Learnings

- Data
  - data platforms — Digest
  - data platforms — Article
  - data warehouse — Digest
  - data warehouse — Article
  - data warehouse use cases — needed vs. overkill

Data

Data

data platforms — Digest

Definition

A data platform is the infrastructure and set of tools an organization uses to collect, store, process, and serve data so applications and people can reliably use it. It's the plumbing that moves data from where it's generated to where it's needed, in usable form.

Real-life usage

- E-commerce: clickstream, orders, and inventory land in a warehouse (Snowflake/BigQuery/Redshift) and feed both BI dashboards and ML models (recommendations, fraud detection) from the same data.
- Fintech: transaction data flows via streaming pipelines (Kafka) into systems supporting both real-time fraud scoring and end-of-day regulatory reporting.
- Healthcare: patient records, device telemetry, and claims data are unified under strict governance (HIPAA) while remaining queryable for research.
- SaaS internal analytics: app DB → ETL/ELT → warehouse → dashboards for product/finance teams.
- Analogy: a city utility grid — treatment plant (ingestion/cleaning), pipes/reservoirs (storage), pumping stations (processing), taps (serving). Break any layer, everything downstream is affected.

5 Whys — Root Cause

- Why do platforms exist? Raw source systems are wrong for two independent reasons.
- **Reason 1 — contention:** OLTP DBs are optimized for row-based point lookups. Large analytical scans evict the transactional working set from buffer cache and hold locks longer, causing cache misses and slow transactional reads (e.g., checkout).
- Read replicas solve contention but not query *shape* — row-store/3NF schemas are structurally bad at large scans/aggregations, hence column-store, star schemas, and pre-aggregation.

- **Reason 2 — incoherence:** each source system encodes its own implicit business logic (e.g., three different definitions of "active customer" across CRM, billing, and app DB). With N sources and M consumers and no shared contract, definitions drift — it's structurally guaranteed, not a one-off mistake.
- Platforms fix this by centralizing logic into governed, computed-once definitions everyone reads from.

Value vs. Risk

- **Value:** understanding the "why" lets you choose stack components by the guarantee a workload needs (ordering, replayability, latency, managed vs. self-hosted) instead of by popularity or familiarity — avoiding redundant tools solving the same problem.
- **Risk — performance/consistency:** black-boxing the platform reproduces the original problems (contention, definitional drift) plus new complexity/understanding debt (unclear why each tool is in the stack).
- **Risk — trust erosion:** when dashboards disagree and no one can explain why, people stop trusting the platform and quietly bypass it for gut-feel or spreadsheets — a costly, hard-to-reverse adoption failure.

Data

data platforms — Article

Definition

A data platform is the infrastructure and set of tools an organization uses to collect, store, process, and serve data so that applications and people can reliably use it. Stripped of jargon: it's the plumbing that gets data from where it's produced to where it's needed, in a form someone can actually use. When you hear "data platform," think less of a single product and more of a layered system — ingestion, storage, processing, and serving — each layer doing a distinct job.

Real-life usage

You've already touched pieces of this: Debezium and NiFi doing change-data-capture and flow-based ingestion, Kafka as the durable transport backbone, Airflow and Azure Data Factory orchestrating batch pipelines, Elasticsearch and MongoDB serving specific access patterns. That's a real, if overlapping, data platform stack.

Beyond your own stack, the pattern repeats everywhere at scale:

- **E-commerce** platforms capture clickstream, orders, and inventory changes, land them in a warehouse (Snowflake, BigQuery, Redshift), and serve both BI dashboards and ML models (recommendations, fraud detection) from the same underlying data.
- **Fintech** systems stream transactions through Kafka into pipelines that support both millisecond-latency fraud scoring and end-of-day regulatory batch reporting — same data, wildly different latency contracts.
- **Healthcare** platforms unify patient records, device telemetry, and claims data while satisfying strict compliance (HIPAA), yet still need to be queryable for research.
- Even a mid-size SaaS company runs a scaled-down version: app DB → ETL/ELT → warehouse → dashboards for product and finance.

