dagster

# Data Platform Fundamentals

Joe Naso
Colton Padden

# Table of Contents

# 01 Data Platforms

# Introduction

Many companies follow a similar maturity curve in data engineering. Initial, small-scale automations grow to support large-scale orchestration needs. Eventually, organizations recognize the need for dedicated ownership—a team or individual tasked with building and maintaining a scalable platform to support their automation and data pipeline requirements.

While organizations may have very similar data needs, headcount, and workflows, it's common for the platforms they build to be significantly different. This is because no distinct solution exists for building a data platform. This can come down to decisions about their cloud provider, preference for open or closed-source software, and budget. Although the underlying technologies may be unique, these organizations often follow common, well-known patterns and principles to design their data platforms. We will review some of those patterns and principles in this e-book.

Dagster believes in an open data platform that is heterogeneous and centralized. It should support specific business use cases and accommodate various data storage and processing tools, all while providing a central unified control plane across all of the processes in the organization.

Only one data platform should exist in an organization. While a single platform may require a larger upfront investment, the standardization it provides and productivity boost for the team pay dividends as the platform grows.

An extensible platform provides better business alignment and less duplicated work. For more information on what Dagster believes about data platforms, see the "What Dagster Believes About Data Platforms" blog post.

This book will be a comprehensive guide for data platform owners looking to build a stable and scalable data platform, starting with the fundamentals:

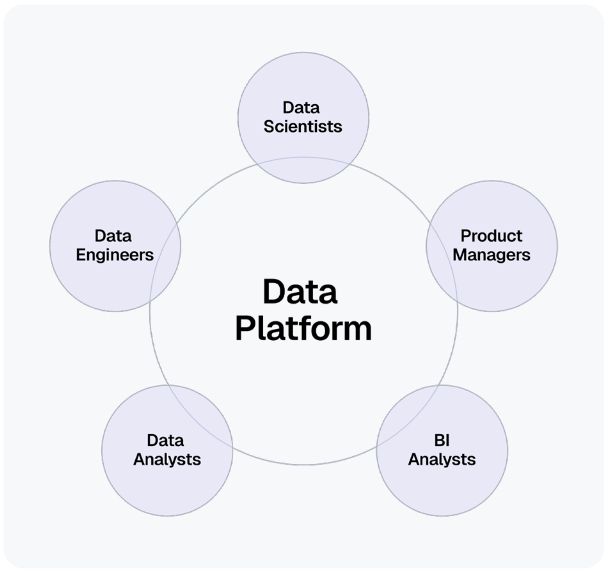
- **Architecting your Data Platform**
- **Design Patterns and Tools**
- **Observability**
- **Data Quality**

Finally, we'll tie it together with real-world examples illustrating how different teams have built in-house data platforms for their businesses.

## An Open Data Platform

You can find the Dagster team's internal Open Data Platform on GitHub. Throughout this e-book, you'll see how Dagster's core tenet of building open-source software can help establish the standards for a scalable, maintainable, and stable data platform.

This book is for data practitioners interested in building and maintaining a data platform. You may be a team of one or part of an established team with a growing area of responsibility. In either case, common issues, patterns, and needs are core to building a data platform. This book will help you navigate those topics and establish a data platform that evolves alongside your business.



## Why do we need data platforms?

Most data tools fall into two categories: all-in-one solutions or unbundled tools. One of the biggest criticisms of the Modern Data Stack is that its design over-indexes on unbundled tools, requiring teams to stitch together a large number of specialized tech to address problems historically managed by larger, managed platforms.

Is it really necessary to adopt a dozen tools to do what was previously managed by only a handful? The problem is not deciding which tools you need but how to integrate those tools into a cohesive whole.

Unbundling your entire stack reveals one major weakness–maintaining consistency and control over various tooling choices and access points. Addressing this flaw is the fundamental purpose of building a data platform.

It helps us bridge the gap between a fully unbundled data stack and a singular, closed platform. And it all comes down to the control plane.

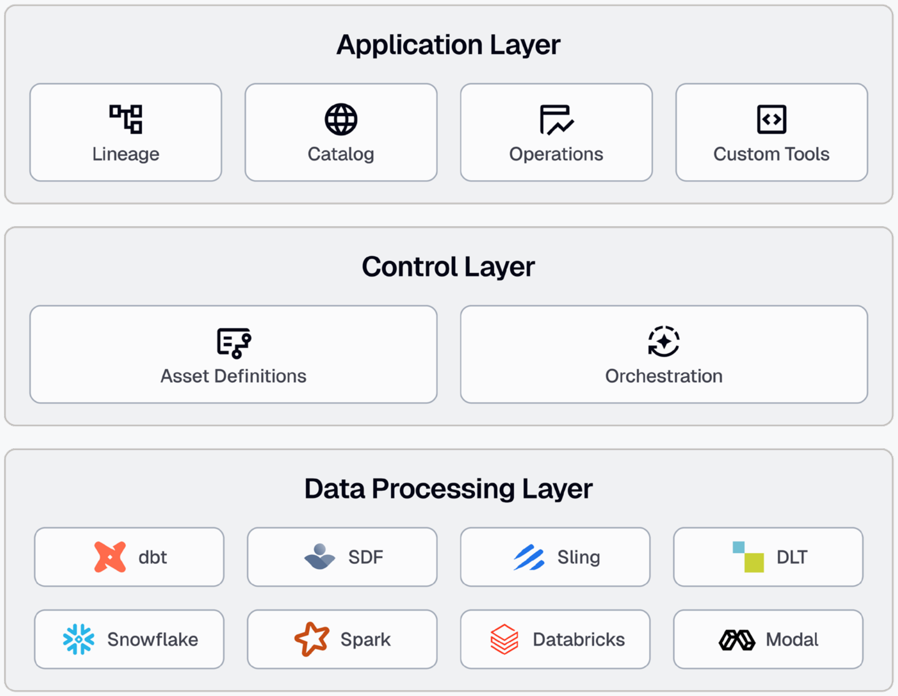
# The Control Plane

The control plane is more than orchestration. For many orchestrators, the ability to schedule and initiate jobs is the bulk of what they provide. Data Platforms require more than that.

Traditionally, the control plane is used to configure and activate the data plane. This is perhaps the most critical piece of the data platform puzzle. It provides a standard of control and enables a single view of all inputs into and outputs of your platform.

The control plane extends beyond creating and modifying datasets; it can connect assets across various systems, resources, and tools. Knowing if your scheduled job failed or alerted you on a failed data quality check is only somewhat useful. Knowing the data lineage of that failure downstream is far more important than the simple observation that “it failed.”

This is the marriage of orchestration, metadata, and observability, and it is the core value-add for most data platforms. In many cases, this requires various tools, which must be integrated.



# Benefits of Data Platforms

The flexibility of an unbundled Data Stack can quickly evolve from a value-add initiative to a maintenance burden. Separate—though integrated—tools often develop opaque and brittle dependencies. An upstream change may have an unexpected impact far downstream.

Testing alone is not the solution to making sure various tools work well together. The true culprit is often disparate codebases and a disjointed developer experience. Many tools often mean many repos, poorly documented dependencies, and an increasingly complex system architecture which leads to siloing within teams and organizations.

These are common problems for many companies and familiar headaches for all data engineers.

Thankfully, a centralized data platform helps address these issues holistically with your team's and system's common language: code.

In software engineering there is the concept of **code encapsulation.** This idea is also prevalent in establishing a data platform. While the organization operates within a single data platform, individual contributors, like data scientists, analysts, data engineers, and machine learning engineers, can operate with freedom from within their own subdivisions of the platform.

**Note**

In the context of data engineering, **"code encapsulation"** refers to the practice of bundling together the functions that operate on, and generate, data into a single unit, typically in the form of modules or classes. This practice helps in managing code complexity, maintainability, and promoting reusability.

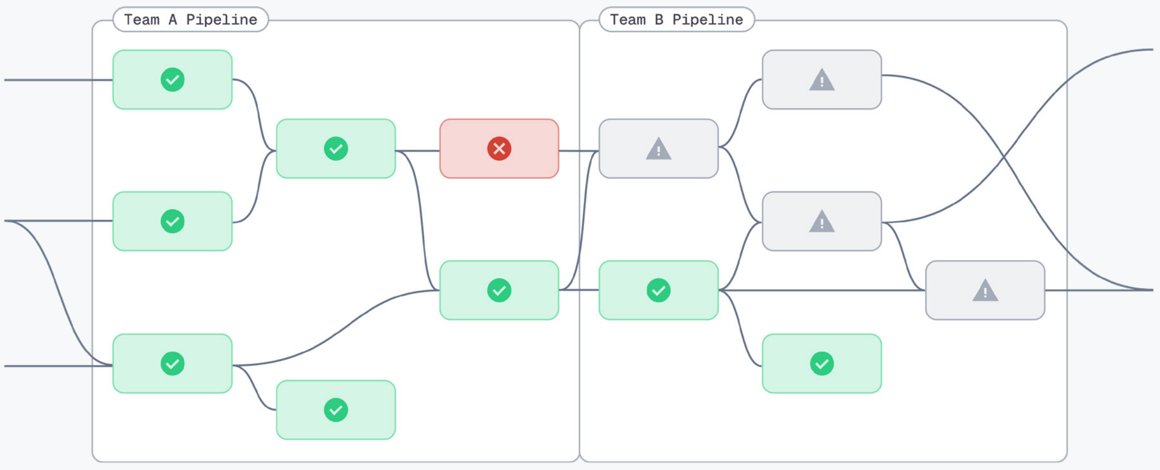
Data platforms help manage and orchestrate workflows, so it is reasonable to expect "code locations" to include both the logical components of a workflow and the configuration related to that workflow. Managing a variety of jobs in disparate systems makes this incredibly difficult. As the complexity of the jobs grows, this becomes increasingly difficult. A centralized platform that introduces standardized workflow patterns and provides a shared interface for multiple jobs is a primary benefit of establishing a data platform. The enforcement of standards by the platform owner enables them to ensure improved validation and support of the platform as a whole.

