

Data Engineering — Essential Concepts

A developer-focused reference covering the core ideas every data engineer needs to know.

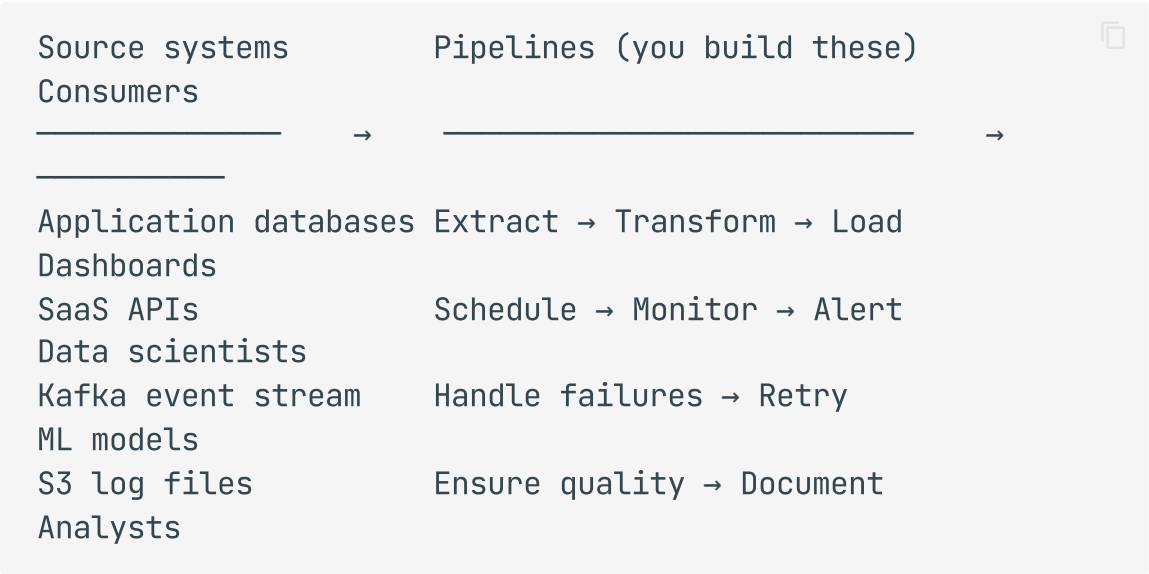
Last reviewed 29 Sep 2026 ·

Prerequisites: None — good place to start

Related: [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) · [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

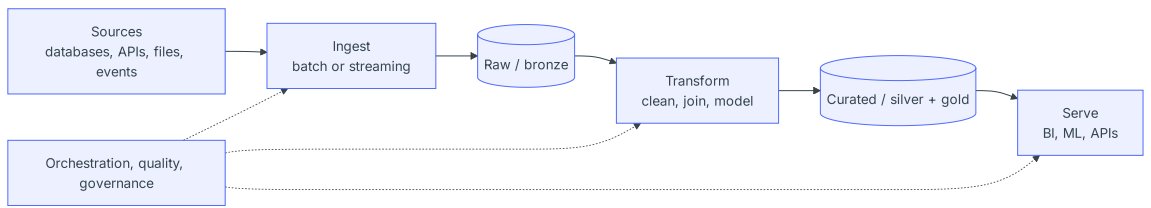
Overview

Data engineering is the discipline of building and operating the systems that move data from where it is produced to where it is used — reliably, on time, and with known quality.



Role in the organization: Source systems produce data and analysts, data scientists, and applications consume it. The data engineer owns what happens in between: ingestion, transformation, storage layout, orchestration, quality, and the reliability of every step.

Typical responsibilities: - Build a pipeline that loads payment events from a SaaS API into the warehouse every hour - Repair a scheduled pipeline after the source API changes its schema - Optimize a slow SQL transformation that exceeds its time window - Monitor freshness and quality, and alert when data is late or incorrect - Trace a discrepancy between two reports back to its root cause



On this page - [Foundations](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#foundations) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#foundations) - [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#data-modeling) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#data-modeling) - [File Formats](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#file-formats) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#file-formats) - [ETL vs ELT](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#etl-vs-elt) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#etl-vs-elt) - [Partitioning & Clustering](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#partitioning--clustering) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#partitioning--clustering) - [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#data-quality) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#data-quality) - [Streaming Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#streaming-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#streaming-concepts) - [Orchestration](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#orchestration) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#orchestration) - [Key Tools Landscape](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#key-tools-landscape) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#key-tools-landscape)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/#further-reading)

Foundations

OLTP vs OLAP

Two fundamentally different workloads that drive almost every architecture decision.

