topic, with keys:

```
kafka-console-producer --broker-list localhost:9092 --topic user_topic
--property parse.key=true --property key.separator=,
> user1, login website
> user1, click the top item
> user1, logout website
```

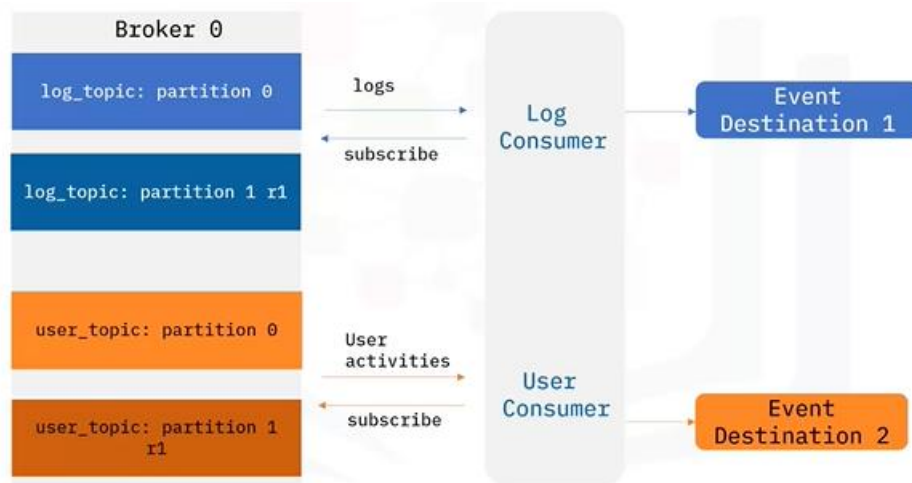
- Accordingly, all events about `user1` will be saved in the same partition to facilitate the reading for consumers.

Kafka consumer –

- Consumers are clients subscribed to topics
- Consume data in the same order
- Store an offset record for each partition
- Offset can be reset to zero to read all events from the beginning again

Kafka consumer in action –

- To read published log and user events from topic partitions, you will need to create log and user consumers, and make them subscribe to corresponding topics.
- Then Kafka will push the events to those subscribed consumers.
- Then, the consumers will further send to event destinations.



Kafka consumer CLI –

- To start a consumer is also easy, using the Kafka consumer script.
 - Let's read events from the `log_topic`.
 - You just need to run the `Kafka-console-consumer` script and specify a Kafka cluster and the topic to subscribe to.
 - Here, you can subscribe to and read events from the topic `log_topic`.
 - Then they started consumer will read only the new events, starting from the last partition offset.
 - Then they started consumer will read only the new events, starting from the last partition offset.
 - After those events are consumed, the partition offset for the consumer will also be updated and committed back to Kafka.
 - Very often a user wants to read all events from the beginning, as a playback of all historical events.
 - To do so, you just need to add the 'from beginning' option.
 - Now, you can read all events starting from offset 0.

```

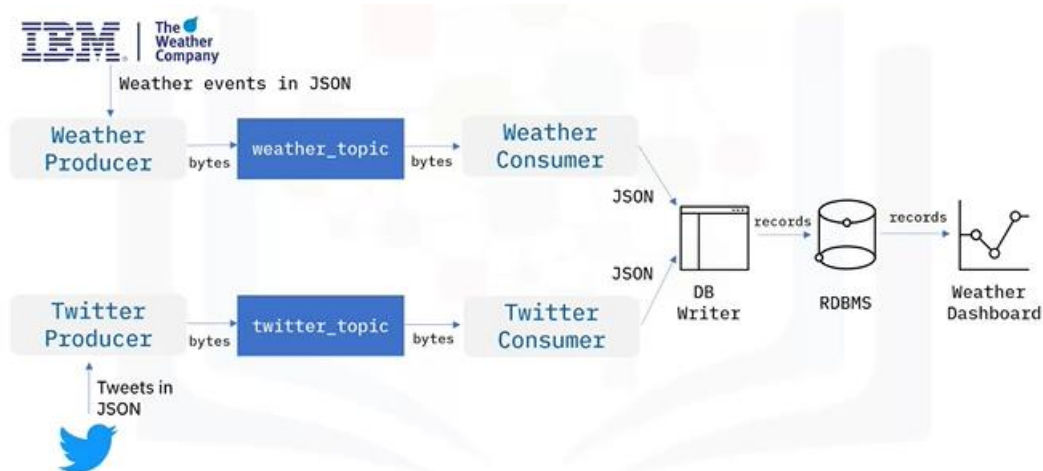
Read new events from the log_topic, offset 1:
kafka-console-consumer --bootstrap-server localhost:9092 --topic log_topic
> (offset 2) log3
> (offset 3) log4
  