## Observability and Developer Experience

Observability and developer experience are two other key benefits of a data platform, alongside code locations and encapsulation. They are also the common threads that help elevate your pipelines from miscellaneous code sets to a stable and mature platform.

We'll explore these topics in later chapters, but observability can take multiple forms. At its core, observability is the ability to reasonably inspect what is happening on the platform. This includes changes to your system inputs, changes in the outputs over time, and other details about the inner workings of your platform. The Data Visibility Primer is another great resource on this topic.

These elements contribute to the developer's experience on the data platform. The biggest pitfall of data platforms - or any large data infrastructure initiative - is an inconsistent experience. This applies to both the creation of new features and datasets within the platform and the maintenance of existing features and datasets. Without the ability to share standards across your code locations, and have a clear view of the workings of the platform, your developer experience will suffer. As a result, your team's velocity will suffer, and your data platform will likely fail to provide the value you expect.



# 02 Architecting Your Data Platform

## Data Platforms often comprise many tools.

Independently, these tools solve discrete problems. Collectively, however, they provide the means to solve broader pain points common to data teams and technical organizations: inconvenient data silos, poor data quality, and ever-changing stakeholder requirements can make delivering value difficult. But, these challenges can be addressed holistically - and preemptively - when architecting your data platform.

Strong consideration for system design at the inception of your project is crucial, as early architecture decisions can influence the effectiveness of your platform: for example, risk of vendor lock-in, scalability limitations, and maintenance overhead. Design decisions and tooling selections should also scale with your team's and your company's growth. It can seem daunting when taken at face value, but established design patterns make this process much more manageable.

This chapter will review the configurations, tooling, and architectures that the data team should consider. Whether starting from square one or rearchitecting a legacy system, you can set yourself up for success by being intentional and thoughtful about the overall system.

## Scaling with the Business

**Data Platforms need to grow and evolve with the business. If data platform development stagnates while the business is still changing, it's likely a byproduct of poor architecture and overly rigid design.**

A centralized data platform is the best way to ensure stability and consistency across all your data initiatives. Designing an extensible platform will allow you to keep up with the business's changing demands.

The architecture of your platform establishes which kinds of data access patterns are possible for the rest of the organization. The data platform, and data teams, must be able to meet the needs of the numerous stakeholders across organizational teams. These can manifest as any number of data deliverables, whether those be dashboards, data exports or machine learning models. These expectations need to be understood early on. If you expect consumers to only interface with the platform through BI tools, you should facilitate a design that ensures data is readily available through that entry point.

If data is primarily shared via APIs or batch exports, your platform should support that as a primary feature. These entry points can evolve over time, but you should expect to support any platform's new entry point for the long term.

Making it easy for users to consume data through a BI tool can be a double-edged sword. Often, "self-service" is the goal, but without strong controls on structure and access, you introduce the possibility of misuse and misinterpretation of the data that's been made available. Centralized data platforms improve this through transparency of data usage, lineage to better understand data origin, and higher quality data through end-to-end pipeline validation.

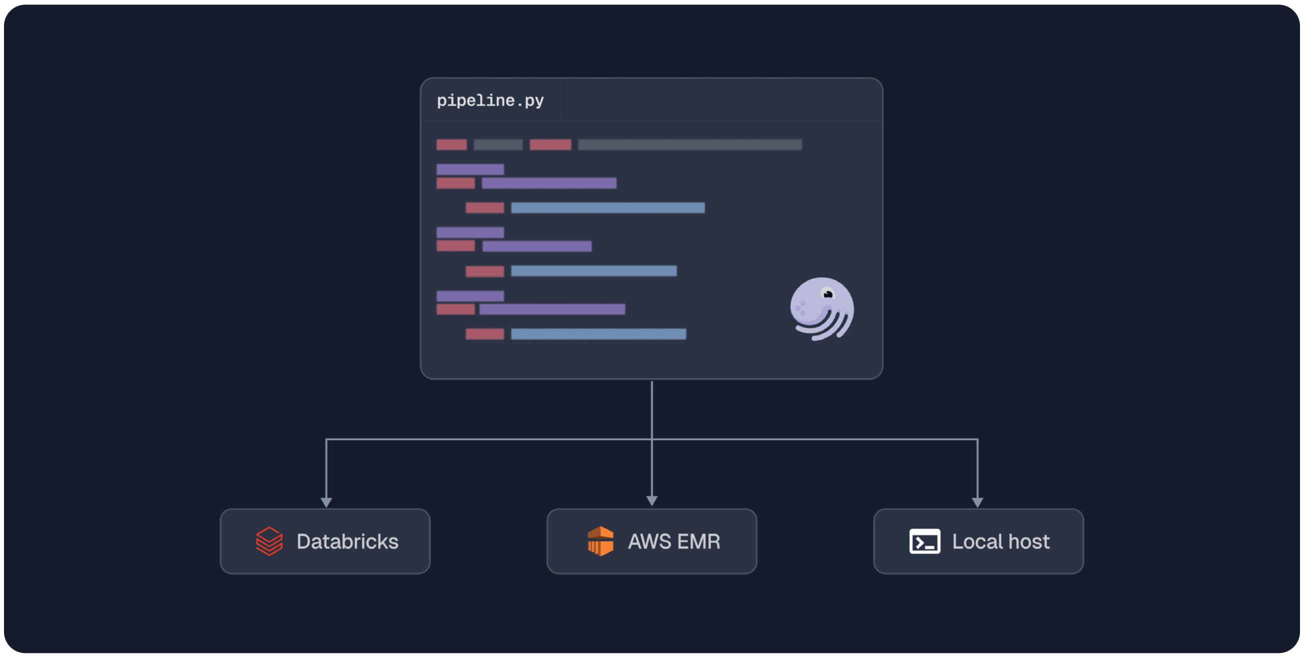
These details must be considered when designing your platform's internals and the entry points for your customers and stakeholders.

## Composability and Extensibility

Rapidly growing datasets, tightened SLAs, and prohibitive costs are common drivers for platform redesign and architecture overhauls.

A composable and extensible platform makes meeting these requirements possible without an expensive redesign. Requirements often change at organizations and the data platform needs to be able to react. A startup may find that as they grow and their data volume increases, their initial set of tools are no longer able to handle the additional traffic.

As the lifespan of the tools within a data platform are tied to the needs of the business, the goal should not be to avoid platform changes but to design a platform that makes operating under this constraint manageable.



## Composability and Extensibility

If you build an extensible and composable platform, you can continue to operate at a high level with minimal downtime. You also afford yourself the flexibility of updating your platform without migrating major system components.

Changing transformation tooling, pipeline design, and migrating data stores are common activities for any Data Platform Owner. The ease with which you can execute those changes depends on the platform's design.

What does an extensible platform look like in the real world? Often, it comes down to abstractions.

**Note**

In object-oriented design and software architecture, the principle to **"prefer composition over inheritance"** advocates for reusability and organization of code over heavy reliance on inheritance.  With inheritance code can become tightly coupled leading to fragility and inflexibility. Whereas with composition, there is a loose coupling between code references, thus leading to improved maintenance, and often clearer and more understandable code.

You probably have heard the phrase **"prefer composition over inheritance"**. in the context of software application design. This applies to data platform design in much the same way, in that the composition of a data platform typically comprises multiple purpose-built tools.

Often the composition of data tools is abstracted and hidden from the downstream analysts, and data engineers, allowing for these individuals to be more efficient in completing their tasks. Data engineering is software engineering after all, and the design and composition of tools, pipelines, and models, should be aligned with software engineering best practices.

Composability and extensibility are key components of a well-architected data platform, and something we will cover in more detail later in these writings.

## Common Data Architectures

Historically, data architectures were considerably simpler than what you see today. As the data needs of organizations have become more demanding, the underlying technologies have evolved to meet them: data needs to be more real time, with higher volume, and through more and more sophisticated modelling.

In recent years, a proliferation of tools has emerged to handle the specialized requirements of data processing and modelling, and a bundling of these tools has been defined with the name of the “Modern Data Stack”. Many teams have become frustrated with the modern data stack, and its many vendors, and  critics often stress that these new tools introduce undue complexity, and some of those concerns are warranted.

But there is no denying that the wide range of data tooling has lent itself to a range of architectures previously unavailable. Let’s look at some of those data architectures and critical components of each design.

