Principles and Practices for Building Effective Data Pipelines

Learning objectives

By the end of this topic, you should be able to:

- define a data pipeline and discuss its role in moving data;
- describe principles and practices for constructing and maintaining data pipelines; and
- list available tools to implement data pipelines.

Contents

1. ETL orchestration and data pipelining

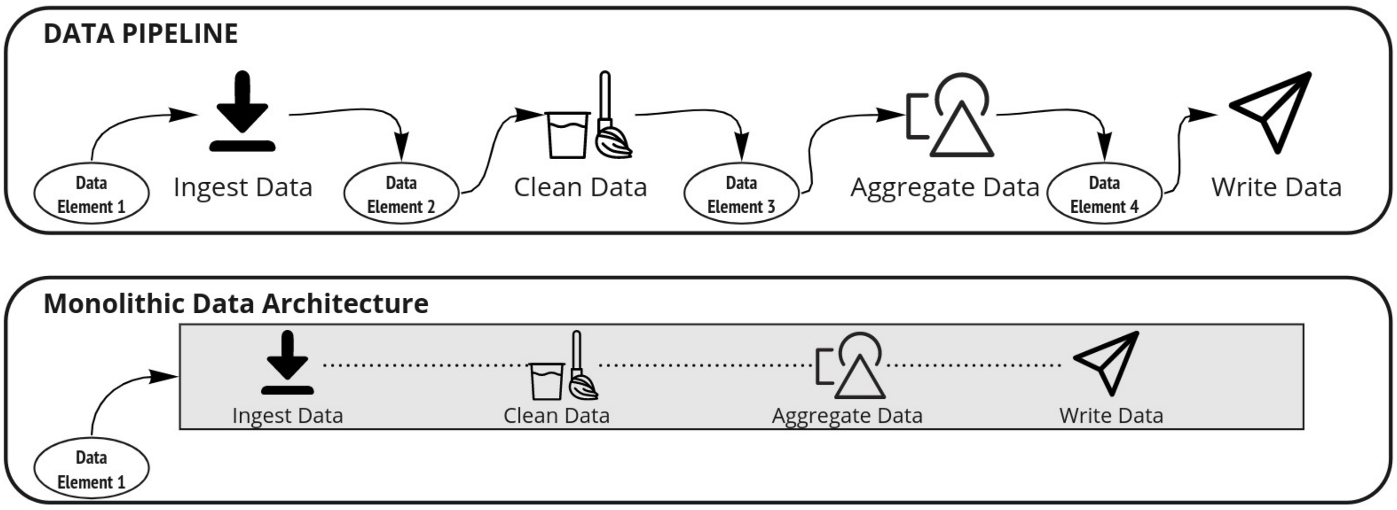

Figure 1: Data pipelines provide greater throughput by processing more data elements per unit time. Here, four times as many data elements can be processed than with a single monolithic application.

Before the industrial revolution, most manufacturing processes had an individual person responsible for the end-to-end production and assembly of a product. Innovations at the Ford Motor Company changed this model, and in 1913 the first moving assembly lines were constructed. In the Ford assembly line, a car moved between workers for assembly, allowing individuals with specialised knowledge of one or two tasks to be constantly engaged in work. This increased throughput and reduced both variations in quality and numbers of defects in manufactured vehicles.

Think of a data pipeline as an assembly line of individual data processing units. These units are like the skilled individuals in the vehicle manufacturing process, arranged into a sequence to perform data transformation. As such, a data pipeline transforms an input data element (file, message, image, table, etc.) into the desired output element. And, as with the move from individual workers to production lines, data pipelines introduce several efficiencies over monolithic data transformation processes:

