O'REILLY®
Technical Guide

Understanding ETL

Data Pipelines for Modern Data Architectures

UPDATED EDITION

Compliments of
databricks

Matt Palmer

databricks

Lakeflow

Built to simplify ETL

Databricks Lakeflow is built to address the key challenges data engineering teams face today. As an end-to-end data engineering solution with built-in data intelligence, Lakeflow unifies and simplifies data ingestion, transformation and orchestration by offering a singular approach to data pipeline creation, empowering data professionals at every stage of ETL.

Ingest

with Lakeflow Connect

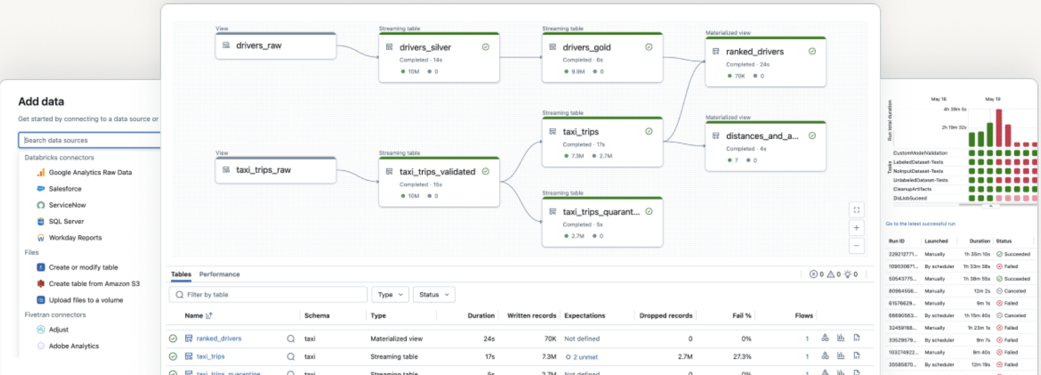
Transform

with Lakeflow Declarative Pipelines

Orchestrate

with Lakeflow Jobs

Learn more



Add data

Get started by connecting to a data source or

Search data sources

Databricks connectors
- Google Analytics Raw Data
- Salesforce
- ServiceNow
- SQL Server
- Workday Reports

Files
- Create or modify table
- Create table from Amazon S3
- Upload files to a volume

Fivetran connectors
- Adjust
- Adobe Analytics

Tables Performance ⊗ 0 ⚠ 0 💡 0

Filter by table | Type | Status

Name	Schema	Type	Duration	Written records	Expectations	Dropped records	Fail %	Flows
✓ ranked_drivers	taxi	Materialized view	24s	70K	Not defined	0	0%	1
✓ taxi_trips	taxi	Streaming table	17s	7.3M	○ 2 unmet	2.7M	27.3%	1
✓ taxi_trips_quarantine	taxi	Streaming table	5s	2.7M	Not defined	0	0%	1

Go to the latest successful run

Run ID	Launched	Duration	Status
229212771...	Manually	1h 35m 10s	✓ Succeeded
109030671...	By scheduler	1h 33m 38s	⊗ Failed
50543775...	Manually	1h 38m 55s	✓ Succeeded
80964556...	Manually	12m 2s	⊖ Canceled
61576629...	Manually	9m 1s	⊗ Failed
66690563...	By scheduler	1h 15m 40s	⊖ Canceled
32459168...	Manually	1h 23m 1s	⊗ Failed
33529579...	By scheduler	9m 7s	⊗ Failed
103274922...	Manually	9m 40s	⊗ Failed
35586870...	By scheduler	12m 19s	⊗ Failed

UPDATED EDITION

Understanding ETL

Data Pipelines for Modern Data Architectures

Matt Palmer

O'REILLY®

Understanding ETL (Updated Edition)

by Matt Palmer

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Gary O'Brien
Production Editor: Kristen Brown
Copyeditor: nSight, Inc.
Proofreader: M & R Consultants Corporation

Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Kate Dullea

March 2024:	First Edition
August 2025:	Updated Edition

Revision History for the Updated Edition

2025-08-22:	First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9798341665118* for release details.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Databricks. See our statement of editorial independence.

979-8-341-66508-8

[LSI]

Table of Contents

Introduction. vii

1. Data Ingestion. 1

Data Ingestion—Now Versus Then 2

Sources and Targets 2

Ingestion Considerations 10

Choosing a Solution 19

2. Data Transformation. 25

What Is Data Transformation? 25

Building a Transformation Solution 33

The Future of Data Transformation 40

3. Data Orchestration. 43

What Is Data Orchestration? 44

Data Orchestration Tools 47

Design Patterns and Best Practices 54

The Future of Data Orchestration 59

4. Pipeline Issues and Troubleshooting. 61

Maintainability 62

Monitoring and Benchmarking 63

Errors 68

Improving Workflows 70

5. **Efficiency and Scalability. 77**
Efficiency and Scalability Defined 78
Understand Your Environment 80
Process Efficiency 87
Conclusion 92

Conclusion. 93

Introduction

Whether your title is data engineer or another data-oriented profession (we see you, analysts and scientists), you've likely heard the term ETL. There's a good chance ETL is a part of your life, even if you don't know it!

Short for extract, transform, load, ETL is used to describe the foundational workflow most data practitioners are tasked with—taking data from a source system, changing it to suit their needs, and loading it to a target.

Want to help product leaders make data-driven decisions? ETL builds the critical tables for your reports. Want to train the next iteration of your team's machine learning model? ETL creates quality datasets. Are you trying to bring more structure and rigor to your company's storage policies to meet compliance requirements? ETL will bring process, lineage, and observability to your workflows.

If you want to do *anything* with data, you need a reliable process or pipeline. This fundamental truth holds true from classic business intelligence (BI) workloads to cutting-edge advancements, like large language models (LLMs) and AI.

The Brave New World of AI

The data world has seen many trends come and go; some have transformed the space, and some have turned out to be short-lived fads. The most recent is, without a doubt, generative AI (GenAI).

At every turn, there's chatter about AI, LLMs, and agents. This recent fascination with AI, largely brought by the release of OpenAI's ChatGPT, extends beyond the media's interest and among researchers—it is now seen by many as an essential strategic investment…and who wants to be left behind?

The true value in LLMs comes from embeddings or fine-tuning models on clean, curated datasets. These techniques allow for the creation of models with domain-specific knowledge, avoiding common errors, like hallucination.

Of course, meaningful embeddings are derived from, you guessed it—clean datasets. In that sense, AI is built on data transformation. Its success depends heavily on the ability to create consistent, high-quality datasets at scale. Data needs to be moved, mutated, and merged in a single location—one might say *extracted, transformed, and loaded*.

That's right—even the most cutting-edge tech has roots back to ETL.

A Changing Data Landscape

In addition to the recent surge in GenAI, other trends have reshaped the data landscape over the past decade. One such trend is the increasing prominence of streaming data. Companies are now generating vast quantities of real-time data through sensors, websites, mobile applications, and more. This shift necessitates the real-time ingestion and processing of data for immediate decision making. Data engineers are therefore challenged to extend beyond traditional batch processing to construct and manage continuous pipelines capable of handling large volumes of streaming data.

Another noteworthy development is the emergence of data lakehouse architectures. The data lakehouse represents a novel concept, seeking to merge the capabilities of data warehouses and data lakes. Leveraging new storage technologies like Delta Lake, which enhance the reliability and performance of data lakes, the lakehouse model combines the cost-effective, scalable storage of data lakes with the efficient transaction processing of data warehouses. This amalgamation enables the execution of both AI workloads (typically handled in data lakes) and analytics workloads (usually conducted in data warehouses) within a singular framework. This integration significantly reduces the complexities associated with maintaining parallel architectures, ensuring consistent data governance, and managing data duplication.

While ETL is a long-standing concept in data management, its relevance remains undiminished in the modern data landscape. A critical consideration now is how ETL processes can adapt to encompass both batch and streaming data, and how they can be effectively integrated within a data lakehouse architecture. This guide aims to illuminate these aspects, helping you understand ETL in light of these evolving trends.

What About ELT (and Other Flavors)?

As you delve into data engineering, you may come across terms like *ELT* in addition to ETL. You might be thinking, "Wow, these guys should hire a proofreader," but rest assured, they're actually different terms.

The key difference in ELT lies in sequence: in ELT, *everything* is loaded into a staging resource, then transformed downstream. ELT has increasingly become the norm, supplanting ETL in many scenarios—as many say "storage is cheap." The term "ETL" has been widely used for so long (since the creation of databases themselves) that it's still commonly referred to, even when ELT is more accurate. We are now in an era of "store first, act later," facilitated by decreasing costs of cloud storage and the ease of data generation.

For analysis, retaining all potentially useful data is prevalent. Technological advancements like the *medallion architecture* and *data lakehouse* support this approach with features like easy schema evolution and time travel. We'll discuss those and more throughout this guide.

Although we predominantly use the term "ETL," it's important to note that the principles and considerations discussed are applicable to both ETL and ELT, as well as other variations like *reverse ETL*—the practice of ingesting cleaned data *back* into business tools from the ware- or lakehouse. No, reverse ETL != LTE, and yes, this is confusing, but we digress.

Whether the term "ETL" precisely describes your current process or not, comprehending the fundamentals of data ingestion, transformation, and orchestration remains crucial. This also extends to best practices in areas like observability, troubleshooting, scaling, and optimization. We hope that this guide will be a valuable resource, regardless of the specific data processing methodology you employ.

O'Reilly Online Learning

> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/understandingETL*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Though we all stand on the shoulders of giants, this guide in particular would not have been possible without mentorship, help, and support from some very dedicated and caring individuals.

First, thank you to my partners from O'Reilly and Databricks: Aaron Black, who gave me the opportunity to write; Gary O'Brien, who was a *stellar* development editor (and confidante!); Ori Zohar, who helped shape the guide as a whole; and both Sumit Makashir and Pier Paolo Ippolito for their excellent and attentive technical reviews.

Thank you to Zander Matheson for your help in understanding streaming and stream processing. Along with developing an *amazing* tool (Bytewax), Zander has been a great friend and a general data guru.

Thank you to Aleks Tordova and the Coalesce team, who partnered to write my *first* guide and have provided me with ample opportunities to learn and grow.

Thanks to my family, who provided unconditional support for my journey—in data and life—despite my flaws, idiosyncrasies, and general tomfoolery. Thank you, Jasmine, Violet, and Paul (and pups Enzo and Rocky!)

Next, I am blessed with some amazing friends who've supported me as I moved across the country, took a new job, wrote this guide, and continued my path of self-discovery. There were many texts, Slacks, phone calls, and memes that helped me through the tough times. In alphabetical order, thank you, JulieAnn, Kandace, Rob, Srini, and Tyson.

Last, thank you to the data community. To the individuals that contribute to open source and present at conferences, the practitioners that wake up every day looking to improve, the educators/mentors that keep us moving forward as a field, and *all* of the authors whose texts, ideas, and content have helped us get to where we are today: I can't wait to see what we accomplish next!

CHAPTER 1
Data Ingestion

Data ingestion, in essence, involves transferring data from a source to a designated target. Its primary aim is to usher data into an environment primed for staging, processing, analysis, and artificial intelligence/machine learning (AI/ML). While massive organizations may focus on moving data internally (among teams), for most of us, data ingestion emphasizes pulling data from external sources and directing it to in-house targets.

In an era where data holds central importance in both business and product development, the significance of accurate and timely data cannot be overstated. This heightened reliance on data has given rise to a multitude of "sources" from which teams extract information to refine decision-making processes, craft outstanding products, and conduct a multitude of other actions. For instance, a marketing team would need to retrieve data from several advertising and analytics platforms, such as Meta, Google (including Ads and Analytics), Snapchat, LinkedIn, and Mailchimp.

However, as time marches on, APIs and data sources undergo modifications. Columns might be introduced or removed, fields could get renamed, and new versions might replace outdated ones. Handling changes from a single source might be feasible, but what about juggling alterations from multiple sources—five, ten, or even a hundred? The pressing challenge is this: "How can a budding data team efficiently handle these diverse sources in a consistent and expandable way?" As data engineers, how do we ensure our reputation for providing reliable and straightforward data access, especially when every department's demands are continuously escalating?

Data Ingestion—Now Versus Then

Though the principles of ingestion largely remain the same, much has changed. As the volume, velocity, and variety of data evolve, so too must our methods.

We've had multiple industry changes to accommodate this—movement to the cloud, the warehouse to the data lake to the lakehouse, and the simplification of streaming technologies, to name a few. This has been manifested as a shift from extract-transform-load (ETL) workflows to extract-load-transform (ELT), the key difference being that *all* data is now loaded into a target system. We'll discuss these environments in the context of transformation in Chapter 2.

We refrain from being too pedantic about the terms ETL and ELT; however, we'd like to emphasize that almost every modern data engineering workflow will involve staging almost all data in the cloud. The notable exception is cases where hundreds of trillions of rows of highly granular data are processed daily (e.g., Internet of Things [IoT] or sensor data), where it makes sense to aggregate or discard data before staging.

Despite constant changes, the fundamental truth of extraction is that data is pulled from a source and written to a target. Hence, the discussion around extraction must be centered on precisely that.

Sources and Targets

While most associate ingestion with *extraction*, it's also tightly coupled with *loading*; after all, every source requires a destination. In this guide, we assume that you have an established warehouse or data lake; therefore, storage will not be a primary topic in this chapter. Instead, we'll highlight both best practices for staging and the hallmarks of ideal storage implementations.

It’s our mission to arm you with a toolkit for architecture design, keeping in mind that a “perfect” solution might not exist. We’ll navigate a framework for appraising sources and untangling the unique knots of data ingestion. Our high-level approach is designed to give you a bird’s-eye view of the landscape, enabling you to make informed, appropriate decisions.

The Source

Our primary consideration for ingesting data is the source and its characteristics. Unless you’re extremely lucky, there will be *many* sources. Each must be separately assessed to ensure adequate resources and set the criteria for your ingestion solution(s).

With the sheer volume of data sources and the nature of business requirements (I’ve seldom been asked to *remove* sources, but adding one is just another Thursday), it’s *highly* likely that you’ll encounter one or many sources that do not fit into a single solution. While building trust takes weeks, months, and years, it can be lost in a day. Reliable, timely ingestion is paramount. So, what’s important in choosing a source?

Examining sources

As a practical guide, we take the approach of presenting time-tested questions that will guide you toward *understanding* the source data, both its characteristics and how the business will get value.

We recommend taking a highly critical stance: it is always possible that *source data is not needed* or *a different source* will better suit the business. You are your organization’s data expert, and it’s your job to check and double-check assumptions. It’s normal to bias for action and complexity, but imperative we consider *essentialism* and *simplicity*.

When examining sources, keep in mind that you’ll likely be working with software engineers on upstream data, but downstream considerations are just as important. Neglecting these can be highly costly, since your error may not manifest itself until weeks of work have taken place.

Questions to ask

Who will we work with?

In an age of artificial intelligence, we prioritize real intelligence. The most important part of any data pipeline is *the people* it will serve. Who are the stakeholders involved? What are *their* primary motives—OKRs (objectives and key results) or organizational mandates can be useful for aligning incentives and moving projects along quickly.

How will the data be used?

Closely tied to "who," *how* the data will be used should largely guide subsequent decisions. This is a way for us to check our stakeholder requirements and learn the "problem behind the problem" that our stakeholders are trying to solve. We highly recommend a list of technical yes/no requirements to avoid ambiguity.

What's the frequency?

As we'll discuss in detail later, most practitioners immediately jump to batch versus streaming. Any data can be processed as a batch or stream, but we are commonly referring to the characteristics of data that we would like to stream. We advocate first considering if the data is *bounded* or *unbounded*—i.e., does it *end* (for example, the 2020 Census American Community Survey dataset), or is it continuous (for example, log data from a fiber cabinet).

After bounds are considered, the minimum frequency available sets a hard limit for how often we can pull from the source. If an API only updates daily, there's a hard limit on the frequency of your reporting. Bounds, velocity, and business requirements will inform the frequency at which we *choose* to extract data.

What is the expected data volume?

Data volume is no longer a limiter for the ability to store data—after all, "storage is cheap," and while compute can be costly, it's less expensive than ever (by a factor of millions; see Figures 1-1 and 1-2). However, volume closely informs how we choose to write and process our data and the scalability of our desired solution.

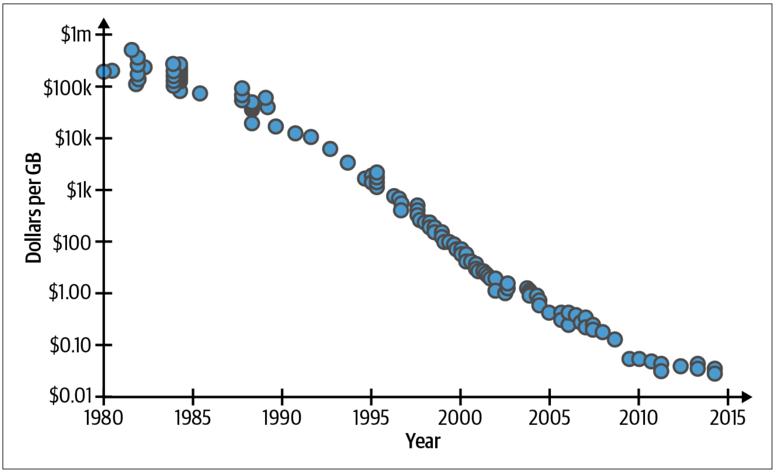


Figure 1-1. Hard drive costs per GB, 1980 to 2015 (Source: Matt Komorowski); the y-axis values are in log scale

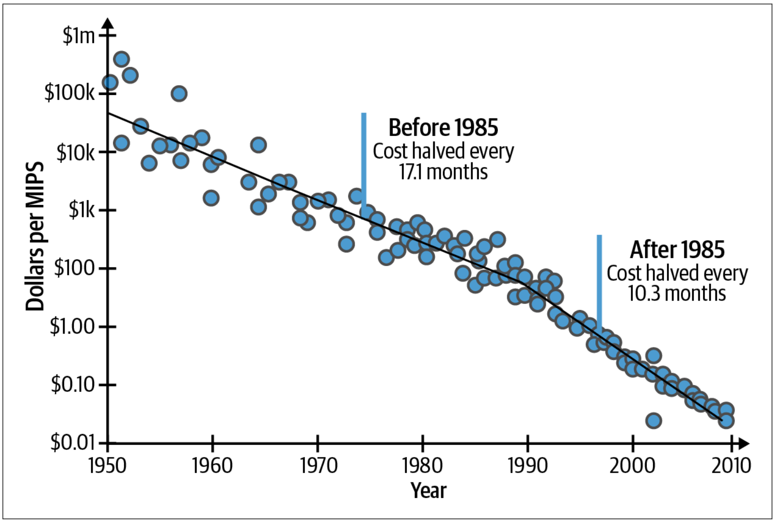


Figure 1-2. Cost of compute, millions of instructions per second (MIPS) (Source: Field Robotics Center); the y-axis values are in log scale

What's the format?