	OLTP	OLAP
Stands for	Online Transaction Processing	Online Analytical Processing

	OLTP	OLAP
Purpose	Run the application	Answer business questions
Operations	INSERT, UPDATE, DELETE	SELECT (mostly reads)
Query shape	Many small, fast queries	Few large, slow queries
Data volume	Current state (GB range)	Historical (TB–PB range)
Schema style	Normalized (3NF)	Denormalized (star/snowflake)
Examples	Postgres, MySQL, Aurora	Redshift, BigQuery, Snowflake

The job of a data pipeline is usually to move data **from OLTP** → **OLAP**, reshaping it along the way.

Batch vs Streaming

	Batch	Streaming
When data moves	On a schedule (hourly, daily)	Continuously, as events arrive
Latency	Minutes to hours	Milliseconds to seconds
Complexity	Lower	Higher
Reprocessing	Easy — rerun the job	Harder — replay from log
Use case	Nightly reports, data warehouse loads	Fraud detection, live dashboards, alerting
Tools	Spark, dbt, SQL scripts	Kafka, Flink, Spark Structured Streaming

Lambda architecture — runs batch and streaming in parallel; merges results at query time. Complex to maintain two code paths.

Kappa architecture — streaming only; reprocess historical data by replaying the event log. Simpler, but requires a replayable log (e.g. Kafka with long retention).

Data Lake, Data Warehouse, Data Lakehouse

Data Lake - Raw storage for any data format (structured, semi-structured, unstructured) - Schema-on-read — you define structure at query time, not load time - Cheap object storage (S3, GCS, ADLS) - Risk: becomes a "data swamp" without governance

Data Warehouse - Structured, curated, query-optimized storage - Schema-on-write — data conforms to a schema on ingestion - Fast analytical queries via columnar storage - Examples: Snowflake, BigQuery, Redshift

Data Lakehouse - Combines lake storage costs with warehouse query performance - Open table formats (Delta Lake, Apache Iceberg, Apache Hudi) add ACID transactions, schema enforcement, and time travel on top of object storage - Examples: Databricks (Delta), Snowflake on Iceberg, BigQuery with open formats

```
Raw files (S3/GCS)
  ↓ ingest
Bronze layer - raw, as-is, append-only
  ↓ clean + validate
Silver layer - cleaned, deduplicated, typed
  ↓ aggregate + model
Gold layer   - business-ready tables, metrics
```

This Bronze → Silver → Gold pattern (Medallion architecture) is the standard way to organize a lakehouse.

Columnar vs Row Storage

Row storage (CSV, Postgres heap): entire row written together. Fast for retrieving a full record. Slow for scanning one column across millions of rows.

Columnar storage (Parquet, ORC, Redshift internal): each column stored together. Fast for analytical queries that scan a few columns. Compresses extremely well because similar values are adjacent.

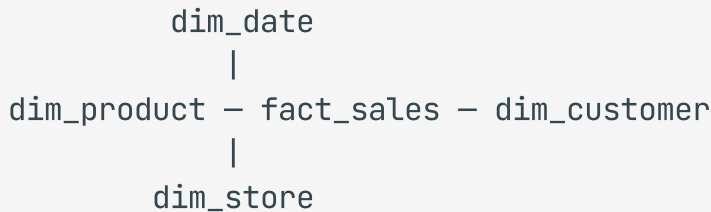
```
Row store:    [id=1, name=Alice, salary=90000] [id=2,
name=Bob, salary=85000]
Column store: [id: 1,2,3...] [name: Alice,Bob,Carol...]
[salary: 90000,85000,92000...]
```

A query like `SELECT AVG(salary) FROM employees` reads only the salary column in columnar storage — skipping name, email, department entirely.

Data Modeling

Star Schema

The standard analytical model. One central **fact table** surrounded by **dimension tables**.



- **Fact table:** numeric measurements (revenue, quantity, duration). One row per event. Large.
- **Dimension table:** descriptive context (who, what, where, when). Smaller. Joined to the fact table.

```
-- Typical star schema query
SELECT
  d.year,
  c.region,
  p.category,
  SUM(f.revenue) AS total_revenue
FROM fact_sales f
JOIN dim_date d ON f.date_key = d.date_key
JOIN dim_customer c ON f.customer_key = c.customer_key
JOIN dim_product p ON f.product_key = p.product_key
GROUP BY d.year, c.region, p.category;
```

Snowflake schema — dimension tables are further normalized (e.g. `dim_product` → `dim_category`). Saves storage but adds joins. Star schema is usually preferred for query performance.