- **Separation of responsibility:** by splitting a pipeline into multiple independent tasks, we know which task is responsible for producing a specific transformation. This increases our ability to localise and correct bugs in our system when they arise.
- **Higher throughput:** by having multiple tasks independently processing single data elements, we can increase the effective throughput of an entire data pipeline. This principle is illustrated in *Figure 1*, where a data pipeline with four tasks is compared to a monolithic application producing the same output data elements. The four tasks of the pipeline can effectively process four data elements at a given instant, whereas the monolithic application can only process one data element at a time. Using independent tasks also enables *parallelisation* of a data pipeline, so computationally intensive tasks can be split into multiple worker instances, further increasing the pipeline's throughput.
- **Built-in logic:** in a monolithic application, we are often required to define logic for data transformation and storage at a low level (embedded within data handling code). Using task stages in a pipeline enables us to use such logic at a higher level *between* tasks with conditionals and rerouting switches. This allows the pipeline to select which processing tasks are used based upon properties such as input values or parameters. Being able to disentangle this logic from the processing tasks means we have increased modularity, isolation of functionality, and improved pipeline observability.
- **Orchestration and automation:** modularity between tasks also provides a way for us to orchestrate effective runs in our data pipelines. By defining the dependence between two or more tasks, we can allow the successful execution of one task to trigger the next. This mechanism can be useful for complex pipelines, where one task might depend on the completion of multiple, asynchronous upstream tasks. Unlike a monolithic application, where complex logic would need to be included to coordinate this step's execution, a data pipeline can run a simple check to ensure all upstream tasks are complete. To use this orchestration to also enable automation within a pipeline, we can simply set an initial task to be triggered by either a predefined schedule or the presence of new source data.
- **Maintainability and reproducibility:** using defined tasks improves our ability to maintain and version our pipeline as we can isolate any changes to our codebase based on the tasks affected. This provides guarantees for a pipeline's reproducibility and automation, for example, monitoring if a given dataset passing through the pipeline produces an identical output each time.

  â„¹ï¸ **Did you know?**

  Monolithic applications arise when code and software components are combined into a single (large) application. In the case of data pipelines, a monolithic application could contain the authorisation, ingestion, cleaning, aggregation, writing, and integration layers of a pipeline â€“ all included within a single script.

2. What does a data pipeline look like?

A data pipeline is a vessel that strings together the various transformations that we want to perform on an ingested dataset:

- Inputs;
- Specific steps to be performed;
- Coordination between these steps;
- Code to apply transformations; and
- Scheduling.

We won't deal with the code used to transform data here, but rather focus on how we bring all of these elements together. To guide our understanding, let's consider an example that we'll repeatedly refer to throughout this train:

> Imagine you're working for a company that sells stationery. As a lead data engineer, you're tasked with setting up a data pipeline to process transactional data for data scientists and business analysts. The data is already available in a data lake, ready for the pipeline to ingest. Databases have also been specified for the storage of processed data. You're tasked with performing quality control with a pipeline, joining the data onto metadata required for analysis, and outputting the data to either business users or data scientists.

