Gartner.

Licensed for Distribution

This research note is restricted to the personal use of Kalani Mertyris (kalani.mertyris@covered.ca.gov).

# Data Engineering Essentials, Patterns and Best Practices

Published 27 May 2021 - ID G00741282 - 70 min read

By Sumit Pal

---

Initiatives: Data Management Solutions for Technical Professionals

With explosive growth in data generated and captured by organizations, capabilities to harness, manage and analyze data are becoming imperative. This research provides data engineering principles and best practices to help data and analytics technical professionals build data platforms.

## Overview

### Key Findings

- Data engineering has evolved beyond extraction, transformation and loading (ETL), and teams are continually required to abstract, generalize and create long-lasting solutions. This involves applying core engineering principles and design practices.
- Data pipelines are growing in size, volume and complexity, with multistage processing and dependencies between various data assets. Today, data engineering comprises 80% to 90% of the work organizations do with data.
- Data engineering and data platforms are evolving quickly with the adoption of cloud, and new best practices, open-source projects, tools and frameworks are being introduced continually. Data engineers have overextended themselves more than ever in order to execute faster, reduce cost and leverage best-of-breed technology to enable data-driven insights.

- Speed of adoption and time to market are causing organizations to often ignore data testing, data validation and data quality monitoring, and this often leads to polluted, mangled and brittle data pipelines that lack maintainability.

### Recommendations

Technical professionals responsible for building end-to-end data pipelines should:

- Develop data with a product mindset — in terms of quality, reliability, availability, interoperability and reproducibility — when developing data pipelines and data-driven systems.
- Create maintainable solutions that use core principles to guide design decisions and practices and that have the flexibility to accommodate new data sources and ever-changing requirements for data ingestion, data processing and data storage. Use software engineering principles/patterns such as single responsibility, dependency inversion, functional programming and immutability.
- Incorporate the testing of data and code, and counterattack data drift, infrastructure drift and code drift with continuous data and code testing practices. Collect data metrics to catch bad data as quickly as possible. Incorporate data observability, and consider factors to build robust data operations and to identify critical issues faster in order to debug and troubleshoot quickly.
- Build data engineering skills that complement and build on top of software engineering skills. Some of the critical data engineering skills include distributed systems architecture, databases (relational and nonrelational), data management, data modeling, programming, data analysis and preferably some domain knowledge.

## Problem Statement

Every organization is now becoming a data company. In most organizations, data engineering is lagging years behind the field of software engineering as far as building data-driven products is concerned. Data engineering remains mired in ad hoc engineering practices. At the same time, for example, building data pipelines is getting more and more difficult with the explosion of a variety of data sources, data types, data volumes, data velocity, and complexity of infrastructure and frameworks. Much time and effort is spent in provisioning and stitching infrastructure and frameworks.

In most cases, data pipelines start off simple and grow in complexity. However, changes to data are accelerating. “Data drift” is defined as unexpected, undocumented changes to data structure, semantics and infrastructure that is a result of modern data movement and architectures.

Data platforms are evolving continuously, from on-premises data lakes and enterprise data warehouses (EDWs) to public cloud services, and engineers have the onerous task for huge replatforming projects while discharging their daily responsibilities. All this is causing data engineers to spend less time architecting and designing data pipelines and more time maintaining and migrating them.

Handling a large number of heterogeneous and ever-growing data sources, with multiple specialized teams building end-to-end data products and workflows, is becoming complex and difficult to coordinate, collaborate, orchestrate and operationalize. Troubleshooting and debugging data transformation logic in a distributed environment is challenging. Building, optimizing, and ensuring reliability and availability of multiple complex interdependent data pipelines at scale is becoming very challenging. All this has caused perpetual delay in reducing and removing data debt. The data landscape is in a constant state of flux; incorporating continuous changes in the underlying technology stack is a nonstop exercise.

Organizations continuously encounter problems when reprocessing data sources and performing back-filling processes. In most cases process logic is too complex and not properly broken into modular, immutable, idempotent tasks. This causes reproducibility problems in modern data pipelines. Hadoop Distributed File System (HDFS) and object stores such as Amazon Simple Storage Service (Amazon S3) store data that cannot be updated with random writes. Datasets like logs, clickstream and event data are immutable. Building data processing tasks that are idempotent based on data that is immutable is necessary for modern data architectures and data flows, especially in today’s regulatory- and compliance-centric use cases.

As data platforms evolve from on-premises to hybrid and multicloud services, the task of replatforming the data stack becomes necessary. This causes data engineers to juggle around with multiple architecture stacks and tool combinations, making an already complex, distributed process more convoluted.

Data often exists across silos, which makes access across varied data formats and systems extremely challenging. This can slow down the building of data-driven systems that can extract insights. With the growing volume and variety of data, existing systems must be able to both adapt quickly to changing workload patterns and handle data skew and data drift.

With all the above-mentioned challenges, it becomes necessary to arm the data team with data engineering best practices to perform complex ingestion, storage and processing of data from a variety of sources across wide formats. You must also ensure fault tolerance

and availability and comply with guaranteed SLAs.

## The Gartner Approach

Before discussing the recommended Gartner approach to data engineering, we will first define data engineering and why it is needed.

### What Is Data Engineering?

Data engineering is the discipline of making data usable. The objective of the data engineering discipline is to help an organization move, store and/or transform/process data and shape it across different data systems — from operational to data lakes and data warehouses — across different data centers. The engineering aspects of this process ensures this can be done consistently, efficiently, scalably, accurately and within compliance.

An alternative definition of data engineering is lucidly explained in Data Engineering Is Critical to Driving Data and Analytics Success.

“Data engineering” is an overarching term covering data ingestion, data storage, processing, modeling, data lake and data warehouse design and implementation, building data pipelines, data integration, data testing, and data orchestration.

The goals of data engineering are:

- Onboard variety of internal and external data sources
- Decrease the time to onboard
- Ensure data quality with data validation and verification
- Develop data pipelines quickly
- Enable analytic teams to develop business logic
- Provision data infrastructure for ingest, storage and processing
- Automate, orchestrate and monitor data pipelines

### Why Data Engineering?

A few years back, data engineering implied managing data across several databases, creating SQL-based pipelines and ETL to move, load and transform data for analytics. However, an explosion of data sources, data types, data storage techniques and analytic advancements has created the desire to incorporate them into businesses for better data-driven insights.

Data almost represents an existential threat to those who don't embrace the insights it can provide, and at the same time, it represents an unparalleled opportunity for those who do. Data is an asset, and without proper data management and data engineering, it can quickly become a liability. Data management and data integration is easy when working with a handful of data sources with minimal data movement and data transformation. However, data management can quickly become complex and convoluted when working with a multitude of data sources at terabyte and petabyte scale with complex transformations. Scale makes it a sophisticated engineering discipline in its own right.

As the number of data systems in an organization increases, along with the ever-growing demand to use those data systems to harness insights, an engineering approach is needed. Use this approach to manage the growth of data systems, the number of data engineers and the number of data pipelines being built.

Professor Michael I. Jordan, one of the most influential people in machine learning and artificial intelligence, makes a strong case for the need for a "field of data engineering." It is similar to the fields of chemical engineering, electrical engineering and civil engineering, which integrate hard engineering concepts and principles with theoretical and practical aspects.

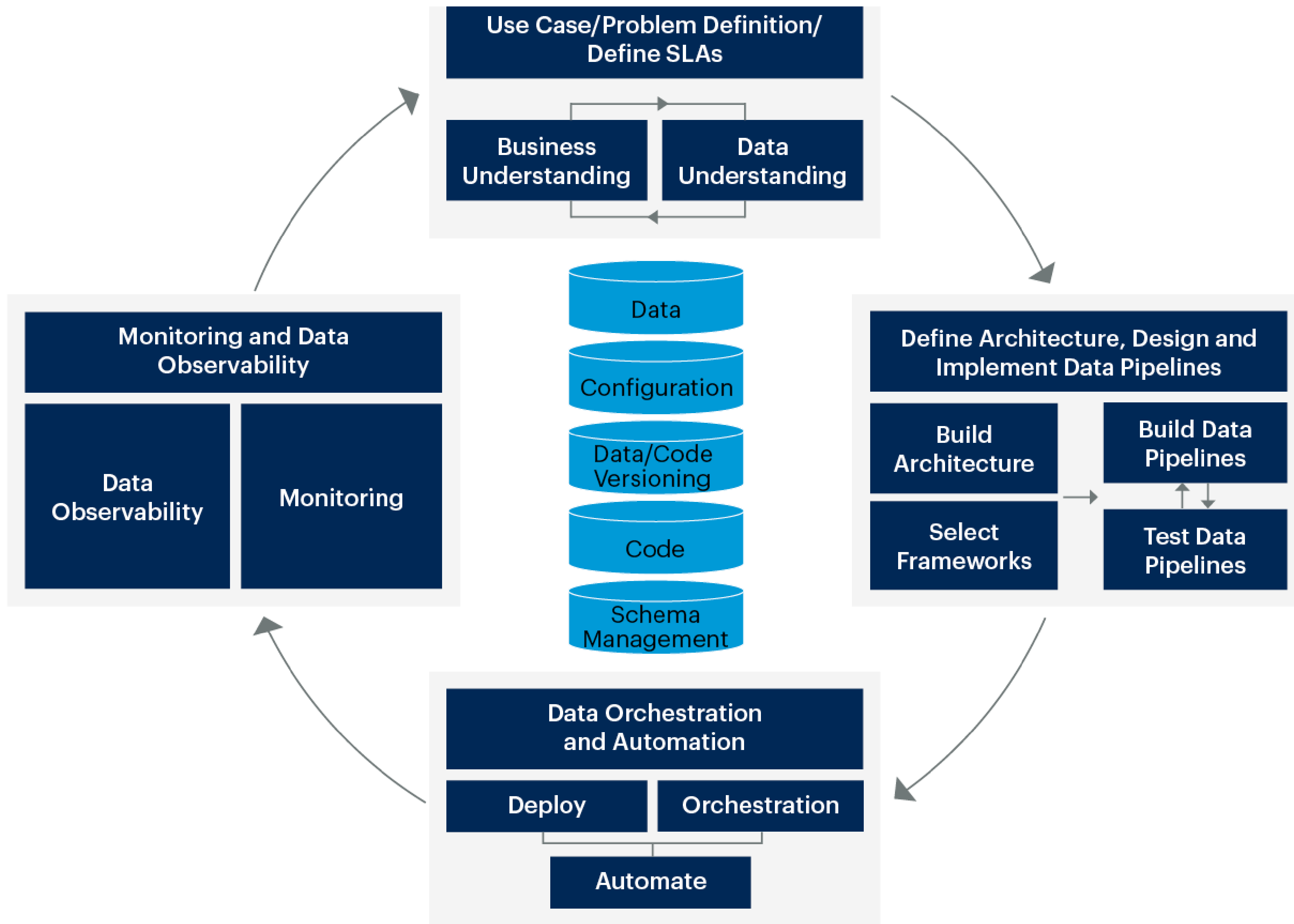
## Gartner's Approach

Gartner's recommended approach applies to organizations looking to modernize their current data engineering management for building and managing their data platforms. Organizations starting small that have a bigger vision for building an end-to-end data platform, where a large number of data sources can be easily onboarded and leveraged for data-driven insights, would benefit greatly from the guidance in this research.

An ideal data engineering life cycle is shown in Figure 1.

Figure 1: Data Engineering Life Cycle

**Data Engineering Life Cycle**




Source: Gartner
741282_C

Typically, organizations have been following a traditional way of developing data architectures — as shown in the Traditional column in Table 1. For organizations starting their data platform architecture and for organizations looking to scale their current data architectures, Gartner highly recommends an approach as shown in the Modern column.