01 Extract-Transform Load (ETL)

02 Extract-Load Transform (ELT)

03 Data Lakehouses

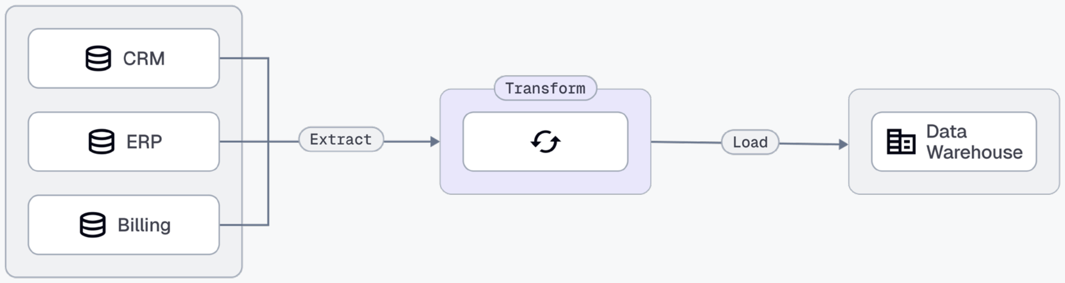
04 Event-Driven

## Common Data Architectures

### Extract-Transform-Load (ETL)

This architecture is the simplest and oldest approach of this list. Data starts from an origin datastore, often a production database, and is transformed in transit before being written to a final storage mechanism. Typically, you'll see aggregations and other denormalization applied to these datasets before they are written to their final destination.

By design, the state of the data in its "final" form is different from what is extracted from its origin. This design is still commonplace, and can be quite effective at reducing compute usage at the destination. The tradeoff is that you may lose the granularity of data, and your downstream usage is limited by the logic applied in transit.



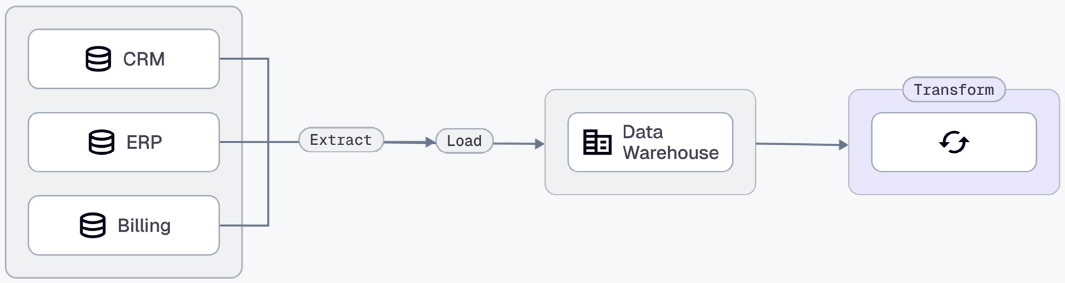
## Common Data Architectures

### Extract-Load-Transform (ELT)

This design has grown dramatically in popularity thanks to the Modern Data Stack and its modular design. It has also become commonplace thanks to the advent of Massively Parallel Processing databases and their modern counterparts.

Instead of applying transformations to your data before it lands in your warehouse, you retain a raw copy of that data as it arrives. This design pattern lends to various extraction methodologies, including change data capture (CDC) and ingestion of data from APIs or file servers. As an example, in the case of CDC, all of the operations that are performed on a dataset, like a table in Postgres, are captured and loaded into storage. Then, that data can be re-constructed in the transformation step, reconstituting the original dataset from the operations that took place. Comparing this to “ETL”, the raw change capture would be lost as the transformation would take place as the data is loaded into storage.

Because the raw data is retained, it is possible to recreate the transformations and models of this data, ultimately providing more flexibility. But, this also results in additional computation, and increased storage costs.



## Common Data Architectures

### Data Lakehouses

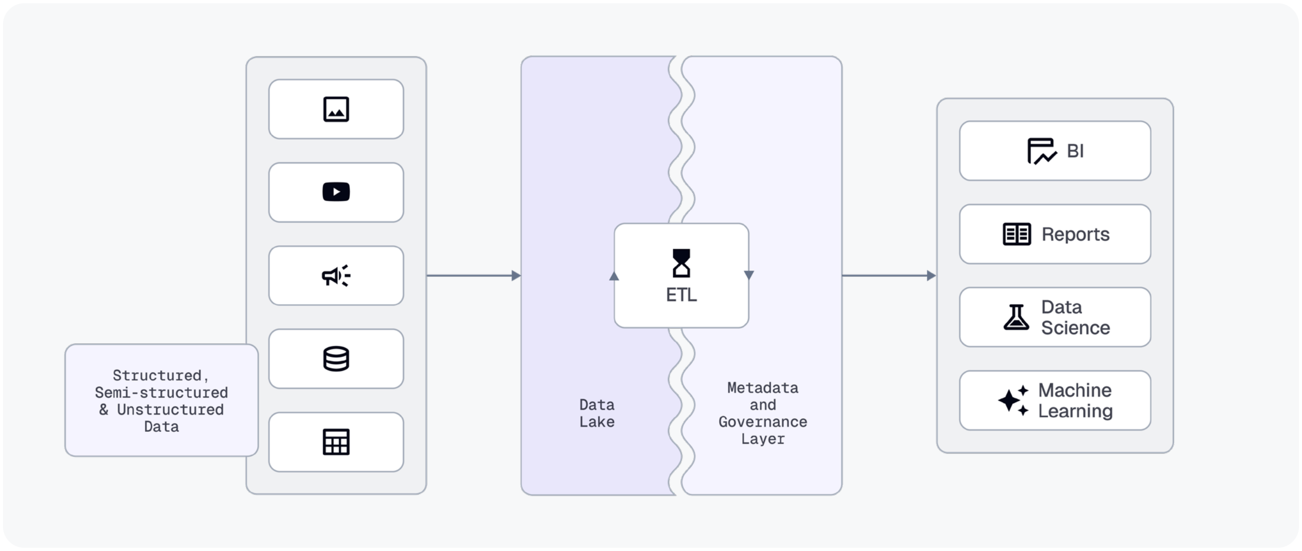
The Data Lakehouse architecture is relatively new, but its underlying components have been around for many years.

Data Lakehouses provide the scale of data lakes with the structure of data warehouses. Their design allows for large-scale data processing and storage, with most of the "heavy lifting" done outside the data warehouse.

**See a term you're not familiar with? Be sure to check out the Dagster data engineering glossary.**

Typically, data is written to columnar file formats like Parquet and stored in cloud storage like S3 or GCS. Recently, Iceberg, Delta, and Hudi formats have also been gaining popularity. In the case of Delta, this is an additional layer on top of Parquet in which additional metadata and transaction logs are included, enabling support for ACID (atomicity, consistency, isolation, durability) transactions.

These files typically follow a Medallion Architecture, which we'll cover in more detail later in this book. These systems often use specialized analytics engines, such as Spark, to transform the data in these files into clean and consumable datasets. The final output is often written for use in a data warehouse or otherwise exposed to downstream consumers.



## Common Data Architectures

### Event-Driven

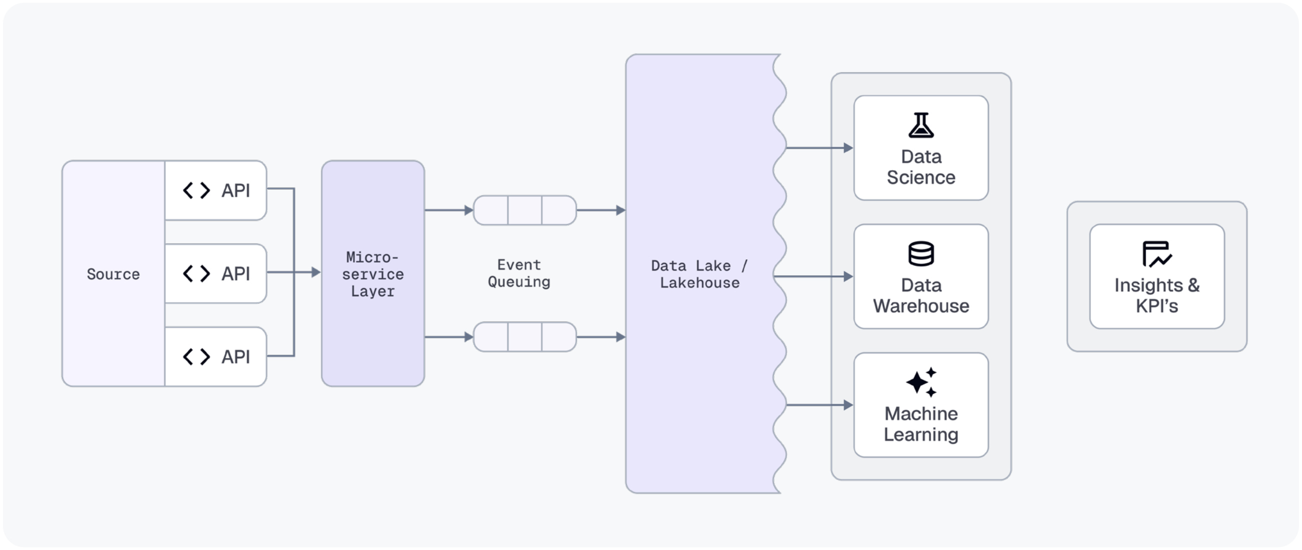
Streaming and "real-time" data platforms blend event-driven design patterns with some of the components of the architectures mentioned above.