```

```

Read all events from the log_topic, offset 0:
kafka-console-consumer --bootstrap-server localhost:9092 --topic log_topic
--from-beginning
> (offset 0) log1
> (offset 1) log2
> (offset 2) log3
> (offset 3) log4
  
```

A weather pipeline example –

- Let's have a look at a more concrete example to help you understand how to build an event streaming pipeline end-to-end.
- Suppose you want to collect and analyze weather and Twitter event streams, so that you can correlate how people talk about extreme weather on Twitter.
- Here you can use two event sources: IBM Weather API to obtain real-time and forecasted weather data in JSON format. And Twitter API to obtain real-time tweets and mentions, also in JSON format.
- To receive weather and twitter JSON data in Kafka, you then create a weather topic and a Twitter topic in a Kafka cluster, with some partitions and replications.
- To publish weather and Twitter JSON data to the two topics, you need to create a weather producer and a Twitter producer.
- The event's JSON data will be serialized into bytes and saved in Kafka topics.
- To read events from the two topics, you need to create a weather consumer and a Twitter consumer.
- The bytes stored in Kafka topics will be deserialized into event JSON data. If you now want to transport the weather and Twitter event JSON data from the consumers to a relational database, you will use a DB writer to parse those JSON files and create database records.
- And then you can write those records into a database using SQL insert statements.
- Finally, you can query the database records from the relational database and visualize and analyze them in a dashboard to complete the end-to-end pipeline.



Summary: -

- The core components of Kafka are:
 - **Brokers:** The dedicated server to receive, store, process, and distribute events.
 - **Topics:** The containers or databases of events.
 - **Partitions:** Divide topics into different brokers.
 - **Replications:** Duplicate partitions into different brokers.

- **Producers:** Kafka client applications to publish events into topics.
- **Consumers:** Kafka client applications subscribed to topics and read events from them.
- The Kafka-topics CLI manages topics
- The Kafka-console-producer CLI manages producers
- The Kafka-console-consumer manages consumers

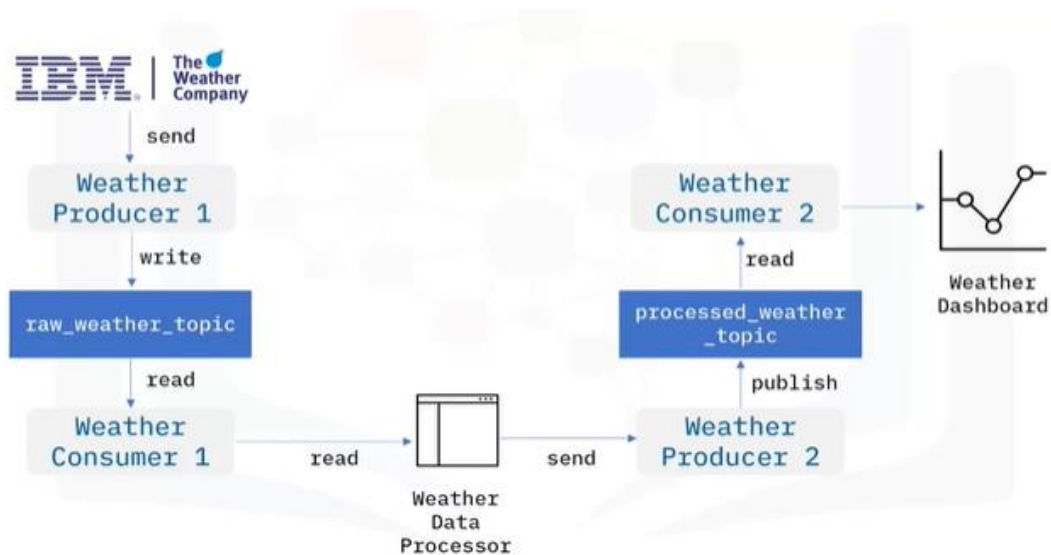
4) Kafka Streaming Process

Ad hoc weather stream processing -

In event streaming, in addition to transporting data, data engineers also need to process data through, for example, data filtering, aggregation, and enhancement. Any applications developed to process streams are called stream processing applications. For stream processing applications based on Kafka, a straightforward way is to implement an ad hoc data processor to read events from one topic, process them, and publish them to another topic.