Fact Table Types

Type	Description	Example
Transaction fact	One row per event	Each sale, each click
Periodic snapshot	One row per period per entity	Account balance at end of each month

Type	Description	Example
Accumulating snapshot	One row per process instance, updated as it progresses	Order lifecycle (placed → shipped → delivered)

Slowly Changing Dimensions (SCD)

How do you handle dimension data that changes over time — e.g. a customer moves cities?

Type	Strategy	Tradeoff
SCD Type 1	Overwrite old value	Simple; history lost
SCD Type 2	Add a new row with date range; mark old as inactive	Full history preserved; table grows
SCD Type 3	Add a <code>previous_value</code> column	Limited history (only one prior value)

SCD Type 2 is the most common in warehouses:

```
-- SCD Type 2 example: customer changed city
-- Old row
id=1001, customer_id=42, city='Mumbai', valid_from='2020-01-01', valid_to='2023-06-14', is_current=false
-- New row
id=1002, customer_id=42, city='Bangalore', valid_from='2023-06-15', valid_to='9999-12-31', is_current=true
```

Normalization vs Denormalization

Normalization (3NF) — eliminate redundancy by splitting data into related tables. Reduces storage, prevents update anomalies. Best for OLTP.

Denormalization — combine tables, accept redundancy to reduce joins. Faster reads for analytics. Best for OLAP.

```
-- Normalized (OLTP): customer address stored once
customers(id, name, address_id)
addresses(id, city, state, country)

-- Denormalized (OLAP): address embedded in customer
dim_customer(id, name, city, state, country)
```

File Formats

Format Comparison

Format	Type	Splittable	Schema	Best for
CSV	Row, text	Yes (by line)	None	Simple interchange, small files
JSON	Row, text	No (unless NDJSON)	None	APIs, semi-structured data
Parquet	Columnar, binary	Yes	Embedded	Analytics, data lakes
Avro	Row, binary	Yes	Embedded	Kafka messages, schema evolution
ORC	Columnar, binary	Yes	Embedded	Hive, heavy analytics

Parquet

The default format for data lakes. Column-oriented, compressed, self-describing.

Key features: - **Column pruning** — only read the columns your query needs - **Predicate pushdown** — skip row groups that can't match your filter (min/max stats stored per group) - **Compression** — Snappy (fast) or ZSTD (better ratio) per column; similar values compress heavily - **Schema embedded** — no external schema required

```
Parquet file structure:
  Row group 1 (128 MB default)
    Column chunk: id      [min=1, max=50000]
```

```
Column chunk: salary    [min=40000, max=250000]
Column chunk: dept      [min='Design', max='Sales']
Row group 2
...
Footer (schema + row group statistics)
```

A query `WHERE salary > 200000` can skip entire row groups where max salary < 200000 — without reading any data.

Avro

Row-based binary format. Schema stored in JSON alongside the data (`.avsc` file or in the header).

Key features: - **Schema evolution** — add/remove/rename fields with backward/forward compatibility rules - **Compact** — no field names repeated per row (unlike JSON) - Preferred for **Kafka** messages and write-heavy pipelines where schema changes are expected

Compression

Codec	Speed	Ratio	Splittable	Use case
Snappy	Very fast	Moderate	No (inside Parquet: yes)	Default for Parquet/Avro
GZIP	Slow	High	No	Cold storage, CSV
ZSTD	Fast	High	No (inside Parquet: yes)	Modern default
LZ4	Fastest	Low	No	Real-time, low-latency
Bzip2	Slowest	Highest	Yes	Hadoop MapReduce (legacy)

ETL vs ELT

ETL — Extract, Transform, Load

Traditional pattern. Transform data **before** loading into the warehouse.

```
Source DB → [Extract] → [Transform in pipeline] → [Load] → Data Warehouse
```

- Transform happens in a dedicated compute layer (Spark job, Python script)
- Warehouse receives clean, ready-to-use data
- Good when: source data is messy, warehouse compute is expensive, or PII must be masked before storage

ELT — Extract, Load, Transform

Modern pattern. Load raw data first, transform **inside** the warehouse.

```
Source DB → [Extract] → [Load raw] → Data Warehouse → [Transform with SQL]
```

- Raw data lands in a staging layer; transformations run as SQL inside the warehouse
- Warehouses like BigQuery/Snowflake have cheap, scalable compute — running SQL there is efficient
- Easy to rerun transformations without re-ingesting source data
- Good when: warehouse compute is cheap, you want full raw history, and your team knows SQL

Idempotency

A pipeline is **idempotent** if running it multiple times produces the same result as running it once. Critical for safe reruns after failures.

```
-- NOT idempotent — appends duplicates on rerun
INSERT INTO orders SELECT * FROM staging_orders WHERE date = '2024-03-15';

-- Idempotent — deletes first, then inserts (run both in one transaction)
DELETE FROM orders WHERE order_date = '2024-03-15';
```

```
INSERT INTO orders SELECT * FROM staging_orders WHERE date
= '2024-03-15';
```

```
-- Idempotent — upsert pattern
INSERT INTO orders (...)
SELECT ...
ON CONFLICT (order_id) DO UPDATE SET ...;
```

Design every pipeline task to be safely re-runnable. Failures happen. Your pipeline will retry.