In an event-driven paradigm, services exchange data and initiate their workload through external triggers rather than relying on the coordination provided by a signal orchestrator. Some examples of triggers include messages landing in a Kafka message queue, a call to a web hook, or an S3 event notification.

These architectures often use specialized tools to enable steam processing for real-time and rolling-window transformations. This approach is often paired with the Lakehouse architecture to serve analytical workflows for a wide range of SLAs.

Often streaming can be paired with batch processing, in that rolling computations can take place on the event stream, and persisted events landing in cloud storage can be later batch processed for additional reporting.

Some considerations of stream based processing is the expected cost of long-running compute, the additional complexity of event processing, and the requirement for specialized tools that may not conform to the existing tools being used at your organization. However, when real-time analytics are required for insights that need to happen fast, like fraud detection, then streaming is a fantastic option.



# 03

# Design Patterns

# Design Patterns for Data Pipelines

As we know, addressing your business's data needs is at the core of your data platform. To do this, we need to capture and process data. Pipeline designs may be constrained by the tools you use within your platform, but don't be fooled by shiny or new technologies. There are only a finite number of data pipeline patterns that you can use to find, ingest, and process new datasets for use downstream.

Countless tools can help you process new data, but only a handful of fundamental patterns exist.

Whether your data platform is entirely nascent or quite mature, the pipelines powering your platform can fall into one of these three categories:

01 Push
02 Pull
03 Poll

There may be layers of abstraction built on top of these patterns, but the fundamental implementations will fall into one of these approaches. We won't be going into much depth in this chapter about pipeline abstractions, but you can find more details about Dagster's approach to abstractions here.

These categories are not unique to data engineering; any software engineer who has worked on system integrations, consumed webhooks from a third-party application, or interacted with a REST API should find these familiar. However, certain patterns can be found quite frequently within a data platform, which we have seen through our experiences implementing data platforms at various organizations, and through relationships with data tooling customers.

## Design Patterns for Data Pipelines

**Push**

For pipelines that are based on the push methodology, the consumer is required to wait for data to be pushed from a source system; the destination could be a location in cloud storage, an API endpoint, or a database.

For push-based systems, the consumer can dictate the schema of the data that is being received, however, a common complexity in these systems is synchronization of schema definitions between producers and consumers. In these cases, schema registries can come in handy for defining a single source of truth for the expected format of data.

An example pipeline that follows the push access pattern may look like this:

01 A process for the producer starts at a scheduled time or preset interval

02 The producer performs some form of processing or creation of data

03 The producer pushes their datasets to a storage location like S3, GCS, or an SFTP Server

04 The consumer ingests that new dataset on their own schedule

**Pull**

Pipelines using a pull methodology have a slightly different definition of ownership. Most notably, the consumer dictates the cadence of ingestion. Typically, this happens on a predefined schedule, though it can also be extended to use an event-driven trigger. We'll look at how this would work next.

With pull based data access patterns, the consumer pulls the data based on the access pattern of the producer. For example, with a REST API, the consumer may have to perform pagination, and supply query parameters in a very specific structure. For replication of data in a database, the consumer would be responsible for defining the SQL query for pulling data, along with the logic for incremental loads. Note that tools like dlt and Sling support this behavior out-of-the-box.

An example pipeline that follows the pull access pattern may look like this:

01 The process starts at a scheduled time or preset interval

02 Consumer job fetches new data available

03 Consumer processes new data as needed for integration into the platform

## Design Patterns for Data Pipelines

**Poll**

The final flavor of the pipeline uses a polling paradigm. It can look similar to the pull pattern, but this implementation typically pairs well with event-driven architectures. It requires the consumer to maintain the state in ways not necessarily needed with other designs.

Polling implementations transfer the cadence of ingestion to the consumer. Maintaining a stateful pipeline - one knowledgeable of what data it has processed - is critical! Without that knowledge, subsequent runs of the pipeline may result in duplicate processing or unexpected downstream changes.

Here is a typical poll-based pipeline trigger through an external webhook:

01 Messages are frequently being published to a Kafka message broker

02 The consumer polls this message broker, and processes the records

03 It keeps an identifier or “cursor” of the record that has most recently been collected

04 The consumer then polls again, at a regular interval, providing the “cursor” to only retrieve new records

# Idempotency

Many of the design patterns that were discussed above benefit from following an idempotent design. Idempotency is the idea that an operation, for example a pipeline, produces the same output each time the same set of parameters are provided. This is often a great practice in designing data pipelines as it creates consistency in processing jobs, but it is especially important for the poll based data access pattern.

Deterministic systems in software engineering is the expectation that the output of a function is only based on its inputs. There are no hidden state changes as a result of calling that function.

For example let's look at two snippets of Python code, one which is idempotent, and the other that is not.

In this example, the non_deterministic_ function is taking a location and destination parameter, and writing the current temperature to a file. Each time this function is called, the output will be different, as it relies on an always changing metric being returned from the MyWeatherService.

Now, let's compare that to something that is idempotent.

**Not Idempotent**

Python

```python
def current_temperature(location: str) -> int:
      return MyWeatherService().get_current_temperature(location)

def non_idempotent_function(location: str, destination: str) -> None:
      with open(destination, 'w') as f:
            f.write(current_temperature(location))
```

**Idempotent**

Python

```python
def get_temperature(timestamp: str, location: str) -> int:
      return MyWeatherService().get_temperature(location, timestamp)

def idempotent_function(timestamp: str, location: str, destination: str) -> None:
      with open(destination, 'w') as f:
            f.write(get_temperature(timestamp, location))
```

## Idempotency

This example differs in that our idempotent_ function requires a timestamp parameter. So instead of getting the temperature at a given location at the time the function is called, we are responsible for explicitly providing the time, and each time this function is called with that specific timestamp, it should return the same value.

Practically, you want your pipelines to produce the same result given the same context. Making this happen - or failing to do so - is often at the root of pipeline complexity.

The means of doing this can be relatively simple. At its most basic, this may mean recording IDs, filenames, or anything that uniquely identifies your input data that has already been processed by the data pipeline. This can often be done by leveraging the context of an orchestrator, for example, the trigger time of a pipeline, or the categorical partition data that can be used as a parameter to your processing code.

For instance, when running a batch pipeline daily, each execution would not look at the last 24 hours of data. Instead, it would use a specific 24-hour period between two specific “bookends.”

By using an explicit window of time rather than something relative to the execution of the job, you remove one potential cause of pipeline drift. Even the best orchestrators can be resource-constrained. Also, reducing your exposure to slightly delayed jobs is a small but essential implementation detail as your platform grows.

Python

```python
@asset(
    partitions_def=DailyPartitionsDefinition(start_date=»2024-10-01»)
)
def daily_partitioned_events(context: AssetExecutionContext)
    «»» Each execution of this asset is configured to be tied to a specific
    day, thanks to the DailyPartitionsDefinition. Subsequent
    materializations of this asset are each tied to a specific day,
    meaning executing them more than once should result in the same
    output
    «»»
       partition_as_str = context.partition_key # 2024-10-01
       # do work on the data relevant for this specific partition
       # this may include using this partition key within a WHERE clause
```

# 04

# Data Modeling

Some patterns are universal when it comes to data architecture and data modeling. Both topics are incredibly vast - and beyond the scope of this chapter. But whether you are implementing a Data Lakehouse, Data Lake, or Data Warehouse as the central data store for your platform, you will find many similarities in their structures.

You've probably heard of the Medallion Architecture. While the term may be most commonly associated with Data Lakehouse architectures, the core concepts are essentially the same as those commonly found in Data Warehouses that follow an ELT design.

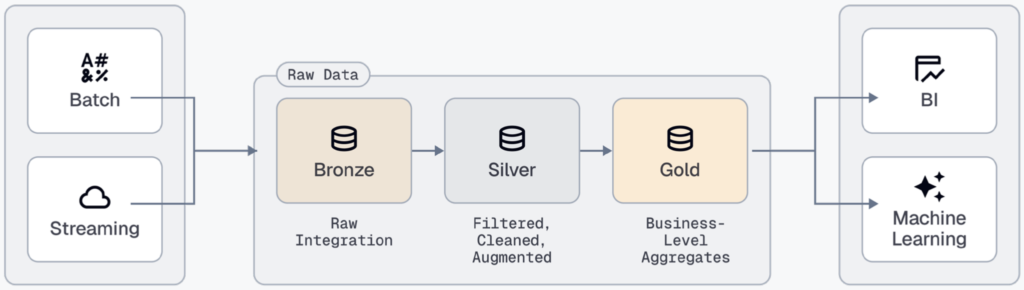
A Medallion Architecture's fundamental concept is to ensure data isolation at specific points in its lifecycle. You'll often hear the terms “bronze”, “silver” and “gold” used to describe this lifecycle.
In a data warehouse context, you may see “staged”, “core” and “data mart”. Some others use “raw”, “transformed”, and “production”. Regardless of the terminology used, the buckets have semantic and functional significance.

| | |
|---|---|
| Bronze | • Raw data is stored as it is received. Ingested data may be schemaless<br>• Often called a “Landing Zone” |
| Silver | • Cleaned and structured datasets derived from the Bronze layer<br>• Primary validation layer for data quality |
| Gold | • Consumer-ready datasets<br>• May include pre-processed aggregates and metrics<br>• Consumers may be business users, applications, or other data stores |

Consumers typically want clean, well-structured data. This may not be the case when working with LLMs or other flavors of ML models, but business use cases always expect clean and structured data.