Example:

- You first request raw weather JSON data from a weather API, and you start a weather producer to publish the raw data into a raw weather topic.
- Then you start a consumer to read the raw weather data from the weather topic. Next, you create an ad hoc data processor to filter the raw weather data to only include extreme weather events, such as very high temperatures.
- Such a processor could be a simple script file or an application which works with Kafka clients to read and write data from Kafka.
- Afterwards, the processor sends the processed data to another producer, and it gets published to a processed weather topic.
- And finally, the processed weather data will be consumed by a dedicated consumer and sent to a dashboard for visualization.



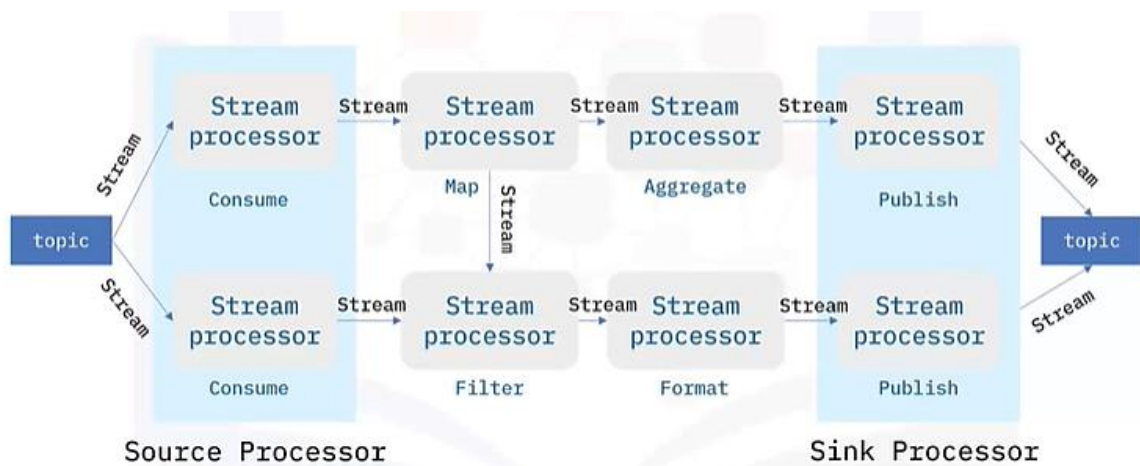
Kafka Streams API –

- Simple client library to facilitate data processing in event streaming pipelines
- Processes and analyzes data stored in Kafka topics
- Record only processed once
- Processing one record at a time



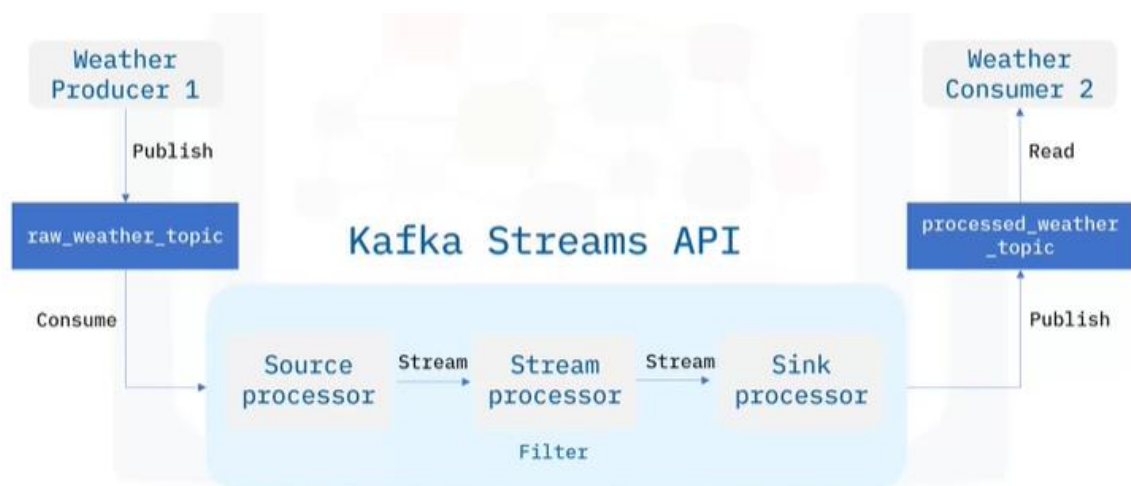
Stream processing topology -

- Kafka Streams API is based on a computational graph called a stream-processing topology.
- In this topology, each node is a stream processor which receives streams from its upstream processor, performs data transformations such as mapping, filtering, formatting, aggregation, and produces output streams to its downstream stream processors. Thus, the edges of the graph are I/O streams.
- There are two special types of processors:
 - On the left, you can see the source processor which has no upstream processors. A source processor acts like a consumer which consumes streams from Kafka topics and forwards the processed streams to its downstream processors.
 - On the right, you can see the sink processor, which has no downstream processors. A sink processor acts like a producer which publishes the received stream to a Kafka topic.