A useful mental model: a data platform is like a city's utility grid. A treatment plant cleans and prepares water (ingestion/cleaning), reservoirs and pipes store and move it (storage), pumping stations push it where it's needed (processing), and taps deliver it on demand (serving/access). Nobody turning on a faucet thinks about the treatment plant — but if any layer fails, everything downstream breaks.

5 Whys — Why Data Platforms Exist

The natural question is: why not just query the production database directly? Working through this honestly gets you to two independent root causes.

Root cause 1: resource contention. OLTP databases are optimized for row-based, indexed point lookups — fast reads/writes of single rows, like fetching one customer's cart. Analytical queries instead scan and aggregate across millions of rows. When a large aggregation runs against the same database, it evicts the transactional working set from the buffer cache. A checkout read that used to be an instant in-memory hit becomes a slow disk read. Combine that with longer-held locks from the scan, and the aggregation job and the checkout flow — which have nothing to do with each other logically — end up fighting over the same finite memory, I/O, and lock resources.

The obvious fix, "just add a read replica," solves contention: reads are offloaded to a copy, so the analytical workload stops starving the transactional one. But it doesn't solve the second problem — *query shape*. A replica has the exact same row-store, normalized (3NF) schema as production, which is structurally bad at scanning and aggregating across huge row counts, replica or not. This is why the industry didn't stop at replicas: it built column-store engines, star/denormalized schemas, and pre-aggregation — physically reshaping the data to match the access pattern of the consumer, not just relocating the same shape to different hardware. Vertical scaling (bigger single machine) eventually hits a ceiling too, which is part of why horizontal separation into distinct, purpose-built systems became the standard architecture rather than just buying a bigger box.

Root cause 2: definitional incoherence. Imagine three teams need "active customer": marketing pulls it from the CRM (logged in within 30 days), finance pulls it from billing (has a paid, non-cancelled subscription), and a data scientist queries the app DB directly (any tracked event in 7 days). All three are querying the same underlying reality, but each source system encodes its own local business logic implicitly — in application code, in how a status field gets set, in team-specific workflow assumptions. There's no shared, enforced contract anywhere. With N source systems and M independent consumers, you get up to N×M different interpretations of the same concept, and nothing structurally prevents that drift — it isn't a one-off mistake someone could have avoided by being careful, it's guaranteed by the absence of a shared definition layer.

Put together: **data platforms exist because raw source systems are wrong for two independent reasons** — they can't serve analytical access patterns without degrading transactional performance, and they can't produce consistent meaning across consumers without centralizing business logic. Every component of a modern data platform — ingestion tools, warehouses, orchestration, semantic/governance layers — exists to solve one or both of these.

Value vs. Risk

Value. Understanding the "why" rather than just the "how to configure X" changes how you make architectural decisions. Instead of picking a tool because it's popular or because you already know it, you pick based on the guarantee the workload actually needs: does it need ordering and replayability (Kafka), low-latency row-level change capture (Debezium), flexible flow-based routing over heterogeneous data (NiFi), managed Azure-native batch orchestration (ADF), or Python-native complex DAG logic (Airflow)? If two tools in your stack solve the same guarantee — say, two orchestrators — that's a signal of accidental complexity worth questioning directly, not a neutral fact about the stack.

Risk of treating the platform as a black box. The immediate risks are a recurrence of the original two problems — contention and inconsistency — plus a new one: complexity debt, where nobody can explain why a given tool is in the pipeline or what it specifically contributes. But there's a third, more expensive risk that's easy to miss because it isn't a technical failure: **trust erosion**. When two dashboards disagree and no one can explain why, people stop trusting the platform itself, not just that one number. The failure mode isn't a wrong query — it's leadership quietly reverting to gut feel or personal spreadsheets because they've learned the platform can't be relied on. That's a behavioral, org-wide cost, and by the time it's visible, the damage — bad decisions already made, the platform already abandoned in practice — is largely done. It's far more expensive to reverse than a performance regression, because it's an adoption and credibility problem, not an engineering one.

Data

data warehouse — Digest

Definition

A data warehouse is a centralized store of structured, historical data — typically modeled with facts and dimensions (star/snowflake schema) — organized specifically to support analytical querying and reporting, as opposed to transactional operations.