**Table 1: Traditional and Modern Data Engineering**

| *Features* | *Traditional* | *Modern* |
|---|---|---|
| Code Generation | Manual | Templatized |
| Programming Model | Imperative | Declarative |
| Data Ingest/Cleaning | Manual | Event-Driven |
| Code and Data Version Management | Code Versioning Only | All Forms of Versioning — Code, Data, Model, Analytics as Code, Infrastructure as Config |
| Deployments | No Automation | Automated |
| Development | Partially Continuous Integration and Continuous Delivery (CI/CD) — Applied only to code | Full CI/CD — Managing data processes and data engineering with CI/CD |

| ***Features*** | ***Traditional*** | ***Modern*** |
|---|---|---|
| Testing | Low Testing | CT (Continuous Testing) |
| Data Orchestration | Minimal | End to End |
| Data Observability/Monitoring | Low — Only monitoring of key metrics and manual feedback process | Imperative — Automated capturing of key metrics and use of advanced analytics to recommend changes to improve pipeline throughput, latency, etc. |
| Data Platforms | On-Premises, Hybrid | Hybrid, Multicloud, Intercloud |
| Data Processing | Batch | Batch and Event-Driven |
| Data Access | Point to Point | Decoupled |
| Data Architectures | EDW, Data Lakes | Data Lake, Data Hub, Data Fabric, Lakehouse |
| Data Processing | ETL | Extraction, Loading and Transformation (ELT), Serverless |

Source: Gartner (May 2021)

Different organizations are at different maturity levels for incorporating data engineering practices. This has been well-described and well-discussed in How to Build a Data Engineering Practice That Delivers Great Consumer Experiences.

Keeping with the data engineering life cycle and a modern approach to data engineering, Gartner recommends a five-step approach to implementing data engineering, as shown in Figure 2.

Figure 2: Gartner Recommended Approach

## Gartner Recommended Approach

Prework

Business Understanding

Problem Definition

Data Understanding

Use Cases, NFRs and Data Sources

- Build use cases
- Finalize NFRs
- Outline data sources

Step 1

Build Architecture and Choose Tools and Frameworks

Step 2

Build Data Pipelines and Follow Software Engineering Principles

Step 3

Data Validation and Testing

Step 4

Orchestrate and Automate Data Pipelines

Source: Gartner
741282_C

Gartner

Gartner's five-step approach to implementing data engineering:

- Prework: Build Use Case; Outline Data Sources, Storage and Processing Requirements; and Define SLAs and NFRs
- Step 1: Lay Foundation — Build Architecture and Choose Tools and Frameworks
- Step 2: Build Data Pipelines and Follow Software Engineering Principles
- Step 3: Build Code and Data Testing, and Ensure Data Quality-Driven Approach
- Step 4: Orchestrate and Automate Data Pipelines
- Step 5: Implement Data Monitoring and Data Observability

## The Guidance Framework

### Prework: Build Use Case; Outline Data Sources, Storage and Processing Requirements; and Define SLAs and NFRs

Do you need data engineering?

Before embarking on the journey to data engineering, the most important thing to figure out is: "Does the organization need a data-engineering-driven approach to building data platforms?" If the organization is small and there is a limited amount of (1) data sources

(less than five) and (2) data processing and data analytics in the roadmap, then there is no need to have a data-engineering-driven approach to building the data platform.

However, if the organization has a multitude of data sources and is struggling to bring them together to build a data-driven organization with analytics on the roadmap, you should start with an engineering mindset to work with data.

The first step in the process — the Prework — sets the scope and goals. The tasks during this stage are outlined in the following subsection. The scope of a piece of the data engineering work should be clearly defined because loosely defined scopes often result in scope creep.

### Tasks

- Identify the business problem, hash out the top five to 10 use cases, prioritize them and define business values.
- Take the top five to 10 use cases and analyze the following for each use case:
  - Identify critical, must-have features.
  - Know what data you have and where it is. Outline data source details — names, locations, formats, type and volume.
  - Outline data storage requirements — Distributed File Service (DFS), object store, other nonrelational data stores and data warehouses (DWs).
  - Outline data processing requirements — batch, stream, incremental batch, etc.
  - Outline the nonfunctional requirements — scalability, fault tolerance, data loss, data delivery guarantees, and latency and throughput requirements.
  - Outline metadata requirements — metadata, schema, configuration, artifact repository, code and data versioning systems, data quality, and data testing.
  - Outline DevOps and DataOps requirements — CI/CD, orchestration, monitoring, observability, security and governance.
  - Outline SLAs/SLOs, even though they may be ballpark numbers.

- Define “success” criteria by choosing metrics that can help identify the success for the data engineering process.
- Develop status alert updates as pipeline execution proceeds.

People involved during this process should include critical decision makers and stakeholders, a data architect, a domain expert and a systems architect. Ideally, at the end of this step, you should have the relevant information for all the columns outlined in Table 2 ready for the project. This table outlines the schema or metadata information that needs to be captured by the organization for each use case.

For each use case, identify the data sources. This table only outlines some of the aspects and requires changes, modifications and additions based on an organization’s requirements and data projects. Contents of this table should be built upon iteratively.

**Table 2: Identify Data Sources for the Use Cases**

| ***Use Case Name:*** ↓ | | |
|---|---|---|
| List of Data Sources for This Use Case: | For Each Data Source, Capture Details: | ■ Name<br>■ Location — On-Premises, Cloud<br>■ Connectivity Details<br>■ Data Type<br>■ Ingest Cadence<br>■ Data SLAs — Max. Size, Min. Size, Max. Rows, etc.<br>■ Preconditions Before Ingest<br>■ Postconditions Before Ingest<br>■ Data Schema<br>■ Monitoring Metrics for Data Ingest |

***Use Case Name:*** ↓

Source: Gartner (May 2021)

## Step 1: Lay the Foundation — Build Architecture and Choose Tools and Frameworks

Data platforms are in a state of continuous innovation and evolution. Data architects and data engineers often have the onerous task for platforming and replatforming their architectures as data platforms evolve and move across data center boundaries. Often, they have several architectural choices with different cost, effort and skill trade-offs across the different NFRs. In many cases, the technically best option may not necessarily be the most appropriate approach.

Remember, time and effort spent developing any data engineering platform and application is typically very small compared with the time and effort needed to maintain the application in production. So while developing the architecture, it is imperative to think not just about the development, but also about operations and maintenance issues — aka "handling second-day problems."

Some of the example principles when building a data architecture for end-to-end data pipelines are:

- Build architectural agility, and build in framework-agnostic solutions. Decision-making processes around tools should be driven by the data engineering needs, scalability and latency requirements of the data-driven products. Ideally, look to use heterogeneous frameworks for data processing (batch, streaming, machine learning and graph processing) — Apache Spark, Apache Flink and Apache Beam.
- Decouple data pipelines from infrastructure. Modern data architectures across different deployment scenarios are a mashup of vendor products, open-source frameworks and tools, and managed service components across data ingest, data storage and analytics. These evolve at their own cadence with bug fixes, new releases and patches. To enable agility to add, upgrade or move away from the frameworks, components should be loosely coupled without having to reimplement foundational pieces of infrastructure. Some ways of achieving this include:
  - Create a configuration-driven approach to building end-to-end data pipelines, and allow for customization across the different components of the pipeline. Create reusable abstract functions/steps that can accept parameters. Decouple development and

deployment using metadata and configuration-driven architecture.

  - Build an abstraction layer. The more parts of a data engineering pipeline that can be abstracted out, the easier it is to add, replace and test components in a data pipeline. Rather than thinking just about the data stack in terms of tools, build abstractions and then slot tools based on appropriate functionality.

- Enable refactoring common elements into libraries to support reuse, and improve time to market. Remove incidental complexities by providing standardized, reusable solutions.
- Automate and avoid manual and repetitive tasks by using declarative semantics and workflows.
- Own the entire data engineering life cycle; don't ignore data testing, data orchestration and data operationalization.
- Ensure workloads can run on any environment with a containerized and Kubernetes-based approach.
- Stick with native languages, which are always helpful because platforms are not programming-languages-agnostic.
- Invest in a centralized logging system because logging is the core of a distributed system.
- Don't compromise on repeatability and traceability. Automation is a necessity for data engineering organizations now. Ideally, automation should be done at each of the layers. This allows for seamless repeatability, reproducibility and rollbacks.
- Create well-defined interfaces at handoff points
- Find a middle ground when creating abstracted tools that data consumers with no technical knowledge can use, but that require a significant engineering effort to create. Avoid providing tools that require so much technical knowledge that training data consumers to use them is not viable.
- Apply where possible a serverless-based approach — where startup latency and operational costs permit — to developing data systems. Organizations should not expect data engineers to invest time to manage and deploy infrastructure. It is important to apply DataOps, but it's more important to know how to avoid doing it.

## How to Start With the Architecture and Tool Selection

This section outlines some of the initial steps organizations should take to kick-start their data engineering efforts. This is divided into macro decisions and micro decisions.

Some of the macro-level decisions:

- Select the data stack — frameworks, vendors, tools, libraries and so on — required based on the business question and requirements. Modern data stacks get defined by themselves with well-known patterns for specific use cases. Although new tools and frameworks are continuously emerging for data ingestion and data processing, most of the existing tools and frameworks have been used successfully across different domains and organizations.

- Identify the type of data ingestion patterns required for your data sources. They would be batch, streaming or CDC (Change Data Capture). It is also required to understand the sources of the data — whether it is coming from external or internal sources — and if the data transfer would happen over the wide-area network (WAN).

- Identify the type of data processing required on these data sources to enable the use cases. These could be batch processing, stream processing, machine learning training or ad hoc querying. If all types of processing are required, choose frameworks that can handle everything with acceptable limitations — for example, Spark. Spark has a very high adoption rate primarily because organizations need to upskill only once to understand the framework. The same framework can be used for different workloads with lower learning curves and deployment challenges. Spark architecture and details are discussed extensively in An Introduction to and Evaluation of Apache Spark for Modern Data Architectures.

- For streaming workloads, clearly identify the SLAs, incoming throughputs, storage requirements, delivery semantics and processing requirements. For more details, see Stream Processing: The New Data Processing Paradigm. If organizations need a distributed messaging platform, the choice is almost unanimously Apache Kafka. Kafka is discussed in great detail in Streaming Architectures With Kafka.

- Identify the data storage requirements — both as the data is ingested into the stack and when it is stored after processing is done. In most cases, the incoming data lands on a data lake and is then processed before being stored for downstream consumption. Decide on the type of storage required for data ingestion and for data consumption postprocessing. Ensure to provision the storage and even better if the storage can autoscale with the incoming data and the data produced post processing. Storage requirements in terms of volume, access controls and so on should be configuration-driven and part of the operationalization framework. Also,

provision enough storage for temporary data produced during execution of data pipelines for checkpointing and caching. Ensure data pipelines clean up temporary stores once data processing is complete.

Once the high-level decisions and approaches are made, it is time to dive down into more details. Some of the micro-level decisions and work that needs to be performed during this step include:

- Define performance metrics for each of the layers — data ingestion, data storage and data processing — including acceptable error thresholds.
- Define error-handling and exception-handling strategies and ensure this is uniformly implemented across the stack from ingestion to storage and processing. For example, what is to be done with bad records ingested? What needs to be done when incoming data is corrupted? Should the ingestion process be restarted or should it continue? Same goes for processing. What happens when processing fails in between stages or tasks? Depending on the SLAs, should the system be architected to restart from the beginning, or should it continue from the failure point? How often checkpointing and save points need to happen should also be strategized.
- Decide on source code control system requirements such as Git, Bitbucket, etc.
- If a data versioning system is required, decide on requirements and tools.
- Decide where the data engineering platform will be developed and where it is to be deployed. Will it support a combination of different environments — on-premises, cloud and hybrid?
- Decide configuration management tools. Most data pipelines use a plethora of components and frameworks. Each of them has out—of-the-box settings that are suboptimal in most cases. Configuration management across these tools can easily get out of control, especially with multiple environments — development, quality assurance (QA), staging and production. Gartner recommends adopting a configuration management system that tracks the configurations used across each of the frameworks and their respective environments. Configuration should be centralized as far upstream as possible and integrated with containerization tools. Configuration flexibility should be determined based on the anticipated frequency of user changes balanced with pipeline stability needs, and never embed configuration within code or scripts. Not all tools and frameworks have the right tooling. For example, managing Kafka topic configuration has been a missing piece in the Apache Kafka ecosystem.