A Medallion Architecture makes this relatively easy to manage and understand. This general pattern is very good at reducing the downstream impact of reporting through separation of data ingestion, cleaning, and consumer-ready datasets; it also pairs with data governance and data security. Compartmentalization is a core concept.

Lakehouse architectures typically apply this structure to files stored in cloud storage services like S3 or GCS. But, this structure is not always necessary, and can be overly complex for many businesses. Instead, you can apply these same logical distinctions within your data warehouse. Companies following a Lakehouse architecture often push the Gold layer into a data warehouse for easier consumption by BI Tools and business users.



Where the Lakehouse architecture uses structured cloud storage service buckets, a data warehouse would instead segregate these datasets by schema.

Modern warehouses like BigQuery, Databricks, Snowflake, and Clickhouse are blurring the boundaries between Data Lakehouse and Data Warehouse. Regardless of your technology choice, segregating your data based on how well it has been validated, cleaned, and transformed is always a good decision.

A bonus of this structure is the separation of reads and writes. Though it adds some overhead, separating these two operations allows for a more stable platform for a few reasons. This separation sets the foundation for more complex CI/CD workflows like Blue/ Green Deployments. It also enables our platform to be resilient to upstream changes. You'll be hard-pressed to find a data engineer who has not dealt with unexpected source data changes or has not had to combine “legacy” and “modern” datasets into one table. Doing so without separating reading data and writing data is very difficult.

Hopefully, it will seem obvious that raw data can be captured in one place and materialized as clean data elsewhere. This single design decision will save you many hours of future headaches. Remember, your current data architecture has a finite lifespan and will evolve. You also should expect your input data to change over time. This is especially true when consuming data from a transactional database. Isolating raw data allows you to abstract away the processing needed for handling miscellaneous edge cases or combining legacy datasets with their current versions. You cannot always rely on upstream systems to provide consistent input data; instead, you should design your data platform to accommodate these inevitable changes.

## Designing a Well-Configured Platform

A well-configured platform makes building a composable and extensible data platform easier, but what constitutes “well-configured”?

Regardless of your data platform’s stage of maturity, you should be designing a system that addresses specific use cases for your team. This will require supporting common functionality typically found within mature software products. Some of that functionality may be bespoke, some from OSS tools, and some from paid vendor tooling.

Well-configured platforms avoid vendor lock-in. To a lesser degree, they may also avoid “tooling” lock-in. In the extreme case, they may even limit your dependency on any single tool in the entire system.

Avoiding lock-in applies to paid software solutions and open-source tooling choices. There are plenty of “modern” data tools, and every tool category has many options. Maintaining your ability to change data pipeline vendors, transformation tooling, or reporting and metrics layers provides a reasonable level of “future-proofing” without requiring a large upfront investment.

So, what components should we consider when designing our future-proofed data platform?

At a high level, your data platform is likely to incorporate tooling for the following components:

01 Orchestration
02 Data Cataloging
03 Storage
04 Replication
05 Ingestion
06 Transformation
07 Observability
08 Data Versioning

## Designing a Well-Configured Platform

You might find that a single tool covers more than one of these categories. But you can certainly find specialized tools for each component as well.

Additionally, as platforms evolve, you may find that multiple solutions are required for any given category. For example, you may require storage in multiple cloud providers, various compute platforms to handle the transformation of data with differing volumes, or different technologies for replication of data from varying sources.

Each of the components of our well-configured platform has many options. Let's run through some popular choices for each category:

### 01 Orchestration

Orchestrators vary greatly, but they all allow you to automatically execute a series of steps and track the results. Typically, they coordinate the inputs and outputs of your platform.

**Popular Choices:** Dagster | Airflow | Kestra | Prefect

### 02 Data Cataloging

Observation and tagging of your data is crucial so that stakeholders and analysts can better understand the data that they use day-to-day. It is common for the catalog to be a joint effort between the data platform team, and the data governance team, ensuring that cataloging is standard, secure, and compliant to organizational policies.

**Popular choices:** Atlan | DataHub | Amundsen | Apache Atlas

## Designing a Well-Configured Platform

### 03 Storage

Storage can take many shapes, including databases, warehouses, blob storage, and various file formats. Depending on your compute engine and how you want to perform transformations, some options are cheaper and better suited.

| | | | | |
|---|---|---|---|---|
| **Popular Warehouses:** | BigQuery | Snowflake | Clickhouse | DuckDB |
| **Popular Cloud Storage:** | AWS s3 | Google Cloud Storage | Hertzner Storage Box | |
| **Popular File Formats:** | Parquet | CSV | JSON | Avro |
| **Popular Table Formats:** | Iceberg | Delta | Hudi | |

### 04 Replication

If you want to get data out of one storage location and available in another, you'll need some replication mechanism. You may be able to do this with some custom code run on your Orchestrator, but there are many purpose-built options to use instead. Change-Data-Capture is a common choice for database replication

| | | | | |
|---|---|---|---|---|
| **Popular Choices:** | Debezium | Estuary | Fivetran | Stitch |
| | dlt | Sling | CData | |

## Designing a Well-Configured Platform

### 05 Ingestion

Many Replication tools also work well for general purpose Ingestion, but it's important your ingestion tooling supports a wide range of inputs - APIs, different file formats, and webhooks are only a small sample.

**Popular Choices:** Airbyte, dlt, Fivetran

> **Note**
>
> The lines can blur between **replication** and **ingestion** tools, and some tools may be suitable for both!

### 06 Transformation

The data coming in from your Replication and Ingestion tools will most likely need to be cleaned, denormalized, and made ready for business users. Transformation tooling is responsible for this. While any language is appropriate, SQL is the most common.

**Popular Choices:** dbt, SDF, SQLMesh

### 07 Observability

Your Orchestrator should provide some observability features, but purpose-built tooling goes a layer deeper. These tools are typically used to proactively monitor your input and output data and alert when anomalies or expensive access patterns are found.

**Popular Choices:** Metaplane, Datafold, Soda, Monte Carlo

## Designing a Well-Configured Platform

### 08 Data Versioning

When working in different environments, or as your data evolves, it can be nice to maintain a history of the changes being made. Data versioning tools can provide a layer on top of your data storage to enable this functionality.

**Popular Choices:** Lake FS | dvc

It's important to note that a given data platform may not need a tool for every one of these categories, but all data platforms utilize tooling - either vendored, open-source, or home-grown - to address these areas of concern.

These tools, however, are not terribly useful on their own. By themselves, they are specialized technology choices that may not fit well within your organization. But, when combined into a data platform, they form a mutually exclusive and collectively exhaustive set of features that provide a strong foundation for platform development.

The key to going from just a collection of tools to an effective platform is incorporating these tools within well-understood design patterns and architectures. And that topic is coming up shortly.

## Must have features

There are many ways to design a data platform, and many pieces that make up that whole. And, though there are plenty of tools available, some features should be considered standard.

These quality of life features that should be required within all data systems, and are often the differentiator between a robust data platform from a collection of one-off scripts:

- **Logging**
- **Retries**
- **Backfilling**
- **Self-healing pipelines**

These features can drastically improve the reliability of your data platform, for example, some upstream data producers or teams can be unpredictable, or, outages can occur for services or compute environments. Without having these common features, the trust in your data and data platform can be significantly tarnished.

Restarting failed pipelines and services, inspecting logs, and backfilling datasets are common, repeated tasks familiar to every data engineer. Before you jump into designing how you'll process your data inputs, you need to address these fundamental pieces. Often these fall under the purview of your orchestrator, but not always.

These tools help you support the evolution of your technology stack without compromising the standard of software your team expects.

You would seldom release an application to production without runtime logging or observability tooling. Data platforms are no different.

It should be clear by this point that there are many parts that make the "whole" of a data platform. Whether tools, pipeline design or general data architectures, there is no shortage of options or decisions to make.

But don't get confused - many of the tools and topics we covered in this book complement one another. The data pipeline designs we covered are not mutually exclusive, and can be used alongside one another. Most of these patterns are just specific names for familiar and common activities found in countless software applications.

Designing an effective data platform from scratch can be challenging, and there are many contradictory examples online. But rather than focusing on tool selection and losing the forest for the trees, it's critical that you recognize the components of your data platform are tools to solve various problems.

Often, teams in and across organizations are solving very similar problems. While the domains and datasets may be different, there is a common pattern to the architectures and design principles that are used to build a robust data platform.

# 05 Data Quality

A core component of data platforms is data quality, or more appropriately, a framework for enforcing data quality standards on the data produced by the pipelines in the data platform. This is because data quality metrics are indicators that are used to determine if data is suitable for use in decision-making.