Real-life usage

- Retail/e-commerce: `fact_sales` (quantity, revenue) joined to `dim_product`, `dim_store`, `dim_date`, `dim_customer` — slice revenue any way, on demand.
- Banking/fintech: regulatory reporting (risk exposure, AML) needs consistent historical snapshots, not just current state.
- Airlines: `fact_bookings` / `fact_flights` joined to `dim_route`, `dim_aircraft`, `dim_time` to analyze load factors and revenue per route over years.
- Marketing: `fact_impressions` / `fact_conversions` joined to `dim_campaign`, `dim_channel`, `dim_audience`.
- Analogy: a physical warehouse doesn't manufacture goods — it receives finished goods and organizes them onto labeled shelves by category (dimensions) with clear counts (facts), so anyone can pull exactly what they need without digging through the factory floor.

5 Whys — Root Cause

- Why not just use a regular RDBMS with a star schema? Row-store databases store full rows together on disk — a query needing 2 of 30 columns still has to read all 30 off disk.
- **Root cause 1 — columnar storage:** warehouse engines store each column contiguously, so aggregation queries read only the columns they need. This also enables better compression and vectorized execution.
- Why also separate facts and dimensions if columnar storage already solves read efficiency? Flattening everything into one wide table causes massive redundancy (a customer's address repeated across every transaction row) and update anomalies

(changing one dimension value means rewriting/missing rows across a billion-row fact table).
- **Root cause 2 — fact/dimension separation:** avoids redundancy and update anomalies by separating high-volume immutable events (facts) from low-volume mutable reference data (dimensions), each managed according to its actual rate of change (see: slowly changing dimensions).

Value vs. Risk

- **Value:** understanding columnar storage + fact/dimension separation changes how you design any analytical store — not just SQL warehouses. E.g., a medallion architecture built on doc-oriented stores (like Elasticsearch) still needs to factor out dimension keys instead of embedding full dimension attributes in every aggregate document, or it silently rebuilds the flat-table problem.
- **Risk — no dedicated warehouse layer:** flat/embedded dimension data means a dimension change (e.g., product recategorized) can't be propagated without reindexing every historical record referencing it — leading to silent inconsistency between old and new records with no single source of truth to detect or correct the drift.

Data

data warehouse — Article

Definition

A data warehouse is a centralized store of structured, historical data — typically modeled with facts and dimensions using a star or snowflake schema — organized specifically to support analytical querying and reporting, as opposed to the transactional operations a production database handles. It's not just "a database with reporting tables"; the storage engine itself and the data model are shaped around analytical access patterns.

Real-life usage

The clearest way to see this is through fact and dimension tables. A retail warehouse might have `fact_sales` — one row per line item, carrying measures like quantity and revenue — joined to `dim_product`, `dim_store`, `dim_date`, and `dim_customer`. Anyone in finance can then slice revenue by region, by month, by category, in combinations nobody had to pre-build.

The same pattern shows up everywhere analytics matters at scale:

- **Banking/fintech**: regulatory reporting (daily risk exposure, AML) depends on consistent historical snapshots — "what was our exposure on March 31st" — not just current state, which is exactly what a warehouse's historical fact tables are built to answer.
- **Airlines**: `fact_bookings` / `fact_flights` joined to `dim_route`, `dim_aircraft`, `dim_time` let analysts study load factors, delays, and revenue per route across years of history.
- **Marketing**: `fact_impressions` / `fact_conversions` joined to `dim_campaign`, `dim_channel`, `dim_audience` answer "which channel actually drives conversions" across time — something no single operational system tracks end to end.

A useful analogy: a physical warehouse doesn't manufacture goods — it receives finished goods from many sources and organizes them onto labeled shelves by category (dimensions), with clear counts (facts), so anyone can walk in and pull exactly what they need without digging through the factory floor. A data warehouse does the same thing with data instead of goods.

5 Whys — Why Data Warehouses Exist as a Distinct Thing

The natural challenge is: you could build a star schema in any relational database — Postgres, MySQL — so why does the industry build dedicated warehouse engines (Snowflake, BigQuery, Redshift) at all?