- Set standards for the development environment and the QA environment across the stack. Ensure developers reuse existing data libraries or build reusable modules to prevent them from doing data transformation in a different way, which can lead to confusion and often inefficient code — both at development time and execution time. When building a data platform designed to grow and evolve, agreeing on schema management, data format, and naming conventions and principles is paramount. Proper organization of the project structure is key to allowing engineers to jump from one project to another and to implement a new feature or a fix without getting tangled up by the complexity of the project itself.

- Decide on data formats. With a data ecosystem, every system has to talk to every other system. This can easily result in impedance mismatches between different systems, causing the integrations to become hard and unmanageable. Numerous data formats exist to represent data. Each of these formats are best-fit for certain use cases as well as storage and processing architectures. Some of these data formats are open standards while others are very proprietary. Not using the right data formats and schema when building data pipelines can result in unoptimized end-to-end data pipelines with high latency and low throughput. Handling high data volume is effective on distributed systems with data formats that are versatile, compact, fast and easy to serialize, deserialize and store efficiently both on disk and in memory. There is no one data format that fits across all use cases and requirements. Select a data format that is best-fit for the use case, and ensure it is compatible across other formats that are used in the pipeline. Otherwise, select one uniform data format for your data pipeline. This can eliminate incompatibilities and multiple back-and-forth serialization and deserialization across formats at the cost of some performance degradation.

- Validate the architecture. Gartner recommends commissioning a thorough external review of your data architecture. Validate architecture against the reference architectures and best practices of the cloud vendors. The following should be validated at a minimum:

  - **Security:** Involve security experts to perform the validation process.

  - **Interoperability:** Validate interoperability when architecting a multicloud environment. Ensure systems can communicate with each other through well-programmed interfaces.

  - **Scalability:** Determine whether the system is scalable at both a global level and a component level and whether it supports auto scaling capabilities if required.

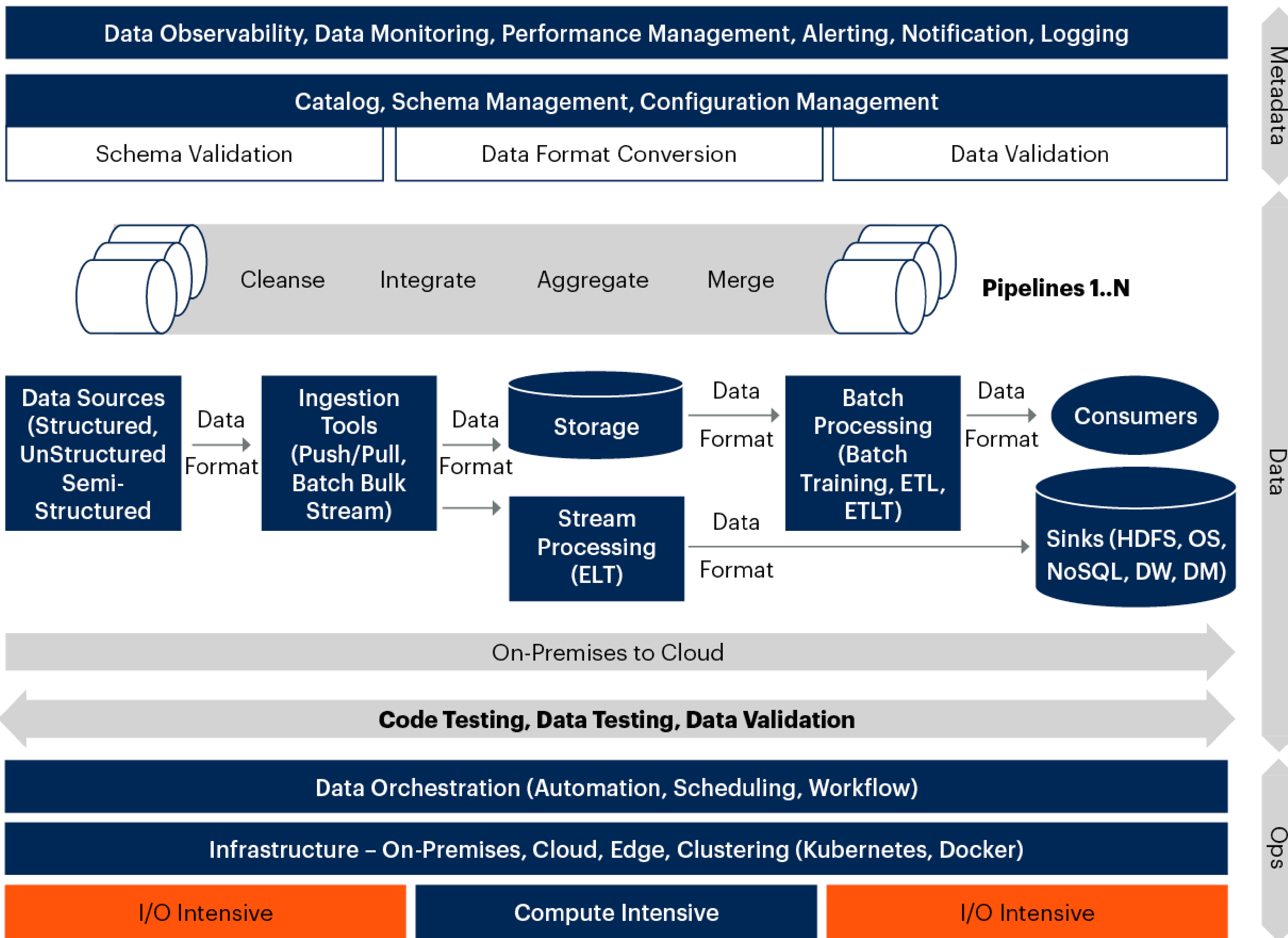
  - **Availability:** Validate the availability of systems and whether systems can be restored within the given SLAs.

- Ensure schema management is in place. Hadoop and nonrelational systems are often labeled the “Wild West” of schemas, where there is no standard format and schema enforcement. Proper schemas provide governance and shape to the data only when schemas are governed to enforce change control. A strong schema ensures that the data that makes its way through the pipeline is usable. Gartner recommends agreeing on schema before implementation because it provides a standard that can be reused. Ideally, you would want a centralized repository containing the necessary schemas across all integrations. This is imperative for preserving, auditing and updating schema for existing data sources, as well as when adding new ones. Schema provides governance on the shape of data and the fields that are required for validity. The industry approach to handling schemas is to include schema with the data. When writing data, write both the schema and data together. When reading data, read schema and then read data based on the schema. Avro is the most popular serialization system being used today across the data ecosystem. Gartner recommends that you ensure schema compliance for both incoming and outgoing data. A shared schema provides the following benefits:
  - Collaboration across different projects when working with a common set of datasets. Schema registry provides a centralized repository to agree upon schemas across all the data-driven applications.
  - Data drifts cause schemas to evolve and morph. This can cause massive data pipeline failures in downstream systems. Without a schema repository that provides schema versioning, there is no formal way to track changes and debug where data pipelines can fail because of schema mismatch.
  - Schema registry can also help with data discovery, a feature provided by most modern data cataloging and metadata management tools. Schema registry can also help in data policy enforcements — for example, preventing newer schema from breaking compatibility and providing lineage and traceability. Schema registry eliminates defensive coding and cross-team coordination, improves data quality, and reduces downstream application failures.

Some of the existing products for schema registry and schema formats include Amazon Web Services (AWS) Glue Schema Registry, Confluent Schema Registry, Iglu Schema Repository and Iteratively.

Data engineers and data architects should examine requirements. Figure 3 shows at a high level the data architecture with different components for building an end-to-end data pipeline.

**Figure 3: Data Architecture for End-To-End Data Pipelines**




Source: Gartner

741282_C

Gartner

## Step 2: Build Data Pipelines and Follow Software Engineering Principles

Data pipelines bring in data from external datasets and multiple sources across the organization and makes it available for downstream consumption, either in raw format or as curated data. Data pipelines can be built using one of two approaches to create abstractions for data ingestion and processing:

- An imperative — the freedom to do anything but with the caveat that everything has to be done by the developer — the "what"
- A declarative (think SQL) — use intent-driven design to abstract away the "how".

This is discussed in more detail later on in the document.

Ideally, a data pipeline should be designed to operate continuously with no manual intervention, completely configuration-driven. Modern data pipelines transcend data center boundaries and should be able to work in dynamic data environments where data can flow seamlessly across multiple data platforms both on-premises and in the cloud.

With a plethora of tools, frameworks, approaches emerging over the past few years, the cognitive load of building data pipelines has continued to increase. There is a lot of confusion, paradigms and approaches toward building data pipelines for an organization's needs. Some of the approaches and best practices for implementing the various aspects of a data pipeline are outlined in this section:

- Building Data Pipelines
- ETL/ELT/ETLT — Best Practices
- Functional Programming
- Imperative/Declarative (SQL/Nonrelational-Based Approaches)

Example solutions:

- SCDs ("slowly changing dimensions")
- Data deduplications

### Building Data Pipelines

Before jumping into building data pipelines, teams should have answers to the following questions based on the data they are working with, the required SLAs and domain requirements:

- How do we handle late arriving/duplicate data?
- How do we ensure pipeline failure part of the way and then rerun again from the point of failure?
- How do we keep track of metrics and monitor pipelines for data quality (DQ)/SLAs?
- How do we maximize performance — throughput or latency?
- How do we orchestrate end-to-end data pipelines?
- How do pipelines handle incrementally loading data?
- How can we debug transformation logic in a highly distributed environment?
- How does the system handle the propagation of upstream changes?
- How do we manage pipeline configuration and state?
- How do we implement metadata-driven data pipelines?

For each of the use cases identified in the Prework stage (see Table 2), identify all the data pipelines that would be needed to support the use cases. Tabulate them accordingly, as shown in Table 3. (Note: The columns just represent a sample set of attributes. Organizations should add other columns and attributes based on their requirements.)

Table 3: Data Pipeline Map

***Data Pipeline Name:*** ↓

Use Cases This Pipeline Supports (Select Use Cases From Table 2)

- Output Data Type/Format
- Pipeline Cadence
- Pipeline SLAs — Max. Time to Process, etc.
- Preconditions Before Pipeline Processing
- Postconditions After Pipeline Processing
- Data Quality/Validations *Before* Processing
- Data Quality/Validations *After* Processing
- Monitoring Metrics for Pipeline

Source: Gartner (May 2021)

The guiding principles when building large-scale data pipelines are shown in Figure 4.

Figure 4: Data Pipeline Guiding Principles

## Best Practices

Well-architected data pipelines can offer a differentiating and strategic advantage to organizations. But building such data pipelines is a daunting, time-consuming and costly activity. The right selection of frameworks and the right architectural mindset are needed. Organizations repeatedly underestimate the effort — in development and operations — and cost involved in building and maintaining these pipelines.

Some of the best practices when building data pipelines are:

- Invest in metadata-driven pipelines, which is the first step in achieving DataOps. Take a configuration-based approach, as discussed in Operationalizing Big Data Workloads.
- Parametrize data pipeline code to allow a single data pipeline to be used for both initial and regular ingestion (incremental ingestion).
- Make data pipelines retriable (idempotent) without unintended consequences (this is where having the benefit of no side effects of functional programming kicks in). For example, a pure task is deterministic and idempotent, implying it produces the same result in every execution. Rerunning a pure task with the same inputs should overwrite previous output without any changes. For example, a function that returns the current time or the current price of a stock symbol is not a pure task. Idempotency is essential to data pipeline operations. When jobs fail, or when logic is changed, the guarantee that reexecution of the task will not either result in side effects such as double-counting or lead to a bad state is imperative. The same concept can be applied to slowly changing dimensions without mutating (immutability) data, using dimension snapshots where new partitions are appended. Dimension is then a collection of dimension snapshots where each partition contains the state of the dimension at a point in time. This is one of the newer ways to solve SCDs with elegance and simplicity.
- Clean up resources by removing temporary folders, closing database connections or truncating temporary tables, and by making sure that you don't end up with duplicates due to inserting the same data several times.
- Make it easy to add, remove or skip components of your data pipeline by using a feature-flag-based approach of software development.
- Use code and data versioning tools to version data pipelines.
- Minimize dependencies among the different components in the pipeline by building and programming to interfaces.