Incremental vs Full Load

Full load — truncate and reload the entire table on every run. Simple, no state to track. Only practical for small tables.

Incremental load — load only new/changed rows since the last run. Requires a high-watermark column (`updated_at` , `created_at` , or a CDC stream).

```
-- Incremental load using a watermark
SELECT *
FROM source_orders
WHERE updated_at > '{{ last_successful_run_timestamp }}';
```

CDC (Change Data Capture) — capture every INSERT, UPDATE, DELETE from the source database's transaction log (e.g. Debezium reads Postgres WAL). The most accurate incremental pattern; no dependency on the source having an `updated_at` column.

Partitioning & Clustering

Partitioning

Divides a table into physical segments based on a column's value. The engine skips entire partitions that can't match a query's filter — called **partition pruning**.

```
-- Unpartitioned: full table scan
SELECT * FROM events WHERE event_date = '2024-03-15';
-- Scans 3 years of data to find one day

-- Partitioned by event_date: reads one directory
```

```

SELECT * FROM events WHERE event_date = '2024-03-15';
-- Reads only 2024/03/15/ partition

-- Create a partitioned table (Hive-style)
CREATE TABLE events (
  event_id    BIGINT,
  user_id     BIGINT,
  event_type  STRING,
  event_date  DATE      -- partition column
)
PARTITIONED BY (event_date);

```

Partition strategies: - **Time-based** (most common) — by day, month, year. Matches how analytics queries filter. - **List-based** — by region, country, status. Good for known, bounded cardinality. - **Range-based** — by numeric range (e.g. user_id buckets).

Avoid over-partitioning. Thousands of tiny partitions (e.g. partitioned by hour + user_id) create excessive metadata overhead. Aim for partition sizes of 100 MB–1 GB.

Clustering / Sorting

Within a partition, **clustering** sorts rows by a column so the engine can skip blocks.

- Redshift: `SORTKEY` — rows physically sorted on disk
- BigQuery: `CLUSTER BY` — rows grouped by column within each partition
- Delta Lake / Iceberg: `ZORDER` — co-locate multiple columns in the same files

```

-- BigQuery: partition by date, cluster by user_id
CREATE TABLE events
PARTITION BY DATE(event_timestamp)
CLUSTER BY user_id, event_type;

-- Queries filtering on user_id now scan far fewer blocks
SELECT * FROM events
WHERE DATE(event_timestamp) = '2024-03-15' AND user_id =
12345;

```

Partitioning vs clustering: - Partitioning: eliminates entire partitions (big skips) - Clustering: eliminates blocks within a partition (fine-grained skips) - Use both together for best performance

Data Quality

The Five Dimensions

Dimension	Question it answers	Example check
Completeness	Is all expected data present?	<code>COUNT(*) > 0</code> , no NULL in required columns
Accuracy	Does it reflect reality?	<code>age BETWEEN 0 AND 120</code> , <code>price > 0</code>
Consistency	Does it agree across systems?	Row count in warehouse matches source
Timeliness	Is it fresh enough?	<code>MAX(updated_at) > NOW() - INTERVAL '2 hours'</code>
Uniqueness	Are there duplicates?	<code>COUNT(*) = COUNT(DISTINCT id)</code>

Common Checks

```
-- Completeness: no NULLs in required columns
SELECT COUNT(*) FROM orders WHERE customer_id IS NULL;

-- Uniqueness: detect duplicates
SELECT order_id, COUNT(*) AS n
FROM orders
GROUP BY order_id
HAVING COUNT(*) > 1;

-- Timeliness: pipeline freshness
SELECT MAX(created_at) AS latest_record FROM events;
-- Alert if this is older than expected

-- Referential integrity: every order has a valid customer
SELECT o.id
FROM orders o
LEFT JOIN customers c ON o.customer_id = c.id
WHERE c.id IS NULL;

-- Distribution check: sudden drop in row count
SELECT DATE(created_at) AS day, COUNT(*) AS n
```

```
FROM orders
GROUP BY 1
ORDER BY 1 DESC
LIMIT 14;
-- Compare today vs 7-day average – flag if drop > 20%
```