Redesigning of Kafka Weather stream processing –

- Suppose you have a raw weather topic and a processed weather topic in Kafka. Now, instead of spending a huge amount of effort developing an ad hoc processor, you could just plug in the Kafka Streams API here.
- In the Kafka Streams topology, we have three stream processors: The source processor that consumes raw weather streams from the raw weather topic and forwards the weather stream to the stream processor to filter the stream based on high temperature.
- Then, the filtered stream will be forwarded to the sink processor which then publishes the output to the processed weather topic.
- Concluding, this is a much simpler design than an ad hoc data processor, especially if you have many different topics to be processed.



Summary: -

- Kafka Streams API is a simple client library to help data engineers with data processing in event streaming pipelines.
- A stream processor receives, transforms, and forwards the streams. Kafka Streams API is based on a computational graph called a stream processing topology.
- And in the topology, each node is a stream processor, while edges are the I/O streams.
- And finally, in this topology, we find two special types of processors: The source processor and the sink processor.

Kafka Python Client

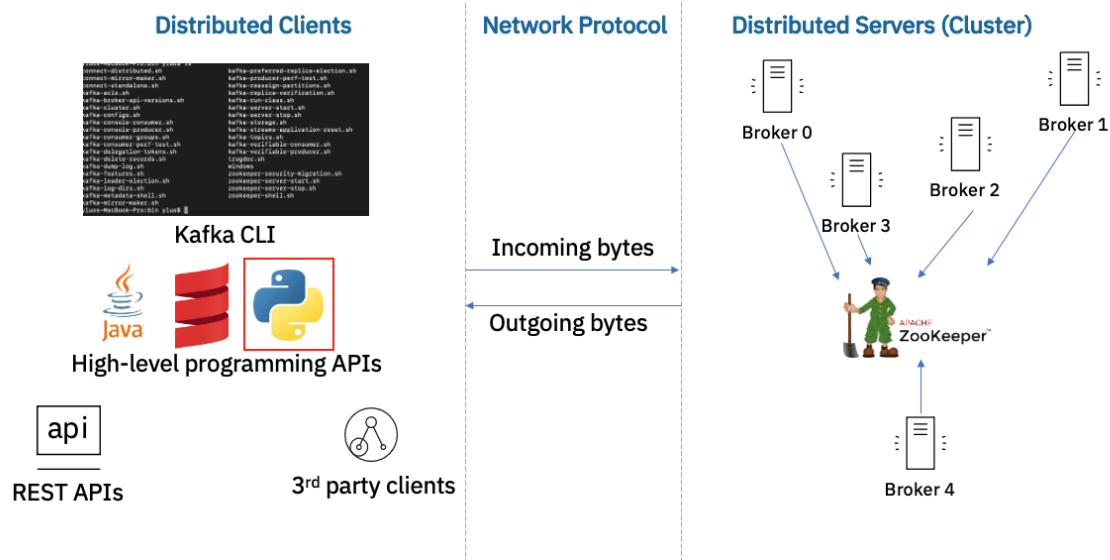
Kafka Python Client

Objectives

After reading this article, you will be able to:

- List the common Apache Kafka clients
- Use `kafka-python` to interact with Kafka server in Python

Apache Kafka Clients



Kafka has a distributed client-server architecture. For the server side, Kafka is a cluster with many associated servers called broker, acting as the event broker to receive, store, and distribute events. All those brokers are managed by another distributed system called ZooKeeper to ensure all brokers work in an efficient and collaborative way. Kafka uses a TCP based network communication protocol to exchange data between clients and servers

For the client side, Kafka provides different types of clients such as:

- Kafka CLI, which is a collection of shell scripts to communicate with a Kafka server
- Many high-level programming APIs such as Python, Java, and Scala
- REST APIs
- Specific 3rd party clients made by the Kafka community

You can choose different clients based on your requirements. In this reading, we will be focusing on a Kafka Python client

called `kafka-python`

Note: Code snippets provided in this article are just for your reference but not the complete working code.

`kafka-python` package

`kafka-python` is a Python client for the Apache Kafka distributed stream processing system, which aims

to provide similar functionalities as the main Kafka Java client.

With `kafka-python`, you can easily interact with your Kafka server such as managing topics, publish, and consume messages in Python programming language.