- Refactor boilerplate and separate code from business logic code, and promote code reuse and loose coupling.
- Pipelines should ideally be built with a plug-and-play kind of componentized architecture. This will increase operational stability and decrease onboarding time to add new datasets and new services, as well as skip components.
- All data pipelines fail eventually, so think of fault tolerance when designing and implementing them. This can be achieved by having monitoring in place so that unexpected failures can be detected and detected early enough. Detection does not solve the problem though. Make sure the data team is prepared to deal with failures when they happen in order to recover fast. This needs the right architecture and pipelines that are built in a componentized way. Monitoring can get complex, especially when running multiple data pipelines simultaneously on a cluster.
- Organizations should invest in tools to manage workloads across different tenants and use isolation between the jobs by creating one cluster for one job. This can make debugging and troubleshooting much easier.
- Try to avoid long-running jobs. The longer the job, the more work is lost when there is a problem, and the longer the recovery time. Instead of doing it all in a single job, break longer jobs into smaller pieces either vertically by separating the data processing into multiple jobs or horizontally by partitioning input data and running multiple jobs to process the whole thing. This can increase your job processing time, but it is all a question of trade-off between performance and fault tolerance.
- Operate in a continuous fashion, with as little manual intervention as possible.
- Ensure resiliency to mitigate data drift by integrating well with schema registry the way Apache Kafka producers and consumers do.
- Ensure portability across different platforms and clouds providers by choosing open-source frameworks like Apache Spark and Apache Beam semantics.
- Capture data flow metrics to enable real-time insights at the pipeline as well as individual-stage-level details within a pipeline for minimizing time to troubleshoot and identify performance problems.
- Write small, well-encapsulated functions. Design your functions to do one (focus) thing and do that well — follow the UNIX philosophy. This also helps in testing and debugging because failure of a single operation can be easier identified, fixed and tested. Smaller functions also promote reusability. The less code to write, the less code to maintain and debug. Scala-based code is very terse and powerful, but can be difficult to read and understand.

### ETL/ELT/ETLT

Organizations deliberate over whether to use ETL or ELT and which is the best approach. There is not one right approach, and the answer depends on the trade-offs and use cases. The pros and cons of the approaches are discussed below, as well as a middle path: ETL and ELT present different strengths and weaknesses, so many organizations are using a hybrid “ETLT” approach to get the best of both worlds.

If writing ETL, follow these best practices:

- Partition data tables: Useful when dealing with large-size tables with long history, easy back-fill
- Load data incrementally: modular and manageable
- Add data checks early and often, and enforce uniqueness
- Separate the data model from transformation that allows proper testing
- Enforce idempotency so that the same code/query runs against the same logic and time range and returns the same result.

When to use ETL approach:

- When collecting and storing large amounts of raw, unstructured data
- When dealing with bulk data
- When data requires complex transformation — grouping, flattening, partitioning, calculations or cleaning
- When sensitive information needs to be removed before loading into the data lake/DW for compliance

When not to use ETL:

- Ideally, data engineers should not be writing ETL. Instead, they provide tools and frameworks that empower others — such as data analysts and data scientists — to perform data ingestion and processing, and even to better deploy applications through self-service tools.

- ETL is rigid because it tightly binds the end users to know how the data is to be used. Changes in data model and data schema can be very costly to implement.
- ETL often lacks visibility because transformations performed can obfuscate or obscure the underlying transformations that have been applied to the raw data.
- ETL also is attributed for lack of autonomy because it requires data engineers to perform the data extract and transform each source of data.

ELT is a good option in these circumstances:

- All transformation needed to curate the data into a usable format is simple enough to be handled via basic SQL operations.
- Business users prefer immediate access to raw or unconsolidated data to experiment first to check quality or explore the dataset before deciding how transformation needs to be done.

ELT moves the raw data into the storage, and it doesn't have the option of removing protected health information (PHI), personally identifiable information (PII) and sensitive data before loading it to the data warehouse. It sacrifices security, compliance and data quality for speed and flexibility.

Hence, ELT can also cause your data warehouse to end up with messy datasets from source systems.

When using ETL, complex transformation happens before loading into the data warehouse, so the greater the data volume, the longer business users have to wait before accessing the data. That's probably one of the biggest drawbacks of ETL when compared with ELT.

Extraction, transformation, loading and transformation (ETLT) is an augmented form of ETL. It breaks transformations into steps that each persist the data in different categories:

- The first step brings data from sources into one place. It is a raw extract into a staging area.
- The second step normalizes the data and allows for filtering, cleaning, selection, renaming, type casting and so on. Apply lightweight transformations in staging areas (tokenize, mask and encrypt sensitive data). Transformations are fast and transforms happen on each source independently. There is no attempt to integrate data sources.

- The final step happens in the analytics to create a dataset that is finally loaded in the data warehouse. Transform and integrate data within the data lake/DW using local tools and frameworks. The second transformation stage integrates multiple data sources and other transformations that apply to data from multiple sources at the same time.

This approach provides flexibility. Any data loading or processing errors can be backfilled from the previous tier and only the affected data needs to be fixed. ETLT allows ingestion faster because it performs lightweight transformations. It overcomes the ELT risks by ensuring essential data compliance and data quality requirements are applied.

### Functional Programming

Functional programming is not new, but it has gained a lot of prominence with the rise of data engineering as a discipline. So, what is functional programming?

Functional programming is a declarative type of programming style, where the focus is on “what to solve” — in stark contrast to imperative style where the focus is on “how to solve.” It is about building functions that work on immutable variables. This style of programming is great for data analysis and machine learning.

A functional data engineering approach is based on defining data processes as pure tasks. This is most beneficial for parallel data processing. When applied to data engineering, it focuses on immutability, idempotency and side-effect-free operations — thus making the tasks “pure.” Functional programming is applicable for working with data as any data engineering task takes input data, applies a function and outputs results. Most modern data engineering frameworks and tools adopt this philosophy and allow us to create reusable code. Such code can also be easily tested and debugged in isolation.

The idea of immutability, where the idea is not to mutate the history but use a snapshot-based approach, is vital for data engineering. This results in reproducible, repeatable code for building data pipelines. Distributed data stores and file systems are built on top of immutable data blocks with a functional approach. This has also been adopted by modern data processing systems built using programming languages that support functional capabilities like Scala that are being used for developing Spark and tools like Scalding from Twitter.

Functional-based architectural approaches are also reflected with modern databases, which use immutable blocks. Today, storage is cheaper than data engineering, and hence, creating copies of data with idempotency and immutability for reproducibility is the mantra. For example, partitions are immutable, and pure tasks should overwrite partitions output. As a part of idempotency and a functional

approach, always create a new index or partition for every new execution of the data pipeline. Even building DAGs (directed acyclic graphs) in data orchestration tools like Apache Airflow (version 2.0) is moving toward a function-based approach to defining DAGs. This allows the user to have a concise, readable way of defining data pipelines.

### Imperative/Declarative (SQL/Nonrelational)

Data engineers are of two archetypes:

1. The business focused ones who approach data engineering using SQL and focus on providing business value as quickly as possible. They look at the data as something that can have an impact on business and try to solve it as quickly as possible without going through the rigors of the engineering aspects of data.

2. The technical focused ones who love to work with Python, Java, Scala and so on and focus on engineering best practices and scalability when building data solutions.

Organizations as well as data engineers and data scientists are getting overwhelmed with the plethora of choices or frameworks, technologies and code in the data ecosystem. This tsunami of tools, frameworks and products has created some common problems in most organizations:

- Dearth of technical knowledge in their data teams
- Severe drought of skilled data engineers — not Coursera-baptized data engineers and data scientists — in the market
- Overabundance of SQL literate analysts, data scientists and data engineers who are unable to write ETL/ELT pipelines for large-scale data projects for numerous end-to-end data pipelines.

This has caused organizations that want to quickly become data-driven to use accelerated query engines — such as Hive, Presto and Impala — not just for data exploration, discovery and analysis, but also for building data pipelines.

There are a few ways to build data pipelines:

- Intent-driven design to abstract away the "how" of implementation from the "what" so that engineers can focus on the business meaning and logic of the data.

- Data pipelines can also be implemented in high-level programming languages such as Scala, Java or Python where data processing is written in structured code. This can be augmented with automated tests to build end-to-end robust data pipelines. This also enables the data pipelines to be treated as code that can take advantage of software development principles and CI/CD processes and tools such as Git, automated tests, builds and deployments.

- Another approach is to use visual tools that are based on a drag-and-drop-based paradigm with the ability to embed hooks and code in SQL, a proprietary language or a standard programming language and use prebuilt operators. These tools generate automated code when deployed. These tools originally did not have support for software engineering principles as version control or code review and writing unit tests. These tools are evolving with support for these features.

Large SQL queries are more often than not untestable and unmaintainable, and they typically are disliked by software engineering developers who transition to data engineering roles. Developing full-blown data pipelines with a SQL-centric approach results in long queries that are difficult to maintain, debug and version control. These queries are embedded with user-defined functions (UDF) written in a different programming language and with type conversions all mashed together to achieve the data processing objective.

This SQL approach also overlooks data type safety, compile-time errors and coding best practices. SQL-centric or low-code tools are adopted by organizations that lack skilled data engineers with the programmatic-based tools and frameworks. Organizations are advised to use SQL for doing data analysis, data exploration and data discovery, and they should use it for building data pipelines for low-complexity jobs.

In addition, SQL has some definite advantages over a pure programmatic approach:

- It is a popular lingua franca for implementing data processing and building data pipelines.

- It can often result in faster development with a quick iteration cycle with no need to compile, deploy, build, containerize and so on.

- For complex data transformation, use in-memory and distributed processing engines such as Spark, Flink and Beam when expressing business logic supported by the best practices of writing testable code.

- It results in a robust, scalable, debuggable and maintainable technology stack.

There is no one approach that fits all use cases and roles. Table 4 summarizes some of the differences between imperative and declarative approaches.

**Table 4: Imperative vs. Declarative**

| | ***Declarative*** | ***Imperative*** |
|---|---|---|
| Pros | ▪ Less code<br>▪ Faster development cycle<br>▪ Less maintenance | ▪ Flexible<br>▪ High levels of control |
| Cons | ▪ Domain specific<br>▪ Difficult to manually operate | ▪ High maintenance<br>▪ Requires state management<br>▪ Failure management can be heavy<br>▪ Everything needs to be done manually:<br>  ▪ Monitoring<br>  ▪ Profiling<br>  ▪ Partitioning<br>  ▪ Intermediate persistence<br>  ▪ Data lineage<br>  ▪ Privacy handling<br>  ▪ Error handling<br>  ▪ Retries — data and task deduplication |

| | *Declarative* | *Imperative* |
|---|---|---|
| | | |

Source: Gartner (May 2021)

## Approaches to Common Data Processing Requirements in Modern Data Pipelines

With the continually evolving data stack and approaches to building data-driven solutions, this section discusses how to approach two common data processing patterns with modern developments in the data ecosystem.