While we will eventually choose a format for storage, the input format is an important consideration. How is the data being delivered? Is it via a JavaScript Object Notation (JSON) payload over an API? Perhaps an FTP server? If you're lucky, it already lives in a relational database somewhere. What does the schema look like? Is there a schema? The endless number of data formats keeps our livelihoods interesting, but also presents a challenge.

What's the quality?
: The quality of the dataset will largely determine if any transformation is necessary. As data engineers, it's our job to ensure consistent datasets for our users. Data might need to be heavily processed or even enriched from external sources to supplement missing characteristics.

We'll use these characteristics to answer our final question:

How will the data be stored?
: As we mentioned, this book assumes some fixed destination for your data. Even then, there are a few key considerations in data storage: to stage or not to stage (is it really a question?), business requirements, and stakeholder fit are the most important.

Source checklist

For every source you encounter, consider these guiding questions. Though it might seem daunting as the number of sources piles up, remember: this isn't a writing task. It's a framework to unpack the challenges of each source, helping you sketch out apt solutions that hit your targets.

While it might feel repetitive, this groundwork is a long-term time and resource saver.

Question	Example
Who will we collaborate with?	Engineering (Payments)
How will the data be used?	Financial reporting and quarterly strategizing
Are there multiple sources?	Yes
What's the format?	Semi-structured APIs (Stripe and Internal)
What's the frequency?	Hourly
What's the volume?	Approximately 1K new rows/day, with an existing pool of ~100K
What processing is required?	Data tidying, such as column renaming, and enrichment from supplementary sources
How will the data be stored?	Storing staged data in Delta tables via Databricks

The Destination

While end-to-end systems require hypothetical considerations that are just that, we assume most readers will be tasked with building a pipeline into an *existing* system, where the storage technology is already chosen.

Choosing data storage technology is beyond the scope of this guide's focus, but we will briefly consider destinations, as they are *highly* important to the total value created by a data system. Thus, when analyzing (or considering) a destination, we recommend using a similar checklist to that of a source. Usually, there are far fewer destinations than sources, so this should be a much simpler exercise.

Examining destinations

A key differentiator in destinations is that the *stakeholder* is prioritized. Destinations have a unique trait: they pivot around stakeholders. These destinations either directly fuel BI, analytics, and AI/ML applications or indirectly power them when dealing with staged data, not to mention user-oriented apps. Though we recommend the same checklist, we suggest framing it slightly toward the stakeholder to be sure it meets their requirements, while working toward engineering goals.

We fully recognize *this is not always possible.* As an engineer, your role is to craft the most fitting solution, even if it means settling on a middle ground or admitting there's no clear-cut answer. Despite technology's incredible strides, certain logical dilemmas do not have straightforward solutions.

Staging ingested data

We advocate for a data lake approach to data ingestion. This entails ingesting most data into cloud storage systems, such as S3, Google Cloud Platform, or Azure, before loading it into a data warehouse for analysis.

One step further is a lakehouse—leveraging data storage protocols, like Delta Lake, which use metadata to add performance, reliability, and expanded capability. Lakehouses can even replicate some warehouse functionality; this means avoiding the need to load data to a separate warehouse system. Adding in a data governance layer, like Unity Catalog, can provide better discoverability, access management, and collaboration for *all* data assets across an organization.

A prevailing and effective practice for staging data is utilizing metadata-centric Parquet-based file formats, including Delta Lake, Apache Iceberg, or Apache Hudi. Grounded in Parquet—a compressed, columnar format designed for large datasets—these formats incorporate a metadata layer, offering features such as time travel, ACID (atomicity, consistency, isolation, and durability) compliance, and more.

Integrating these formats with the medallion architecture, which processes staged data in three distinct quality layers, ensures the preservation of the entire data history. This facilitates adding new columns, retrieving lost data, and backfilling historical data.

The nuances of the medallion architecture will be elaborated upon in our chapter on data transformation (Chapter 2). For the current discussion, it's pertinent to consider the viability of directing all data to a "staging layer" within your chosen cloud storage provider.

OLAP Versus OLTP Databases

The biggest choice in data warehousing is whether to use a cloud native database or a cloud-hosted traditional database—the main difference being the distributed nature and column-oriented architecture of cloud native solutions. These are more commonly referred to as OLAP (online analytical processing) and OLTP (online transaction processing):

- OLAP systems are designed to process large amounts of data quickly. This is commonly accomplished via distributed processing and a column-oriented architecture. Newer, cloud native databases are OLAP systems; the big three OLAP solutions are Amazon Redshift, Google BigQuery, and Snowflake.
- OLTP systems are engineered to handle large amounts of transactional data originating from multiple users. This usually takes the form of a row-oriented database. Many traditional database systems are OLTP: Postgres, MySQL, etc.

OLAP systems are most commonly used by analytics and data science teams for their speed, stability, and low maintenance cost. Here are some considerations for data warehouse selection:

Computing platform of choice

The integration of these technologies largely depends on the rest of your tech stack. For example, if *every* tool in your organization lives in the Amazon ecosystem, Redshift might be more cost-effective and simple to implement than Google BigQuery. Similarly, the Databricks ecosystem contains a broad array of functionality—from storage (Lakehouse, Databricks SQL) to governance (Unity Catalog) and compute (Apache Spark).

Functionality

BigQuery has great support for semistructured data and ML operations. Redshift Spectrum has proven to be a useful tool for creating external tables from data in S3. Databricks separates storage and compute through the use of Spark, Databricks SQL, and Delta Lake. Every warehouse has strengths and weaknesses; these need to be evaluated according to your team's use case.

Cost

Pricing structures vary wildly across platforms. Unfortunately, pricing can also be opaque—in most cases, you won't *truly* know how a database will be used until you're on the platform. The goal with pricing should be to understand *how to use a database to minimize cost* and what the cost might be if that route is taken. For example, in cases where cost is directly tied to the amount of data scanned (BigQuery), intelligent partitioning and query filtering can go a long way. Using a lakehouse saves you the redundancy and replicated data of a lake and a data warehouse. This is especially important since warehouses tend to have steeper storage costs and limitations.

There will be no direct answer on cost, but research is worthwhile, as warehouses can be expensive, especially at scale.

Change data capture

Change data capture (CDC) is a data engineering design pattern that captures and tracks changes in source databases to update downstream systems. Rather than batch-loading entire databases, CDC transfers only the changed data, optimizing both speed and resource usage.

This technique is crucial for real-time analytics and data warehousing, as it ensures that data in the target systems is current and in sync with the source. By enabling incremental updates, CDC enhances data availability and consistency across the data pipeline. Put simply by Joe Reis and Matt Housley in *Fundamentals of Data Engineering* (O'Reilly, 2022), "CDC…is the process of ingesting changes from a source database system."

CDC becomes important in analytics and engineering patterns—like creating Slowly Changing Dimension (SCD) type 1 and 2 tables, a process that can be unnecessarily complex and time-consuming. Choosing platforms or solutions that natively support CDC can expedite common tasks, letting you focus on what matters most. One example is Lakeflow Declarative Pipelines (LDP) on Databricks, which provide native support for SCD type 1 and 2 in both batch and streaming pipelines.

Destination checklist

Here's a sample checklist for questions to consider when selecting a destination:

Question	Example
Who will we collaborate with?	Human Resources
How will the data be used?	Understand how tenure/contract duration affects bottom-line results in a multinational organization.
Are there multiple destinations?	Data is staged in Delta Lake; final tables are built in Databricks SQL.
What's the format?	Parquet in Delta Lake. Structured/semi-structured in Databricks SQL.
What's the frequency?	Batch
What's the volume?	Approximately 1K new rows/day, with an existing pool of ~100K
What processing is required?	Downstream processing using LDP. ML models and GenAI applications in Spark.
How will the data be stored?	Blend of Databricks SQL and external tables in Delta Lake.

Ingestion Considerations

In this section, we outline pivotal data characteristics. While this list isn't exhaustive and is influenced by specific contexts, aspects like *frequency, volume, format, and processing* emerge as primary concerns.

Frequency

We already mentioned that the first consideration in frequency should be bounds, i.e., is the dataset *bounded* or *unbounded*. Bounds and business needs will dictate a frequency for data ingestion—either in batch or streaming formats. Figure 1-3 neatly shows the difference between batch and streaming processes; streaming captures events as they occur, while batch groups them up, as the name would suggest.

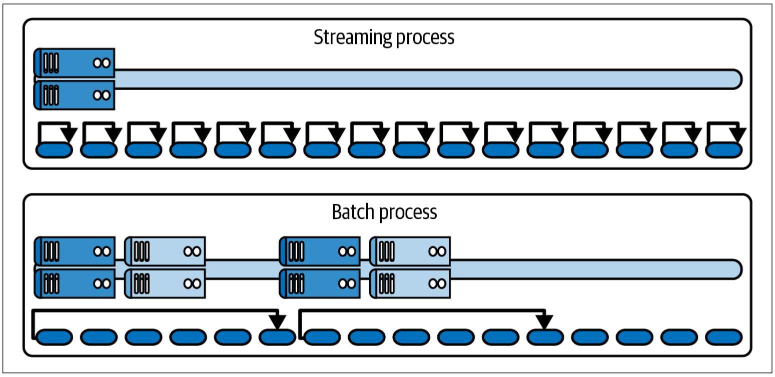


Figure 1-3. Latency is the property that defines "batch" or "streaming" processes; beyond some arbitrary latency threshold, we consider data "streamed" (courtesy of Denny Lee)

We'll present batch, micro-batch, and streaming along with our thoughts to help you select the most appropriate frequency and a compatible ingestion solution.

Batch

Batch processing is the act of processing data in *batches* rather than all at once. Like a *for loop* that iterates over a source, batch simply involves either chunking a bounded dataset and processing each component *or* processing unbounded datasets as data arrives.

Micro-batch

A micro-batch is simply "turning the dial down" on batch processing. If a typical batch ingestion operates daily, a micro-batch might function hourly or even by the minute. Of course, you might say, "At what point is this just semantics? A micro-batch pipeline with 100 ms latency seems a lot like streaming to me."

We agree!

For the rest of this chapter (and book), we'll refer to low-latency micro-batch solutions as streaming solutions—the most obvious being Spark Structured Streaming. While "technically" micro-batch, latency in the hundreds of milliseconds makes Spark Structured Streaming effectively a real-time solution.

Streaming

Streaming refers to the continuous reading of datasets, either bounded or unbounded, as they are generated. While we will not discuss streaming in great detail, it does warrant further research. For a comprehensive understanding of streaming data sources, we suggest exploring resources like *Streaming 101*, *Streaming Databases,* and, of course, *Fundamentals of Data Engineering*.

Methods. Common methods of streaming unbounded data include:

Windowing
: Segmenting a data source into finite chunks based on temporal boundaries.

Fixed windows
: Data is essentially "micro-batched" and read in small fixed windows to a target.

Sliding windows
: Similar to fixed windows, but with overlapping boundaries.

Sessions
: Dynamic windows in which sequences of events are separated by gaps of inactivity—in sessions, the "window" is defined by the data itself.

Time-agnostic
: Suitable for data where time isn't crucial, often utilizing batch workloads.

Figure 1-4 demonstrates the difference between fixed windows, sliding windows, and sessions. It's crucial to differentiate between the *actual* event time and the processing time, since discrepancies may arise.

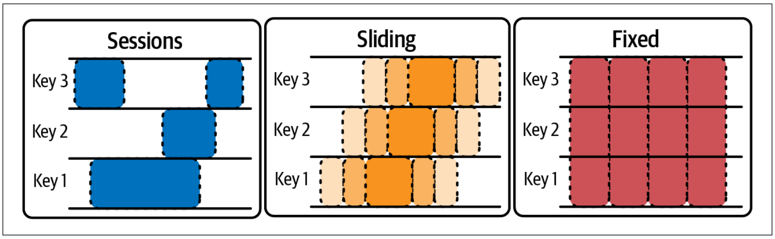


Figure 1-4. Fixed windows, sliding windows, and sessions treat the arrival of streaming data quite differently

Message services. When we say "message services," we refer to "transportation layers" or systems for communicating and transporting streaming data. One important note is that this *is not a direct comparison*; while there is overlap in these services, many operate under fundamentally different architectures, rendering "Kafka versus Pub/Sub" or "Kinesis versus Redpanda" discussions largely irrelevant.

Apache Kafka
: Originated at LinkedIn in 2011, Apache Kafka started as a message queue system but quickly evolved into a distributed streaming platform. While Kafka's design allows for high throughput and scalability, its inherent complexity remains a hurdle for many.

Redpanda
: Developed as an alternative to Kafka, Redpanda boasts similar performance with a simplified configuration and setup. Redpanda is based on C++ rather than Java and compatible with Kafka APIs.

Pub/Sub
: Pub/Sub is the Google Cloud offering for a durable, dynamic messaging queue. Unlike Kafka, Google Pub/Sub scales dynamically to handle variable workloads. Rather than dealing in "streams" and "shards," Pub/Sub opts for "topics" and "subscriptions." A big draw to Pub/Sub is the elimination of most "maintenance" tasks—it's almost fully managed.

Kinesis

Kinesis is another robust, fully managed service. As an Amazon service, Kinesis offers the obvious ease of integration with other Amazon Web Services (AWS) offerings while bringing automatic scalability and real-time data processing. Like Pub/Sub, Kinesis stands out for its managed service nature, offering a lower operational burden than Apache Kafka.

Stream processing engines. Stream processing is about analyzing and acting on real-time data (streams). Given Kafka's longevity, the three most popular and well-known stream processing tools are:

Apache Flink

An open source engine that continuously processes both unbounded and bounded datasets with minimal downtime. Apache Flink ensures low latency through in-memory computations, offers high availability by eliminating single points of failure, and scales horizontally.

Apache Spark Structured Streaming

An arm of the Apache Spark ecosystem designed to handle real-time data processing. It brings the familiarity and power of Spark's DataFrame and Dataset APIs to streaming data. Structured Streaming might be an attractive option given the popularity of Apache Spark in data processing and ubiquity of the engine in tools like Databricks.

Apache Kafka Streams

A library built on Kafka that provides stateful processing capabilities, but ties to Java can be limiting.

Simplifying stream processing. Several relatively new solutions simplify stream processing by offering straightforward clients, with a focus on performance and simple development cycles.

Managed platforms

Taking Databricks as an example: leveraging tools like Lakeflow Declarative Pipelines or simply running Spark Streaming jobs on the Databricks runtime can be a powerful abstraction of complexity and drastically simplify the process of building streaming systems.

Confluent Kafka
: An attempt to bring Apache Kafka capabilities to Python, although it remains rudimentary compared with its Java counterpart. Confluent Kafka is simply a client library in the same way psycopg2 is a Postgres client library.

Bytewax
: A library that aims to bridge the gap by offering a more intuitive, Pythonic way of dealing with stream processing, making it more accessible to a wider range of developers. Built on Rust, Bytewax is highly performant, simpler than Flink, and boasts shorter feedback loops and easy deployment/scalability.

Still newer tools that seek to unify stream processing—like Apache Beam or Estuary Flow—or combine stream processing directly with databases (streaming databases) are growing in popularity. We recommend *Streaming Systems* and *Streaming Databases* for an in-depth look.

The streaming landscape, while complex, has seen strides in simplification and user-friendliness, especially when considering managed platforms, like Databricks, and low-latency micro-batch solutions, like Spark Structured Streaming.

While many think of real-time data as the ultimate goal, we emphasize a "right-time" approach. As with any solution, latency can be optimized infinitely, but the cost of the solution (and complexity) will increase proportionally. Most will find going from daily or semi-daily data to hourly/subhourly data a perfectly acceptable solution.

Payload

The term "payload" refers to the actual message being transmitted, along with any metadata or headers used for routing, processing, or formatting the data. Data payloads are inherently broadly defined, since they can take almost any shape imaginable. In this section, we'll discuss typical payload characteristics.

Volume

Volume is a pivotal factor in data ingestion decisions, influencing the scalability of both processing and storage solutions. When assessing data volume, be sure to consider factors like:

Cost
: Higher volumes often lead to increased costs, both in terms of storage and compute resources. Make sure to align the cost factor with your budget and project needs, including storage/staging costs associated with warehouses, lakes, or lakehouses, depending on your solution.

Latency
: Depending on whether you need real-time, near-real-time, or batch data ingestion, latency can be a critical factor. Real-time processing not only requires more resources, but it also necessitates greater efficiency when volume spikes. For any data volume, be sure your systems can handle latency requirements.

Throughput/scalability
: It's essential to know the ingestion tool's ability to handle the sheer volume of incoming data. If the data source generates large amounts of data, the ingestion tool should be capable of ingesting that data without causing bottlenecks.

Retention
: With high volumes, data retention policies become more important. You'll need a strategy to age out old data or move it to cheaper, long-term storage solutions. In addition to storing old data, security and backfilling (restoring) lost data should be considered.

For handling sizable datasets, using compressed formats like Apache Avro or Parquet is crucial. Each offers distinct advantages and constraints, especially regarding schema evolution. For each of these considerations, be sure to look to the future—tenable solutions can quickly disintegrate with order-of-magnitude increases in data volume.

Structure and shape

Data varies widely in form and structure, ranging from neat, relational "structured" data to more free-form "unstructured" data. Importantly, structure doesn't equate to quality; it merely signifies the presence of a schema.