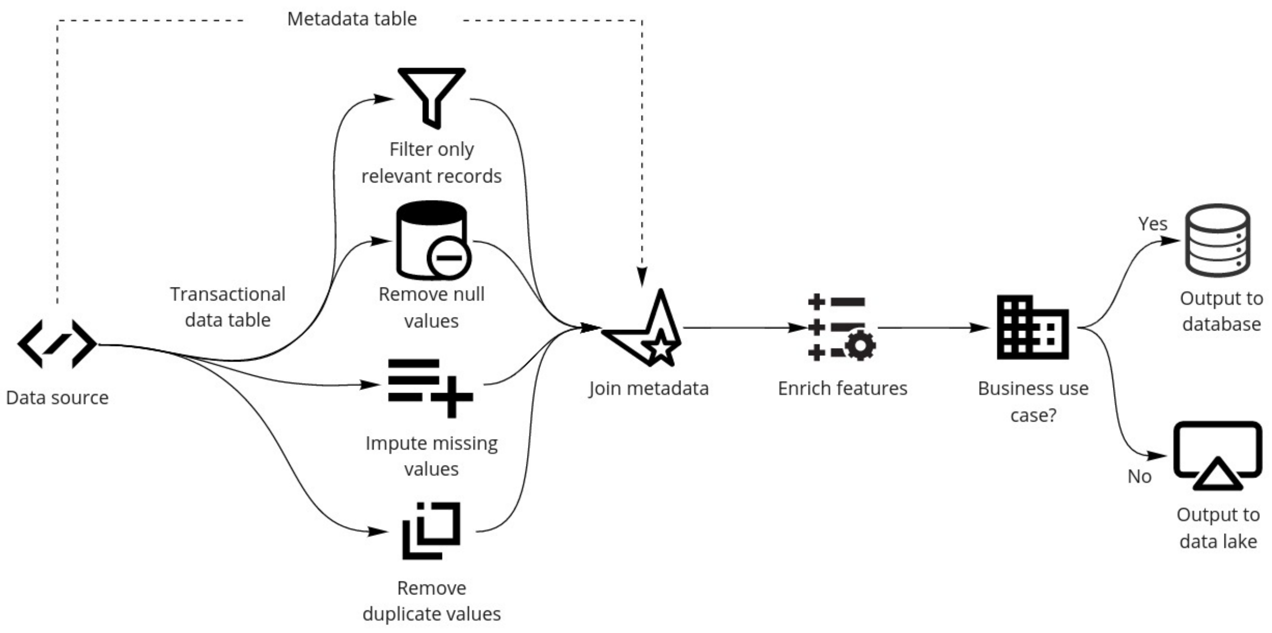

Figure 2: Data flow diagram of a basic data pipeline for transforming and transferring data in our stationery company example. The pipeline becomes active when input data is presented. Next are four parallel data transformations, each one typically a piece of code or individual notebook. The cleaned data is joined onto metadata in another piece of code, followed by stages of feature enrichment and conditional output into either a

data lake for data science or a data warehouse for business analysts.

Discussion

Let's look at how this proposed design aligns with the characteristics and advantages of data pipelines introduced in Section 1.

- Each of the transformation steps represents a piece of code that should be isolated or organised into a notebook or module. This ensures **separation of responsibility** between the developed tasks. Thus, if the input data changes unexpectedly or there is a scenario that we have not planned for in the code, we can quickly isolate the concern and make the necessary changes.
- With independent steps, we can have concurrent runs of the pipeline depending on its underlying architecture. For example, while performing the feature enrichment task in one run, we can simultaneously initiate the cleaning task of a second run, thus **increasing throughput**. Further, if our pipeline is run on a distributed architecture, we can have multiple runs in a parallel or staggered fashion.
- We **build logic** into our data pipeline at an inter-task level by branching out during the cleaning of our data and by having a *conditional* for outputting the data either to a database (possibly a data warehouse) or a data lake. This conditional could check if the data leaving the feature enrichment stage is aggregated, to determine where it should be sent.

  â„¹ï¸ **Did you know?**

  Conditionals are commands that deal with decisions. A conditional chooses between code blocks to execute based on a statement that evaluates to either true or false.

- Within our pipeline, each task is sequentially dependent on the previous (upstream) tasks. We can only start the cleaning stage once the data source becomes available. Only once all of the cleaning steps are complete can we start joining onto the metadata, and so on. This means that the **orchestration** of this pipeline is entirely dependent on an event (the availability of the initial data), allowing the pipeline to be fully **automated**. We could also increase the complexity of orchestration by using a trigger to orchestrate the latter part of the processing pipeline whenever there is any change to the metadata.
- We commit the pipeline to version control and apply best practices in software development for code versioning and deployment. This action means that the pipeline's behaviour is consistent unless changes are made to the code. Rerunning the pipeline with two identical datasets should produce identical outputs for full **reproducibility**.

3. Data pipeline principles

Using data pipelines is advantageous. However, not all data pipelines are created equal, and it's all too easy to write a pipeline that increases complexity, slows down processing, or simply makes an application unusable.

Let's consider the principles and practices for writing effective data pipelines. These principles are generic, that is, agnostic to the specific project or underlying technology stack you may be working with. Applying these principles will ensure that pipelines are robust, secure, and scalable. Pipelines should also be platform agnostic, in other words, data pipelines implemented as python scripts and scheduled via shell scripts should be comparably robust to pipelines implemented using the solutions from a cloud provider.

For each principle, there are one or more corresponding best practices that we can implement in our data pipelines. These describe how we apply the principles to our system, and they represent the actions we can take based on sound theory.

3.1. Principle 1: Data pipelines should adapt to the data sources they ingest

Data sets come in various shapes, sizes, and formats. During the design phase of a pipeline, we don't always have all of the *non-functional* requirements for the system. These include the volume, variety, or velocity of data that must be processed. As such, pipelines should be designed to adjust to changes in volume, variety, and velocity.

Principle 1: Associated practices

- Design for any data source and type;
- Design for changes in data volume, variety, and velocity; and
- Enable high scalability.