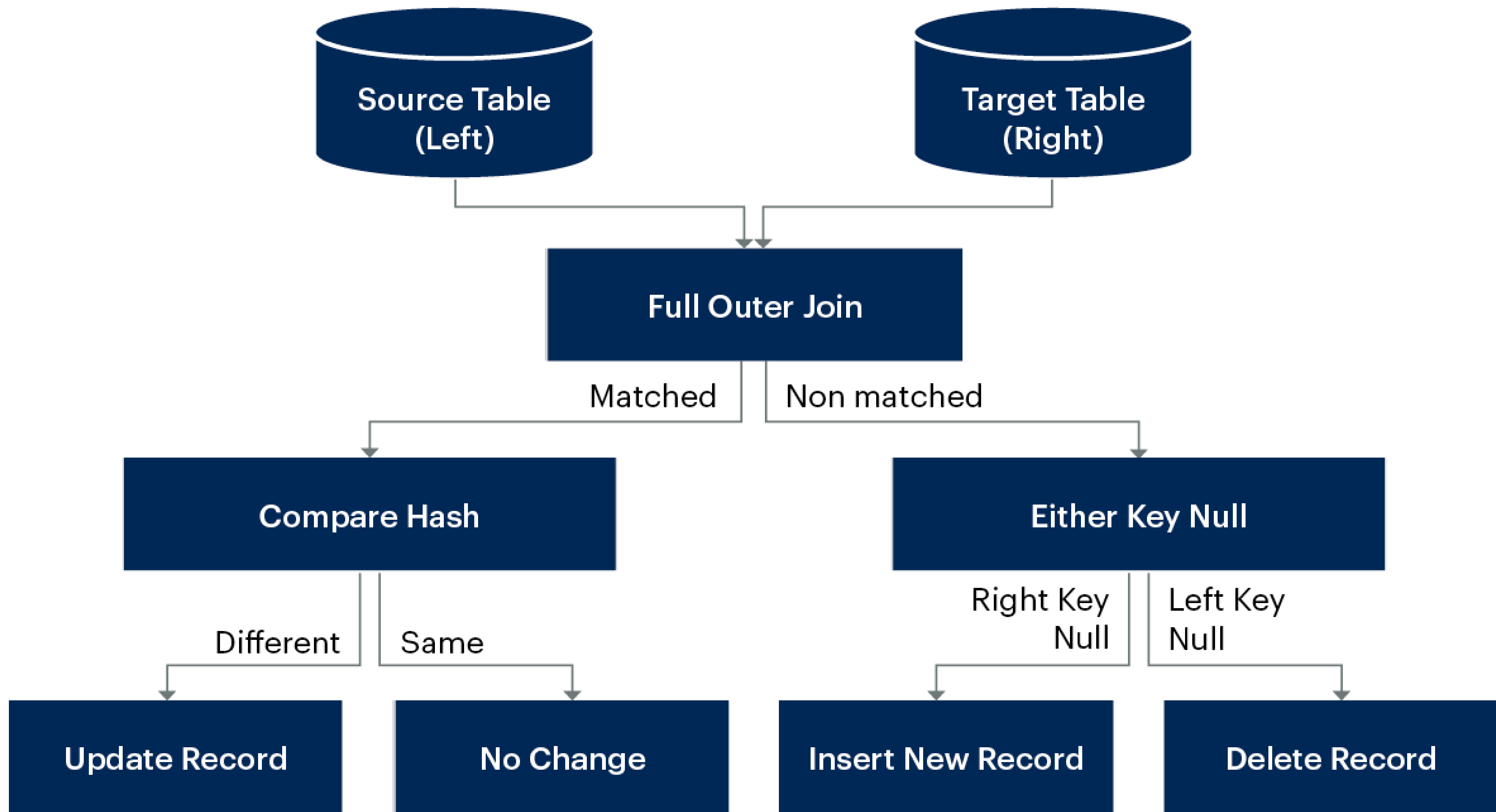
### How to Handle Slowly Changing Dimensions (SCDs) — A Very Common Requirement in Data Pipelines

The goal of SCDs is to preserve a history of changes for a dimension. For example, when customers change their phone number, an SCD Type 2 allows data analysts to link back to the customers and their attributes at the time of a transaction. This would be difficult to link to if the history were not retained. SCDs deal with the unpredictable and infrequent changes on the dimension tables from dimensional schemas.

According to data warehousing principles and best practices, there are six ways to handle SCDs in a relational datastore. At a high level, the logic of an SCD is shown in Figure 5.

Figure 5: Handling SCDs

## Handling SCDs




Source: Gartner
741282_C

Gartner

But doing distributed joins for dimensions in the large-scale datasets can be problematic. There are basically four approaches (last one being very recent) to handling SCDs at a conceptual level within a larger data ecosystem beyond a data warehouse:

- **Do it the "old" way — six SCD types:** This is very cumbersome to manage, error-prone for large datasets, full of mutations and hardly reproducible. Managing surrogate keys on dimensions and performing surrogate key lookup can be painfully slow for ETL jobs.
- **Use a snapshots-based approach:** These are dimension snapshots where a new partition is appended at each ETL schedule. This is simple and reproducible. Dimension becomes a collection of snapshots of multiple partitions, where each contains the full dimension of a point in time. With storage and compute becoming cheaper compared with data engineering time, dimension snapshotting is becoming easier and is consistent, and it allows reproducible results. This is appropriate for large dimensions.
- **Use nested data types:** A rather new approach to store historical values in dimensions is using complex and nested data types as arrays and map-based data structures within the datasets. As an example, to track a dimension's attribute history over time, add a "attribute_history" column as a map data structure where keys are dates and values are the actual values of this attribute at that point in time. This is very effective because the history can be tracked without any schema changes and without altering the grain of the table.
- **Use a delta lake:** This is the most advanced approach — everything happens under the hood.

### How to Handle Deduplications

Running into duplicate data issues is not a major problem for many applications, but there are applications where duplicate data is a must-fix issue, especially in the financial sector. Applications can *fail* for a variety of reasons — from hardware and software issues to network and service issues — and most applications are coded to retry operations with some exponential back-off strategies. This can cause the same user activity to emit similar messages multiple times, which leads to data loss and/or corrupt data, and this has a chain effect on downstream systems — for example, double counting.

To address this issue, data engineering pipelines need to handle duplicate data. However, there are huge challenges to deduplication at scale, which exacerbates in streaming systems when events flow at very high throughput in a distributed environment. In these cases, performing deduplication can result in high latency.

A few approaches for deduplication of data at scale:

- Common methods for data deduplication in storage architecture include hashing, binary comparison and delta differencing.
- At a database layer, you can define unique keys — based on one or more fields identified for ensuring uniqueness. The idea is to generate a table with the unique key based on the fields identified for deduping and then check the look-up for the existence of an entry.
- Another approach is to leverage Bloom filters — a space-efficient probabilistic data structure that is designed to rapidly decide whether an element is a member of a set. It allows elements to be added to the set, but not be removed. This is a probabilistic approach to data structure that tells whether the element either *definitely is not* in the set or *may be* in the set. In other words, false negatives *are not* possible, but false positives *are* possible.
- Another approach is to leverage process messages exactly once in order to avoid overcounting. This works with Kafka because the messaging system with exactly once message processing semantics can prevent duplicate message processing from happening.

## Step 3: Data Validation and Testing

Data testing skills are often not taught with a focus on data processing and data engineering skills, and in most organizations, testing during data engineering, as a formal process, is largely ignored and not implemented. Rigorous systematic data testing and quality monitoring is lacking in practice across most organizations. For software engineers coming to data engineering, it is essential to remember that code testing is *not* equal to data testing. With data pipelines, there is the need for both runtime data quality tests as well as standard code validation tests.

Validate what is expected from the data before running the pipelines, and describe expected behavior. For example, testing during development with data and code happens during software product development, but testing and data validation postdeployment does not happen. However, in data engineering, it is a recommended best practice to perform data validation and testing before starting the data pipelines in production.

Data teams should validate data across all stages of a data pipeline. Testing for schema guarantees and fixed data tests can confirm that data pipelines are functioning according to expectations and can help detect issues before they cause data discrepancies in downstream systems.

Adoption of data testing and quality monitoring is lacking in practice in most organizations. Automated data testing should be a first-class citizen and integrated into the CI/CD pipeline for data engineering. Data testing is often tightly coupled with data pipeline code,

creating a lack of visibility of data quality issues that are often inherent in deep pipeline code.

Even if tests are implemented, the final goal should be to automate them. Otherwise:

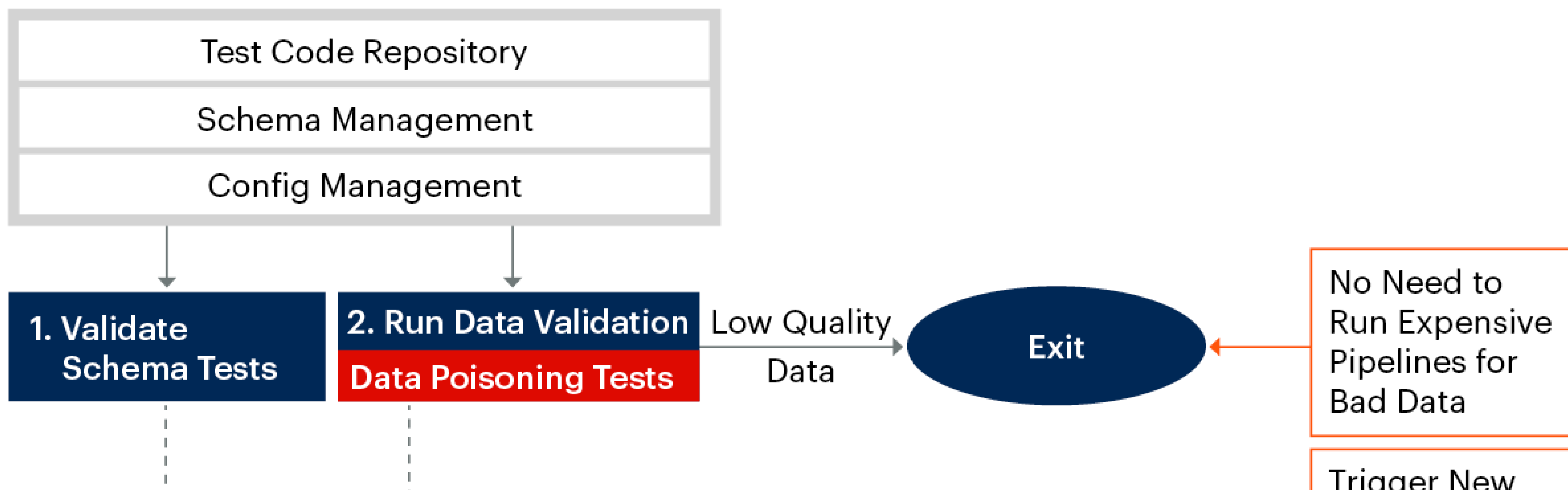
- Data engineers become skeptical of changing any code.
- Code rot can easily set in.
- A DataOps-based approach to a data-driven organization can remain a distant reality.

Data testing and data validation are a subset of data quality as a practice. Data quality at an organizational level is discussed in more detail in Data Quality Fundamentals for Data and Analytics Technical Professionals.

### How to Perform Data Tests and Data Validations

At a high level, data testing and data validation are to be done as shown in Figure 6.

Figure 6: Data Testing and Validation Framework



Data Sources

Data Ingestion

Data Transform

Pipeline Component to Run Data-Salting-Based Partitioning

3. Run Data Skew Tests

Skewed Data

Test Code Repository

Schema Management

Config Management

Source: Gartner

741282_C

Identify data tests and validations and define them at two different levels: the data source level and the pipeline level. For each of the data sources (as defined in Table 2 in the Prework section), define a set of tests and validations, as outlined in Table 5.

Table 5 shows the data test and data validation approach. Add additional tables if needed (not shown here) or if constraints and validations are needed to be formalized at a dataset level. For example, dataset sizes should not exceed “n number of rows” per cadence.

For each data source (as defined in Table 2), outline in Table 5 the details for the data pipeline tests and validations.

**Table 5: Data Test and Data Validation Approach**

| ***Data Sources Name (From Table 2)*** ↓ | |
|---|---|
| For Each Column | - Data Types — Categorical, Nominal<br>- Unique Data Values<br>- Max. Value<br>- Min. Value<br>- Allowed Types<br>- Valid Data Value |
| | |

Source: Gartner (May 2021)

For each data pipeline as defined in Table 3, outline the details in Table 6 for the data pipeline tests and validations

**Table 6: Data Pipeline Tests and Validations**

| ***Data Pipeline Name:*** ↓ |
|---|
| Data pipeline post execution tests and conditions to be satisfied |
| For each data source used in the pipeline, preconditions to be satisfied |
| Conditions to be satisfied for each stage of data pipeline execution |

***Data Pipeline Name:*** ↓

Source: Gartner (May 2021)

Always ensure to validate the schema before the data and validation tests.