In today's AI-driven landscape, unstructured data's value is soaring as advancements in large language and machine learning models enable us to mine rich insights from such data. Despite this, humans have a penchant for structured data when it comes to in-depth analysis, a fact underscored by the enduring popularity of SQL—*Structured* Query Language—a staple in data analytics for nearly half a century.

Unstructured. As we’ve alluded, unstructured data is data without any predefined schema or structure. Most often, it’s represented as text, but other forms of media represent unstructured data, too. Video, audio, and imagery all have elements that may be analyzed numerically. Unstructured data might be text from a Pierce Brown novel:

> “A man thinks he can fly, but he is afraid to jump. A poor friend pushes him from behind.” He looks up at me. “A good friend jumps with.”

Such data often feeds into machine learning or AI applications, underlining the need to understand stakeholder requirements comprehensively. Given the complexity of machine learning, it’s vital to grasp how this unstructured data will be utilized before ingesting it. Metrics like text length or uncompressed size may serve as measures of shape.

Semi-structured. *Semi-structured* data lies somewhere between structured and unstructured data—XML and JSON are two popular formats. Semi-structured data might take the form of a JSON payload:

```
'{"friends": ["steph", "julie", "thomas", "tommy",
"michelle", "tori", “larry”]}'
```

As data platforms continue to mature, so too will the ability to process and analyze semi-structured data directly. The following snippet shows how to parse semi-structured JSON in Google BigQuery to pivot a list into rows of data:

```
WITH j_data AS (
    SELECT
      (
      JSON '{"friends": ["steph", "julie", "thomas",
"tommy", "michelle", "tori", “larry”]}'
      ) AS my_friends_json
), l_data AS (
  SELECT
    JSON_EXTRACT_ARRAY(
      JSON_QUERY(j_data.my_friends_json, "$.friends"),
'$'
```

```
        ) as my_friends_list
    FROM j_data
)
    SELECT
        my_friends
FROM l_data, UNNEST(l_data.my_friends_list) as my_friends
ORDER BY RAND()
```

In some situations, moving data processing downstream to the analytics layer is worthwhile—it allows analysts and analytics engineers greater flexibility in how they query and store data. Semi-structured data in a warehouse (or accessed via external tables) allows for flexibility in the case of changing schemas or missing data while still providing all the benefits of tabular data manipulation and SQL.

Still, we must be careful to properly validate this data to ensure that missing data isn't causing errors. Many invalid queries result from the improper consideration of `NULL` data.

Describing the shape of JSON frequently involves discussing keys, values, and the number of nested elements. We highly recommend tools like JSONLint and JSON Crack for this purpose. VS Code extensions also exist to validate and format JSON/XML data.

Structured. The golden "ideal" data, structured sources are neatly organized with fixed schemas and unchanging keys. For over 50 years, SQL has been the language of choice for querying structured data. When storing structured data, we frequently concern ourselves with the number of columns and length of the table (in rows). These characteristics inform our use of materialization, incremental builds, and, in aggregate, an OLAP versus OLTP database (column-/row-oriented).

Though much data today lacks structure, we still find SQL to be the dominant tool for analysis. Will this change? Possibly, but as we showed, it's more likely that SQL will simply adapt to accommodate semi-structured formats. Though language-based querying tools have started appearing with AI advancements, SQL is often an intermediary. If SQL disappears from data analysis, it will likely live on as an API.

Format

What is the best format for source data? While JSON and CSV (comma-separated values) are common choices, an infinite number of format considerations can arise. For instance, some older SFTP/FTP transfers might arrive compressed, necessitating an extra extraction step.

The data format often dictates processing requirements and available solutions. While a tool like Airbyte might seamlessly integrate with a CSV source, it could stumble with a custom compression method or a quirky Windows encoding (believe us, it happens).

If at all possible, we advise opting for familiar, popular data formats. Like repairing a vehicle, the more popular the format, the easier it will be to find resources, libraries, and instructions. Still, in our experience it's a rite of passage to grapple with a perplexing format, but that's part of what makes our jobs fun!

Variety

It's highly likely you'll be dealing with multiple sources and thus varying payloads. Data variety plays a large role in choosing your ingestion solution—it must not only be capable of handling disparate data types but also be flexible enough to adapt to schema changes and varying formats. Variety makes governance and observability particularly challenging, something we'll discuss in Chapter 5.

Failing to account for data variety can result in bottlenecks, increased latency, and a haphazard pipeline, compromising the integrity and usefulness of the ingested data.

Choosing a Solution

The best tools for your team will be the ones that support your sources and targets. Given the unique data requirements of each organization, your choices will be context-specific. Still, we can't stress enough the importance of tapping into the knowledge of mentors, peers, and industry experts. While conferences can be pricey, their knowledge yield can be priceless. For those on a tight budget, data communities can be invaluable. Look for them on platforms like Slack and LinkedIn and through industry newsletters.

When considering an ingestion solution, we think in terms of *general* and *solution-specific* considerations—the former applying to all tools we'll consider, and the latter being specific to the class of tooling.

General considerations include extensibility, the cost to build, the cost to maintain, the cost to switch, and other, system-level concerns.

Solution-specific considerations are dependent on the class of tooling, which commonly takes one of two forms:

- *Declarative solutions* dictate outcomes. For example, "I use the AWS Glue UI to build a crawler that systematically processes data" or "I create a new Airbyte connection via a UI."
- *Imperative solutions* dictate actions. For example, "I build a lambda that calls the Stripe API, encodes/decodes data, and incrementally loads it to Snowflake."

Each of these solutions has its pros and cons. We'll briefly discuss each and present our recommended method for approaching data integration.

Declarative Solutions

We classify declarative solutions as legacy, modern, or native, largely dependent on their adherence to modern data stack (MDS) principles, like infrastructure-as-code, DRY (don't repeat yourself), and other software engineering best practices. Native platforms differ from the first two—they are integrated directly into a cloud provider:

Legacy

Think Talend, WhereScape, and Pentaho. These tools have robust connectors and benefit from a rich community and extensive support. However, as the data landscape evolves, many of these tools lag behind, not aligning with the demands of the MDS. Unless there's a compelling reason, we'd recommend looking beyond legacy enterprise tools.

Modern

Here’s where Fivetran, Stitch, and Airbyte come into play. Designed around “connectors,” these tools can seamlessly link various sources and targets, powered by state-of-the-art tech and leveraging the best of the MDS.

Native

In the first two solutions, we’re working from the assumption that data must be moved from one source to another—but what if you had a managed platform that supported ingestion, out of the box? Databricks, for example, can natively ingest from message buses and cloud storage:

```
CREATE STREAMING TABLE raw_data
AS select *
FROM cloud_files(“/raw_data”, “json”);

CREATE STREAMING TABLE clean_data
AS SELECT SUM(profit)...
FROM raw_data;
```

While there is no “right” type of declarative solution, many will benefit from the reduced cost to build and maintain these solutions, *especially* those native to your existing cloud provider/platform.

Cost to build/maintain

Declarative solutions are largely hands-off—here’s where you get a bang for your buck! Dedicated engineers handle the development and upkeep of connectors. This means you’re delegating the heavy lifting to specialists. Most paid solutions come with support teams or dedicated client managers, offering insights and guidance tailored to your needs. These experts likely have a bird’s-eye view of the data landscape and can help you navigate ambiguity around specific data problems or connect you with other practitioners.

For example, solutions such as Databricks’ Lakeflow Connect offer a combination of modern connectors that are also native to the platform, allowing users to get all the benefits of an easy-to-configure, fully managed connector without having to integrate an additional vendor solution.

Extensibility

Extensibility revolves around how easy it is to build upon existing solutions. How likely is that new connector to be added to Airbyte or Fivetran? Can you do it yourself, building on the same framework? Or do you have to wait weeks/months/years for a product team? Will using Stitch suffice, or will you need a complementary solution? Remember, juggling multiple solutions can inflate costs and complicate workflows. In declarative solutions, extensibility is *huge*. No one wants to adopt a solution only to learn it will only solve 15% of their needs.

Cost to switch

The cost to switch is where the limitations of declarative tools come to light. Proprietary frameworks and specific CDC methods make migrating to another tool expensive. While sticking with a vendor might be a necessary compromise, it's essential to factor this in when evaluating potential solutions.

Imperative Solutions

Imperative data ingestion approaches can be in-house Singer taps, lambda functions, Apache Beam templates, or jobs orchestrated through systems like Apache Airflow.

Typically, larger organizations with substantial resources find the most value in adopting an imperative methodology. Maintaining and scaling a custom, in-house ingestion framework generally requires the expertise of multiple data engineers or even a dedicated team.

The biggest benefit of imperative solutions is their extensibility.

Extensibility

By nature, imperative is custom—that means each tap and target is *tailored to the needs of the business*. When exploring data integration options, it quickly becomes apparent that no single tool meets every criterion. Standard solutions inevitably involve compromises. However, with an imperative approach, there's the freedom to design it precisely according to the desired specifications. Unfortunately, without a large, dedicated data engineering organization, this extensibility is incredibly expensive to build and maintain.

Cost to build/maintain

While imperative solutions can solve complex and difficult ingestion problems, they require quite a bit of engineering. One look at the Stripe entity relationship diagram should be enough to convince you that this can be incredibly time-consuming. Additionally, the evolving nature of data—like changes in schema or the deprecation of an API—can amplify the complexity. Managing a single data source is one thing, but what about when you scale to multiple sources?

A genuinely resilient, imperative system should incorporate best practices in software design, emphasizing modularity, testability, and clarity. Neglecting these principles might compromise system recovery times and hinder scalability, ultimately affecting business operations. Hence, we suggest that only enterprises with a robust data engineering infrastructure consider going fully imperative.

Cost to switch

Transitioning from one imperative solution to another might not always be straightforward, given the potential incompatibility between different providers' formats. However, on a brighter note, platforms based on common frameworks, like Singer, might exhibit more compatibility, potentially offering a smoother switch compared with purely declarative tools such as Fivetran or Airbyte.

Hybrid Solutions

Striking the right balance in integration often means adopting a hybrid approach. This might involve leveraging tools like Fivetran for most integration tasks, while crafting in-house solutions for unique sources, or opting for platforms like Airbyte/Meltano and creating custom components for unsupported data sources.

Contributing to open source can also be rewarding in a hybrid environment. Though not without faults, hybrid connectors, like those in Airbyte or Singer taps, benefit from expansive community support. Notably, Airbyte's contributions to the sector have positively influenced market dynamics, compelling competitors like Fivetran to introduce free tiers. We also encourage a proactive approach to exploring emerging libraries and tools. For instance, dlt (data load tool) is an open source library showing significant promise.

Consider data integration akin to choosing an automobile. Not every task demands the prowess of a Formula One car. More often, what's required is a dependable, versatile vehicle. However, while a Toyota will meet 99% of uses, you won't find a Sienna in an F1 race. The optimal strategy? Rely on the trusty everyday vehicle, but ensure access to high performance tools when necessary.

CHAPTER 2
Data Transformation

While data ingestion simply transfers data from point A to B, data transformation turns raw data into valuable insights through various stages of the data lifecycle. This chapter delves into the diverse languages, platforms, and technologies available to data practitioners for executing data transformations.

We'll see how to ensure that data transformations are conducted efficiently and in a well-coordinated manner, laying the groundwork for more detailed discussions on efficiency, scalability, and observability later in the guide.

What Is Data Transformation?

Data transformation is the art of manipulating and enhancing data to better serve users and processes. Transformation involves taking some data, whether in a raw or nearly pristine state, and performing one or many operations to move it closer to the intended use. In an ETL pipeline, transformation occurs in not one, but *many* places. Data might be transformed upon ingestion and again at any number of points downstream. The goal of data transformation is to turn data into an asset—using analysis and science to create something of value for the business.

Transformation might be as simple as removing unwanted records, e.g., filtering, or as complex as restructuring the source data entirely. Transformation exists on a spectrum; there's an almost infinite number of ways to transform data—that's what keeps things interesting!

Similarly, transformation can be orchestrated in any language with any technology, unhindered by the bounds of cloud providers, servers, or sources. This could manifest as choreographed Spark jobs, lambda functions, SQL workflows, or maybe something different—whether that be in Python, JavaScript, Rust, Scala, R, or (hold your breath) a spreadsheet. While we advocate for picking familiar, popular languages, we'd be remiss if we didn't mention the ubiquity of the practice of "transforming" data.

In this chapter, we'll provide you with an overview of what transformation is and how we transform data as practitioners. Then, we'll walk through building a transformation solution, starting from common patterns of transforming and updating data and continuing with best practices and considerations for streaming transformation. Our aim is to provide a series of considerations for building an efficient transformation system and equip you with the tools to adapt this knowledge to your own unique problems.

Where Are We Now?

In the ever-evolving landscape of data engineering, the history of data transformation is a tale of growth, simplification, and adaptation. In the early days of the internet, behemoths like Google and Yahoo were trailblazers. They innovated with big data frameworks like Hadoop and MapReduce. However, these technologies were only slightly less complex than graduate-level linear algebra.

As the industry matured, tooling simplified and democratized. Spark emerged, offering a streamlined distributed engine with APIs in Python, SQL, and Scala. Not long after, Databricks emerged with a mission to simplify Spark deployment. Simultaneously, data warehouses hit the gym. BigQuery, Redshift, and Snowflake emerged as technologies that could scale, eventually separating storage from compute. Presto, Trino, and Athena joined the fray, offering lightning-fast query performance and scaling SQL to a big data language.

Combined with the power of data warehouses, the flexibility of cloud storage, and the availability of transformation tools, data has *never* been more accessible. Today, lakehouses have emerged as an additional option for managing data at scale in a unified, cost-efficient manner.

How Do We Transform Data?

As we've alluded, data transformation is an inherently broad topic. We'll attempt to bring structure and scope to data transformation by discussing transformation environments, languages, frameworks, and best practices.

Environments

Where we transform data often dictates *how* we transform data. Here are the most common environments for data transformation:

Data warehouses
: Transformation in the warehouse is performed using SQL. Modern data warehouses have some interesting functionality that enables transformation for a variety of data—materialization, incremental logic, and partitioning, to name a few. Serverless data warehouses can dynamically scale to handle intense workloads, making them prime for data transformation on large, structured datasets.

Data lakes
: Data lakes excel at storing large amounts of data economically, making them a prime area for staging. Unlike data warehouses, data lakes do not have any computing resources, so transformations need to be orchestrated and executed using external services.

Data lakehouses
: Lakehouses can combine aspects of data lakes and warehouses to provide a solution with greater flexibility, potentially at a lower cost. Utilizing services like Apache Spark, practitioners can transform data in PySpark or Spark SQL, solving the issue of compute in a data lake. Databricks SQL is a warehouse that sits atop the Databricks lakehouse platform, allowing the benefits of a serverless data warehouse.

The choice among data lakes, warehouses, and lakehouses hinges on the specific needs of a project, the team's expertise, and the long-term data strategy of the organization, each catering to different facets of data transformation with its distinct set of tools and challenges.

Here are some questions to consider when selecting *where* your transformations will take place:

- How do the costs compare among maintaining a data lake, warehouse, and lakehouse, especially concerning the complexity of data transformations?
- Does any platform better fit the structure of your team or organization? Which is best for maintenance, development, or expansion, given your current and expected workflows?
- What are the considerations for data security and compliance?

When choosing a data platform, consider your budget, the skill and experience of your team, and both current and future requirements. A balance between well-established practices and cutting-edge technology is crucial for long-term success.

Data staging

Between transformations, data is often *staged*—written in a temporary state to a suitable location, often either cloud storage or an intermediate table. Staged data may be kept temporarily or indefinitely, but it plays a crucial role in transformation.

Whether you're taking a lake plus warehouse approach or going with a pure lakehouse implementation, how will you stage data? Medallion architecture (Figure 2-1) is an approach that describes three distinct "layers": Bronze for raw, Silver for light transformation, and Gold for "clean."

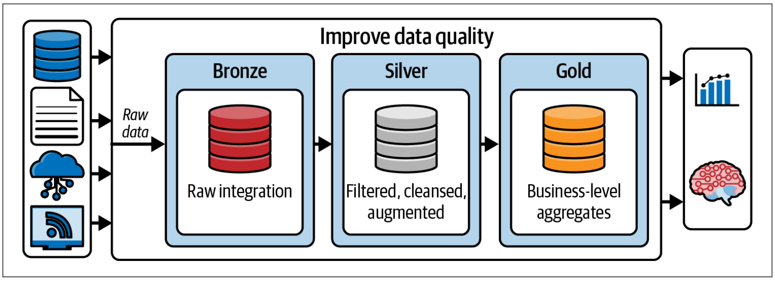


Figure 2-1. A medallion architecture promotes three distinct layers of data preparedness

Using terms from medallion architecture, staged data usually follows this pattern:

- *Bronze* data is raw and unfiltered, often directly from integration sources. For example, your raw tables might be written directly from an API.
- *Silver* data is filtered, cleaned, and adjusted. You might remove unnecessary, extraneous information, apply transformations to ensure consistency, and enrich data as necessary.
- *Gold* data is generally stakeholder-ready and sometimes aggregated. Your consumers (analytics, engineering) should be querying Gold tables a *vast* majority of the time, with only occasional need for Silver tables. Generally, production assets (visualizations, reports) should *not* hit anything other than Gold tables.

Downstream from your lake, a similar approach can be used in a warehouse, with *staging*, *curated*, and *marts* tables denoting levels of transformed data. This is a time-tested way to transform data, and the approach dbt currently recommends.

Thus, the architectural pattern for staging transformation is to take each storage layer and break it down into discrete "stages" of data cleanliness. Interwoven in this approach is leveraging storage functionality, like Delta Lake. In any case, patterns like write-audit-publish can be applied to any stage.

Languages

Of course, *how* we transform data is largely dictated by the tools at our disposal. Our biggest choice, programming language, plays a key role in the libraries, frameworks, and methods we use to reach a solution. Here are a few of the most popular libraries in data:

Python

Python has remained a steadfast choice in the evolving landscape of the digital era, particularly in data science and transformation. Figure 2-2 shows Python's dominance in the last decade. At the heart of Python's data science capabilities is the Pandas library, bolstered by a range of I/O libraries for data manipulation. Historically, scaling Python for large datasets has been challenging, often necessitating the use of libraries like Dask and Ray. However, recent advancements have marked a significant evolution in Python-based data processing, which we view as a renaissance in the field.