  â„¹ï¸ **Did you know?**

  Non-functional requirements (NFR) are specifications that describe the characteristics that a system should have to enhance its functionality. For example, we may want our data pipeline to ensure data integrity, be adaptable, stable, and scalable, and provide sufficient throughput. All these requirements are complementary to the functional goal of our pipeline to move and transform data from one point to another.

3.2. Principle 2: Data pipelines should be automated

A fully automated data pipeline allows us to do the following on a regular schedule, or in response to an external event, without human intervention:

- Extract data from a source system;
- Clean the data;
- Perform quality control;
- Transform the data in various ways;
- Join the data to other data sources; and
- Load the data to a destination.

Having these elements automated means that we will have better data management and auditability, improved business intelligence, increased data utilisation and mobility, and it will enable the ability to capture real-time insights. As we progress our data-driven journey we want to have more and more up-to-date data, eventually ending in streaming data into our environment in real-time. Automated data pipelines will allow us to gradually increase the velocity of the incoming data, with the data pipeline just adapting to the change in load.

Principle 2: Associated practices

- Configure triggers and scheduling.

3.3. Principle 3: Data pipelines should be robust and observable

Data pipelines should not fail. However, when they do, we have to be able to find and isolate the failure(s) quickly. Testing and monitoring should allow us to rapidly correct errors and patch our production systems.

Principle 3: Associated practices

- Establish inherent data quality logging and testing;
- Perform validation and testing; and
- Configure monitoring and alerting.

4. Data pipeline practices

We've discussed the principles that characterise performant and functional data pipelines. Now, let's look at some practices that benefit their design and implementation. As before, note that only once we have the correct practices in place will we consider which tools or frameworks to use to solve the problem at hand.

Within each of the subsections below is a description of the practice, followed by a grounded explanation of how it relates to the example of our stationery business data pipeline.

4.1. Practice 1: Design for changes in data volume, variety, and velocity

Data has to flow into a central location to be used within our applications. It's important to note, however, that data sources are not created equal. We can have batch processes, streaming workloads, or manual ingestion of data. Each of these settings has unique considerations.

> **Example**
>
> Think about our stationery retailer example. We may need to implement a solution where the company only receives data once a day through a batch process. However, we have to anticipate future company growth and build our pipeline to be compatible with streaming data to provide data and insights in real-time.

4.2. Practice 2: Design to accommodate changes in data sources and types

We have to consider the structure, source, and type of incoming data, both now and in the future. These attributes, along with the factors in the previous principle, will influence the tools and degree of task modularity that we select for our processing pipeline.

> **Example**
>
> The competitive nature of the retail sector sees companies continuously changing their systems and processes. In our stationery company, such a change could be the organisation upgrading its point-of-sale systems to make customer transactions more efficient. This could completely alter the format and structure of incoming data.
>
> Consider a situation where the new system only supports the transmission of XML data as opposed to CSVs. Moreover, the data fields incoming from the new system could have significant differences when compared to the old system, such as new attributes being introduced, attributes being deprecated, or attributes being referenced by different names. The linking job that dealt with the CSV data would need to be adapted to support the ingestion of XML data along with the new attributes contained in the data file.
>
> As such, we need to build our pipelines to support these types of updates efficiently and robustly. Failing to do so could cripple any business activities relying on our data. We can, again, tip our hats to modular pipeline design here, as only a single transformation task would need to be updated.

4.3. Practice 3: Ensure your pipeline can scale effectively

We can never be certain of the processing capacity required to ingest and transform our data. The volume or velocity of input data can change quickly. Similarly, we might implement a new algorithm as a step in our pipeline that requires additional computing power. In all these cases, we need to elastically scale our pipeline's resources to meet current demands. This is an area where cloud service providers thrive, with their data pipeline service offerings natively supporting automated scaling based on predefined metrics.

> **Example**
>
> For our stationery retailer, we may start with only implementing the pipeline solution for one or two branches before extending further. Once we've developed this as a proof of concept, we'd want to expand to the whole organisation. To ensure this goes according to plan, we'd have had to properly design the system initially to deal with a future increase in load.
>
> Another scenario requiring scaling arises during periods of increased demand, for example, at the start of the school year when the stores are busier with more school children buying stationery. Some of the techniques to do this include the use of auto-scaling clusters from a cloud provider or choosing tools that are resilient to increasing loads. Examples of such tools include message brokers for source data ingestion or components of streaming architectures that can catch up on a heavy workload during times of reduced demand.

