

Data Engineering System Design

How to design data platforms and pipelines end to end — requirements, estimation, architecture patterns, trade-offs, and worked examples.

Last reviewed 29 Sep 2026 ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) · [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) · [Data Ingestion & CDC](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/)

Related: [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) · [Kafka](https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/04-streaming/kafka-reference/) · [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Choosing a Stack](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/) · [Cost Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

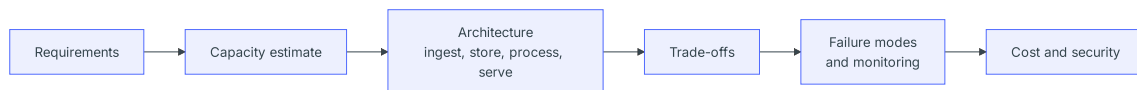
Challenge: Individual tools are well documented; combining them into a system that meets real requirements is not. The same question — "build a pipeline for order analytics" — has very different answers depending on data volume, freshness needs, consumers, budget, and compliance constraints. Designs that ignore those constraints end up either over-engineered (a streaming platform for a daily report) or fragile (a nightly script expected to serve real-time decisions).

Solution: A repeatable design process: clarify requirements, estimate scale, choose an architecture pattern, select components for each layer, and then address reliability, data quality, security, and cost explicitly — documenting the trade-offs behind each decision.

```
Requirements → Estimation → Architecture pattern →  
Components per layer → Cross-cutting concerns  
who, what,          volume,          batch / streaming /
```

ingest · store ·	reliability · quality ·
how fresh,	throughput, lambda / kappa /
process · serve ·	security · cost ·
SLAs, budget	storage CDC / lakehouse
orchestrate	observability

Relevance to data engineering: system design is how senior data engineering work is evaluated — in design reviews and in interviews. The goal isn't a perfect architecture; it's a defensible one whose trade-offs are explicit.



On this page

Basic - [A Design Process](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#a-design-process) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#a-design-process) - [Clarifying Requirements](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#clarifying-requirements) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#clarifying-requirements) - [Capacity Estimation](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#capacity-estimation) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#capacity-estimation)

Intermediate - [Building Blocks by Layer](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#building-blocks-by-layer) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#building-blocks-by-layer) - [Architecture Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#architecture-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#architecture-patterns) - [Batch or Streaming?](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#batch-or-streaming) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#batch-or-streaming) - [Storage and Serving Choices](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#storage-and-serving-choices) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#storage-and-serving-choices)

Advanced - [Reliability and Correctness](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#reliability-and-correctness) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#reliability-and-correctness) - [Scalability](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#scalability) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#scalability) - [Security, Privacy, and Compliance](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#security-privacy-and-compliance) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#security-privacy-and-compliance) - [Cost](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#cost) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#cost) - [Worked Design: Clickstream Analytics](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-clickstream-analytics) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-clickstream-analytics) - [Worked Design: Operational Database to Lakehouse](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-operational-database-to-lakehouse) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-operational-database-to-lakehouse) - [Worked Design: Daily Business Metrics with an SLA](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-daily-business-metrics-with-an-sla) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#worked-design-daily-business-metrics-with-an-sla)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#interview-questions)

997.github.io/data-engineering-handbook/08-architecture/system-design/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/#further-reading)

A Design Process

Step	Output	Typical time in a 45-minute interview
1. Clarify requirements	Consumers, use cases, freshness, SLAs, constraints	5–8 min
2. Estimate scale	Events/day, peak throughput, storage growth	3–5 min
3. High-level architecture	Diagram: sources → ingestion → storage → processing → serving	8–10 min
4. Deep dive on critical parts	Data model, the hardest pipeline, failure handling	10–15 min
5. Cross-cutting concerns	Quality, reliability, security, cost, monitoring	5 min
6. Trade-offs and evolution	What you chose, what you gave up, what changes at 10×	3–5 min

State assumptions out loud, write them down, and revisit the design when an assumption changes.

Clarifying Requirements

Functional - Who consumes the data — dashboards, analysts, data scientists, ML models, applications, external partners? - Which questions or products must it support? Which metrics, with what definitions? - What are the sources — databases, event streams, APIs, files? Who owns them? - Is history required (point-in-time analysis, audits), or only current state?

Non-functional

Dimension	Questions	Why it matters
Freshness	Seconds, minutes, hours, or daily?	Drives batch vs streaming — the biggest cost and complexity decision
Volume and growth	Events per day now and in 2 years? Peak vs average?	Sizing, partitioning, technology choice
Correctness	Exactly-once needed? Tolerance for late or duplicate data?	Idempotency, deduplication, reconciliation design
Availability / SLA	By when must data be ready? What if it's late?	Scheduling, alerting, redundancy
Query patterns	Aggregations over months? Point lookups by key? Ad hoc exploration?	Storage format, modeling, serving layer
Retention	How long to keep raw and modeled data?	Storage cost, lifecycle policies, compliance
Security / compliance	PII? Regulated data? Residency rules?	Access control, masking, region choices
Budget and team	Cost ceiling? Team skills? Build or buy?	Managed vs self-hosted, tool choice

Capacity Estimation

Rough numbers justify design choices. Round aggressively and show the arithmetic.