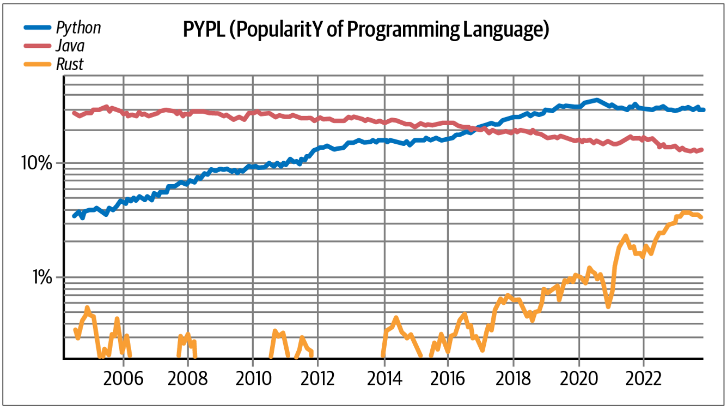


Figure 2-2. Though Rust is on the rise, Python continues a trend of dominance (adapted from PYPL); the y-axis values are in log scale

When transforming data in Python, the first step is to choose a suitable library, followed by selecting a framework for orchestrating and executing the transformation. We recommend starting with Pandas, renowned for its robustness. However, emerging libraries such as Polars and DuckDB are also worthy of consideration. This chapter will delve into various transformation techniques, and we will explore orchestration in more depth in Chapter 3.

SQL

SQL boasts the title of a declarative language, a term that now makes its second appearance (see Chapter 1). This title implies that the language is more about describing desired outcomes than outlining the step-by-step process to achieve them. For instance, the statement `SELECT * FROM views.active_users` isn't code in the traditional sense; it's more of a polite request, saying, "Please fetch all active users."

We argue that SQL can be used as a declarative *or* an imperative language, as we've seen some queries that are pretty innovative (for better or worse), but SQL is still inherently limited by a lack of functionality that languages like, say, Python are not.

This inherent limitation nudges most SQL transformations into the companionship of a templating solution, facilitating scalable SQL transformation. Languages like Jinja/Python (via dbt) and JavaScript (via Dataform) often complement SQL workflows.

Most SQL transformations occur in a data warehouse or leverage the data warehouse's compute to execute a query.

Rust

Rust is an up-and-coming transformation language that has many claiming it's the future of data engineering. Developed by Mozilla, it was first released in 2010. Its primary edge comes from its speed compared with Python and its strongly typed nature, which is beneficial for production workflows. However, Rust is relatively new, and Python has a solid foothold given its extensive community support, rich library ecosystem, and widespread adoption across industries.

While exploring Rust is encouraged, for production services, sticking to well-adopted languages is advisable. A pragmatic approach to Rust might be engaging with a Python library written in Rust, like Polars.

Frameworks

Transformation frameworks are multilanguage engines for executing data transformations across machines or clusters. As parallelization and distributed computing has become the norm, these frameworks have powered transformation at scale.

The benefit of a framework is that multiple APIs exist for executing workflows, which means transformations may often be manipulated in various languages, like Python or SQL. We'll discuss the two most popular engines, Hadoop and Spark, though you could also think of your data warehouse as an engine:

Hadoop

Apache Hadoop is an open source framework that emerged as a groundbreaking solution for handling big data. Its history, use by large corporations, and current state are closely intertwined with the rise of Apache Spark and the advent of serverless data warehouses. Hadoop gained significant traction in the mid-2000s, with tech giants like Yahoo, Facebook, and later Google adopting it to manage their growing data volumes.

While Hadoop was instrumental in handling large-scale data, it had limitations. Optimized for batch processing, Hadoop's MapReduce was not well suited for real-time or iterative workloads. This led to the rise of Apache Spark in the early 2010s, offering in-memory processing, faster analytics, and support for various workloads, including batch, streaming, and machine learning.

While Hadoop remains relevant, its dominance has waned. Many organizations have shifted toward hybrid data architectures built around cloud-based solutions and serverless data warehouses.

Spark

Apache Spark is a powerful open source data processing framework that revolutionized big data analytics by offering speed, versatility, and integration with key technologies. Its history is closely linked to the need for an alternative to Hadoop's MapReduce, as it was developed at University of California, Berkeley in response to MapReduce's limitations. Spark's key innovation is resilient distributed datasets (RDDs), enabling in-memory data processing and faster computations.

Apache Spark also shares a deep connection with Databricks, a company founded by its original creators, providing a unified analytics platform for data engineers, scientists, and analysts. Databricks simplifies Spark adoption and enhances collaboration, making it a pivotal player in modern data analytics.

With high-level APIs in Java, Scala, Python, and R, Spark is accessible to almost any data professional, and the recent addition of Spark Declarative Pipelines to the framework further expands this accessibility. To get started transforming data in Spark, we recommend considering a managed platform, like Databricks, for the low barrier to entry and ease of getting started.

Database/SQL engines

With the rise of the serverless data warehouse, one might (validly) question if big data engines are still necessary. Serverless data warehouses can scale up or down to meet anticipated demand. Combined with the appropriate use of data lakes/lakehouses and external tables, they can be a highly efficient way to crunch and transform large amounts of data. Not only that, but the simplicity and ubiquity of SQL stands in stark contrast to the complexity of other technologies.

> At a certain point, it's likely your team will need to leverage Spark (or even Hadoop), but query engines, like BigQuery, Databricks SQL, and Redshift should not be disregarded as viable solutions, especially in the early stages of a data team. With recent advancements in in-memory computation (like MotherDuck and DuckDB), it's likely we'll see data warehouses' transformation capabilities continue to expand. Siding with history, *never* doubt the durability or extensibility of the relational database and plain-old SQL.

Other approaches

An alternative to traditional transformation methods is the use of low- or no-code platforms. Due to their user-friendly interfaces, these platforms are particularly suited for organizations with less technical expertise. While such platforms offer accessibility, this often comes at the cost of reduced customization, scalability, and control, so they're certainly not for everyone.

Relying solely on graphical user interface (GUI) tools might hinder your ability to fine-tune software to specific needs, leading to inefficiencies or functionality gaps. Additionally, GUI-based solutions can be less transparent and harder to version control, making it challenging to collaborate with other developers.

Other, more downstream methods of transformation—for example, BI tools or Google Sheets—are largely out of the domain of engineering. While they make sense for analysis or certain stakeholders, we recommend ownership of those solutions lie directly with users, i.e., analysts. Why? These solutions often lack the ability to be productionalized—a lack of continuous integration/continuous deployment (CI/CD), version control, and transformations as code make them unqualified for software development.

Building a Transformation Solution

Up to this point, we've discussed the pieces of the puzzle—languages, environments, and frameworks for transforming data—but how do we put those pieces together to create something valuable? The next section will discuss the transformation patterns and best practices on *structured* data used in modern data pipelines.

Data Transformation Patterns

In this section, we'll provide a suite of the most common data transformation patterns, along with examples of their application in a production setting. Data transformation is largely a pattern-mapping exercise—understanding *what* transformations exist and *when* they should be applied is most of the job!

Enrichment

This entails enhancing existing data with additional sources, for example, appending demographic information to customer records to provide a more complete picture. Enrichment might involve joining on internal structures or fetching additional data from external sources.

Example: Your raw orders table has a status column that represents the order status as an internal code. This is a non-human-readable code that makes analysis confusing. You join on orders to enrich status codes: "1337" to "complete."

Joining

Joining involves combining two or more datasets based on a common field, akin to a `JOIN` operation in SQL. Joining is vital when integrating data from disparate sources and plays a pivotal (no pun intended) role in data architecture. Which fact and dimension tables should be joined to improve analysis? Which should remain disparate? An understanding of different `JOIN` types is paramount.

Example: In talking with your analytics team, you realize that reports frequently analyze sales data by the originating country of the order, but that data isn't present in your cleaned sales table. Joining sales data with user data allows you to pull in the originating country and saves your analytics team time and confusion.

Filtering

When filtering, you select only the necessary data points for analysis based on certain criteria. This can reduce the volume and improve the quality of the data that is loaded into the target system.

Example: A new system should contain no data from years prior to 2025, but you find some errant records exist. You filter the data to eliminate erroneous records: `SELECT * FROM purchases WHERE date >= '2025-01-01'`.

Structuring

Structuring involves translating data into a required format or structure. This could mean transforming JSON documents into a tabular format or vice versa.

Example: A financial data API returns JSON data. You unpack and structure the data, then append it to a Parquet data store in an S3.

Conversion

Conversion is changing the data type of a particular column or field, like converting a string to a date/time format or an integer to a float. This is one of the most common types of transformation, especially when converting between semi-structured and structured data sources.

Example: An upstream API returns a timestamp in the string format "2011-01-01 00:00:00," but for easy retrieval and visualization, you cast timestamps to the corresponding format in your database. A more advanced date/time conversion might involve Unix timestamps, which can't simply be cast.

Aggregation

Aggregation is summarizing and combining data, such as calculating the total sales for a certain period or the average of a set of values. Data aggregation is essential for drawing conclusions from large volumes of data. As engineers, we often try to shift-right aggregation and bring it as close to analysis as possible. This ensures data can be analyzed at the finest grain. The goal of aggregation is to enable us to draw insights from vast amounts of data and inform business decisions, creating value from our data assets.

Example: You receive millisecond-level data from IoT sensor devices deployed by your services team. Because of the incredibly large quantity of information, you store a small subset of this data at a given time and aggregate on the second in downstream tables.

Anonymization

Anonymization is masking or obfuscation of sensitive information within a dataset to protect privacy. As data practitioners, we *must* do our best to protect the privacy of our users, for both legal and ethical reasons. Anonymization might look like hashing emails or otherwise removing personally identifiable information (PII) from records.

Example: Emails are hashed and stored as a unique identifier to prevent sensitive customer information from reaching downstream data users. Information is retained in tables with very limited access.

Splitting

Splitting, which is dividing a single complex data column into multiple columns, can be thought of as a simple form of denormalization.

Example: You would like to provide information on users' email domains while keeping data private. You split emails into prefix and domain columns, dropping the prefix to keep data anonymous, while retaining the domain.

Deduplication

The process of deduplication, part of data normalization, is the act of removing redundant records to create a unique dataset. Deduplication might occur through aggregation, filtering, or other methods.

Example: An event stream occasionally picks up duplicate events (with identical universally unique identifiers [UUIDs]) for a single occurrence. Stakeholders agree the earliest record should be retained. A transformation is created that drops duplicate event UUIDs.

Data Update Patterns

Part of transforming data is understanding how to update data that *already* exists in your target system. Here, we present ways of updating data to ensure smooth-running transformation processes:

Overwrite

The simplest form of updating data involves a complete drop of an existing source or table and an overwrite with entirely new data. While this is absolutely the simplest form of data update, it can quickly become untenable as the size of your data increases. Still, it might be a good starting point, given its simplicity.

Insert

Instead of overwriting data, you might choose to append new data to your existing source. This is the process of *inserting*. Inserting data only adds new rows; no existing rows are changed.

An ideal insert use case is one where new data is entirely independent of old data; for example, a table of transactions or stock market data for a given day. In these cases, simply incrementally inserting new records is a fast and simple way to maintain up-to-date data sources.

Upsert

Upsert, short for update and insert, is a more complicated update pattern, with applications for change data capture, sessionization, and deduplication, making it particularly useful, albeit complex.

Changed data is identified by a predefined unique identifier, and then pertinent records are updated and/or inserted. Given the inherent complexity of DIY upsert transformations, platforms like Databricks have `MERGE` functionality that drastically simplifies the process.

This simplification is just one of many benefits offered by using a declarative pipeline framework, which handles many of the complexities, allowing you to focus on the business logic you are implementing.

The following example from Databricks showcases `UPSERT` nicely:

```
MERGE INTO people10m
USING people10mupdates
ON people10m.id = people10mupdates.id
WHEN MATCHED THEN
 UPDATE SET
    id = people10mupdates.id,
    firstName = people10mupdates.firstName,
    lastName = people10mupdates.lastName,
    birthDate = people10mupdates.birthDate,
    ssn = people10mupdates.ssn,
    salary = people10mupdates.salary
```

```
WHEN NOT MATCHED
  THEN INSERT (
    id,
    firstName,
    lastName,
    birthDate,
    ssn,
    salary
  )
  VALUES (
    people10mupdates.id,
    people10mupdates.firstName,
    people10mupdates.lastName,
    people10mupdates.birthDate,
    people10mupdates.ssn,
    people10mupdates.salary
  )
```

Delete

The concept of data deletion is often misunderstood in data engineering. We encounter two main types of deletion: "hard" and "soft." A soft delete might involve marking a record as "deleted" (e.g., setting status = "deleted"), while a hard delete means completely removing the record from the database.

In the context of privacy regulations like the General Data Protection Regulation, there is a tendency to favor hard deletes. However, the utility of soft deletes should not be overlooked. For instance, in a software as a service (SaaS) company managing digital assets, soft deletes enable the creation of a historical record for an asset's status. This can be crucial for maintaining communication with users, analyzing account terminations, and other operational aspects. In contrast, hard deletes eliminate these historical records, which might be problematic in cases of data recovery.

Best Practices

While transformation can look quite different depending on the system, some themes emerge as constants. Consider these best practices when building out your transformation solution, regardless of where it lives:

Staging

We've already discussed how you can use a medallion architecture to stage data in a lake or warehouse, but it's worth reiterating. Staging should be a consideration whenever data is transformed, to protect against data loss (improve recoverability) and ensure a low time to recovery (TTR) in the event of a failure.

Idempotency

Idempotency is a fundamental concept that ensures consistency and reliability. While idempotent is a great vocabulary word, it can seem a bit opaque. However, it's actually a simple concept: doing something multiple times yields consistent results. In that sense, idempotency is like "reproducibility." Say you run a pipeline twice with no new source data—does your output data look the same? Idempotency is crucial for handling failures and ensuring data consistency in distributed systems.

Normalization and denormalization

These terms can be confusing to new data practitioners (I know I was confused when I first heard them). *Normalization* involves refining data to the clean, orderly format we'd expect. This can get complicated quickly, so we'll stay brief: normalized data is unique and sometimes has a primary key.

Denormalization is precisely the opposite. It involves duplicating records and information to make a system more performant. In large data systems, redundant elements can be utilized to improve the performance of analytic systems or downstream elements.[1]

Incrementality

On the topic of data updates, incrementality defines whether our pipeline is a simple `INSERT OVERWRITE` or a more complex `UPSERT`. The volume, compute, and cost will largely determine if nonincremental workflows are possible. There exist many predefined patterns for building incremental workflows in tools like dbt and Airflow. Databricks also has a robust library of resources for incremental pipelines.

[1] For a deeper dive on normalization and denormalization, see "Data Normalization Explained: How to Normalize Data" and "Data Denormalization: The Complete Guide".

> If you're loading data into a database, using `MERGE` instead of `INSERT` can do wonders. If you're loading data into a lake, using "stateful processing" or "bookmarking," i.e., keeping track of *where* a transformation left off, can be valuable.

Real-Time Data Transformation

In our discussion on real-time data in Chapter 1, we presented the differences between batch, micro-batch, and streaming transformations. Here, we'll simply note that micro-batch transformations can be effective streaming transformations when latency is low enough.

Streaming data transformation refers to the processing of data in real time as it is generated or received. True streaming transformations (Beam, Flink, Kafka) can introduce a level of complexity that most are unprepared for. Unless absolutely essential, we recommend a micro-batch approach with something like Spark.

Apache Spark, particularly through its components PySpark and Spark SQL, excels in this area by offering micro-batch/streaming transformations. These are simpler to implement compared with true, single-event transformations, which are more complex due to real-time data processing constraints like windowing.

Spark Structured Streaming is a popular streaming application among developers, as it efficiently handles incremental and continuous updates. Key features include streaming aggregations, event-time windows, and stream-to-batch joins, all facilitated by Spark's Dataset API in a more straightforward manner than other methods.

Although technically using a micro-batch processing engine, Spark Structured Streaming achieves latencies as low as 100 milliseconds, with exactly once fault tolerance, which is sufficiently low to be considered as "streaming." Additionally, Continuous Processing, introduced in Spark 2.3, can reduce latencies to as little as 1 millisecond, offering at-least-once guarantees, further enhancing its capability for streaming data transformation.

The Future of Data Transformation

The modern data stack is amid a *second* renaissance, spurred by new technologies addressing shortcomings in traditional data workflows. Concurrently, advancements in AI are reshaping not only how we work, but *the actual systems we build*.

On a seemingly weekly basis, new tools and technologies emerge that redefine what it means to transform data. Thankfully, the concepts and patterns provided in this chapter are timeless—regardless of your tooling, you'll need to be sure to adhere to time-tested strategies for managing data and creating cleaned assets to drive value at your organization.

The future of data transformation is simultaneously similar to and different from transformation today—similar in that the strategies, skills, and communication required to architect an efficient system will not change and different in that tooling will be supercharged and process automated...well, hopefully.

Supercharged tooling and automations can be a blessing and a curse, as they bring a tremendous amount of power to practitioners. But, as they say, with great power comes great responsibility. It's our job as engineers to be sure we can deliver well-planned and -executed transformation systems with a high value-to-cost ratio.

CHAPTER 3
Data Orchestration

Though we've already discussed ingestion (*E, L*) and transformation (*T*), we've only scratched the surface of ETL. Contrary to viewing data pipelines as a series of discrete steps, there exist overarching mechanisms that operate on a meta level, aptly dubbed "undercurrents" by Matt Housley and Joe Reis in *Fundamentals of Data Engineering*:

- Security
- Data management
- Data operations (DataOps)
- Data architecture
- Data orchestration
- Software engineering

In this chapter, we'll explore dependency management and pipeline orchestration, touching on the history of orchestrators, which is important for understanding *why* certain methods of orchestration are popular today. We'll present a menu of options for you to orchestrate your own data workflows and discuss some common design patterns in orchestration.

Throughout will be a discussion of how an “orchestrator” has historically been separate from a “transformation” tool. We’ll touch on why this *has been* true and why it *might not be* true in the future, though we still believe a separate orchestrator is the preferred approach.

What Is Data Orchestration?