Select tools to run data tests and data validations. Tools in this space are discussed below. Tools like Great Expectations are being widely adopted across the industry. For example, Great Expectations allows you to declare an expectation of the dataset as a declarative statement that describes the property of the dataset. For example, it allows you to define expectations as values in a given column for a dataset between two values X% of time and also allows you to ensure certain column values are unique in a declarative way. It also helps assert facts about data (uniqueness of columns and maximum/minimum values) that need to be enforced across updates, and it has the capability to automatically profile and propose such assertions. For example, below is an example of a declarative data validation test:

*checks:*

*- isComplete:*

*column: movieid*

*- isUnique:*

*level: warn*

*column: rating- dataFrameName: moviesWithRatings*

These tests should be stored in a test repository, which should be version controlled. Configuration settings required for these tests should be stored in a corresponding configuration repository.

Data tests and data validations should be integrated into the CI/CD pipelines and also deployed with the data pipeline code. These tests and validations for data sources should be run at ingest time, and the tests for data pipelines should be run at data pipeline

execution time.

These tests should be orchestrated by the data orchestration tool. The orchestration tool should take inputs to these tests and validations from the configuration repository to decide the thresholds associated with each test and validation.

### Best Practices

Some of the recommended best practices for data testing and validation are outlined below:

- Determine early during the development phase the strategy regarding what to do when data testing fails. This could take one of the following forms:
  - Do nothing. Continue to run the pipeline and log failure or alert the team.
  - Isolate the "bad" data — for example, move rows that fail to a separate file and continue to run the pipeline.
  - Stop the pipeline.
- Decode what to do to make your tests more robust against these unknown unknowns:
  - Use automated profilers to generate data tests to increase test coverage in areas that might not be obvious.
  - Do manual spot checks on your data, possibly also with the help of a profiler.
- Validate data across every stage of the pipeline through end-to-end testing.
- Do not tightly coupled your data testing code with data pipeline code.
- Automate data tests across every stage of the pipeline to detect and identify issues before they become data disasters.
- Add data checks early in the data pipeline and test often the ability to turn on or off certain types of checks based on configuration. This can be a very useful trick to debug and troubleshoot failed pipelines in production. When processing data, it is useful to write data into a staging table and check the data quality (often called the stage-check paradigm as a defensive mechanism).

- Define pipeline regression tests, and always track your test metrics. Ideally, test-result metrics should be saved for the long term and can be linked with a data versioning system for better troubleshooting. Make sure data used in data regression tests is updated and represents real production data. It is often recommended to use data from the latest successful production pipeline run for testing purposes.
- Don't stop with unit tests; include higher-level tests for the whole data pipeline. In distributed systems and databases, state matters — integration tests are more important than unit tests.
- Test databases are good to have, but they can fall out of sync quickly if they're not set up with automation to constantly pull in current data.
- Use tools to monitor data drift because perfect queries can often fail as underlying data generation changes and as test logic needs to be updated.
- While writing tests, think about data dependencies and things that could go wrong with data.
- If possible, implement schema tests (this is relatively easy to implement with database tools) in every data source before starting the ingesting pipelines.
- Write tests to make sure compliance is kept.
- Log failed test results along with the data points, and fully investigate any test failure to fix the root cause. As shown in Figure 7, log data can be useful for root cause analysis. Use the log data to understand if:
  - Data is correct, but tests need to be adjusted.
  - Data is "broken," but it can be fixed.
  - Data is corrupted and can't be fixed
- Always run data quality and data validation tests during ingestion. Do not run expensive data pipelines on low-quality data. Data quality and data validation metrics can also be coupled with data orchestration tools to inform the infrastructure provisioning modules to set up data infrastructure based on the data profiles of the data. As shown in Figure 7, detecting data skew in the

ingested data can trigger an additional step of salting the data before the data processing starts to avoid data skewing from creating straggler jobs.

Figure 7: Data Testing and Validation Flow

## Data Testing and Validation Flow

Source Data

Data Testing and Validation

Pass

Run Ingestion Pipeline Extract

Data Testing/ Data Validation

Pass

Run Transform Pipeline

Data Testing/ Data Validation

Pass

Promote to Product

Fail

Fail

Fail

Logging (for Root Cause Analysis)

- Data Is Correct, but Tests Need to Be Adjusted.

- Data Is "Broken," but It Can Be Fixed.
- Data Is Corrupted and Can't Be Fixed.

Source: Gartner
741282_C

- Another way of looking at Figure 7, with minor changes, could be to show circuit breaker patterns in data pipelines. When data quality issues are detected in a data pipeline, the data flow circuit opens, preventing low-quality data from propagating to downstream processes. Also, when the circuit opens, data teams are alerted of the issues and a diagnosis, along with a root cause issue, is performed.
- Test the infrastructure independently from the data pipelines.
- On failures of data validation, alert stakeholders and downstream data consumers. Ideally, an automated alerting system should be in place when data discrepancies are detected.
- A successful test does not mean everything is all good — it could also mean not enough testing is being done. For example, a data pipeline could be testing if record counts suddenly dropped, but may not be checking if the record counts suddenly doubled.
- Iterate and have a consensus, based on domain experts and data stewards, about what good data is.
- Track, monitor and observe data quality over time as data pipelines are run repeatedly.

### Tools

Some of the recommended tools for data validation and data testing include:

- Conexus SQL-for-ETL Constraint Validator — SQL-based ETL process
- Great Expectations — Python framework automates data profiling, testing and documenting
- Deequ — Open-source library on Apache Spark for defining "unit tests for data"

- Database tools that help create repeatable transformations so that the data loading workflow created on the original data can be applied as new records arrive

Other vendors in this space include Acceldata, Datafold, HyTrust-DataGravity, Soda, Bigeye, HoloClean, NexusData and OwlDQ (acquired by Collibra).

## Step 4: Orchestrate and Automate Data Pipelines

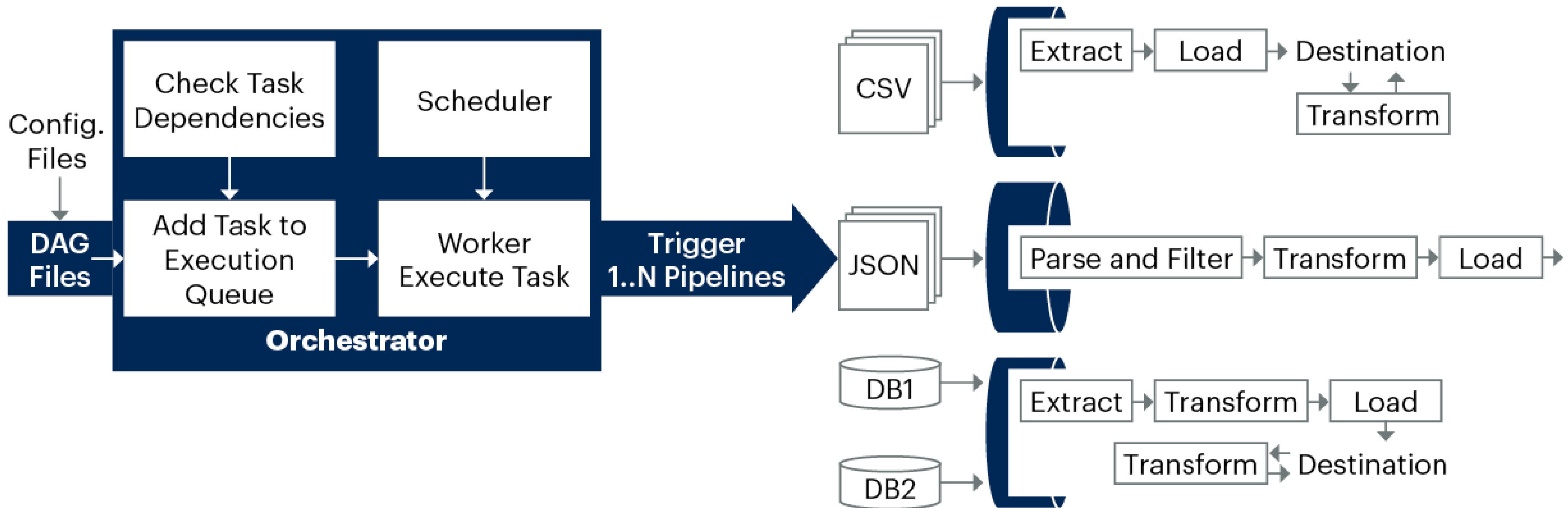
Data pipeline and machine learning workflows involve data movement between multiple subsystems at data ingestion, storage, preparation, training and scoring time. Data needs to be moved, wrangled and massaged into the right format before being consumed by downstream systems. Data orchestration systems allow composing these complex workflows, scheduling, and executing them reliably at scale in production. A large part of a data engineering team's focus should be on orchestration and automation, a prerequisite to building, running and deploying data pipelines at scale. Data orchestration stitches together the different moving parts of a data flow.

Data orchestration can be used for many processes like:

- Cleansing, organizing and publishing data from a data lake into a DW
- Computing business metrics
- Applying rules for doing automated data quality checks
- Maintaining data infrastructure
- Continuous delivery and continuous model training
- Creation of an event stream of task completions, workflow completions, and data arrivals to automatically trigger tasks
- Use of GitHub Actions as CI/CD triggers; code that is pushed to a master branch triggers the data orchestration system to generate and deploy a container in a Kubernetes cluster that performs the transformations

Figure 8 shows high-level internals of a data orchestration engine.

Figure 8: Data Orchestration Engines




Data pipeline orchestration is evolving from a manual approach to a more template-driven approach. Some of the trends happening in the data orchestration space include a move toward a more data-driven approach to data orchestration.

First-generation data orchestration systems like Airflow and Luigi have a task-driven architecture. Tasks in Airflow are idempotent — no matter how many times you run a task, it needs to result in the same outcome for the same input parameters. These frameworks are unaware of the data being passed across the tasks. This lack of data awareness prevents data pipelines from benefiting in caching and reuse of computation, which could provide significant efficiency gains as pipelines become more complex and long-running. Such frameworks have problems, including restarting long-running pipelines after failure and restarting from where the failure happened.

These pipelines cannot accommodate incremental changes during pipeline iterations and cannot take advantage of storing in-memory intermediate results. They are also hard to integrate with context-specific choices of job scheduling.

Data orchestration in data engineering pipelines is slowly moving toward adopting the next-gen orchestration tools like Dagster, Flyte and Reflow. These frameworks are data-driven, which means they are aware of the types of input and output tasks. They version the output tasks and often include caching the data to avoid rerunning the tasks. Data-driven orchestration like Flyte (from Lyft) intelligently understands how the data splits into different tasks.

Detailed coverage and how to approach orchestration and scheduling are discussed in depth in Operationalizing Big Data Workloads — especially the Step 4: Orchestration and Scheduling section of that research — and will not be discussed here.

### How to Perform Data Orchestration

Data orchestration and scheduling should be identified and properly defined at two different levels: the data source level and the pipeline level.

For each dataset defined in Table 2, use the data in Table 7 to build the orchestration flow.

**Table 7: Data Orchestration for Data Source Ingestion**

| ***Data source (or sources) this orchestration is associated with:*** ↓ |
| --- |
| Error handlers to be invoked for wrong data |
| Exception handlers to be invoked for bad data |
| Actual ingestion framework details associated with this orchestration |
| |

Source: Gartner (May 2021)

For each data pipeline defined in Table 3, use the data in Table 8 to build the orchestration flow.

Table 8: Data Orchestration for Data Pipelines

| ***Data pipeline this orchestration is associated with:*** |
|---|
| Error handlers to be invoked |
| Exception handlers to be invoked |
| List of different DAGs (directed acyclic graphs) associated with this orchestration and associated metrics |
| |

Source: Gartner (May 2021)

These configurations could be specified in the configuration repository. The orchestration tool should be configured with these settings and should integrate with the configuration management tool to use these configurations at production time, as shown in Figure 8.