Deduplication

```
-- Keep the latest record per entity
WITH ranked AS (
  SELECT *,
    ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY
updated_at DESC) AS rn
  FROM raw_orders
)
SELECT * FROM ranked WHERE rn = 1;
```

Streaming Concepts

Core Terminology

Term	Meaning
Event	An immutable record of something that happened (a click, a purchase, a sensor reading)
Producer	The service that publishes events
Consumer	The service that reads and processes events
Topic	A named, ordered, durable log of events (Kafka's unit of organization)
Partition	A topic split into parallel ordered logs for horizontal scaling
Offset	A sequential integer identifying each message's position in a partition
Consumer group	A set of consumers sharing the read load; each partition is assigned to one consumer in the group

Term	Meaning
Broker	A server that stores and serves topic data

How Kafka Works

Producers → Topic (3 partitions) → Consumers (Consumer Group A)

Topic: user-events

Partition 0: [offset 0] [offset 1] [offset 2] ...

Partition 1: [offset 0] [offset 1] [offset 2] ...

Partition 2: [offset 0] [offset 1] [offset 2] ...

Consumer Group A:

Consumer 1 → reads Partition 0

Consumer 2 → reads Partition 1

Consumer 3 → reads Partition 2

- Consumers **commit offsets** to track how far they've read. On restart, they resume from the last committed offset.
- Adding partitions = more parallelism. One consumer per partition is the maximum useful scale for a group.
- Messages are **retained** for a configurable period (e.g. 7 days) regardless of whether they've been consumed.

Event Time vs Processing Time

Event time — when the event actually happened (in the source system). Stamped by the producer.

Processing time — when the event was processed by the pipeline. Can be minutes or hours later due to network delay, retries, or backfill.

Event happened: 10:00:00
Kafka ingestion: 10:00:03 (3s network lag)
Stream job sees: 10:04:30 (job was behind)

Always use **event time** for business metrics. Processing time gives you misleading numbers when a consumer falls behind or when replaying historical data.

Late data — events that arrive after their event-time window has already been processed. You must decide: drop them, reprocess, or hold the window open longer (watermark).

Windowing

Aggregations over a continuous stream require a window — a bounded slice of time.

Tumbling window — fixed-size, non-overlapping. Every event belongs to exactly one window.

```
|--10:00--|--10:01--|--10:02--|  
Count=5      Count=8      Count=3
```

Sliding window — fixed size, overlapping. An event can appear in multiple windows.

```
Window size: 5 min, slide: 1 min  
[9:55-10:00], [9:56-10:01], [9:57-10:02] ...
```

Session window — groups events separated by a gap of inactivity. Size is dynamic.

```
User active 10:00-10:03, idle, active 10:15-10:18 → two  
sessions
```

Delivery Guarantees

Guarantee	Meaning	Risk
At-most-once	Message delivered 0 or 1 times	Data loss possible
At-least-once	Message delivered 1 or more times	Duplicates possible
Exactly-once	Delivered exactly once	Hardest; requires idempotent consumers or transactions

Most production systems aim for **at-least-once + idempotent consumers** (deduplicate on the consumer side). True exactly-once end-to-end is expensive.

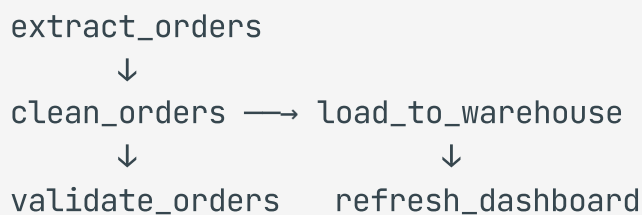
Orchestration

What It Solves

Raw scripts and cron jobs break silently and have no dependency management. An orchestrator: - Defines task **dependencies** (task B only runs after task A succeeds) - **Retries** failed tasks automatically - Provides **observability** — logs, alerts, run history - Supports **backfill** — rerun historical date ranges - Manages **concurrency** — don't run 50 jobs at once

DAGs — Directed Acyclic Graphs

A pipeline is modeled as a DAG: tasks are nodes, dependencies are directed edges. "Acyclic" means no circular dependencies.