Every workflow, data or not, requires sequential steps: attempting to use a French press without heating water will only brew disappointment, whereas poorly sequenced data transformations might brew a storm far more bitter than a caffeine-deprived morning (though the woes of the decaffeinated are not to be trivialized). In data, these “steps” are often referred to as tasks and “workflows,” or directed acyclic graphs (DAGs), a term we’ll dive into shortly.

Orchestration is a process of dependency management, facilitated through automation. The data orchestrator manages scheduling, triggering, monitoring, and even resource allocation. Orchestrators are distinctly different from schedulers, which are solely cron-based. Orchestrators, on the other hand, can trigger on events, webhooks, schedules, and even intra-workflow dependencies. Data orchestration provides a structured, automated, and efficient way to handle large-scale data from diverse sources.

Orchestration is, first and foremost, about ensuring pipelines produce *accurate and timely* results. A good orchestrator should also have a focus on efficiency, scalability, and speed, though as we’ll discuss shortly, operations will largely take place *outside* the orchestrator.

Why Orchestrate?

Orchestration steers workflows toward efficiency and functionality, with an orchestrator serving as the tool enabling these workflows. Typically, orchestrators trigger pipelines based on a schedule or a specific event. Event-driven pipelines are beneficial for handling unpredictable data or resource-intensive jobs. Here’s a breakdown of the perks that come with having an orchestrator in your data engineering toolkit:

Workflow management
: Orchestrators assist in defining, scheduling, and managing workflows efficiently, ensuring tasks are executed in the right order by managing dependencies.

Automation
: As engineers, we should be automating as much as possible (or feasible). Orchestrators can be used to automate routine, repetitive, and even complex tasks, saving time and ensuring tasks run as scheduled without manual intervention.

Error handling and recovery
: Orchestrators often come with built-in error handling and recovery mechanisms. They can retry a failed task, notify the team, or trigger other tasks to fix the issue.

Monitoring and alerting
: Monitoring workflows and receiving alerts for failures or delays are crucial for maintaining reliable data pipelines. Orchestrators provide these capabilities.

Resource optimization
: By managing when and where tasks run, orchestrators help optimize resource use, which is crucial in environments with limited or costly resources.

Observability and debugging
: Orchestrators provide a visual representation of workflows, log management, and other debugging tools, which is invaluable for troubleshooting and optimizing workflows.

Compliance and auditing
: Orchestrators maintain an audit trail of all tasks, which is crucial for compliance with data governance and other regulatory requirements.

Employing an orchestrator is a strategic step toward building robust, efficient, and scalable data pipelines, ensuring that data engineering processes are well coordinated, monitored, and managed.

The DAG

The "directed acyclic graph" is possibly the most unnecessarily complex term in data engineering. Borrowed from graph theory (for some reason), it's simply used to describe a "tree" of data execution where "tasks" are represented as nodes and "dependencies" are edges (Figure 3-1). These trees are:

Directed
: The execution of tasks follows a specific direction from one end of the tree to the other, indicating the presence of dependencies within the graph.

Acyclic
: There are no *cycles* in our tree; that is, if we execute route 1 → 2 → 5 in the image in Figure 3-1, we will not then execute 5 → 1 within the same pipeline.

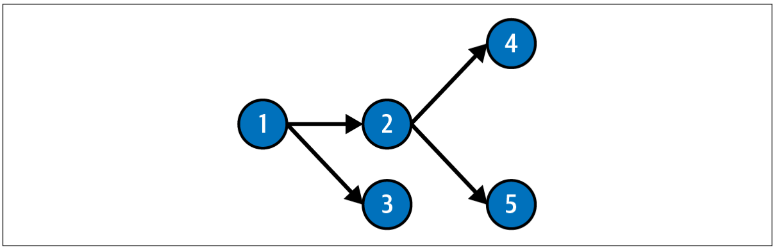


Figure 3-1. A very simple DAG

DAGs bring order, control, and repeatability to data workflows. By contrast, my life is an undirected cyclic graph—chaotic events seem to occur repeatedly without any rhyme or reason. Thus, I'm at least grateful for the standard DAGs impose on data engineering.

The main purpose of DAGs is to manage dependencies. Dependencies can get *very* complex *very* quickly. Before you know it, you'll be working with something that looks like a bifurcation diagram, which is apt, because chaos will likely ensue.

In the domain of data engineering, DAGs are pivotal for orchestrating and visualizing pipelines. They are the blueprint for mapping out a sequence of tasks, ensuring a structured and predictable flow of data. This becomes indispensable in managing complex workflows, particularly within a team or a large-scale setup.

Take a data pipeline where tasks are designated for data collection, cleaning, transformation, and, finally, loading into a database. A DAG serves as a clear roadmap defining the order of these tasks—ensuring, for instance, that data isn't prematurely loaded into the database before undergoing cleaning and transformation.

DAGs were popularized by workflow orchestration tools like Apache Airflow, which facilitates the creation, scheduling, and monitoring of data pipelines through the lens of DAGs.

Data Orchestration Tools

The journey of data orchestration tools reflects a narrative of remarkable evolution over the past few decades. Initially, data engineers resorted to legacy or makeshift solutions to orchestrate data workloads—ranging from custom Python scripts to adapted CI/CD tools. However, as data landscapes grew in complexity, the call for specialized tools grew louder, resulting in the creation of Apache Airflow and Luigi.

The trajectory of the field is on a continuum of evolution, driven by the open source community and the demands of big data. While Apache Airflow has emerged as a dominant force, there are a number of alternatives like Dagster, Prefect, and Mage. In addition, a more holistic approach in recent years offers orchestration as part of a unified solution of tools, such as Databricks' Lakeflow, that let you do ingestion, transformation, and orchestration right out of the box. It is important to choose the right tool for the job, as each tool has its own strengths and weaknesses. The future of data orchestration is likely to be a diverse ecosystem of tools, each with its own niche.

Choosing an Orchestrator

As you can likely tell by now, I enjoy analogy. Orchestration, quite literally, is what the conductor of an orchestra does. Thus, data orchestrators are aptly named, but the similarities run deeper. If the musicians in the symphony are the members of our data stack, it's the conductor who ensures they're playing in harmony and makes adjustments in tempo, phrasing, or repetitions, relaying their vision to the team.

Now, the important piece here is that the conductor is not also playing an instrument. Somewhere along the line, we confused orchestration with a data platform. To be fair, this is an easy mistake to make. For example, Airflow is simple enough to install and learn (and even easier to deploy, thanks to the availability of managed options), and, once everything is configured, you have this beautiful blank canvas to write.… Pure Python.

Rather, using orchestrators for their purpose, to orchestrate, makes the most effective use of their competitive advantage. Map out your dependencies in external services—Fivetran, dbt, Databricks, and the like—then use your orchestrator to trigger and monitor these services in the appropriate order. A conductor can't effectively conduct *and* play the trombone, so why do we expect our orchestrator to?

A consistent theme in this guide has been the need for hybrid tools that balance declarative and imperative frameworks to provide the flexibility of code with the rigor and efficiency of software engineering best practices. We feel that orchestrators are no different. You can either build such a framework yourself *or* pay a vendor/platform to do it for you—that depends largely on your situation.

Characteristics

With the above in mind, we can move on to considerations for selecting an orchestrator. Here are some characteristics to keep in mind:

Scalability

As data processing needs grow, orchestrators help in scaling workflows to handle increased loads. We'll discuss the benefits of containerized orchestration shortly, but consider whether your orchestrator will be able to scale vertically (increasing the number of parallel tasks) or horizontally (increasing the compute of each task).

As we've alluded, ideally the process of *orchestrating* your data will be separate from the process of *transforming* your data, so scale should be a matter of handling complex logic and dependencies, not necessarily compute. Think through what your workflows might look like with 10x the DAGs, dependencies, and tasks—is it manageable?

Code and configuration reusability

Unlike the more mature realms of software development, data engineering is notoriously bereft of established best practices. That's not to say they don't exist, only that they won't exist if you don't make an honest effort to ensure their implementation (choosing the right tools, ensuring frameworks exist, etc.).

Your orchestrator should foster code and configuration reusability, simplifying access to similar services and reutilization of common tasks across pipelines. For a good example of how this is managed in Airflow, see the Astro SDK. Anything that is done more than once should be converted to a function or class.

Connections

Orchestration, like data integration, is about *connections*. What do we mean? If a significant portion of your workflows transpire on a particular platform, housing your orchestrator there is sensible. Consequently, every major cloud provider offers a hosted version of Airflow—Amazon (Managed Workflows for Apache Airflow), Google (Cloud Composer), and Azure (Azure Data Factory Managed Airflow).

Platform-embedded alternatives to Airflow, such as Databricks Lakeflow Jobs, are arguably more aligned to customers with Databricks deployments. Since orchestrators serve to kick off external processes, ensuring a wide library of native connections, rather than flimsy webhooks, will give you the most visibility into the tasks you're orchestrating.

Support

Popular orchestrators have strong community support and continuous development, ensuring that the tool remains up to date with the latest technologies and best practices. When investigating an orchestrator, as with any tool, do some digging into how frequently you can expect updates. Also, assess the tool's maturity—can your team take on the deployment of a pre-v1 tool? Is frequent updating feasible, or is a more stable solution preferable?

If you're opting for a closed-source or paid solution, support is just as important. Do you have solutions engineers helping your team with implementation? Can you count on support to help you resolve issues? In some cases, these support teams can be as valuable as consultants, given their view into other organizations with similar implementations.

Observability

Areas of the modern data stack often bleed together. While orchestrators aren't *necessarily* observability tools, yours should provide you with insight into your transformation flows. Did a job fail? Was one retried?

Think about your own process. What makes the most sense for an alert system? Is it a Slack message, an email, a handwritten letter delivered by carrier pigeon?[1] Make sure your orchestrator supports whatever method best suits your team *and* provides the necessary visibility into the status of your data pipelines.

Orchestrator options

Today, to implement an orchestration solution, you can:

Build a solution

We do not advocate this option for the simple reason that the scale required for this to make sense is on the Uber/Airbnb-level.

Buy an off-the-shelf tool

This can be a valuable option for teams looking to abstract away DevOps work and simplify their deployment.

Self-host an open source tool

If you have the resources to self-host, this option can be a great way to avoid vendor lock. It should be noted that self-hosting comes with an entirely different set of challenges and headaches, however.

Use a tool included with your cloud provider or data platform

If you're heavily platform dependent, solutions like Azure Data Factory or Databricks Lakeflow Jobs can be a simple, effective way to orchestrate your workflows.

There are pros and cons for every approach. We'll discuss a few options, starting with the tool that serves as the basis for a number of cloud native options (Amazon Managed Workflows, Google Cloud Composer, Azure Data Factory Managed Airflow) and warrants its own category, Apache Airflow:

[1] Note: we do not advocate this method of alerting, unless you're Johannes Kepler and the year is 1612.

Apache Airflow

Developed by Airbnb, Airflow has blossomed into the de facto orchestration tool, with a vibrant community, massive adoption, and support by *every* major cloud provider. It remains a popular choice, thanks to its ease of adoption, simple deployment, and ubiquity in the data space.

There's no denying that Airflow has not only been instrumental in the development of the modern data stack but also that it *continues* to be an effective tool for orchestration. However, we've observed a number of flaws inherent with the tool.

Conceived in the early 2010s, Airflow was engineered to orchestrate, not to transform or ingest. Despite Astronomer's commendable efforts, fundamental barriers in transformation workflows persist within Airflow due to its original blueprint.

Airflow is deceptively easy to spin up, but much more difficult to build at scale and in production. The lack of constraints allows for simple missteps, often surfacing without seasoned guidance.

We *highly* recommend checking out newer tools in the orchestration and transformation space to see just what's out there. If Airflow is your only choice—great news, it's still an excellent tool for orchestration with a wealth of community support.

Other open source tools

Many popular orchestrators, Airflow included, are built on open source with the option for paid, hosted service and support. There are a number of newer tools that are innovating in the space, with Prefect and Dagster the leaders in both funding and adoption. A number of newer still, pre-v1 tools have been on the rise, like Mage, Keboola, and Kestra.

Choosing an open source option can be attractive for a number of reasons, like community support and the ability to directly modify source code. Of course, open source tools are *heavily dependent* on such support for continued development. Additionally, choosing pre-v1 software will likely get you fast updates and new features but at the risk of project abandonment and instability.

When considering a solution, take into account its history, support, and stability. Compare that with the requirements of your team, both now and in the future. Many open source tools offer an option for paid hosting and support. If your team is in a position to offload the DevOps work of managing a hosted instance, it might be worth researching. Otherwise, be sure to understand the support options (community) of your chosen tools *and* the complexity of deployment.

Paid, closed-source tools

As with managed open source options, paid solutions come with a number of benefits: support with predefined service-level agreements (SLAs) around response times, possible architecture consulting/guidance, and *complete removal* of deployment headaches.

The main drawback is vendor lock on proprietary tooling. Because most data teams quickly find themselves with dozens or even *hundreds* of data pipelines, this can be incredibly costly in the long term. We recommend thoroughly vetting *any* paid proprietary solution for reputability, functionality, and longevity.

Platforms

Data platforms offer their own orchestration solutions that are tightly coupled to other services on the platform, like Lakeflow Jobs. These orchestrators are ideal for teams that are all-in on a particular platform and offer:

Improved observability

Integrated directly into the platform, these solutions have a maximum level of visibility into the granular details of pipeline runs.

Maximum compatibility

As we've discussed, the orchestrator exists to interface with external services. A platform-embedded option is necessarily compatible with other tools on that platform.

Increased support

As with other paid services, platform tooling provides you with paid support and solution engineers whose job is to help your team succeed. This service should not be dismissed and can be invaluable for a new data team that needs pseudo-consulting on implementation details.

While platform-specific orchestrators increase vendor lock and come at a price, they can drastically reduce the work necessary to maintain and scale your solution. That value should not be dismissed…and if you're on a platform like Databricks already, their solutions make quite a bit of sense.

Orchestrating SQL

Thus far, we've been largely concerned with orchestrating entire data systems, but there's a subset of data transformation that has its own concept of orchestration—relational databases. Often, data is written to a source system, then transformed sequentially with complex SQL queries and manipulations.

Before the modern data stack, teams relied on custom infrastructure to manage these transformations alone. Today, we use tools like dbt, Lakeflow Declarative Pipelines, Dataform, or SQLMesh. These tools are *orchestrators* in the sense that they evaluate dependencies and conditions, then optimize and execute commands against a database to produce a desired result.

This overlap gains significance in the context of data orchestration. For instance, consider an Airflow DAG that:

1. Ingests data to a stage.
2. Lightly transforms the data.
3. Triggers a dbt project.

The interesting thing is that #3 is actually just a DAG—one that gets executed against a data warehouse (Figure 3-2).

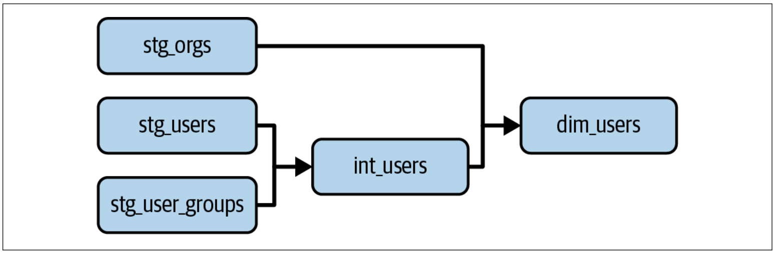


Figure 3-2. A dbt pipeline—looks familiar, no?

However, this structure hints at a potential limitation. There exists a lineage outside of the SQL orchestrator and another within, leading to a need for a mechanism to observe data across these layers—from ingestion to transformation.

This is a common pitfall many data teams encounter, often resulting in a disconnection between sources and cleaned data. Identifying errors in downstream data becomes a formidable challenge: discerning whether an anomaly was introduced by an analyst or a data engineer is like finding a needle in a haystack.

Lineage not only streamlines the orchestration process but also fosters a more comprehensive view of data observability, bridging the gap between different data layers and ensuring a smoother error detection and resolution process. This is one reason to choose a platform-specific data orchestrator—greater visibility, both between and within data workflows.

We'll discuss observability further in Chapter 4, but for now, we'll simply mention that SQL-based transformation frameworks are largely *orchestrators*, and finding an orchestrator that delivers end-to-end observability can be invaluable for ETL workflows.

Design Patterns and Best Practices

Design patterns, when employed judiciously, can significantly enhance the efficiency, reliability, and maintainability of data orchestration processes within a modern data engineering environment.

These patterns exist outside the considerations mentioned in "choosing an orchestrator," since *most* can be accomplished independent of your orchestration choice. It should be noted, however, that some orchestration solutions make these patterns significantly easier—an important consideration in selection:

Backfills

If you're building a data system, data likely existed prior to its implementation. That means you'll need to backfill the old data before you start recording new data. Many will fall prey to "one-time thinking" and create a hacky solution that they can't replicate.

A best practice is that *anything done once should be repeatable*. In orchestration, we have an excellent opportunity to build backfill logic into our pipelines themselves—a best practice is to ask, "If this data disappeared tomorrow, could I recreate it?" While that exact scenario might not happen, it's likely that you'll need to add a column or pick up a few extra dates in the future.

In tools like Airflow, we recommend building DAGs that allow you to simply set a past *start_date* to trigger a run that recreates historical data. This is sometimes easier said than done and will require *idempotent* pipelines.

Idempotence

We discussed idempotence in Chapter 2, but it's worth reiterating—idempotence, or ensuring doing something multiple times yields consistent results, is *essential* for reliable data engineering. It's especially important when running a backfill—if you accidentally have overlapping dates, will data be duplicated? Can we run a pipeline for past dates, in addition to future ones?

Event-driven orchestration

Event-driven orchestration allows for reactivity to changes in data or system states. Triggering on events means your data will be as up to date as possible or occur at just the right time. For example, if your data warehouse is heavily dependent on a Fivetran ingest job, you might trigger it once that completes. That way, you can avoid weird cron syntax or the potential that a long-running job overlaps and causes issues. Databricks' Lakeflow Jobs, for example, offers triggering based on new data file arrival or table update events.

Event-driven orchestration can also be used to lower cost. Have a burdensome pipeline with infrequent source data? You can build a system to only trigger that pipeline when the source updates.