Root cause 1: columnar storage matches the analytical access pattern. In a normal row-store database, a table's rows are stored together on disk — all of row 1's columns, then all of row 2's columns, and so on. A typical warehouse query like `SELECT SUM(revenue) FROM fact_sales WHERE region = 'EU'` only needs 2 of maybe 30 columns, but a row-store still has to pull every column of every matching row off disk into memory before it can discard the ones it doesn't need. That's wasted I/O and wasted cache space, proportional to the columns you don't care about.

Warehouse engines fix this with columnar storage: each column is stored contiguously on disk, so a query reads only the columns it actually references. This also unlocks two side benefits — far better compression (similar values stored together compress much more efficiently than mixed-type rows) and vectorized execution (operating on whole columns of values at once instead of row by row). This is the physical, mechanical reason a warehouse engine behaves differently from a row-store RDBMS running the exact same schema.

Root cause 2: fact/dimension separation avoids redundancy and update anomalies. Given columnar storage already solves read efficiency, it's fair to ask why not just put everything — customer name, product category, store region, revenue, quantity — into one big flat, columnar table. The answer is what happens to descriptive attributes under that design. If "customer address" lives inline in every sales row, and that customer has 500 transactions, the same address string is stored 500 times. At billions of rows, that's significant redundancy even under compression, and it creates an update anomaly: if the customer moves, do you rewrite all 500 (or 5 million) rows referencing them? Miss one, and the data is now silently inconsistent.

The deeper issue is conceptual: a "sale" and a "customer's current address" are different kinds of data that change at fundamentally different rates. Facts are an append-only event log; dimensions are comparatively static reference data that occasionally changes (which is exactly why "slowly changing dimensions" exists as a defined technique — dimensions do change, just rarely, and need an explicit strategy for handling it). Separating facts from dimensions keeps each kind of data managed according to its actual rate of change, instead of conflating an immutable event stream with mutable reference data in one structure.

Put together: **data warehouses exist because (1) columnar storage matches the "few columns, many rows" read pattern of analytical queries in a way row-stores structurally cannot, and (2) fact/dimension modeling avoids the redundancy and update anomalies that come from mixing high-volume immutable events with low-volume mutable reference data in a single flat structure.**

This root cause generalizes beyond SQL warehouses. A doc-oriented store like Elasticsearch actually does provide columnar-style reads for aggregations, via a structure called `doc_values` — a separate, purpose-built columnar layer distinct from its inverted index for text search. So root cause 1 can genuinely be satisfied outside a traditional warehouse engine. But root cause 2 is a modeling discipline, not a storage-engine feature — it has to be deliberately designed in. A medallion-style `agg_` index built from flat JSON documents that embed both facts and dimension attributes in every document reproduces the exact flat-table problem: dimension values are duplicated across every document that references them, and because a document store has no native join to factor dimensions out, a dimension change either goes unpropagated (silent inconsistency) or requires reindexing every historical document that referenced it.

Value vs. Risk

Value. Understanding these two root causes changes how you evaluate any analytical data store, not just SQL warehouses. It tells you precisely what to check before trusting a design: does the storage layer read only the columns/fields a query needs (columnar or equivalent), and are high-volume facts kept separate from low-volume, referenced dimension data — or are they flattened together for convenience? For a system like a medallion-on-Elasticsearch setup, this means factoring dimension keys/IDs into fact-like documents and maintaining separate dimension indices, resolving them at query or ingest time, rather than embedding full dimension attributes into every aggregate document.

Risk. Keeping facts and dimensions flattened together, whether in a wide SQL table or in JSON documents, defers a cost rather than avoiding it. As data volume grows and dimension values change more often, the concrete failure looks like this: a report segmenting historical sales "by current product category" silently returns wrong numbers, because only some historical records were ever updated to reflect the new category — and there's no single source of truth to detect or correct the drift from, since nothing enforces referential consistency across the duplicated copies. This isn't a performance problem; it's a correctness problem that erodes trust in the numbers exactly the way inconsistent metric definitions do at the platform level.