4.4. Practice 4: Ensure your pipeline is automated by using triggers or scheduling

Data pipelines that are manually run are not effective. Triggers and scheduling enable automation by orchestrating pipelines based on either a schedule or an event. Event-based triggering starts a pipeline as soon as a file of a specified format arrives in storage, making sure pipelines only start once the correct data is available for processing.

> **Example**
>
> In the case of our stationery retailer, we can enable pipeline automation in several different ways depending on how the company's systems are configured. If the accounting department required a daily report of cashflows to monitor profits, a batch job could be

designed to obtain the relevant data and have it dropped into an appropriate location at the end of the business day or on request, if need be. This means we could set up an event-based trigger to run only once a new batch of data becomes available.

Alternatively, the stationery store may want to track the traffic of its e-commerce site to provide real-time price updates, product recommendations and/or online deals to its consumers. In this case, our pipeline may instead need tighter coupling with transactional data being read from a message broker topic. A regular schedule may also be helpful in a streaming scenario, initiating pipeline processing at short intervals.

4.5. Practice 5: Build data quality validation and logging into your pipeline

In traditional software systems the input to an application is generally stable, consisting of predefined user interactions. However, when working on a data pipeline, data enters our system from an external environment and is prone to variations in quality and format. We should ensure that our systems are robust to changes in incoming data quality. To do this, we can build processes to validate data quality at various stages in our pipeline and constantly log any changes in data format or quality.

Example

Returning to our example, let's imagine that at one of the company's outlets the point-of-sale system received a software patch from their vendor. This inadvertently introduced a bug and caused the data pipeline to ingest and produce erroneous results. To be able to rapidly identify the root cause, logging tasks should be incorporated at critical key points within the data pipeline. This means each modular section of the pipeline should have a task that performs validation to ensure that we have a robust pipeline. It should ideally be done during the design phase.

If we have these measures in place, how would we identify the cause of this problem? We'd check the log files produced by each section of the pipeline and look for any anomalous entries or pipeline behaviours. We can first look at the initial ingestion of the data to see if we are receiving a full set of data with no error flags. Once that is done, we can move on to check the four transformation jobs. When looking at the four transformation jobs, we notice that the job that usually removes null values from the data is barely doing any work. Upon further investigation, we find that all of the integer fields that were previously `null` values are now being recorded as`NaN`. We can now retrace our steps to see where and when the error was first introduced. In our case, the data being ingested only had `NaN` values after the patch was received. From here, we can take the following actions to keep the pipeline and point-of-sales system running:

- Implement a quick fix to flag the new`NaN` values; and
- Send these logs to the vendor and log a case to have this fixed at the root.

4.6. Practice 6: Always ensure that your pipeline is testable at multiple levels

Since data pipelines have a large number of components, small errors can go undetected and propagate through multiple tasks, causing rapid and cascading errors! To mitigate such disasters, we implement testing at multiple levels of our data pipeline. Similar to all good software applications, this testing should be done at the unit, integration, and end-to-end levels.

Example

If configured correctly, the test coverage of our stationery pipeline should be similar to any other software system that we may encounter. We should have a suite of unit tests throughout our codebase, aiming for high coverage of functions and modules. Where we have components that have interfaces in our application, we should be writing integration tests, for example, between the ingestion and processing layer or between our processing or database layer. Finally, we need to be sure that our application works as a whole. This can be done through end-to-end tests, using a defined mock dataset entering our application and making sure it exits the application in an expected format and state.

4.7. Practice 7: Configure continual monitoring and event-based alerts for your pipeline

Once we have working data pipelines, we need to ensure that they remain operational. Having a single place to monitor our pipelines, such as a dashboard, is central to good architecture. We should be able to view run duration and any change indicating server overload or increases in data volume. Individual runs should also be inspectable from the dashboard, enabling investigation of specific failed pipeline runs. To enrich a good monitoring system, we should set up alerts to pick up failed pipeline runs quickly. Alerting and monitoring are usually based on the logs generated within the system.