Conditional logic

The interesting thing about data and ETL, specifically, is that inputs are always changing. The question isn't "Will it break?" but "When will it break, and how frequently?" As such, an orchestrator should be able to handle conditional logic—"if True then X, if False then Y"—to direct data workflows.

Conditional logic means your team will be able to further automate the ETL process. If a common exception arises in a pipeline, adding conditional logic can account for errors, failures, or malformed data, allowing your team to free itself from exception-handling hell.

Concurrency

Given the existence of this book, we know that ETL is hard (or at least not very easy). It's highly likely one or more of your pipelines will be bottlenecked by a large amount of small, individual tasks. For example, you might have to ingest thousands of tiny zipped CSVs. Looping through each would take hours, especially with the wrong execution scheme.

Instead, we can employ *concurrency*, sometimes referred to as dynamic tasks, to save time. Orchestrators are obviously capable of running concurrent pipelines, but most can fan out tasks to be executed simultaneously. Figure 3-3 shows an example of a concurrent, dynamic pipeline, executed in Apache Airflow.

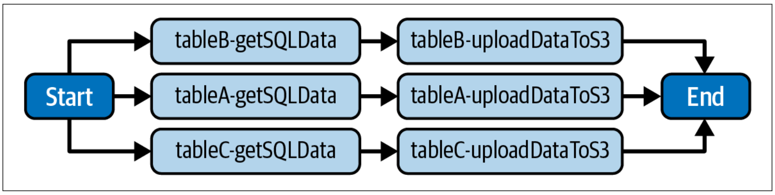


Figure 3-3. Dynamic task mapping is one way to programmatically create tasks executed in parallel in Apache Airflow

Fast feedback loops

Orchestrators are inherently complex and often trigger actions that are difficult to test. This makes them notoriously difficult to develop locally. We've heard the phrase "the easiest way to stop a local Airflow instance is to reboot my machine," which, while hyperbole, has a ring of truth.

We recommend investing significant time and energy into finding a tool that allows you to fail fast and identify errors quickly, achieve parity with a cloud-deployed instance, and have a local environment that's developer friendly and quick to spin up.

With the ease of deploying infrastructure (thanks to containerization), it's also possible to deploy something that's incredibly cumbersome to maintain and extend. If you find yourself in this trap, it can be a very, very painful experience.

Retry and fallback logic

In the same way we need to be prepared for backfilling pipelines, retries and failures are inevitable. In a complex data stack, handling failures well ensures data integrity and system reliability, which, in turn, facilitates smooth data operations and reduces downtime.

Part of idempotent pipelines is handling failures in a way that doesn't omit or duplicate data, but rather sets up a scenario for either retrying the operation or skipping and alerting the proper parties. Most orchestrators will allow you to directly set a "retry" parameter at both the task and pipeline level.

Conditional logic can be paired with retry/fallback logic to create scenarios that gracefully handle errors, lowering both the time you spend triaging them *and* the stress levels across your data team.

Parameterized execution

Along with *retry* and *conditional logic*, *parameterized execution* allows for further malleability in your orchestration. Adding parameters simply means allowing your orchestrator to accept variables.

This can be a valuable way to not only handle different cases but also reuse pipelines for multiple purposes. Parameterized execution can be invaluable in facilitating backfills, for example. By architecting a DAG that accepts a *date* as a parameter for fetching data from an API, for example, you could create a simple structure to backfill a pipeline by sequentially executing a list of dates to backfill for.

Lineage

Lineage refers to the path traveled by data through its lifecycle. Given the visual nature of the DAG, it's also a great place to understand the different actions taken on data. Be sure your lineage solution is robust, as lineage is instrumental for debugging issues and extending pipelines. Ideally, strive for *column-level lineage*: a column-level view of data's journey, from ingestion to analysis.

Column-level lineage is illuminating the pathway of data through transformation pipelines, amplifying traceability and debugging prowess. This granularity is poised to become an industry norm in SQL orchestration. I would argue column-level lineage should be *table stakes* in any new implementation. Advanced lineage capabilities are one obvious benefit of a platform-integrated orchestration solution—like Unity Catalog and Lakeflow Declarative Pipelines.

Pipeline decomposition

A good strategy for ensuring tidy, readable pipelines is to break them down into smaller, more manageable tasks that facilitate better monitoring, error handling, and scalability (Figure 3-4).

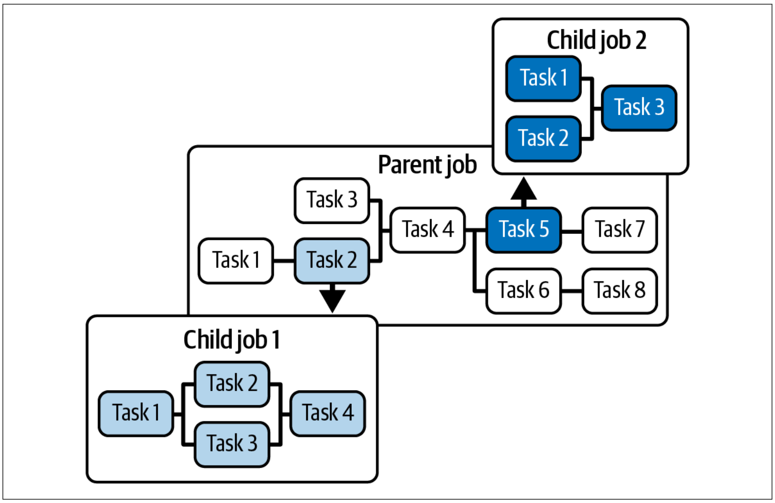


Figure 3-4. An example of pipeline decomposition—breaking pipelines into microservices and orchestrating them with meta-DAGs

Borrowing from software engineering—take pipelines as microservices: a key function of microservices is their autonomy and ability to operate independently of each other. Aim to build autonomous DAGs that can be run in parallel to mitigate dependencies and critical failures. Modular design will also make it easier to build and debug your workflows.

The Future of Data Orchestration

Wondering why this guide has such a strong focus on the history and future of the modern data stack? In a rapidly evolving industry, grasping trends is crucial—it can be the difference between adopting a tool that will become obsolete in a few years and one that will gain traction.

The right tools, gauged in terms of time and resource savings, can catapult well-led data teams far ahead of their counterparts. Although we've expressed reservations about merging orchestration and transformation tooling in this chapter, the evolution of transformation tools, as discussed in Chapter 2, might change the narrative.

Containerized infrastructure, which facilitates both vertical and horizontal autoscaling, along with a new breed of tools aiming to eliminate complex big data tooling (like MotherDuck, PyArrow, modern orchestrators), may reduce the demand for powerhouse transformation solutions. Hybrid transformation tools like Prefect and Dagster warrant serious consideration.

The evolution in SQL orchestration augurs well for enhanced observability and monitoring within our data warehouses (or lakehouses). Emerging tools like SQLMesh are poised to challenge established players like dbt, although dbt continues to innovate and regularly update its platform.

As platforms mature, solutions like Lakeflow Jobs are becoming increasingly appealing from a plug-and-play perspective. The fact that these solutions are native to their respective platforms instills confidence in their functionality and seamless integration.

Staying up to date in such a rapidly changing environment is a Herculean task, but that's why I have a job writing guides like these. Regardless, these developments should be exciting, since they'll enable us to do our jobs more effectively and deliver what we're all after: quality data in a timely, robust manner.

CHAPTER 4

Pipeline Issues and Troubleshooting

As we've alluded, the primary objective of any ETL is to deliver business value. This goal, however, is often hindered by fragile systems characterized by prolonged recovery times, leading to excessive resource allocation toward troubleshooting rather than innovation and value creation.

A well-designed ETL pipeline, therefore, must be robust and maintainable, embodying key characteristics such as high data quality, efficient error handling, and effective issue identification—all integral components of system observability. Regardless of your engineering prowess, your systems *will* fail—not everything is within our control!

Planning for failure means engineering pipelines that:

- Are easy to maintain and extend—that is, allow for quick error triage *and* new feature development
- Provide automated ways to handle errors in real time and recover from failure
- Incorporate a framework for improvement based on learning and experience

This approach ensures maintainability and seamlessly ties into another critical aspect of ETL systems: scalability.

As we delve deeper, we'll explore how maintaining a system goes hand in hand with scaling it, addressing challenges and strategies to ensure your ETL pipeline is both resilient and adaptable to evolving business needs.

Maintainability

In a chapter on issues and troubleshooting, we're mainly addressing the topic of maintainability—quite literally, the ability to maintain the things you've already built.

As engineers, we sometimes goal-seek on solutions exclusively. While we're huge advocates for minimum viable products and the 80/20 principle, when collaborating on systems to drive business value, the ability to *maintain* and *scale* (the topic of Chapter 5) those systems is imperative. The inability to maintain a pipeline will come at a direct and indirect cost to your team. While the former is obviously more salient, the latter is more likely to sink your battleship:

Direct cost

Most are quite aware of direct cost—if you want something expensive, you usually have to pony up and make a good case for it. When interest rates were near zero and venture capital money flowed like the Rio Grande, many overlooked the cost of things like cloud computing, storage, and DataOps. Efficiency starts with pattern design. Bad patterns mean low efficiency across an entire organization.

An ETL system with a low maintainability will incur a high direct cost in the form of inefficient operations and long runtimes. This will not only incur a high bill from your provider of choice, but it will also result in slow jobs and delayed data from your team. Low maintainability is often the result of a fragmented stack—a patchwork of integrated tools is more costly to maintain. This is why many organizations opt for a unified ETL solution.

Indirect cost

Indirect costs can *far* outpace direct costs. Creating systems that fail often and require constant triaging can be dismal for resource allocation. Your team's time and energy are the most valuable commodities you have.

> Spending the majority of your working hours fixing DAGs, responding to alerts, and fighting fires might make you *look* busy, but generally manifests itself as unsustainable and unproductive. Teams that win build efficient systems that allow them to focus on feature development and data democratization. Inefficient systems mean more fire drills, more sleepless nights, and fewer hours spent building things that matter.
>
> The main selling point of SaaS is that the benefit outweighs the cost, namely the cost of developing and maintaining in-house solutions. If you're on a team with limited resources or experience, the most maintainable solution might be one you buy.
>
> Of course, these outages don't just impact data engineers—those downstream are also affected. That means a loss of trust from stakeholders, loss of revenue from customers, and even loss of reputation from the general public. The indirect cost of a data failure, especially one that exposes PII, can be catastrophic.

So, why focus on maintainability? Winning teams are built on maintainable systems.

If your goal is to move your organization forward with timely, efficient data, you'll need to minimize issues and troubleshooting, lowering the direct and indirect costs of operation. In the rest of this chapter, we'll discuss precisely how you can do that through observability, data quality, error handling, and improved workflows.

Monitoring and Benchmarking

Here, we make the distinction between observability and monitoring/benchmarking. Benchmarking and monitoring systems is essential—a subset of observability. But observability isn't just about troubleshooting and maintenance. Monitoring and benchmarking our systems is required for minimizing pipeline issues and expediting troubleshooting efforts.

Tightly monitored and neatly benchmarked systems are set up quite nicely as "observable" and make it easier to improve the maintainability of our data systems and lower the costs associated with broken data. Without proper monitoring and alerting, your team might not even know things have gone wrong! The *last* thing we want is for stakeholders to be the ones to discover a data error…or worse, the error to go undiscovered and poor business decisions to result.

We should observe data across ingestion, transformation, and storage, handling errors as they arise (gracefully, if possible) and alerting the team if (when) things break. This is where unified ETL solutions shine, providing a unified view of every aspect of your pipeline.

Observability isn't just for troubleshooting, however. It also helps us scale, as we'll discuss in Chapter 5.

Metrics

Here are some essential measures used to assess the reliability and usefulness of data within an organization. These metrics help ensure that the data being collected and processed meets specific standards and serves its intended purpose effectively:

Freshness

Freshness refers to the timeliness and relevance of data in a system. It's the measure of how up to date and current the data is compared with the real-world events it represents. Maintaining data freshness is crucial in ensuring that analytics, decision making, and other data-driven processes are based on accurate and recent information. Data engineers work to design and implement systems that minimize latency in data updates, ensuring that stakeholders have access to the most current data for their analyses and operations.

Common freshness metrics for a dataset include:

- The length between the most recent timestamp (in your data) and the current timestamp
- The lag between source data and your dataset
- The refresh rate, e.g., by minute, hourly, daily
- Latency (the total time between when data is acquired and when it's made available)

Volume

Volume refers to the sheer amount of data that needs to be processed, stored, and managed within a system. It's a fundamental aspect of data handling. Dealing with data at scale presents challenges, such as efficient storage, quick retrieval, and processing speed. High data volume demands specialized infrastructure and techniques like distributed computing, parallel processing, and data compression.

Volume metrics include:

- The size of a data lake (gigabytes, terabytes, petabytes)
- The number of rows in a database
- The volume of daily transactions in a system

Quality

Quality involves ensuring that data is accurate, consistent, and reliable throughout its lifecycle. Data quality revolves around accuracy, consistency, reliability, timeliness, completeness, security, documentation, and monitoring. Addressing these aspects guarantees high-quality data for informed decision making and analysis.

Here are some sample data quality metrics:

- Uniqueness: are there duplicate rows in your dataset?
- Completeness: how many nulls exist? Are they expected?
- Validity: is data consistently formatted? Does it exist in the proper range, e.g., greater than zero?

Methods

Monitoring implies the ability to see everything that's happening across your stack and detect errors in a timely fashion. Data quality means implementing strict measures that improve the quality of the observed data.

The following are patterns and techniques you can apply to directly improve the *quality* of the data you're monitoring, hopefully leading to fewer errors and less downtime:

Logging and monitoring

The first step to debugging is to check the logs. But to check logs, there have to be logs to check! Be sure that your systems are logging data verbosely. For systems that *you* build, make logging mandatory. Define and codify logging best practices, including libraries and *what* to log. Not only must you be an exceptional data plumber, but you also have to be a data lumberjack, a term we deem much more flattering.

Lineage

Simple in concept, lineage—the path traveled by data through its lifecycle—is one of the most important ways to observe your data. Having visual and code representations of your pipelines and systems is important for everyday runs, and will save you endless amounts of time in triaging and debugging issues.

For lineage to be useful, it needs to be both complete and granular. Complete in the sense that all systems are observed, including interconnectedness between systems. Ideally, your lineage will be at the most granular level possible. For tabular data, that's the column level. Column-level metadata gives the most granular insight possible. It enhances a team's ability to triage errors, simplifies workflows, and improves productivity…and the experience of working on data. As shown in Figure 4-1, lineage exercises start with the column.

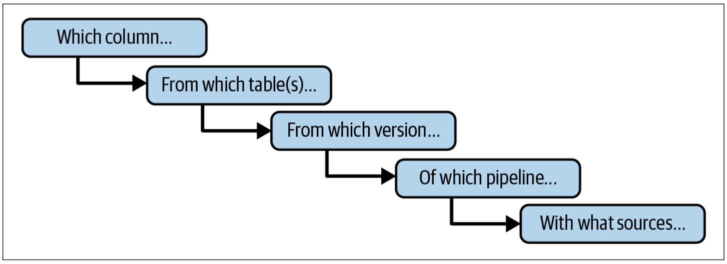


Figure 4-1. A first principles assessment of data origins reveals the complexities of lineage

A good start is implementing systems with self-contained lineage solutions. A more complete lineage system might observe *all* of your data processes. Managed platforms are good for this, since they're generally self-contained. Other tools, like Monte Carlo, might provide insight into the entire stack.

Anomaly detection

The trickiest thing about data is that you can't see it all. Most of our errors arise from stakeholders or analysts returning unexpected results, which begs the question, "What gremlins are alive in my data today?"

Anomaly detection is one way to know that, within some threshold, anomalous data does not exist. Anomaly detection systems work through basic statistical forecasts to analyze time-series data and return data that lies outside some confidence interval.

Anomaly detection can be a great way to catch errors that might originate outside your systems—for example, a payments processing team introducing a bug that underreports purchases in Europe. No alarms will sound, but an anomaly detection system pointed at the right table will catch the bug.

Data diffs

Data engineering contains an additional dimension software engineering does not: data quality. Changing code changes outputs in ways that can be difficult to understand. What's the trickiest thing about data?

Data diffs are systems that report on the *data changes* presented by changes in code. These tools will inform row and column changes, primary key counts, and more specific effects. Their primary purpose is to be sure accurate systems stay accurate.

For data diffing solutions, we recommend tools like Datafold. Newer SQL orchestrators, like SQLMesh, also have data diffing functionality. Data diffing is tied closely to CI/CD and is accented nicely by assertions and tests. We'll discuss all of the above shortly.

Assertions

Assertions are constraints put on data outputs to validate source data. Different from anomaly detection, assertions are much simpler. For instance, you might say, "Assert that users.purchases only contains prices in the plans.pricing table." If a value that does not exist in pricing appears in purchases, you know either (*a*) a new price was introduced or (*b*) there's an error in your system.

While manual in nature (assertions require some business context and understanding of what should be), assertions are one of the few ways that we can be entirely confident data is not erroneous (in the way we specify, at least). For assertion solutions, check out libraries like Great Expectations or look for systems that have the built-in ability to define assertions, such as the expectation capabilities in Lakeflow Declarative Pipelines.

Errors

Now that we've discussed observation and preventing errors, we come to the simple truth. Regardless of how robust your solution is, there will still be errors! Thus, error handling becomes incredibly important, both for your sanity and that of your team.

Separate from handling errors is recovering from their effects, whether that be lost data or downtime. An important part of recovery is retrospectives, postmortems, and reflections on how to prevent similar errors in the future.

Error Handling

Error handling is how we *automate* error responses or boundary conditions to either keep our data systems functioning *or* alert our team in a timely and discreet manner. The following approaches detail some methods for processing errors gracefully and efficiently:

Conditional logic
: When building your data pipelines, conditional logic can be useful for unreliable or inconsistent sources. The ability to say "If X then A, else if Y then B" adds a powerful component to your orchestration and transformation tooling. Seek out solutions that allow conditional logic.