Data

data warehouse use cases — needed vs. overkill

When a data warehouse is genuinely needed

- **BI/reporting** — ad hoc slicing across dimensions (region, time, product) at speed; row-stores choke on this at scale.
- **Historical/point-in-time analysis** — cohort retention, trend analysis, "what did exposure look like on a given date." Needs immutable historical snapshots that production systems don't retain.
- **ML feature engineering / training data prep** — building wide feature tables by joining large historical fact/dimension sources; needs efficient scans/joins at volume.
- **Regulatory & compliance reporting** — audits need reproducible, consistent historical figures with lineage, not a moving production DB.
- **Cross-domain analytics / data mesh consumption** — joining multiple business domains' data together; exactly the "N sources, M consumers, no shared contract" problem centralized modeling solves.
- **Forecasting & capacity planning** — needs large historical time series at consistent granularity.
- **Aggregate-level anomaly/trend detection** — e.g. "revenue by region dropped 20% week over week"; inherently an aggregation-over-time problem.

Common thread: all need to scan/aggregate across large historical volumes with consistent, centrally-defined meaning — the two root causes (columnar reads + fact/dimension governance).

When a data warehouse is overkill or the wrong tool

- **Single-record, low-latency lookups** (get this customer's cart/profile) — the OLTP point-lookup pattern; columnar storage actually hurts here. Use the operational DB.
- **Real-time operational decisions inside a live transaction** (fraud scoring at checkout, inventory decrement) — needs millisecond latency and current state; warehouses are typically minutes-to-hours behind via batch/CDC.
- **Full-text search / relevance ranking** — not built for inverted indices or scoring; use Elasticsearch/OpenSearch.
- **Unstructured/semi-structured blob storage** — raw images, PDFs, logs, un-modeled JSON; forcing this into a star schema is premature structure — use a data lake (or the raw layer of a lakehouse) first.
- **High-frequency small writes** — IoT sensor ingestion event-by-event; warehouses are read-optimized/batch-oriented. Use a time-series DB or streaming store.
- **Simple, low-volume reporting for a small team** — a few thousand rows, one monthly summary; the ETL/governance/infra overhead isn't justified. A well-indexed Postgres view or spreadsheet suffices.
- **Graph-shaped queries** — path-finding, fraud rings, network traversal; star schemas/columnar engines aren't built for relationship traversal. Use a graph database.

Common thread: low-latency point access, unstructured content, full-text relevance, high-frequency writes, or relationship traversal are all problems a warehouse wasn't designed to solve.

Grouped by business domain

Telecom

- *Needs it:* CDR analysis across billions of records, churn prediction feature tables, network capacity planning, regulatory usage/coverage reporting.
- *Overkill/wrong tool:* real-time call routing, live network fault detection (streaming/time-series), subscriber lookup during a support call (OLTP).

Retail / e-commerce

- *Needs it:* sales analysis across store/product/time, inventory trend forecasting, customer segmentation/LTV, marketing attribution.
- *Overkill/wrong tool:* checking current stock for one SKU at checkout, product search/autocomplete (search engine), real-time cart/session state.

Consulting

- *Needs it:* multi-client benchmarking and longitudinal engagement analysis, internal utilization/profitability reporting across practices and time.
- *Overkill/wrong tool:* a single short engagement's ad hoc analysis on a one-off dataset — a spreadsheet or lightweight Postgres/DuckDB is enough.

Industry / manufacturing

- *Needs it:* production yield analysis across plants/lines/time, supply chain and demand forecasting, quality trend analysis joining defects to equipment/shift/supplier.
- *Overkill/wrong tool:* real-time sensor/IoT ingestion and shop-floor anomaly alerts (time-series/streaming), machine-level current-status lookups.

Finance / banking

- *Needs it:* regulatory historical reporting, risk exposure trend analysis, AML pattern detection across large transaction histories.
- *Overkill/wrong tool:* fraud scoring at the moment of a transaction (real-time), account balance lookup (OLTP).

Healthcare

- *Needs it:* population health trend analysis, claims/cost analysis across providers and time, research cohort analysis.
- *Overkill/wrong tool:* pulling a single patient's current chart during a visit (OLTP), clinical decision support needing sub-second current data.

Pattern across every domain: warehouse = historical, cross-dimensional, aggregatable analysis at real volume. Anything current-state, single-record, sub-second, or small-scale is the wrong job for it, regardless of industry.