Each task in a DAG should be: - **Atomic** — does one thing - **Idempotent** — safe to rerun - **Decoupled** — doesn't share state with sibling tasks in memory

Scheduling Patterns

```
# Cron expressions
0 2 * * *      - daily at 2am
0 * * * *      - hourly
*/15 * * * *   - every 15 minutes
0 2 * * 1      - every Monday at 2am

# Airflow schedule examples (Airflow 2.4+ / 3.x use `schedule=`)
schedule='@daily'
schedule='0 6 * * *'           # 6am UTC daily
schedule=timedelta(hours=6)
```

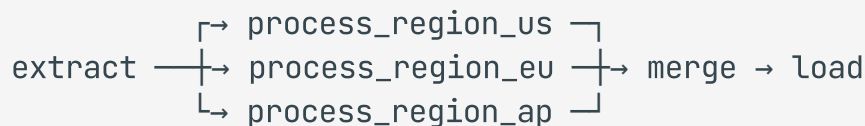
Important: a scheduled Airflow run processes a **data interval** and runs *after* that interval ends: the daily run covering March 15 starts on March 16. In Airflow 2, the run's `logical_date` (formerly `execution_date`) is the interval *start* (March 15); Airflow 3's default cron timetable sets it to the run time instead. Read `data_interval_start` / `data_interval_end` explicitly and you're correct on both. This trips up almost everyone the first time.

Common Pipeline Patterns

Sensor — wait for an external condition before proceeding (file arrives in S3, table row count > 0).

Branch — conditionally run different downstream tasks based on runtime logic.

Fan-out / Fan-in — split work across parallel tasks, then merge results.



SLA — define a deadline for task completion; alert if missed.

Backfilling

Re-running a pipeline for historical dates — to fix a bug, onboard a new table, or apply a new transformation.

Requirements for safe backfilling: 1. Tasks must be **idempotent** (rerunning produces the same result) 2. The pipeline must be **parameterized** on date/time (not hardcoded `NOW()`) 3. Source data must still be available for the historical range

Key Tools Landscape

A map of what each tool solves — not tutorials, just the mental model.

Processing

Tool	What it does	When to reach for it
Apache Spark	Distributed batch + streaming processing	Large-scale transformations, ML pipelines, anything that doesn't fit in memory on one machine
dbt	SQL-based transformation layer in the warehouse	ELT pipelines, modeling raw data into analytics-ready tables, documentation, testing
Apache Flink	Stateful stream processing	Low-latency streaming, complex event processing, exactly-once guarantees
Pandas	In-memory DataFrame operations	Small-to-medium data that fits in RAM, quick exploration, prototyping

Ingestion

Tool	What it does	When to reach for it
Apache Kafka	Distributed event log / message broker	High-throughput event streaming, decoupling producers from consumers
Debezium	Change Data Capture from databases	Streaming database changes (inserts/updates/deletes) to Kafka from Postgres, MySQL, etc.
Airbyte / Fivetran	Managed connectors for batch ingestion	Pulling data from SaaS tools (Salesforce, Stripe, etc.) into your warehouse
Apache NiFi	Data flow automation	Complex routing, transformation and delivery of data between systems

Orchestration

Tool	What it does	When to reach for it
Apache Airflow	DAG-based workflow orchestration	Scheduling and monitoring batch pipelines; the de facto standard
Prefect	Python-native workflow orchestration	Airflow alternative with better local development and dynamic workflows

Tool	What it does	When to reach for it
Dagster	Asset-centric orchestration	When you think in terms of data assets, not tasks; strong typing and observability

Storage

Tool	What it does	When to reach for it
Delta Lake	Open table format on object storage	ACID transactions, time travel, and schema enforcement on S3/ADLS (Databricks native)
Apache Iceberg	Open table format on object storage	Multi-engine support (Spark, Flink, Trino, BigQuery); partitioning evolution
Apache Hudi	Open table format with upsert support	<u>CDC</u> -heavy workloads that need efficient record-level updates
Trino / Presto	Distributed SQL query engine	Federated queries across multiple data sources without moving data