Retry mechanisms
: Systems, even well-built ones, fail. Unforeseen errors can cause one-off API timeouts or other oddities. For that reason, even well-functioning code can produce errors. Retry logic is important in any orchestration tooling. Of course, for these to work, sensible retry settings must be configured. Be sure to incorporate a retry strategy that can handle data oddities or nuances but that isn't wasteful and doesn't result in endless runs.

Pipeline decomposition
: We previously mentioned the concept of modularity in Chapter 3, but breaking pipelines down into "microservices" is an effective way to keep the impact of errors contained. Consider building DAGs and systems that *only* require absolutely necessary tables and connections.

Graceful degradation and error isolation

Error isolation is enabled through pipeline decomposition—whenever possible, systems should be designed to fail in a contained manner. You wouldn't, for example, want your product usage data to live upstream of a financial reporting pipeline, or else an unrelated failure might have your CFO returning late metrics to the board, a situation that will end well for precisely no one.

Graceful degradation is the ability to maintain limited functionality even when a part of the system fails. By isolating errors and decomposing pipelines, we're effectively enabling graceful degradation. You might experience an error that only one part of the business notices because the rest of your systems work so well. Trust us, this is far better than errors that *everyone* notices.

Alerting

Alerting should stand as a last line of defense—receiving alerts is necessarily *reactive*. We see that something bad has happened and drop everything to fix it. While *proactive* resolution of common errors is best, the unexpected will prevail eventually. Alerts might come in the form of an email or Slack message—we prefer Slack, since it's highly visible and team members can add comments with helpful context.

When alerting, be cognizant of alert fatigue. *Alert fatigue* refers to an overwhelming number of notifications diminishing the importance of future alerts. Isolating errors and building systems that degrade gracefully, along with thoughtful notifications, can be powerful mechanisms for reducing alarm fatigue and creating a good developer experience for your team.

Recovery

So we now have insight into our systems, we're actively enforcing quality constraints, and we have dedicated methods for handling errors. The final piece is building systems for recovering from disasters, which might include lost data. The following are a few methods and concepts that will help you bounce back after the inevitable failure:

Staging

We've mentioned staging quite a bit thus far, but an additional benefit of staged data is disaster recovery. With Parquet-based formats like Delta Lake and patterns like the medallion architecture, time travel makes it possible to restore data (up to a certain point). While staged data should be treated as ephemeral, it's an important pattern for redundancy.

Backfilling

It's likely that you'll either (*a*) want data from some period before you began running your pipeline or (*b*) have lost data that needs to be backfilled. *Backfilling* is the practice of simulating historical runs of a pipeline to create a complete dataset. For example, Airflow has a backfill command that runs a pipeline for every date between two dates.

When building systems, keep backfills in mind. Systems that are easy to backfill will save you quite a bit of time when something breaks. Seek out tools that support simple backfills, as backfill logic is something that can get very complex, very quickly. Idempotent pipelines will also make your life easier. Check your orchestrator of choice for backfill functionality out of the box.

Improving Workflows

As much as we wish improving processes were entirely within our control, it's not. Our job is inherently collaborative—proverbial "plumbers," we give stakeholders pipelines for internal and external data. This makes our job inherently collaborative.

We've mentioned it briefly, but data engineering is truly a question of *when* things break, not if. Even in the best systems, anomalies arise, mistakes happen, and things fall apart. The *Titanic* is a lesson for a reason, right?

The deciding factor becomes your ability to adapt and resolve issues…and that's the point of this chapter! Starting with systems that prioritize troubleshooting, adaptability, and recovery is a great way to reduce headaches down the line. In this section, we'll provide a few ways for you to continually improve your processes.

Start with Relationships

Consider the following example: semi-regularly, your software team changes a production schema without warning. This not only breaks your daily ETL job from the prod dataset, it also runs the risk of *losing* data, since your export method lacks CDC.

Understandably, this leaves your team pretty frustrated. Not only are you losing time and resources by fixing these issues, but they *seem* entirely avoidable. Before reacting harshly, ask yourself the question, "Is the software team *trying* to reduce overall productivity and invoke worse outcomes?" If the answer is yes, we recommend a job search, but in 99.999% of cases the answer will be "No, of course not."

In fact, if you ask yourself exactly what that team is trying to do, it's probably the same thing you are—do the best work possible given your resources. Great, now you have empathy for their motivations.

Next is to understand their workflows and communicate your frustrations. From there, you can begin to craft a process to improve efficiency. Here are some ways you can ensure healthy relationships through a structured, pragmatic approach:

SLAs

Service-level agreements (SLAs) are common in the provider space for runtime and uptime guarantees, but they can be used just as effectively within and between teams. If you're struggling with data quality, consider an SLA that formally defines things like *performance metrics, responsibilities, response and resolution times, and escalation procedures* so that *everyone* has a clear understanding of what's required for incoming data.

By communicating requirements clearly and assigning ownership, SLAs can be a surprisingly effective way to improve the quality of data that's outside of your control, just by writing a few agreements down on paper.

Data contracts

Data contracts have gained traction in the past few years as an effective way to govern data ingested from external sources. Popularized by dbt, contracts are a type of assertion that check for metadata (usually column names and types) before executing part or all of an ETL pipeline.

We like the term "contract" because it implies an agreement between two parties, like an SLA. We recommend first defining an SLA, even if it's simply a back-of-the-envelope agreement, then implementing that SLA in the form of data contracts on external assets. If (when) a contract returns an error, the SLA will dictate exactly whose responsibility it is to resolve that error and how quickly it should be expected.

APIs

APIs can be a more formal method of enforcing contracts and SLAs. In a sense, SQL is itself an API, but there is no reason internal data can't be fetched (or provided) via an API. APIs are just a method of transmitting an expected set of data, but, implemented correctly, they provide an additional layer of standardization and consistency to the source. APIs also come with more granular access control, scalability benefits, and versioning, which can be useful for compliance.

Compassion and empathy

You might be used to these terms appearing in texts where dbt is capitalized and refers to something other than a transformation tool, but they're just as important in engineering as in psychology. Understanding your coworkers (and partners) and their motivations, pain points, and workflows will allow you to effectively communicate your concerns *and* appeal to their incentives.

In this digital age, it's far too easy to take an adversarial approach or assume ill intent, especially from vague video meetings and terse snippets of text. We advocate going the extra mile to understand your coworkers, whether through one-to-ones, long-form written communication, or in-person meetings, where possible.

Align Incentives

To foster meaningful progress, we need to align incentives and outcomes.[1] If your team's only mandate is to "build lots of stuff," little time will be spent on making that stuff resilient and robust.

[1] This section is adapted from a Monte Carlo guide by Barr Moses, "12 Data Quality Metrics That ACTUALLY Matter".

Setting key performance indicators (KPIs) around common incident management metrics can help justify the time and energy it takes to do the job right:

Number of incidents (N)
: Counting the number of incidents over some timeframe will provide you with a window into the frequency of incorrect, incomplete, or missing data.

Time to detection (TTD)
: TTD describes the average time it takes for an incident to be detected.

Time to resolution (TTR)
: This metric gauges the swiftness with which your systems can resume normal operations after a disruption. It's a direct measure of your system's resilience and recovery capabilities.

Data downtime (N × [TTD + TTR])
: Using the above three metrics, we can arrive at an average "data downtime." This summary metric can help you understand the severity of your outages and health of your systems.

Cost
: From downtime, you can calculate the cost of these failures. Cost is highly specific to your team and organization. We recommend a bespoke cost calculation, factoring in the specifics of your deployment.

Improve Outcomes

Thomas Edison's famous quote, "I've not failed, I've just found 10,000 ways that don't work," aptly applies to the process of building exceptional data pipelines and the frameworks that support them. The journey toward excellence in this field is marked by a series of educated guesses, experiments, and, crucially, the ability to adapt and correct course when faced with challenges.

If you've followed along thus far, you know great systems are built on great frameworks. Here are a few ways to iterate and improve your processes in the wake of failures and adjust your good pipelines to make them *great*:

Documentation

If your data is staged and backfill-able, the only thing left is to know how to fix it! Be tedious and pedantic when it comes to documenting your systems processes and code. While it might feel like a chore, we can guarantee it will be less time-consuming (and stressful!) then trying to reengineer your code during a failure.

Postmortems

In any large event or outage, a postmortem can be valuable for analyzing the failure. A postmortem involves reflecting on what went wrong and performing an analysis to understand *why*. Postmortems, and reflection in general, are excellent ways to learn, educate, and grow. Ideally, your postmortems will lead to fewer events that require recovery in the first place.

Unit tests

Unit testing is the process of validating small pieces of code (the components of a system) to ensure they produce results as expected. Like any other engineering system, code in a data engineering system should be unit tested. That means any custom code or bespoke systems you create should have tests to ensure they're producing desired results.

These are specifically different from *assertions*, since unit tests check the underlying code rather than the output data. Building unit tests into your code is an excellent preventative practice to minimize future errors.

While many platforms are slowly adopting unit tests/assertions, there are a surprising number of data teams operating without them. We advocate their adoption in *every* data system.

CI/CD

Continuous integration/continuous deployment (CI/CD) is a term that refers to how you integrate and deploy your code—that is, how *changes* (like pull requests) are assimilated, tested, and rolled out to your code base.

In practice, CI/CD for data engineering might include any number of things we've already discussed and a few things we haven't. Unit tests, assertions, linting, and data diffs will ensure a consistent code base that functions well and allows you to seamlessly collaborate with others to build something awesome (at scale).

Simplification and abstraction

More of a design pattern than a tool, as an engineer, you should seek to simplify and abstract as much complex logic out of your code as possible. This not only makes it easier to collaborate, but it also reduces the likelihood that you will introduce errors.

When you're writing a piece of code, think to yourself, "If I don't look at this for six months, how difficult will it be for me to understand it?" If something breaks in six months, you'll likely be looking at it under some form of pressure, so keep that in mind.

Data systems as code

Data diffs hit on a concept that we've yet to discuss—building data systems as code. By versioning and codifying *every* system, it becomes possible to roll back changes and revert to previous states. In a sense, staging systems with a medallion architecture let us do something similar with time travel.

Building your data systems with software engineering best practices and implementing logic as version-controlled code will drastically improve your observability, disaster recovery, *and* collaboration. Be wary of any tooling that is difficult or impossible to version control or otherwise manipulate through code, even if only JSON or YAML configs.

With responsibilities defined, incentives aligned, and a full monitoring/troubleshooting toolkit under your belt, you're ready to begin automating and optimizing your data workflows. Recognizing that some tasks *cannot* be automated and edge cases will always exist, the art of data engineering is balancing automation and practicality. Now, with robust, resilient systems, we're ready to scale.

CHAPTER 5
Efficiency and Scalability

In this final chapter, we focus on the crucial aspects of optimizing and scaling the data pipelines we've developed. We start by defining what we mean by "efficiency" and "scalability" to set the boundaries for our discussion.

Our journey begins with resource allocation, which hinges on a thorough understanding of our operational environment. This understanding enables us to optimize our processes effectively.

The chapter culminates with a dual-focused discussion. First, we explore the process of collaboration, particularly how to scale effectively in terms of team size and skill set. Second, we delve into creating an optimal developer experience, a key factor in efficient data pipeline management.

Throughout the chapter, we weave in ongoing themes such as tooling and platform considerations, the pros and cons of managed versus custom-built solutions, and architectural strategies for crafting superior ETL systems. These discussions aim to provide a comprehensive view of building and maintaining efficient, scalable data systems.

Efficiency and Scalability Defined

Efficiency is about optimizing workflows to deliver business value through data. It measures our ability to generate impactful outputs with the resources at our disposal, encompassing aspects of code, services, and teamwork. The ultimate measure of efficiency is the impact produced relative to the finite resources used.

Scalability refers to the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth. In the context of data transformation, it's about the ability of data processing systems to handle increasing volumes of data and more complex transformation tasks without compromising performance or *efficiency*.

Imagine a data transformation system that initially handles data from a small ecommerce website. As the business grows, so does its data—from thousands to millions of customers, transactions, and product interactions. Scalability in this context means the system can expand its data processing capabilities to manage this growth.

Data teams frequently encounter trade-offs between *price* and *performance* when scaling data transformation systems. Higher performance often comes with higher costs—be it through more powerful compute, sophisticated solutions, or expanded cloud storage.

In data, as in life, balance is key. Though it's easier said than done, our goal is to find solutions that provide *just* the right amount of performance at a sustainable price. Usage-based pricing, for example, can be a great way to start assessing services, though it typically is eclipsed by subscription or flat pricing at some point. Data teams must evaluate the cost-effectiveness of scaling decisions, ensuring they're getting the optimal performance for their investment and not overspending for unnecessary capacity.

Unified ETL

Another consideration that is often overlooked is the overhead and complexity imposed by the need to integrate multiple tools. More complex integration often goes hand in hand with lower efficiency and with difficulty in scaling. Imagine your ETL pipeline uses one tool to ingest data and another to transform it—fail to seamlessly integrate the two, and you may find bottlenecks or failures when ingestion scales but transformation lags behind. This is where unified ETL tools are beneficial—having everything in one place can help with both efficiency and scalability. For example, Databricks introduced Lakeflow, which includes Lakeflow Connect for ingestion, Lakeflow Declarative Pipeline for transformation, and Lakeflow Jobs for orchestration—all fully integrated and built to work together efficiently and at scale.

So how do we balance efficiency, scalability, and cost-effectiveness?

As we discussed in Chapter 4, setting clear metrics around data quality, freshness, and volume is crucial. These metrics can justify investments in optimization or increased computing resources. For example, if your team needs to deliver data hourly but the current system is too slow or expensive, these metrics make it easier to argue for optimization or budget adjustments.

Metrics around efficiency and scalability can be useful too. Here are a few examples of key results (KRs) around efficiency/scalability:

- Increase average throughput speed by 25%
- Maintain a latency of <200 ms in streaming jobs
- Reduce data storage costs by 15%
- Increase spot instance usage by 30% without degrading performance

KRs can be obtained by setting KPIs that correlate with their outcomes:

Processing time
The time taken to process a given volume of data

Throughput
The amount of data processed in a given timeframe

System uptime
The percentage of time a system is available and operational

Error rate
The frequency of errors in your data pipelines

Aligning team OKRs and KPIs with stakeholder expectations ensures that resources are appropriately allocated to scalability and optimization efforts.

Understanding the tools and resources at your disposal, which we'll discuss in this chapter, along with constraints like data freshness or quality, allows you to fine-tune your workflows for optimal performance and cost.

Understand Your Environment

Scale in data engineering is fundamentally rooted in architecture. Establishing the right framework and foundations is crucial for effective scalability. While data advancements have made it easier to start building, this ease also brings the risk of creating systems that function initially but become maintenance burdens or struggle to scale.

As the adage goes, hindsight is 20/20, and every implementation will inevitably lead to lessons learned and opportunities for optimization. The key to successful scaling is a deep understanding of your environment, enabling the construction of a robust and enduring foundation for your data systems.

Frameworks

Optimizing a system begins with a thorough understanding of it. Thus, before we dive into allocating resources, improving efficiency, and optimizing workflows, we have to *understand* the framework we're operating on.

The better you comprehend execution mechanics, the more effectively you can optimize your tasks. If you're using Pandas in Spark, for example, spend time reading up on best practices; the docs are always a good place to start. If you're running SQL against BigQuery, understand query plans and execution order. Once again, docs can be your savior.

Other fundamental concepts for scalability to consider include: distinguishing between OLAP and OLTP databases, comprehending Spark's logical and physical plans, differentiating between spot and on-demand instances, and understanding how your chosen distributed compute system—be it Databricks, Kubernetes, or even AWS Lambda—allocates resources and scales.

Resource Allocation

With a robust architectural foundation in place, the scalability of your data systems significantly hinges on effective resource allocation. Having the right tools is only part of the equation; it's how you utilize them that truly counts. The key is to leverage strengths in the most appropriate context. So, what factors should we consider in resource allocation? The next sections discuss some critical aspects to keep in mind.

Parallelization and concurrency

If you're not utilizing a fully managed, serverless platform for your data workflows (which is a feasible option), you will generally have a degree of control over how your ETL pipelines are executed. In the current landscape, distributed computing, which harnesses the power of clusters and nodes, is predominantly conducted on shared, virtual machines located in data centers around the world.

Your setup might involve a self-hosted execution environment, such as Kubernetes, or a shared one, like Databricks. Each option presents its own set of considerations in terms of resource allocation, scalability, and maintenance.

Clusters

When managing ETL workflows, an important decision you'll face involves the configuration of clusters. Clusters are collections of servers working together as a unified system, with each server referred to as a "node." These nodes are responsible for executing and monitoring your workflows.

Clusters can be all-purpose or job-specific; that is, they can be used for ad hoc queries and general analysis or specific tasks in your data pipelines. The process of configuring clusters involves decisions about the number and types of nodes. These choices will affect key factors like memory and processing power, directly influencing the compute capacity and associated costs.

A nuanced understanding of cluster configuration allows for the customization of your workflows to various setups, or even the automation of scaling and configuration. This enables you to strike an optimal balance between cost efficiency and performance. In the coming paragraphs, we'll explore some effective techniques for managing clustered execution.

Spot versus on-demand cluster instances

Spot instances offer a cost-efficient solution, typically priced significantly lower than on-demand instances, making them a practical choice for budget-sensitive projects. They are particularly effective for tasks that can accommodate flexibility and tolerate interruptions, such as batch processing or data analyses that don't necessitate immediate results. However, their availability is variable, reliant on market demand, and providers can reclaim them with little warning, potentially disrupting ongoing tasks.

In contrast, on-demand instances guarantee continuous service, offering reliability and consistent performance. They are ideally suited for tasks that are critical to operations and require real-time processing, where consistent availability is crucial. However, this level of reliability and consistency comes at a higher cost compared with spot instances.

The decision to use spot or on-demand instances should be based on the specific needs of each task. This includes considering factors such as the importance of cost efficiency and the ability to handle potential disruptions with spot instances, against the need for stability and uninterrupted performance with on-demand instances.

Pooling

Pools are utilized in data engineering to avoid the costly process of repeatedly creating and destroying resources. Specifically, a pool refers to a group of clusters that are kept idle and ready to use.

Opting for a platform or system that supports the use of pools can significantly enhance the cost efficiency of your solutions. By having clusters readily available, pools can reduce the time it takes to start clusters and auto-scale them.