> **Data is used to both run and improve the ways that the organization achieves their business objectives … [data quality ensures that] data is of sufficient quality to meet the business needs.**
>
> **David Loshin**
> **The Practitioner's Guide to Data Quality Improvement**

Having a framework defined for how to enforce data quality, along with a set of standards for what is considered acceptable is a crucial step in building a data platform that an organization can trust. Defining these standards often originates from the governance team. However, it is often a cross-organization and multi-team collaboration for well-defined data quality standards. Many rules come from the stakeholders who have expertise in the underlying data and reporting needs, but it is often common for these stakeholders too to not have a full understanding of the origin of their data. This is why it's important for the definitions of data quality to be a result of a team effort.

# The Dimensions of Data

There exist 6-commonly defined dimensions for defining rules of data quality. These include: **timeliness, completeness, accuracy, validity, uniqueness** and **consistency.**

These dimensions can be referenced in defining rules of data quality and implementing quality checks.

## Timeliness

Timeliness refers to how up-to-date data is. If data is being produced by an upstream system every hour, and the downstream models haven't been updated in the past week, then this report is not timely. On the contrary, if you have data being produced hourly, and the replication and modeling of that data triggering just after it is produced, then that is a timely pipeline.

**Example:** An organization expects a financial report to land in an S3 bucket on a weekly basis, however, this report hasn't been received for the past month

**Impact:** Data not received within the expected latency window can result in operational impacts and inaccuracies in analysis and reporting

| date_week_end | date_report_received | status |
|---|---|---|
| 2025-01-01 | 2025-01-01 | ON_TIME |
| 2025-01-08 | 2025-01-08 | ON_TIME |
| 2025-01-15 | 2025-01-15 | ON_TIME |
| 2025-01-22 | NULL | MISSING |
| 2025-01-29 | NULL | MISSING |

## The dimensions of data

### Completeness

No required field should be missing for a record of data, whether that be a column in a database table, or an attribute of new-line delimited JSON. If an attribute is indicated as optional, then completeness may not apply, however, validation of these missing records may be more complex, as completeness still applies to them, but only when they are populated.

**Example:** Customer record — first name, last name, e-mail address

**Impact:** Missing customer information can result in skewed analysis

| user_id | first_name | last_name | email |
|---|---|---|---|
| 1 | Augustus | Gloop | gloopinator@gmail.com |
| 2 | Violet | Beauregarde | NULL |
| 3 | Charlie | Bucket | golden_tix@gmail.com |

### Accuracy

Data values should be accurate to the expected numerical or categorical values for a given data point.

**Example:** A medical record indicates that allergies of a given individual, this data should be consistent and accurate across systems, and be an accurate representation of the real world

**Impact:** Inaccurate data can significantly impact downstream reporting and real-world processes

| first_name | last_name | allergies |
|---|---|---|
| Augustus | Gloop | Chocolate |
| Violet | Beauregarde | Blueberries |
| Charlie | Bucket | - |

## The dimensions of data

### Validity

Categorical data entries must adhere to an expected list of values. Values must match the structure or format, for example, pattern matching, or schema.

**Example:** A bank maintains a table of customer accounts, with an account type field that can either be checking or savings, but a value is incorrectly entered as loan

**Impact:** Failures and errors in transactional processing

| user_id | account_type | balance |
|---|---|---|
| 13ed536c-851e-41ae-8cf1- | checking | 530.44 |
| 4d5829a4-700f-419b-8b2f- | checking | 2.50 |
| ac3ea5af-92d6-48fe-a81a- | loan | 3.75 |

### Uniqueness

Data values must be free of duplicate values when there are not expected to be any, for example, for values that are primary keys in a relational database.

**Example:** An online retailer maintains a list of products with the expectation that Stock Keeping Unit (SKUs) are unique, but there is a duplicate entry.

**Impact:** duplicate SKUs could result in inventory tracking errors, incorrect product listings, or order fulfillment mistakes.

| sku | name | price_usd | weight_lbs |
|---|---|---|---|
| CW21001 | Fizzy Lifting Drinks | 12.99 | 0 |
| CW21001 | Everlasting | 4.95 | 0.05 |
| CW21002 | Magnet Bites | 6.72 | 0.10 |

## The dimensions of data

### Consistency

Data is to be aligned across systems and sources–often tied to replication of data from a source system to a data warehouse.

**Example:** A data team replicates a postgres table to delta live tables for analytics, however the schema of the schema of the replicated table incorrectly uses an integer value whereas the upstream data uses floats

**Impact:** Incorrect reporting and analysis of data can occur resulting in invalid metrics

**Source Dataset**

| measurement_time | temperature | humidity_pctt | pressure_inhg |
|---|---|---|---|
| 1722885084 | 92.50 | 78.44 | 33.40 |
| 1722898737 | 92.55 | 78.78 | 33.75 |

**Replication Dataset**

| measurement_time | temperature | humidity_pctt | pressure_inhg |
|---|---|---|---|
| 1722885084 | 92 | 78 | 33 |
| 1722898737 | 92 | 78 | 33 |

## Tools and Technologies

In enforcing data quality standards on a data platform, there are a wide variety of tools available within the data ecosystem. Some tools are better suited to specific data processing frameworks, for example the Deequ framework works particularly well with Apache Spark, whereas other tools are more general, for example Great Expectations which may work with a number of data tools and formats. However, the enforcement of rules should remain consistent across tools.

A non-exhaustive list of tools that we've found to be prevalent in data platforms includes:

| | |
|---|---|
| Soda | Data quality testing for SQL-, Spark-, and Pandas-accessible data |
| Great Expectations | General purpose data validation tool with built-in rulesets |
| Deequ | Library built on top of Apache Spark for defining "unit tests for data", which measure data quality in large datasets |
| dbt tests | Assertions you make about your models and other resources in your dbt project |
| Dagster | Asset checks |

A benefit of tools like Soda and Great Expectations is that they provide a large set of rules out-of-the-box. Instead of having to write custom logic for validating the 6 data dimensions that were previously mentioned, it is possible to apply the logic of already existing rules.

Ultimately, it's important to choose the data quality framework that is most appropriate for the other tools in your data platform, and it is perfectly acceptable to compose multiple tools that suit your needs. But it's important to design your platform in such a way that you have a central observation layer over these disparate tools and frameworks, likely done at the orchestrator, so that the stakeholders can have a strong understanding and confidence in their data.

# Enforcement of Data Quality

## The full data lifecycle

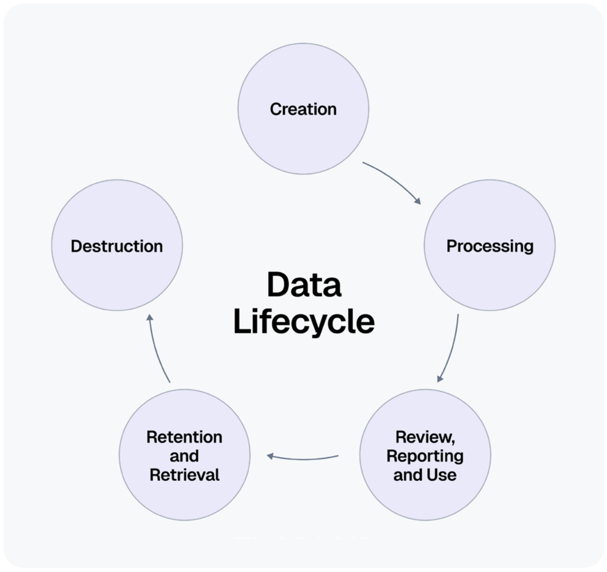
It should be noted that the enforcement of data quality should occur throughout all stages of the data lifecycle. This includes the application layer, data replication, the analytical layer, and reporting.

To provide an example, imagine you are a SaaS company that maintains a list of their users. The e-mail address that is provided by the user should be verified at the application layer, both the client side and server side.

When this data is replicated in an analytical warehouse, validation should also occur to ensure that the data that is present in the warehouse matches what exists in the source database table. It is important to ensure that the record exists and that no duplicate records have been created from some faulty replication logic. Additionally, while the e-mail address should be structured in a valid format for an e-mail address, for defensive data quality enforcement, the structure of the address should also be validated at the warehouse layer.

In the analytical layer, it is common for many downstream models to be produced that reference application data, like this user table. It's important to validate that the representation of this user is accurate and that there are no faults in how these models are being formed or how this data is being joined with other datasets. This can be particularly tricky as data from many datasets come together.

Finally, these models will likely be surfaced in some kind of way, whether that be through dashboards, some kind of alerting, or through reverse-ETL loaded back into the application database. It is important to validate that the data being surfaced meets the expectations of the business.