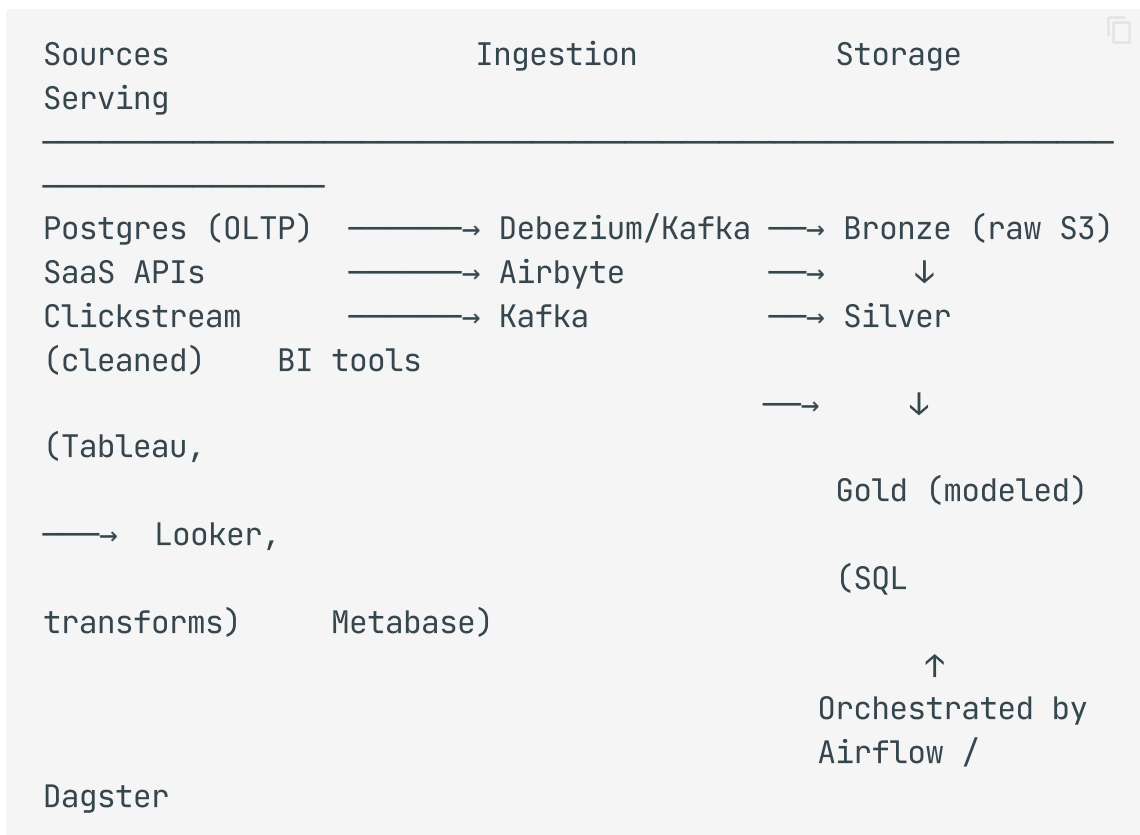
Warehouse

Tool	What it does	When to reach for it
Snowflake	Cloud data warehouse	Separate compute + storage scaling, multi-cloud, strong ecosystem
BigQuery	Serverless cloud data warehouse	No infrastructure, pay-per-query, tight GCP integration
Redshift	AWS-native data warehouse	Heavy AWS workloads, tight Glue/S3 integration
DuckDB	In-process analytical database	Local analytics on files (Parquet, CSV), replacing Pandas for medium data

Putting It Together

A typical modern data stack looks like this:





The key insight: each layer has one job. - **Bronze** — land raw data, never modify it. It's your source of truth for reprocessing. - **Silver** — clean, validate, deduplicate. Schema is enforced here. - **Gold** — business logic lives here. Star schema, aggregations, metrics.

Common Pitfalls

Pitfall	Symptom	Fix
Non-idempotent loads (<code>INSERT</code> only)	Duplicate rows after every retry or backfill	<code>MERGE</code> /upsert on a key, or <code>DELETE</code> + <code>INSERT</code> for the partition being loaded
Using <code>NOW()</code> / <code>CURRENT_DATE</code> inside the pipeline	Backfills silently load today's data for every historical date	Parameterize every run on the logical date the orchestrator passes in
Aggregating on processing time	Metrics shift when a consumer lags or you replay history	Use event time for business metrics; handle late data with watermarks
Modifying or deleting Bronze data	Can't reprocess after a bug — the raw truth is gone	Keep Bronze append-only and immutable; fix things in Silver

Pitfall	Symptom	Fix
Over-partitioning (e.g. by hour + user_id)	Millions of tiny files, slow planning, high metadata cost	Partition by low-cardinality columns (usually date); aim for 100 MB–1 GB partitions; cluster on the rest
Relying on <code>updated_at</code> for incremental loads	Hard deletes and rows updated without touching <code>updated_at</code> are missed	Use <u>CDC</u> from the transaction log, or add periodic full reconciliation
Assuming exactly-once delivery	Occasional duplicates in downstream tables	Design for at-least-once and deduplicate on a stable key
Choosing streaming because it sounds modern	Twice the operational complexity for a dashboard refreshed daily	Start with batch; move to streaming only when latency is a real business requirement
No freshness or volume checks	Stakeholders notice stale dashboards before you do	Alert on <code>MAX(event_time)</code> lag and on row counts vs a trailing average

Cheat Sheet

Pick the architecture

Question	If yes	If no
Need results in seconds?	Streaming (Kafka + Flink/Spark)	Batch
Mostly SQL-savvy team, cloud warehouse?	ELT (SQL transformations in the warehouse)	ETL in Spark/Python
Many engines reading the same data?	Lakehouse with Iceberg	Warehouse-native tables
Table under ~1 GB and cheap to reload?	Full load	Incremental (watermark or <u>CDC</u>)
Source DB exposes a transaction log?	<u>CDC</u> (Debezium)	<code>updated_at</code> watermark + periodic reconciliation