At the same time, employing on-demand instances for tasks that require quicker execution times or have stringent requirements can enhance performance speed. This approach allows for a balanced use of resources, optimizing both cost and efficiency in your data engineering workflows.

Cluster sharing

Shared clusters offer a cost-effective approach to resource allocation in data engineering. By enabling multiple users to attach to and execute their workloads concurrently on the same compute resource, shared clusters lead to notable cost savings. They simplify the management of clusters and facilitate comprehensive data governance, including precise access control measures. Despite the implication of the term "shared," these clusters provide a secure method of reducing costs through efficient resource utilization.

Autoscaling

There are two types of scaling. *Horizontal scaling* refers to increasing the number of nodes or machines upon which a task is executed; *vertical scaling* means increasing the size or power of existing resources. For instance, if a database is running low on storage, you can vertically scale it by increasing its storage capacity. Alternatively, to improve the performance of a large, resource-intensive job, you might opt for horizontal scaling by adding more machines. Figure 5-1 provides a visual representation.

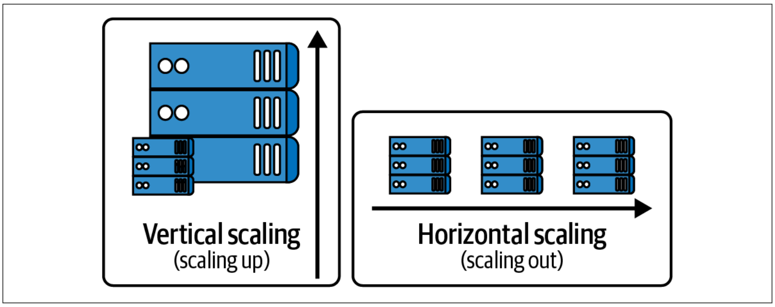


Figure 5-1. Vertical versus horizontal scaling—both have an appropriate time and place

Autoscaling encompasses the tools and techniques that automatically adjust your resources to suit fluctuating workloads. A classic example is U.S. retail websites on Black Friday, when there is a significant spike in web traffic. Maintaining high-level compute resources throughout the year would be prohibitively expensive. Instead, autoscaling can be employed to efficiently handle these traffic surges, even when they're unexpected.

Nowadays, an increasing number of cloud services offer autoscaling features, ranging from data warehouses to Spark executors.

Autoscaling as a managed service can save significant time and resources, reducing the burden of DevOps tasks. However, this convenience comes with a cost, and the degree to which you utilize managed autoscaling services should be based on your team's specific needs and stage of development.

Serverless

Serverless computing is a model where backend services are provided based on actual usage rather than a predetermined allocation of resources. While servers are still integral to this model, the distinctive aspect is the pricing structure: companies using serverless computing services are billed based on their usage, not on a fixed bandwidth or a set number of servers.

Serverless solutions, exemplified by BigQuery and Databricks services like Databricks SQL and Lakeflow, offer the ability to autoscale according to fluctuating workloads. This adaptability significantly reduces—or, in some cases, eliminates—the necessity for meticulous tuning of resources. While serverless compute often has a bespoke pricing structure, the elimination of certain maintenance makes it an attractive option for many.

Data Processing Techniques

Regardless of how your resources are allocated, you'll still need to process data for optimal efficiency. Here are some concepts to consider when you're building out your pipelines, whether that's in a warehouse or a data lake.

Incremental processing

Processing entire datasets can become prohibitively expensive, especially as the volume of data increases. While truncate and reload is a straightforward approach, it reaches a point of being unfeasible or excessively costly at scale. Incremental processing, which involves adding only new data (`INSERT`) or updating existing data while inserting new ones (`UPSERT`), presents a more scalable alternative. Incremental processing logic can get complex quickly—we recommend seeking out tooling with prebuilt patterns for incremental processing, like dbt incremental models, change data capture, or Databricks `MERGE`.

Column-oriented data stores

For data and analytics, column-oriented formats and databases offer substantial performance enhancements. Since data analysis often involves reading by column, storing data in a columnar format is inherently more efficient. In contrast, traditional OLTP databases like Postgres are row-oriented, which suits scenarios where data is frequently written or read one row at a time, as in production environments.

Today, column-oriented stores are preferred by data and analytics teams. To achieve optimal analytic performance, consider utilizing data formats such as Parquet and Avro and databases like BigQuery and DuckDB. See Figure 5-2 for a visualization of the two data stores.

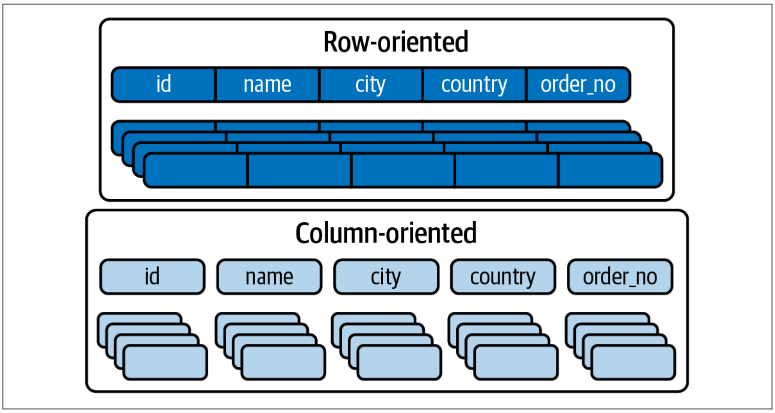


Figure 5-2. Row- and column-oriented databases play pivotal roles in data and app development but have vastly different use cases

When these columnar storage methods are combined with partitioning (or another way of segmenting data), they create an exceptionally powerful and efficient pattern for data handling. Databases like Databricks SQL are built directly on Parquet-based formats (Delta Lake), unlocking cost optimizations and performance enhancements, like liquid clustering.

Data partitioning

The act of "partitioning" data simply means breaking it up into parts. Partitioning is highly important for read/write performance and should *not* be overlooked. Using formats like Parquet and libraries like PyArrow, partitioning data can be simple.

The effectiveness of partitioning becomes particularly pronounced with columnar datasets. Let's take a common scenario—a dataset containing 100 days of data across 10 columns. Consider the query:

```
SELECT user_id FROM data WHERE date = '2024-01-01'
```

In an unpartitioned, row-based dataset, this query would scan every row and column, resulting in slow execution times and increased compute costs.

However, in a column-oriented database partitioned by date, the query efficiently processes only the necessary data. If we assume an equal distribution of data across days, accessing data from one column for one day in a partitioned columnar database is one thousand times more efficient. That might not make much of a difference in one query, but stack that up over hundreds of dashboards, queries, and pipelines and you can quickly reap benefits. Figure 5-3 provides an illustration of how partitioning works to split datasets and improve performance.

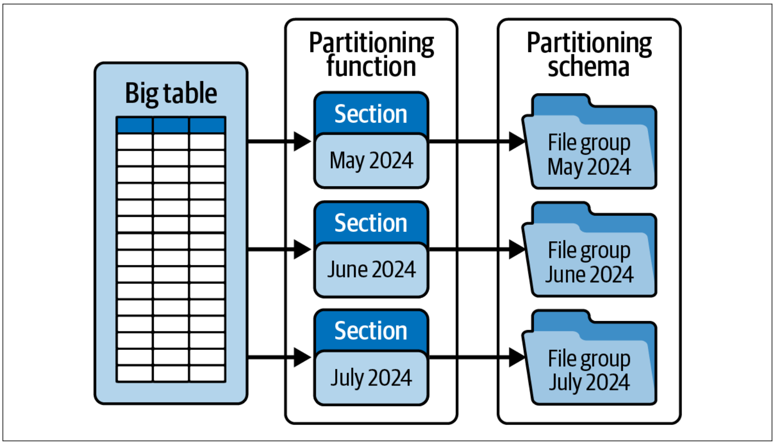


Figure 5-3. Partitioning data improves both query performance and storage efficiency

Managed services, which eliminate the need for creating and maintaining partitions, are also worth exploring. Partitioning can be a difficult process that requires domain knowledge and thoughtful engineering to create and maintain. Systems that automatically partition, order, and optimize datasets may very well be worth their cost.

Materialization

Materialization refers to the creation of physical data stores, namely tables. This term might be a bit misleading in the digital age, as "physical" is more conceptual than literal. Modern databases often support "views," which are essentially virtual representations of data derived from specific queries. For instance, you could execute a query like `SELECT * FROM users WHERE status = 'active'` and define this as a *view*, active_users. This view then functions as if it were a table, providing a convenient way to access analytics-ready datasets that need minimal transformation from the source data.

An "external view" refers to a view created on data stored externally—for example, in a data lake. A growing trend involves storing semi-structured data in a data lake and querying it directly as an external view. This approach represents a "shift right" in data engineering, where the bulk of data transformation is moved to the data warehouse. It allows analysts and other stakeholders to interact directly with semi-structured data, streamlining the process.

When planning data transformations, it is crucial to carefully consider whether to use tables, views, or materialized views, as each has its own implications and benefits in the architecture of data systems.

Process Efficiency

It's essential to have a functioning prototype or production solution in place. Once you have a working system, observability becomes key. This means having the ability to monitor and measure the performance of your solution. Questions to consider include: How well does the solution manage resources? Are there any bottlenecks? What occurs post-execution?

It’s crucial to reassess your initial assumptions and decisions once you’ve obtained insights from observing your system in action. Engineering, by nature, involves continuous learning and adapting based on new insights that weren’t apparent during the planning phase. Optimization is an ongoing process, a constant cycle of reevaluating requirements, seeking simplification, and exploring automation.

Data (Engineering) Democratization

Data democratization refers to making data-driven decision making accessible across an entire organization, beyond just the data team. Commonly associated with analytics, it often entails providing self-service analytics tools that enable stakeholders to explore data independently, without relying on an analytics team.

For data engineering teams, data democratization involves creating systems and frameworks that simplify resource creation and ETL job development. Here’s an example:

> A stakeholder needs to enrich data with a new custom API data source (absent from a connector, like Fivetran). The analytics team comes to you asking for an ETL job to pull from the data source, write to a stage table, and ingest into your data warehouse.

Now, many I’ve worked with would say, “Sure, we can fit that in five sprints from now.” That’s a lose-lose from the business perspective: it adds to the data engineering backlog *and* delays important supplemental data. Five sprints and two months later, the company has rolled out new KPIs, the initiative has been relegated, and the pipeline sits unused. When so far divorced from the *user* (our stakeholders!), EOD turns into “end of December,” and things become much less efficient.

Democratizing data engineering means building systems and platforms that are self-service. The goal of these systems is to enable analysts or other stakeholders to do most of the legwork, unblocked for work they need. Building a system follows this process:

1. Identifying common requests or patterns that data engineers frequently encounter
2. Assessing the necessity of these requests
3. Simplifying the process as much as possible

4. Exploring SaaS offerings or custom solutions usable by less technical users
5. Continuously simplifying and optimizing
6. Reaping the benefits

Such systems might still require code reviews or pull requests, but they should largely operate autonomously. This could involve setting up a repository where analysts submit SQL and YAML to automatically generate an Airflow job, or employing hybrid GUI/code solutions for interactive pipeline development.

Tools facilitating this approach are becoming increasingly prevalent (see dbt Canvas and Lakeflow Designer). The ability of data teams to implement such systems is emerging as a key differentiator between teams that effectively scale and those bogged down with endless requests—the differentiator between a team that scales and one whose JIRA board is filled well into the next year.

Data democratization represents a unique aspect of scalability. It alleviates bottlenecks in data teams by enabling self-service, whether through low-code, no-code, or custom in-house platforms. Without self-service capabilities, data teams risk being perpetually overwhelmed by ad hoc requests.

Developer Experience

Efficiency and scalability in ETL systems focus on optimizing and scaling code workflows, but for the data engineering teams behind these systems, it's about enhancing the developer experience (DevEx). Noda et al. define the "core dimensions" of DevEx as flow state, feedback loops, and cognitive load,[1] represented in Figure 5-4.

[1] Noda, A., et al. 2022. "DevEx: What Actually Drives Productivity? The developer-centric approach to measuring and improving productivity." In *Proceedings of the 2022 ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)* (pp. 32–47). ACM. *https://dl.acm.org/doi/10.1145/3610285*.

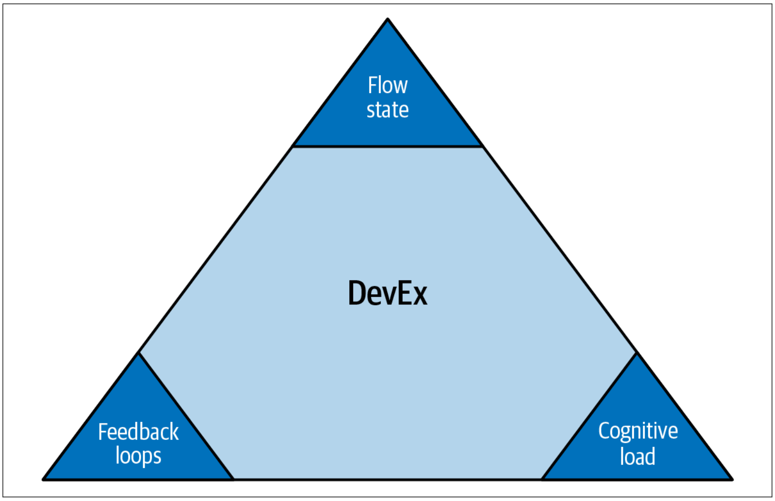


Figure 5-4. The developer experience can be broken down into three main areas: flow state, feedback loops, and cognitive load

Feedback loops

Feedback loops can be defined as the speed and quality of responses to actions. Consider the process of developing a DAG in Airflow. Can you develop locally with ease? When you make a change, can you immediately see how that change affects your system? Your data? We believe data engineers should target a "frontend DevEx." Frontend engineers can code in one window and see real-time changes in another, then deploy changes with a few simple commands. How can you build your systems to produce a similar local DevEx?

Cognitive load

Cognitive load is the amount of mental processing required for a task. How hard is it to configure your local development environment? To create a new dbt model? To understand the existing tables, infrastructure, and metadata in your environment? Are you commonly switching between seven tools to do your job? As engineers, we should seek to simplify and thus minimize the cognitive load to development. Invest in systems that make *configuration* and *building* as simple as possible so you can focus on impactful work.

Flow state

Flow is the mental state of immersion, with feelings of focus, involvement, and enjoyment. Everyone has heard of flow state, and it's what we're all after. How do we get there? Doing *meaningful, impactful* work with short feedback loops and a low cognitive load. The first component is largely driven by interest, so we can't help you there, but tuning your systems to promote swift development in a low-stress environment will help you find flow.

Collaboration

Our last section is about collaborating. Big surprise—you'll get more done with others than you will by yourself! In the world of data engineering, this means striving to create solutions that are not only effective but also easy for others to understand, extend, and engage with:

Infrastructure as code

Every piece of your infrastructure should exist as version-controlled, testable code. This will allow you to leverage technologies like Git and providers like GitHub to build with the rigor of an engineer. Yes, you'll likely find yourself combing through Stack Overflow for "that one Git command" that you forgot, but it will be much better than breaking something and having to rebuild it entirely.

This is *especially* important with GUI-based low-/no-code tools. Don't completely write them off, but be sure that your solutions *at least* have a versioned configuration file and some way to back up/restore settings. The last thing you want is a tool that requires 10 clicks to rebuild a pipeline. The first time you experience a failure, the nightmare of rebuilding those one hundred pipelines will not be fun.

Documentation and knowledge sharing

Documentation is often looked at as a chore, but there's an art to communicating and authoring effective wikis. *Anything and everything* you do should be documented and saved off for posterity (and yourself). Let us save you the trouble; it can be a nightmare to try to remember that thing you built last year *or* to try to dig into someone else's code without any resources. Document the code you write. Document the data. It's guaranteed to be worth the effort.

Conclusion

Efficiency and scalability in data engineering extend far beyond the realm of constructing the quickest and highest-performing pipelines. They encompass a broader understanding of resource limitations, which we all inevitably face, as well as the vital roles of collaboration and the necessity of making trade-offs.

While technical expertise in coding and implementation of efficient algorithms and data structures is crucial, the softer skills are equally important. Effective communication, thorough documentation, and adept team management are all integral components of a successful data engineering practice. Our aim with this chapter, and indeed with the entire guide, is to equip you with a comprehensive skill set that enables you to scale your data pipelines effectively and quickly—to quote a similarly ambitious fictional astronaut—"to infinity and beyond."

Conclusion

Thank you for engaging with our technical guide, *Understanding ETL*. In this journey, we've explored the intricacies of data ingestion, delved deep into the nuances of data transformation, and unraveled the complexities of ETL undercurrents: orchestration, troubleshooting, and scalability. Along the way, we've also shed light on the evolving landscape of data engineering, focusing on enduring methodologies for crafting clean and reliable datasets.

While we've aimed to cover concepts with lasting relevance, it's important to recognize that the field of data engineering is likely to evolve significantly in the coming years. However, the fundamental process of ETL—extracting data from a source, transforming it, and loading it into a target—is expected to remain a cornerstone, regardless of future technological advancements or changes in the methods of building ETL systems.

This guide has provided a foundational overview of data engineering, intending to help you identify and fill gaps in your knowledge. The landscape of data can be vast and complex, but understanding the key components and the bigger picture is crucial in navigating it effectively.

Beyond the technical aspects, we've woven in themes like solution architecture and enhancing the developer experience, both of which are pivotal in data engineering and beyond. Emphasizing the importance of understanding your coworkers, users, and the challenges they face will not only make you a more proficient engineer but also lead to more precise and effective solutions. Adopting a philosophy of relentless simplification and measured decision making can be transformative in your work.

Finally, we encourage you to actively participate in the wider data community. Whether it's contributing to open source projects, attending conferences, writing articles, or sharing insights on social media, your involvement can be immensely fulfilling. Engaging with the community not only fosters personal growth and learning but also contributes to the collective advancement of data engineering practices. Good luck—we hope to see you out there!

About the Author

Matt Palmer is a developer experience engineer at Replit, with a background in product analytics and data engineering. In his free time, he enjoys writing, hiking, and strength training. He lives in the Bay Area and spends most of his weekends exploring California.