# Enforcement of Data Quality

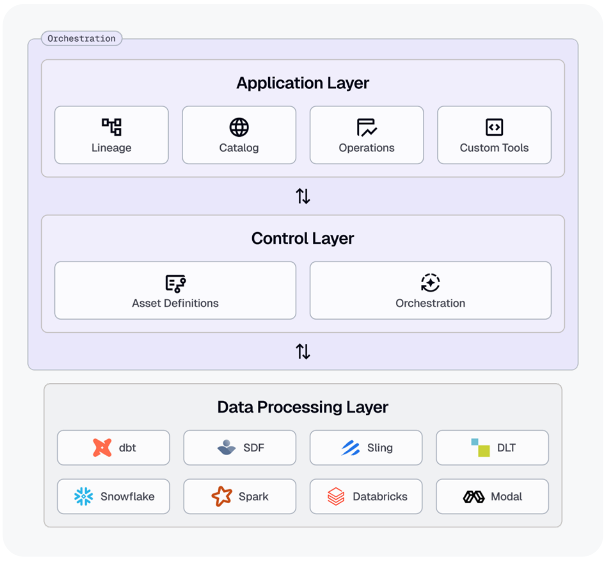
## A central control plane (orchestrator)

When constructing a data platform, having a central control plane, like an orchestrator, makes it much easier to enforce data quality standards at a high level and throughout all stages of the data lifecycle.

There are several benefits to this approach, one being that the data platform owner can act as the overseer to the standards. If the organization has a number of teams, each with its own set of rules, these can be consolidated into a single playbook for enforcing standards by the data platform owner through the orchestrator.

Additionally, the orchestrator is responsible for determining which jobs should run, if any data quality issues arise, it has control over the downstream jobs that may surface data to a dashboard or application. By having full visibility into the quality of data at the various points of its life in the orchestrator, reporting of this problematic data can be prevented.

Finally, the orchestrator is also home to system-wide alerting, along with updating of data in data catalogs. The orchestrator can keep the engineers and platform owners informed of problematic data in a standard way for on-call duties.

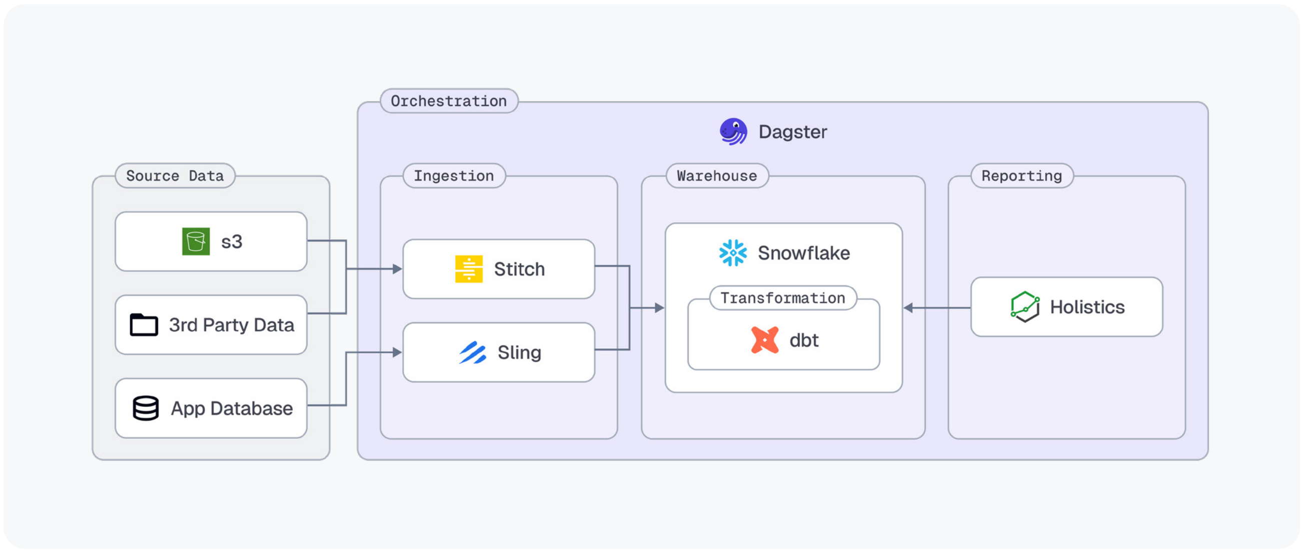


# 06 Example Data Platforms

To better understand how one might architect a data platform, let's walk through several example data platform stacks. In these examples, we outline the origin of data, the warehouse, and the orchestrator, how data is ingested, how it's transformed, and what is used for reporting. There are many ways to architect a data platform; however, these examples provide a referential starting point.

## The Lightweight Data Lake

| | |
|---|---|
| Source Data | S3, various third-party |
| Warehouse | Snowflake |
| Orchestrator | Dagster |
| Ingestion | Stitch, Sling |
| Transformation Framework | dbt |
| Reporting | Holisitics |



A hypothetical SaaS company is used by thousands of e-commerce stores–their core application uses MySQL for transactional data, and transactional events are captured and written to S3 via AWS Kinesis.

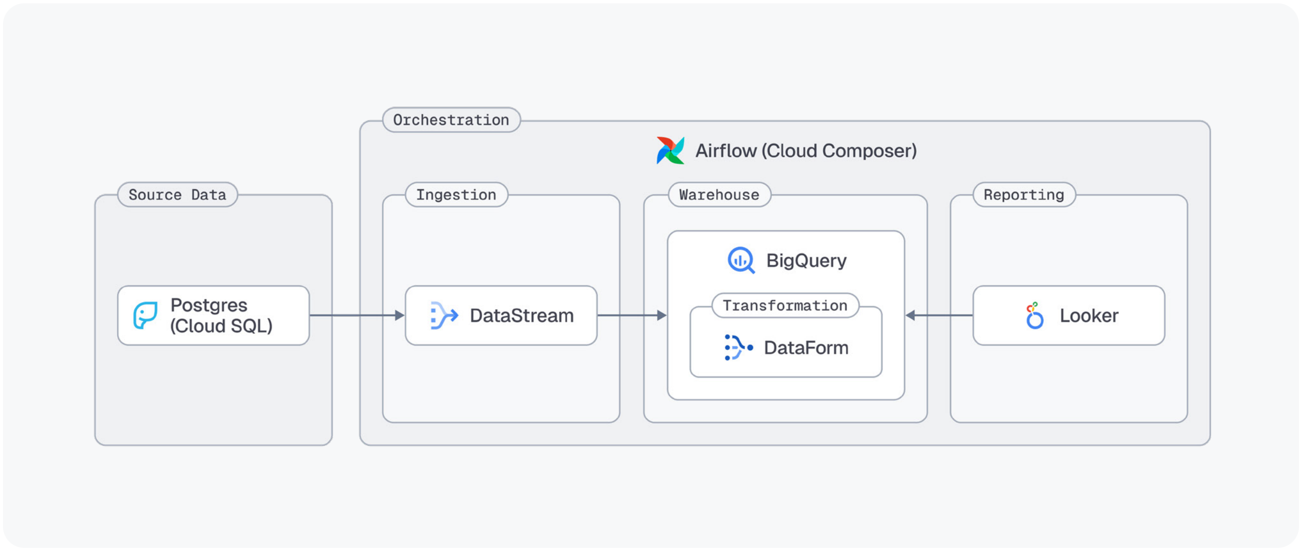
The company used Stitch to ingest third-party data, but relied on a custom pipeline orchestrated with Dagster to process and ingest the data in s3.

This traditional E-T-L method allowed for in-transit aggregation, reducing the compute load on the warehouse and limited exposure of PII in the pipeline. The decision to aggregate in-process was intentional, as downstream users were only interested in aggregate trends based on the event data.

Third-party data was processed in the typical E–L-T pattern.

# The GCP Stack

| | |
|---|---|
| Source Data | Postgres (Cloud SQL) |
| Warehouse | BigQuery |
| Orchestrator | Airflow (Cloud Composer) |
| Ingestion | Custom, DataStream |
| Transformation Framework | DataForm |
| Reporting | Looker |

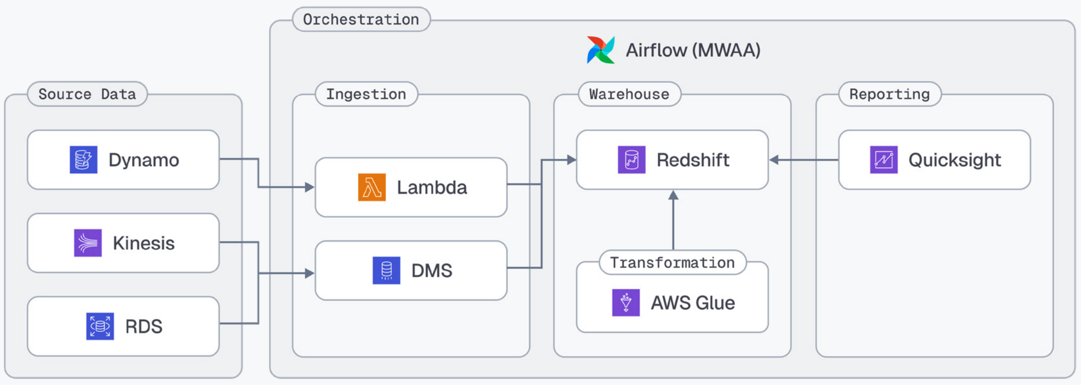


This company opted for a simple GCP-native architecture due to its existing usage of GCP. This stack leans heavily toward simplicity and convenience, favoring readily available tools rather than third-party vendors.

A common practice is to use Airflow (Cloud Composer, in this case) for simple E/L operations that rely on simple SQL queries. This works fine for scrappy, fast-paced development but can quickly get out of hand. The move to DataStream is a prudent one, specifically for replicating transactional database tables.