Example

We want to implement a suite of dashboards for the retailer's various stakeholders. At an executive level, we need a dashboard showing the performance of the overall system, as well as some key metrics of the organisation, for example, sales figures and branch performance, among others. We also want to implement a performance dashboard for the operations team that tracks data flowing through the system. Finally, we want to implement email or push notification alerts for the team in charge of maintaining the system to catch bugs in real-time.

4.8. The final system

Combining the theory from Section 2 through Section 4, we arrive at something like this:

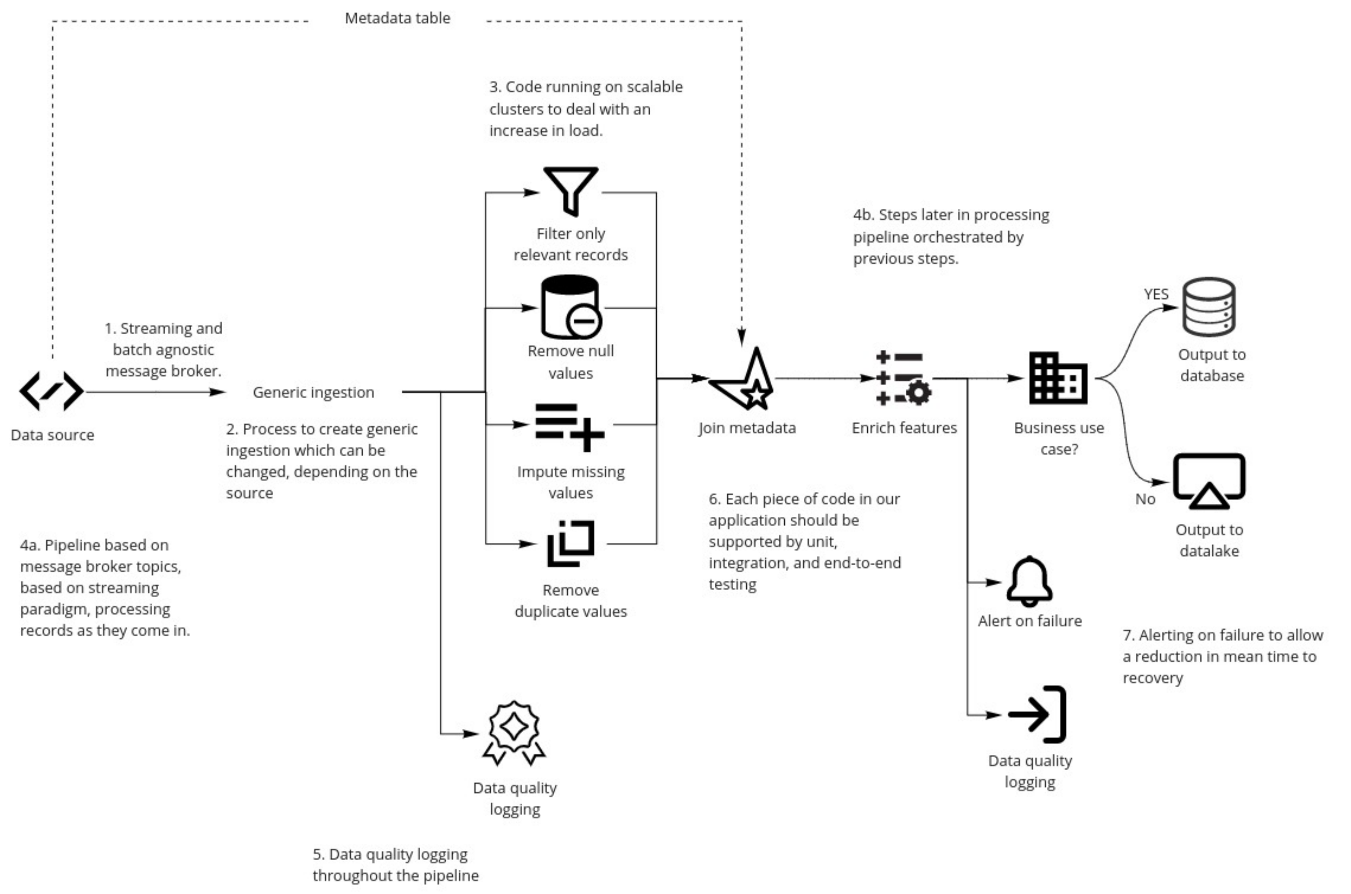
Metadata table
3. Code running on scalable clusters to deal with an increase in load.
Filter only relevant records
Remove null values
Impute missing values
Remove duplicate values
4b. Steps later in processing pipeline orchestrated by previous steps.
1. Streaming and batch agnostic message broker.
Data source
Generic ingestion
2. Process to create generic ingestion which can be changed, depending on the source
Join metadata
Enrich features
Business use case?
YES
Output to database
No
Output to datalake
6. Each piece of code in our application should be supported by unit, integration, and end-to-end testing
4a. Pipeline based on message broker topics, based on streaming paradigm, processing records as they come in.
Alert on failure
7. Alerting on failure to allow a reduction in mean time to recovery
Data quality logging
Data quality logging
5. Data quality logging throughout the pipeline