Core definitions

Term	One-liner
OLTP / <u>OLAP</u>	Run the app (small writes) / answer questions (big reads)
Medallion	Bronze = raw, Silver = clean, Gold = business-ready
Idempotent	Running twice gives the same result as running once
Partition pruning	Skip whole partitions that can't match the filter
Watermark	How late an event can arrive and still be counted
<u>SCD</u> Type 2	New row per change, with <code>valid_from</code> / <code>valid_to</code> / <code>is_current</code>

Formats

Need	Use
Analytics on a lake	Parquet (+ Iceberg/Delta for ACID)
Kafka messages with schema evolution	Avro + Schema Registry
Hand-off to humans or legacy tools	CSV
API payloads, raw landing	JSON / NDJSON
Compression default	ZSTD (or Snappy for speed)

Interview Questions

Q: What is the difference between OLTP and OLAP? Give an example of each. A: OLTP (Online Transaction Processing) systems run the business — they handle many small, fast read/write queries like inserting a new order or updating an account balance. Examples: PostgreSQL, MySQL. OLAP (Online Analytical Processing) systems answer business questions — they run few but complex analytical queries over large datasets. Examples: Snowflake, BigQuery. The key difference: OLTP is normalized for writes; OLAP is denormalized for reads.

Q: What is the medallion architecture and why do we use it? A: Bronze/Silver/Gold — a three-layer pattern where raw data lands in Bronze unchanged, is cleaned and validated in Silver, and becomes business-ready (star schema, aggregates) in Gold. We use it because

it separates concerns: Bronze is the safety net (can always reprocess), Silver enforces quality, Gold optimizes for queries. Each layer has a clear owner and a clear definition of done.

Q: What is the difference between ETL and ELT? A: ETL (Extract-Transform-Load) transforms data before loading it into the destination — traditional, needed when the destination is expensive or slow. ELT (Extract-Load-Transform) loads raw data first, then transforms it using the destination's compute — modern approach enabled by cheap cloud warehouses. dbt is an ELT tool: you load raw data into Snowflake, then transform it with SQL inside Snowflake.

Q: What is idempotency in data pipelines and why does it matter? A: An idempotent pipeline produces the same result whether it runs once or ten times. It matters because pipelines fail and get retried — if a retry inserts duplicate rows, your data is wrong. Common patterns: use MERGE/upsert instead of INSERT, use DELETE+INSERT with a date partition, or use deduplication logic (dbt's `unique_key` on incremental models).

Q: What is partitioning and how does it improve query performance? A: Partitioning divides a large table into sub-groups based on a column value (usually date). When you query with a filter on the partition column (`WHERE date = '2024-03-15'`), the query engine only reads that partition's files — skipping 99%+ of the data. Without partitioning, every query scans the entire table. For time-series data (orders, events), partitioning by day is almost always the right choice.

Q: What is the difference between a data lake, a data warehouse, and a lakehouse? A: A data lake stores raw files in any format on cheap object storage (S3) — flexible but no schema enforcement or transactions. A data warehouse stores structured, optimized data in a proprietary format — great for queries but expensive and schema-rigid. A lakehouse combines both: open file formats (Parquet/Delta/Iceberg) on object storage, with a metadata layer that adds warehouse features (ACID, schema enforcement, time travel). Databricks and Delta Lake are examples.

Q: What is a data contract and when would you need one? A: A data contract is a formal agreement between the producer of a dataset and its consumers — specifying schema, data types, SLA (freshness guarantee), quality rules, and ownership. You need one when multiple teams depend on a dataset: the contract prevents the upstream team from silently breaking downstream pipelines with schema changes or delayed delivery.

Further Reading

- *Fundamentals of Data Engineering* — Joe Reis & Matt Housley (O'Reilly). The best overview of the whole lifecycle.

- *Designing Data-Intensive Applications* — Martin Kleppmann (O'Reilly). Storage, replication, partitioning, streams, from first principles.
 - *The Data Warehouse Toolkit* — Ralph Kimball & Margy Ross (Wiley). The original source for star schemas and SCDs.
 - *Streaming Systems* — Tyler Akidau, Slava Chernyak & Reuven Lax (O'Reilly). Event time, watermarks, and windowing explained properly.
 - [Apache Parquet documentation](https://parquet.apache.org/docs/) (https://parquet.apache.org/docs/)
 - [Debezium documentation](https://debezium.io/documentation/) (https://debezium.io/documentation/)
-

Next: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/)

· **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026