## The AWS Stack

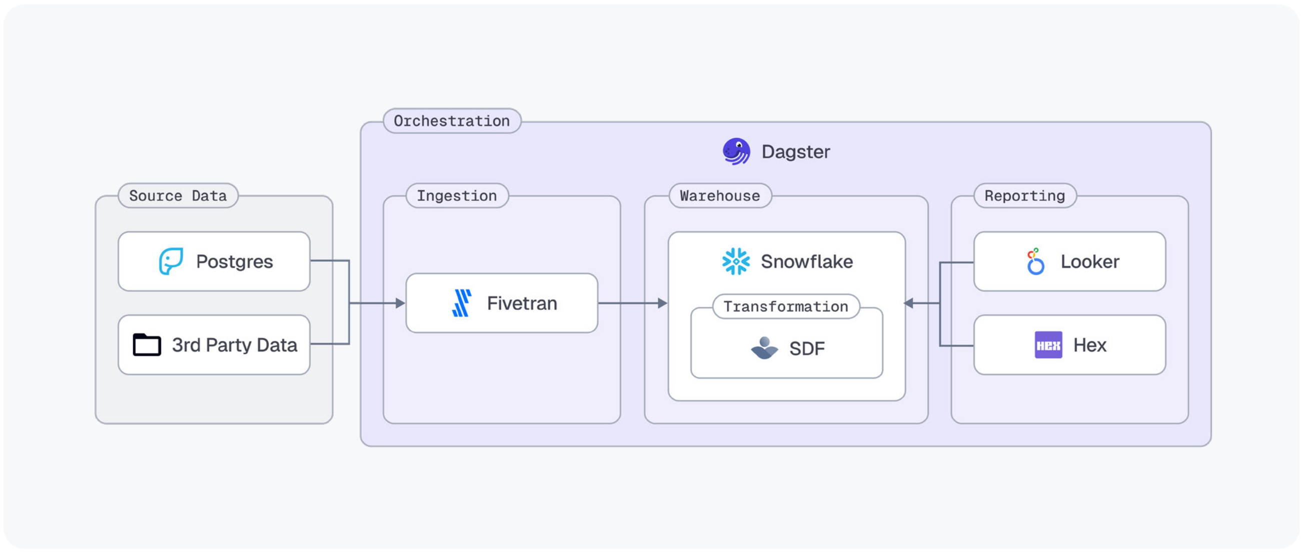
| | |
|---|---|
| Source Data | Amazon RDS, DynamoDB, |
| Warehouse | Redshift |
| Orchestrator | Airflow (MWAA) |
| Ingestion | Lambda, DMS |
| Transformation Framework | AWS Glue |
| Reporting | Amazon QuickSight |



In this stack we've elected to use the services explicitly provided by Amazon, with source data coming from DynamoDB and Amazon RDS, and replication occurring through custom Lambda processing and AWS DMS. Data is stored in S3 cloud storage, and transformed using AWS Glue. Finally, analysis and reporting occurs with Amazon Athena, and Amazon QuickSight.

## MDS by the Book

| | |
|---|---|
| Source Data | Postgres |
| Warehouse | Snowflake |
| Orchestrator | Dagster |
| Ingestion | Fivetran, dlt, Sling |
| Transformation Framework | SDF |
| Reporting | Looker, Hex |

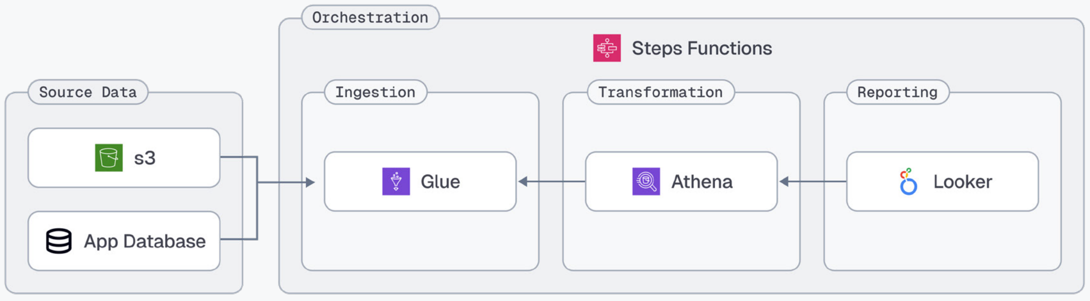


The "Modern Data Stack" is popular and used by thousands of companies, it is often a combination of cloud provider tools, open source frameworks, and data services. A common theme of the modern data stack is that it is a composition of several smaller purpose-built tools with an orchestrator acting as a unified control plane and single observation layer.

This stack falls squarely into the E-L-T pattern, with heavy reliance on third-party vendors. It is very fast to get started.

## Data Lakehouse

| | |
|---|---|
| Source Data | Postgres, s3 |
| Warehouse | - |
| Orchestrator | AWS Step Functions |
| Ingestion | AWS Glue |
| Transformation Framework | Spark, Athena |
| Reporting | Looker |



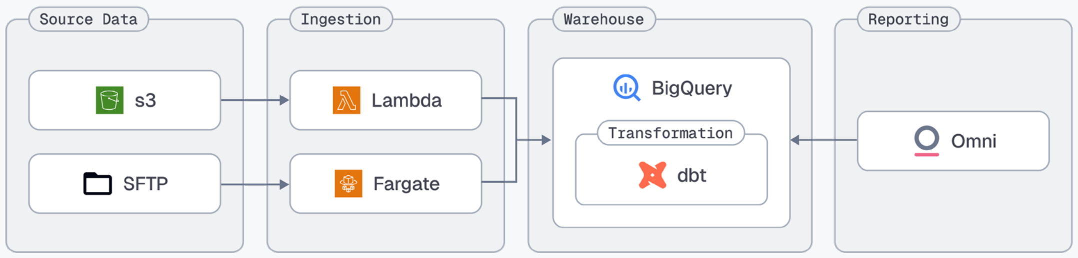
If you are dealing with extremely large data sets, you may be interested in foregoing a traditional data warehouse in favor of a compute engine like Spark. This company processed many millions of events on a daily basis and relied heavily on Spark for large-scale computation

In lieu of a warehouse, data in s3 was indexed by AWS Glue, transformed with AWS EMR (Spark), and consumed downstream through Athena.

This architecture is not complicated but it is a significant deviation from the data stacks relying on cloud warehouses for storage and computing.

## Event Driven

| | |
|---|---|
| Source Data | SFTP, s3 |
| Warehouse | BigQuery |
| Orchestrator | - |
| Ingestion | AWS Lambda, AWS Fargate |
| Transformation Framework | dbt |
| Reporting | Omni |



Event-driven design does not necessarily need an orchestrator; the events themselves “orchestrate” the execution of the pipeline. In this company’s case, files landed in SFTP and were eventually copied to s3.

s3 provides native functionality to trigger AWS Lambda functions when new files arrive. As an alternative, long-running processes were also executed via AWS Fargate when the AWS Lambda runtimes limits were not sufficient.

These files were ingested into BigQuery and ultimately transformed with dbt. End users consumed the data through Omni dashboards.

Event-driven architectures are a way to process data in a near-real-time fashion, but they have their downsides. Re-execution of the pipeline can be tedious and missed or dropped events may go unnoticed without sufficient checks in place.

The key consideration is that most of these components are run within a Docker container.

# Conclusion

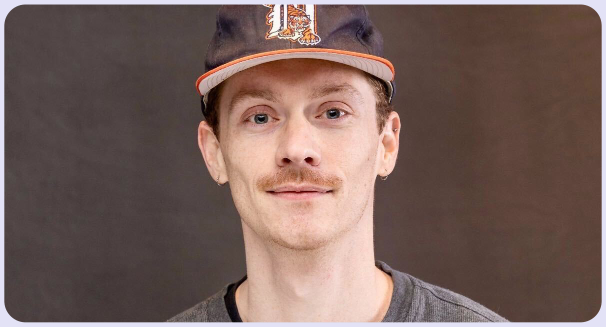
That concludes the first edition of Fundamentals of Data Platforms. We've covered many topics from an introduction to what a data platform is, common architecture and design patterns, important concepts like data quality, and we even dove into some examples of tools and architectures.

The data ecosystem is broad, but hopefully this introduction gives you the information that you need to get started in designing and building a data platform for either your organizations, or even if you're just tinkering with a side project.

Also keep in mind that tools in the data ecosystem are always evolving, but don't fret, as many of the core architectures and design patterns covered in this book have stood the test of time.

Thank you for taking the time to read Fundamentals of Data Platforms, and we hope you learned something new.

## Meet the authors



**Joe Naso**
Joe is a Fractional Data Engineer. He helps SaaS companies build reliable data platforms and monetize their data. He's previously managed multiple data teams at different startups. You can find him talking about the intersection of data and business on Substack, or you can give him a shout on LinkedIn.

**Colton Padden**
Colton is a data engineer, and developer advocate with experience building data platforms at fortune 500 institutions, government agencies, and startups. He is currently employed by Dagster Labs helping build and educate engineers on the future of data orchestration, and is a strong advocate for open source software and community.