Choose the right orchestration tool. Some of the tools in this space are discussed below. Airflow is a mature tool that is increasingly being used across many organizations. Airflow 2.0 has overcome limitations such as support for REST API, support for Airflow TaskFlow API and support for Kubernetes. It is being offered as a managed service by cloud providers like AWS and Google Cloud Platform (GCP). Airflow has a lot of moving pieces and includes scheduling and a monitoring UI. Tools like Airflow and Luigi are suitable for ETL and data engineering tasks, but they are not suitable for ML-based workflows.

### Tools

Here are a few things to consider when selecting tools for data orchestration:

- The tool should check whether SLAs are being met.
- The tool should contain hooks to send email or integrate with other systems like slack to an administrator to report events such as errors during test runs.
- The chosen tool should include Kubernetes Operator support.
- The tool should have the ability to exchange data between tasks.
- It should support well-documented APIs to create automated tasks with a declarative style.
- Ideally, the tool should have a high-availability scheduler.
- The tool should be able to resume your workflow from where it failed.
- The tool should save/log all the parameters, outputs and metadata from each step in an appropriate registry or repository.
- The tool should provide a visual guide of the graph of operations being performed and where in the path the current execution context is.
- Look for tools that support custom hooks that send data pipeline errors/warning alerts to, for example, a Slack channel.

Other tools in the space include Argo AI, Astronomer, AWS Step Functions, Dagster, Flyte, Kubeflow Pipelines, Luigi, Prefect and Tecton

### Implementing Automation

To scale data pipeline development and deployment automation is a perquisite. At the first level, ingestion, processing, storage and deployment should be automated to build end-to-end data driven platforms. Next stage, should be to automate metric collection and feedback the results of data observability and monitoring to improve the efficiency of data flow, choice of hardware and autoscaling. Not to be left behind — automated tests also provide high and sustainable data quality. Automation improves speed to market or velocity in agile speech, and it mitigates errors.

Automation doesn't work well without architecture. Pipeline architecture defines standards, patterns, templates and reuse opportunities. Metadata is essential for pipeline automation. Automation should be configuration-driven and tightly coupled with

metadata. Automation of pipeline operation includes functions to schedule jobs, execute workflows, coordinate dependencies among tasks and monitor execution in real time.

Ideally, organizations should look to automate these four things at a minimum:

- Automated tests
- Automated infrastructure provisioning
- Automated deployment
- Automated pipeline execution

Infrastructure automation built using tools such as Terraform, Ansible and Chef allows creation and change management of infrastructure. It also reduces provisioning time and improves accuracy of deployments based only on configuration. IaaC (infrastructure as configuration) provides DevOps productivity and an audit trail of deployments.

Data pipelines are often handcrafted with little attention to standards, reusable components and repeatable processes. Pipeline automation shifts pipeline development away from handcrafting toward true engineering discipline. Pipeline automation must be used to gain efficiency and improve effectiveness for data pipeline development and operations. This is imperative to reduce backlogs and meet future demands.

### Best Practices

Some of the best practices for data orchestration include:

- The configuration and execution of pipelines should be as close to one-click deployment as possible for users.
- DAG workflows should instantiate runtime parameters once at the beginning of the workflow and push parameters to the tasks.
- Because storage is inexpensive, you should establish a checkpoint and persist intermediate data computations across data extraction, preprocessing and training.
- Decouple orchestration of the data pipeline from business logic.

- Architect application so that the structure of your DAGs is independent from the structure of business logic.
- Don’t manually keep track of filenames in complex data workflows because it is not scalable.

Some of the best practices for automation are outlined below:

- Choose carefully where to automate — don’t seek to automate everything and everywhere.
- When starting with pipeline automation — begin with straightforward pipeline requirements, gain experience and then add complexity.
- Perform automatic anomaly detections on data assets.
- Embed architectural standards into automated tools, and apply and adapt the templates and patterns that are built into tools to best fit enterprise requirements.
- Improve productivity, including automated scheduling and real-time monitoring of pipelines.
- Automate data protection so that sensitive data connects with policies for data protection, and applies those policies for data masking and obfuscation.
- Automate for data lineage to gather metadata about data pipelines’ data sources, flows and processes that shape the data as it moves.

### Step 5: Implement Data Monitoring and Data Observability

Organizations often do not measure the quality of their data and do not know that they have a problem or where to improve it. It is often left to the data engineering team to write tests, identify errors and then fix them. The ability to tackle broken data pipelines is lagging. This is tedious and time-consuming and signals a bigger disaster waiting to happen for data-driven organizations. Data pipelines are growing complex with multiple tools, frameworks, and stages of data storage and processing and with multiple dependencies between data assets and systems. Changes made to one dataset almost always have unintended side effects that have a direct impact on trust ability and correctness of downstream systems, products and datasets.

In order to avoid the above problems and avoid blind spots, it is becoming increasingly critical for data teams to perform end-to-end data monitoring and/or data observability. Data observability provides visibility by alerting to minimize and prevent data downtime.

Data pipelines behave very differently than software applications and infrastructure. Just like the field of software reliability is woven into organizations building software engineering practices, data reliability needs to be interwoven with data engineering practices.

Like Google's SRE monitoring systems, data monitoring systems should address two questions: "What's broken?" and "Why is it broken?" Data pipelines have more aspects that need to be monitored and often require granular reporting on metrics. These get exponentially compounded when data pipelines are using complex systems such as Spark, Kafka or Kubernetes.

Data observability provides end-to-end monitoring, and leverages machine learning to identify data issues and to assess impact and root causes of failures. It allows organizations to understand data health and can often minimize data downtime. Data observability uses automation to identify data quality issues, preventing downstream data issues.

Observability is often confused with monitoring. Monitoring is the act of collecting data of certain metrics, while data observability is the ability to infer the internal states of the system from the monitoring metrics. Observability provides context to the metrics monitored and provides a deeper view of the system. Observability goes one step beyond just monitoring and collecting metrics. As data pipelines get more complex and distributed over a network across multiple processes and tasks, determining why parts fail becomes a complex science. This is where data observability helps. Observability is an essential part of data engineering when building a data platform because it provides visibility into system internals to identify issues and troubleshoot more reliably and quickly. Better observability leads to better monitoring. Observability can mean how transparent the internal state of a system is from the outside.

Only through better and continuous observability can organizations hope to achieve resilience and reduce the MTTR (mean time to repair or restore) and reduce time needed to troubleshoot. Observability cannot be an afterthought; it must be built right from the beginning.

Figure 9 shows typically how monitoring is done.

**Figure 9: Spark Monitoring Example**

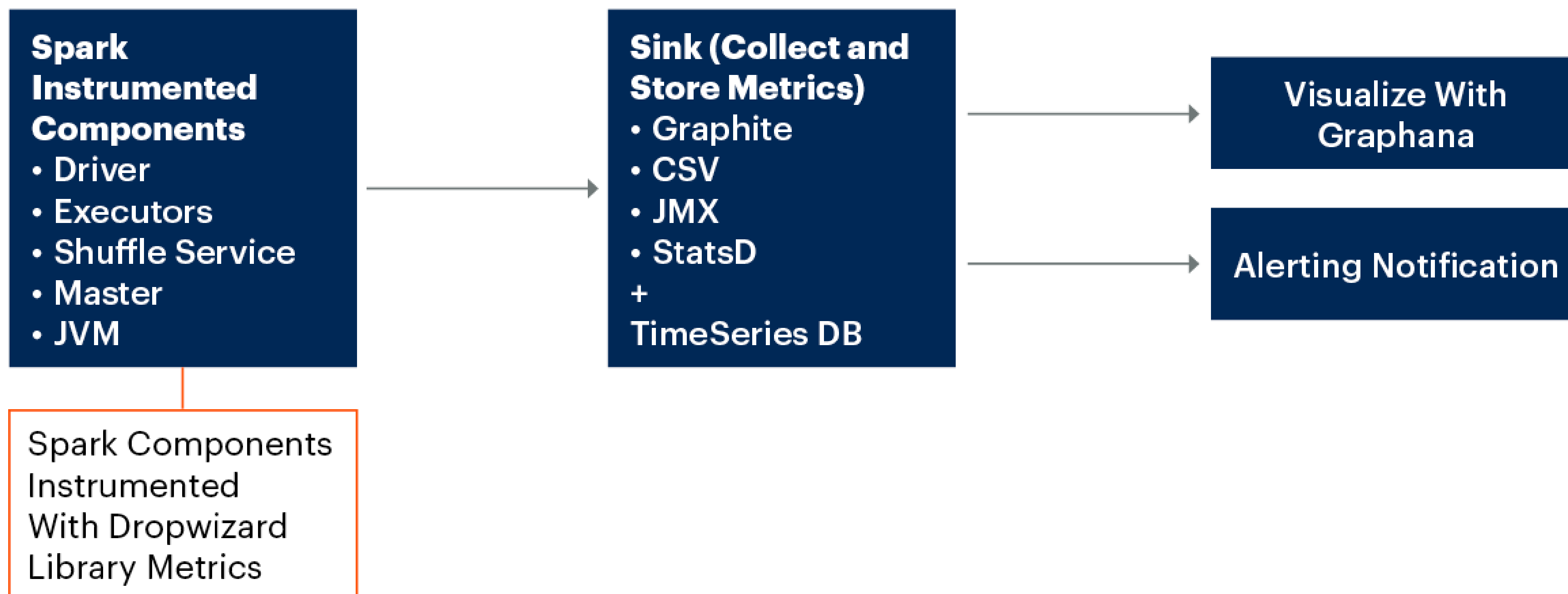
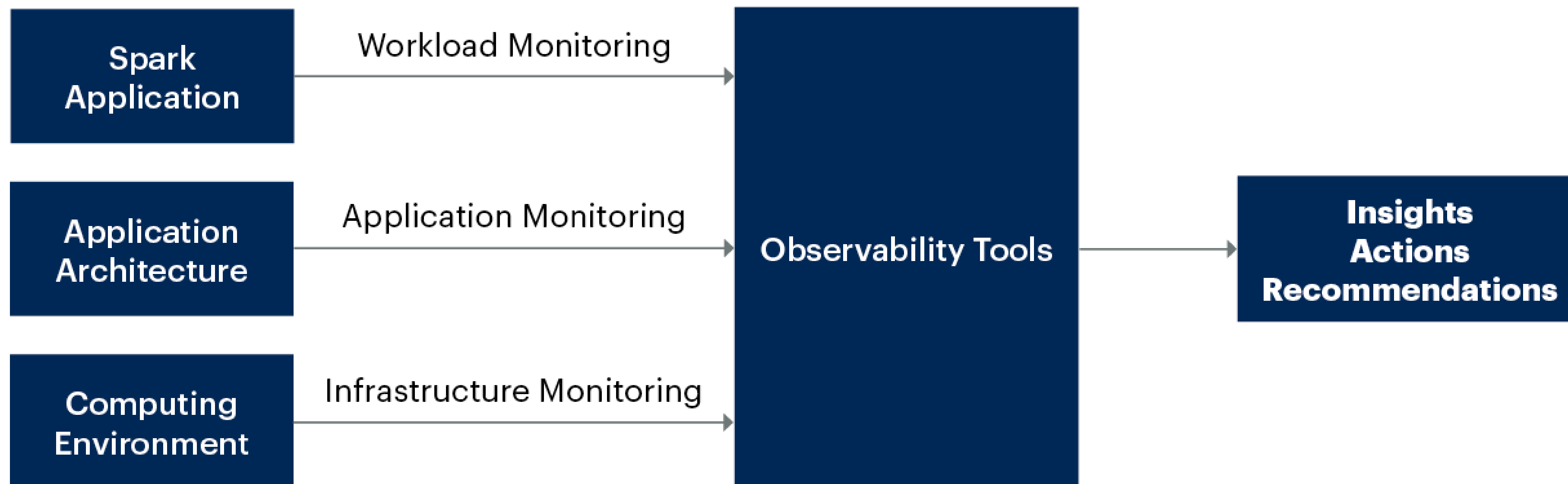



Figure 10 shows how observability is beyond just monitoring.