Figure 3: We improve the data pipeline from Figure 1 by considering each of the principles and practices. For data source considerations, we add in a message broker to provide resilience against changes in data velocity and variety. We use a generic ingestion notebook that can be changed whenever the data source changes. Code for transformation should be run on clusters or compute resources to allow scaling when load increases. We implement a streaming paradigm, allowing processing to happen as data arrives and subsequent steps triggered by earlier ones. Data and code are monitored and tested throughout the pipeline. Finally, we incorporate alerts based on failures.

5. Available data pipeline solutions

These principles and practices provide guidelines to build pipelines with any combination of tools. However, the increased prominence of big data in recent times has led to open-source and cloud providers providing data pipeline tools that encapsulate best practices.

5.1. Apache Airflow

Started at Airbnb in 2014, Apache Airflow is a data pipelining tool that allows workflow automation and scheduling programmatically. Airflow provides a powerful user interface, allowing deployed pipelines to be monitored at multiple levels of granularity. It draws strongly on the principles of configuration-as-code, including the full configuration of data pipelines in the code.

Interestingly, Apache Airflow is written in Python with workflows created as Python scripts. This means that developers can, in addition to specifying the workflow logic, import existing Python libraries that may be required when implementing data transformations in a pipeline.

5.2. AWS Glue

AWS Glue is a fully managed, serverless data pipeline service. Glue aims to provide the developer with a simple user interface and a low technical barrier to begin developing data pipelines. To help with this, AWS Glue includes companion features in the form of AWS Glue Data Catalog, Glue ETL Engine, and Glue Scheduler.

The Glue Data Catalog is a persistent metadata store that can index data and provides additional metadata for any dataset. The catalogue also automatically computes statistics for ingested data, enabling efficient and cost-effective queries, and it keeps an extensive history of the schema and schema changes for datasets.

AWS Glue Studio also removes the hassle of coding up the pipeline. This allows us to build data pipelines via a user interface, which it then transforms into Spark Python (PySpark) or Scala code for backend execution.

Glue also simplifies scheduling pipeline runs. We can invoke pipelines based on a schedule, an event, or on-demand. This means that Glue will take care of most of the integration for us.

5.3. Azure Data Factory

Similar to AWS Glue, Azure Data Factory provides a serverless data pipeline service, but on the Azure cloud platform. Data Factory includes a code-free user interface to specify data movement and transformation, with a range of connectors to both cloud and on-premise services. This means that with only a couple of clicks and configuration settings, we can move data between two locations in the cloud or even from an on-premise location. Data Factory also incorporates support for data transformation within Databricks, Data Lake Analytics, Machine Learning, and various other Azure services.

Finally, Data Factory also provides transparent pipeline monitoring, offering a view on pipeline runs, as well as deeper insights into components that succeed or fail during runs.

Conclusion

In this topic, we looked at what it takes to design a data pipeline that is future proof. We identified needs, highlighted principles that should be applied when designing pipelines to address the identified needs, and looked at practices that will enable you to adhere to those principles.

Finally, we provided a brief overview of the prominent data pipeline offerings currently available through open-source projects and cloud providers.

Resources

- A comprehensive guide to data pipelines
- AWS Glue 101
- What is Azure Data Factory?
- Why do we need an automated data pipeline?