Install `kafka-python`

Install `kafka-python` is similar to other regular Python packages:

1. `pip install kafka-python`

Next, let's see the use cases of the main functions provided by the `kafka-python` package.

`KafkaAdminClient` Class

The main purpose of `KafkaAdminClient` class is to enable fundamental administrative management operations

on kafka server such as creating/deleting topic, retrieving, and updating topic configurations and so on.

Let's check some concrete code examples:

Create a `KafkaAdminClient` object

To use `KafkaAdminClient`, we first need to define and create a `KafkaAdminClient` object:

1. `admin_client = KafkaAdminClient(bootstrap_servers="localhost:9092", client_id='test')`

`bootstrap_servers="localhost:9092"` argument specifies the host/IP and port that the consumer should contact to bootstrap initial cluster metadata

- `client_id` specifies an id of current admin client

Create new topics

Next, the most common usage of `admin_client` is managing topics such as creating and deleting topics.

To create new topics, we first need to define an empty topic list:

1. `topic_list = []`

Then we use the `NewTopic` class to create a topic with name equals `bankbranch`, partition nums equals to 2, and replication factor equals to 1.

1. `new_topic = NewTopic(name="bankbranch", num_partitions=2, replication_factor=1)`
2. `topic_list.append(new_topic)`

At last, we can use `create_topics(...)` method to create new topics:

1. `admin_client.create_topics(new_topics=topic_list)`

Above `create` topic operation is equivalent to using `kafka-topics.sh --topic` in Kafka CLI client:

1. `"kafka-topics.sh --bootstrap-server localhost:9092 --create --topic bankbranch --partitions 2 --replication_factor 1"`

Describe a topic

Once new topics are created, we can easily check its configuration details using `describe_configs()` method

1. `configs = admin_client.describe_configs(`
2. `config_resources=[ConfigResource(ConfigResourceType.TOPIC, "bankbranch")])`

Above `describe` topic operation is equivalent to using `kafka-topics.sh --describe` in Kafka CLI client:

1. `kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic bankbranch`

KafkaProducer

Now we have a new `bankbranch` topic created, we can start produce messages to the topic.

For `kafka-python`, we will use `KafkaProducer` class to produce messages.

Since many real-world message values are in the format of JSON, we will show you how to publish JSON messages as an example.

First, let's define and create a `KafkaProducer`

1. `producer = KafkaProducer(value_serializer=lambda v: json.dumps(v).encode('utf-8'))`

Since Kafka produces and consumes messages in raw bytes, we need to encode our JSON messages and serialize them into bytes.

For the `value_serializer` argument, we define a lambda function to take a Python dict/list object and serialize it into bytes.

Then, with the `KafkaProducer` created, we can use it to produce two ATM transaction messages in JSON format as follows:

1. `producer.send("bankbranch", {'atmid':1, 'transid':100})`
1. `producer.send("bankbranch", {'atmid':2, 'transid':101})`

The first argument specifies the topic `bankbranch` to be sent, and the second argument represents the message value in a Python dict format and will be serialized into bytes.

The above producing message operation is equivalent to using `kafka-console-producer.sh --topic` in Kafka CLI client:

1. `kafka-console-producer.sh --bootstrap-server localhost:9092 --topic bankbranch`

KafkaConsumer

In the previous step, we published two JSON messages. Now we can use the `KafkaConsumer` class to consume them.

We just need to define and create a `KafkaConsumer` subscribing to the topic `bankbranch`:

1. `consumer = KafkaConsumer('bankbranch')`

Once the consumer is created, it will receive all available messages from the topic `bankbranch`. Then we can iterate and print them with the following code snippet:

1. `for msg in consumer:`
2. `print(msg.value.decode("utf-8"))`

The above consuming message operation is equivalent to using `kafka-console-consumer.sh -`
`-topic` in Kafka CLI client:

1. `kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic bankbranch`

Summary & Highlights: -

- An event stream represents entities' status updates over time
- The main components of an ESP are Event broker, Event storage, Analytic, and Query Engine
- Apache Kafka is a very popular open-source ESP
- Popular Kafka service providers include Confluent Cloud, IBM Event Stream, and Amazon MSK
- The core components of Kafka are brokers, topics, partitions, replications, producers, and consumers
- The Kafka-console-consumer manages consumers
- Kafka Streams API is a simple client library supporting you with data processing in event streaming pipelines
- A stream processor receives, transforms, and forwards the processed stream
- Kafka Streams API uses a computational graph
- There are two special types of processors in the topology: The source processor and the sink processor