**Figure 10: Observability With Spark**




In most cases, data engineers use standard monitoring tools that were originally built for applications and infrastructure monitoring and use it to monitor the data pipelines. These provide visibility into, at a high level, job tasks and data store performance, but they lack visibility into the right level of information needed to manage pipelines. This can cause teams to spend a lot of time tracking issues, debugging and building hypotheses and drawing inferences manually. Data engineering teams need to be smart. They should be able to anticipate failures based on monitoring data. This is where data observability becomes indispensable. The ability to peek deeper inside data pipelines and intermediate results as a result of complex DAGs is important to catch errors.

### How to Perform Data Monitoring

When building the monitoring solution and deciding what is to be monitored, refer to Table 2, outlined in the Prework step. Similarly, for the data pipeline, refer to Table 3 in the Building Data Pipelines section.

Monitoring has to be done at multiple levels. Monitor the datasets, data pipelines, frameworks being used — for example, Spark and Kafka — and finally the infrastructure. Develop a set of tables (as shown in Tables 9 and 10) that outline the list of metrics to be monitored.

Table 9: Dataset Monitoring Map

| ***For each dataset defined in Table 2, monitor:*** ↓ |
|---|
| The number of bad records |
| The number of null values across columns |
| The data distributions |
| The total row counts |
| |

Source: Gartner (May 2021)

Table 10: Data Pipeline Monitoring Map

| ***For each data pipeline defined in Table 3, monitor:*** ↓ |
|---|

| ***For each data pipeline defined in Table 3, monitor:*** |
|---|
| Execution time for each stage of the data pipeline |
| Input data count for each stage |
| Output data count for each stage |
| End-to-end pipeline execution time |
| |

Source: Gartner (2021)

Frameworks like Spark provide out-of-the-box metrics to be monitored. Also, tools like Dropwizard's library in Spark can be used to develop hooks into your applications to monitor Spark metrics as well as to create additional metrics to monitor.

For each framework used in the pipeline, outline a set of metrics that would be useful to monitor. For example, a sample set of metrics to monitor for Spark would look similar to Table 11.

**Table 11: Framework Monitoring Map**

| ***Framework*** | ***Pipeline Name*** | ***Metrics*** | ***Metrics*** | ***Metrics*** | ***Metrics*** | |
|---|---|---|---|---|---|---|

| Framework | Pipeline Name | Metrics | Metrics | Metrics | Metrics | |
|---|---|---|---|---|---|---|
| Kafka | P1 | Producer Level Metrics | Consumer Level Metrics | Broker-Level Metrics | JVM-Level Metrics | Topic-Level Metrics |
| Spark | P2 | Driver- Level Metrics | Executor-Level Metrics | JVM-Level Metrics | | |
| | | | | | | |

Source: Gartner (May 2021)

Infrastructure monitoring is out of scope for this document. However, along similar lines, organizations should set up a set of metrics for infrastructure monitoring. These should be clearly outlined and set up using the infrastructure monitoring tool of choice (see Table 12).

**Table 12: Infrastructure Monitoring Map**

| Infrastructure | Type (Hardware) — Specs, Framework — Spark | CPU Metrics Monitoring List | Memory Metrics Monitoring List | I/O Metrics Monitoring List | Networking Metrics Monitoring List | Other |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |

Source: Gartner (May 2021)

Also, the following could be monitored for additional visibility and troubleshooting:

- Changes in frequency and volume of incoming data. Any significant change could indicate some real-world event.
- Track statistics mean, median, variance, standard deviation, number of outliers and so on about your data. Advanced organizations can also track the change in distribution of the incoming data.
- Trigger alerts whenever minimum/maximum ranges change for certain features beyond acceptable thresholds.
- Monitoring how often data is changing, and if there is a need to train models more frequently. This can also be used to understand when to trigger autoscaling of the data pipelines to adhere to the SLAs.
- Only tracking the above metrics can lead to a better understanding of the need to upgrade the infrastructure with growing demand and data volumes. This can also trigger the need to reevaluate chosen data processing frameworks and data storage solutions or hardware when certain limitations are reached.

### Questions

Some questions that data engineers should ask regarding the data sources, data pipeline and infrastructure include the following. The right set of monitoring and observability tools would help data engineers and architects to home in on the answers to these questions:

- When was the dataset last updated?
- Why is a dataset missing?
- Is the data within an accepted range?
- Is the data complete?
- Did the data suddenly become skewed?
- Has the training data been poisoned?
- What is the lineage with time stamps of mutations that have been applied to the dataset?

- Where did the data pipeline break?
- Which dependent datasets or downstream reports were affected?
- *When* should a dataset be considered late?
- *How frequently* are datasets late?
- *Why* is a dataset late?
- What is a reasonable SLA for a dataset?
- When a dataset is late, how do you understand why?
- What is the data source?
- What is the schema for a data source?
- How often is it updated?
- What has changed?

### Best Practices

Best practices that can be applied to data pipelines include:

- Ensure timeliness of alerts, and ensure testing of alerting mechanism failures.
- Don't ignore infrastructure monitoring to ensure service uptime and optimal performance.
- Implement validations to provide alerts when data does not match expectations.
- Make notifications and alerts clear for stakeholders.
- Don't let errors pass silently.

- Categorize which errors need alerting and which ones can be simply logged. Any critical errors should go to a different channel than errors that only require logging.
- Ensure the mediums you choose for the alerts are ones which don't go unnoticed.
- Incorporate metadata and cataloging integration with monitoring and observability tools. This is useful to have a centralized, single pane-of-glass view into the data ecosystem.
- Automate monitoring and alerting for data downtime and reducing latency when data errors happen.
- Incorporate lineage tracking of data changes and dependencies.
- Ensure logging and monitoring is auditable to enable the data engineering team to see the history of events. Ensure that logs are searchable, have the time stamps and are properly structured with the right tagging.
- Use the analogy from software development to perform root cause software failures in production; use a Five Whys method to perform the diagnosis.
- Save monitoring data, and version it with the related code and the data on which the monitoring results were collected.
- Do not overdo monitoring, because any instrumented system results in slowdown of the running systems.
- Always drive monitoring tools through a configuration-based system, and do not do hardcoding or one-off-based monitoring.

Some metrics that should be monitored regularly:

- Data source throughput rates
- Data pipeline error rates
- Execution time by stage
- PII detection failures
- Schema drift frequency

- Data skew frequency
- Semantic drift frequency

#### Tools

When choosing monitoring and observability tools, ensure that the tool enables:

- Understanding operational dependencies
- Impact analysis
- Troubleshooting: What has changed since the last time it worked?
- Ideally, integration with your data orchestration tools — for example, Databand integrates with Airflow.

Some of the tools and vendors in this space include Datakin, Bigeye, Datafold, Databand, Atlan, Datadog, New Relic, Librato, Dynatrace, TICK stack, PagerDuty, Monte Carlo, Cribl, OpenTelemetry, Soda, Superconductive, SparkMeasure, Unravel and Dashbird.

## Risks and Pitfalls

Organizations undertaking data engineering initiatives face several risks and potential pitfalls for their project. Organizations that plan to be data-driven and yet overlook right data engineering practices run the risk of falling behind the data curve, time to market and return on investments.

### Pitfall: Ignoring Testing, Orchestration and Monitoring

There is a huge downside to not putting enough emphasis and stress on Steps 3, 4 and 5. Organizations' vision can easily get blurry with shiny new products for data ingestion, storage and processing. They lose track of cross-cutting steps needed to make an organization data-driven from end to end.

Organizations aiming to transition to an end-to-end data-driven model absolutely need automated testing of data pipelines to manage complexity. Test data instead of just testing code; that's where the real complexity is. Inculcate the habit of testing when new data arrives.

For long-running and complex DAG-based pipelines, failed jobs can result in partial results being read downstream. Ensure proper failure, exception and error handling is applied throughout the pipeline for debugging and troubleshooting purposes. Ensure idempotency by using a functional approach to developing data pipelines and by checkpointing at appropriate places within the pipelines.

## Risk: Managing Drift

Drift is dangerous, whether it is in water, in the data, or in machine-learning-based models. Managing data, framework, schema, infrastructure, code and models can be difficult to track and correct. Automated tools to detect, manage and remediate drift, especially in data and machine learning models, are still in infancy, and hence, engineering practices must be in place to reduce the time to detect and remediate drifts.

## Risk: Underengineering, Overengineering and Tight Coupling

Avoid overengineering, overabstraction and overgeneralization following the principles of software development — that is, YAGNI (you ain't gonna need it). Data engineering tools are already complex, and distributed systems are inherently complex.

Avoid the urge to build everything in-house. Most of the problems an organization encounters have been already seen and solved with well-known patterns and best practices. Stand on the shoulders of giants and leverage both commercial and mature open-source frameworks. Integrate them for data engineering problems.

## Pitfall: Pipeline Debt

When organizations are looking for velocity of product delivery, tight coupling between frameworks and tools can happen often, making the pipelines brittle and interconnected. Pipeline debt is a technical debt that infests data pipelines. One of the best ways to address pipeline debt is automated testing of data pipelines with code and data.

Data teams generally cut across different departments whose data is used and interconnected within the pipelines. Differences in domain knowledge and data tribal knowledge, missing context, often allows pipeline debt to creep in. Pipeline complexity is in data, with edge cases and continuous data drift often exacerbating pipeline debt.

## Pitfalls: Data Engineers and Data Scientists Using Notebooks-Based Approach to Data Engineering

Notebooks are good for writing rough/scratch code and trying out experiments. The biggest downside of notebooks is that they often violate all best practices of software development and end up with code duplicated and code scattered all around. Tools for notebooks

lack governance around usage, naming scheme, parametrization and versioning.

### Pitfall: Lack of Schema Management

Lack of Schema enforcement can lead to type inconsistency and corruption. Modern data systems like HDFS and nonrelational data stores can be like the Wild West, with schema management with no agreed-upon standards in serialization and versioning. Schema management is extremely critical for large organizations and large teams looking to collaborate and share data and metadata across silos. Kafka and AWS Glue Schema Registry provide great examples of managing schemas, eliminating defensive coding, and promoting cross-team collaboration and coordination to improve data quality and mitigating application failures.

### Pitfall: Trying to Find Just One Tool and Choosing Wrong Frameworks and Tools

The entire data ecosystem is exploding with tools. There is no single tool that can help with data engineering across all components in the complex data stack. Added to that is the constantly changing and evolving data ecosystem and its tool stack. Organizations should be careful while selecting tools. However, tool selection, tool integration and development of tool expertise should not become full-fledged projects. If an organization does not find the right commercial tool for a specific purpose, it should search the open-source space. This should be undertaken only if the organization has in-house skills to understand, integrate and sometimes extend open-source products.

Understand the best-fit use cases of tools before using them. Be careful when adding tools to the tool chain that do not facilitate automation and integration with other tools in the organization. Be wary of tools that claim to take away complexity and capabilities to build data products easily.

### Pitfall: Not Hiring for the Skills Required

When undertaking data projects, organizations should look to hire not just data engineers, but also DataOps, data architects and distributed system engineers. For cloud-based deployments, organizations should look for cloud engineers and cloud architects to complement their data teams.

## Recommended For You

How to Operationalize Data Workloads

Demystifying the Data Fabric

Decision Point for Selecting Semistructured Data Stores

Working With Graph Data Stores

Adopting a Logical Data Warehouse

## Supporting Initiatives

Data Management Solutions for Technical Professionals

© 2021 Gartner, Inc. and/or its affiliates. All rights reserved. Gartner is a registered trademark of Gartner, Inc. and its affiliates. This publication may not be reproduced or distributed in any form without Gartner's prior written permission. It consists of the opinions of Gartner's research organization, which should not be construed as statements of fact. While the information contained in this publication has been obtained from sources believed to be reliable, Gartner disclaims all warranties as to the accuracy, completeness or adequacy of such information. Although Gartner research may address legal and financial issues, Gartner does not provide legal or investment advice and its research should not be construed or used as such. Your access and use of this publication are governed by Gartner's Usage Policy. Gartner prides itself on its reputation for independence and objectivity. Its research is produced independently by its research organization without input or influence from any third party. For further information, see "Guiding Principles on Independence and Objectivity."



© 2021 Gartner, Inc. and/or its affiliates. All rights reserved.