Useful constants

Quantity	Approximation
Seconds per day	$\sim 86,400 \approx 10^5$
Peak-to-average ratio	2–10× (use 3–5× if unknown)
JSON event	0.5–2 KB
Parquet compression vs JSON	5–10× smaller

Quantity	Approximation
One modern server/executor	Tens of MB/s of parsing and transformation per core

Worked example: 500 million events per day, ~1 KB each

Average throughput events/s	$500M / 86,400 \text{ s}$	$\approx 6,000$	
Peak throughput events/s	$\times 5$	$\approx 30,000$	
Raw volume per day (JSON)	$500M \times 1 \text{ KB}$	$\approx 500 \text{ GB/day}$	
As Parquet	$\div \sim 7$	$\approx 70 \text{ GB/day}$	
Per year (Parquet)	$70 \text{ GB} \times 365$	$\approx 25 \text{ TB/year}$	
Kafka partitions (conservative)	$30 \text{ MB/s} \div \sim 5 \text{ MB/s per partition}$	$\approx 6 \rightarrow$ provision 12-24 for headroom	
Daily partition size	$70 \text{ GB} \div \sim 512 \text{ MB target files}$	≈ 140 files/day	

Conclusions from the numbers: one streaming cluster of modest size handles ingestion; storage growth is moderate and fits object storage cheaply; daily partitions with ~140 files each are healthy; aggregations over a year (25 TB) require a distributed engine or pre-aggregated tables for interactive dashboards.

Building Blocks by Layer

Layer	Purpose	Options (examples, not endorsements)
Sources	Where data originates	OLTP databases, SaaS APIs, event streams, files, logs
Ingestion	Move data into the platform	Batch extract, CDC (Debezium, cloud DMS), managed connectors, event streaming (Kafka, Kinesis, Pub/Sub)
Storage	Durable, queryable data	Object storage + open table formats (Iceberg, Delta, Hudi); cloud warehouses (BigQuery, Snowflake, Redshift, Synapse)
Processing	Transform and model	SQL in the warehouse, Spark, Flink, streaming SQL, Python for small data (DuckDB, Polars)

Layer	Purpose	Options (examples, not endorsements)
Orchestration	Schedule and coordinate	Airflow, Dagster, Prefect, cloud-native schedulers, event-driven triggers
Serving	Deliver data to consumers	Warehouse/BI, semantic layer, OLAP stores (ClickHouse, Druid, Pinot), key-value stores, APIs, reverse ETL, feature stores
Quality	Detect bad data	Tests in the transformation layer, validation frameworks, anomaly detection, data contracts
Governance	Know and control data	Catalog, lineage, access policies, classification, retention
Observability	Know the system is healthy	Freshness, volume, and schema monitors; pipeline metrics; cost dashboards

Choose per layer based on requirements — and prefer fewer, well-understood components over many specialized ones.

Architecture Patterns

Batch lakehouse / warehouse (medallion)

```
sources —batch/CDC—> bronze (raw, immutable) —> silver
(clean, conformed) —> gold (modeled, aggregated) —> BI / ML
```

orchestrator runs

each layer on a schedule, with tests between layers

Use for: most analytics; freshness of an hour to a day. **Strengths:** simple, cheap, easy to backfill. **Weakness:** latency.

Streaming (Kappa)

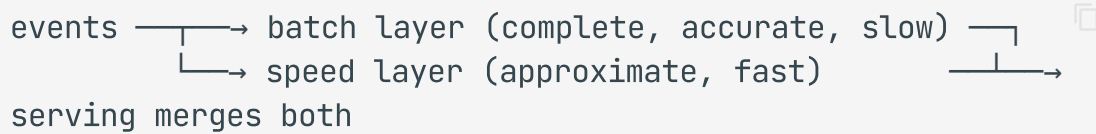
```
producers —> event log (Kafka) —> stream processor —>
serving store / lakehouse tables
```

└─ replay the log to reprocess history

Use for: second-level freshness, operational use cases (fraud, alerting, live dashboards).

Strengths: one code path for real-time and reprocessing. **Weakness:** operational complexity; long retention needed to replay.

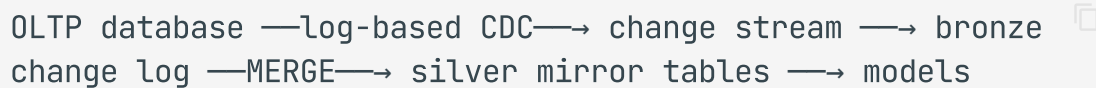
Lambda (batch + speed layer)



Use for: legacy designs or when batch correctness and real-time views are both required.

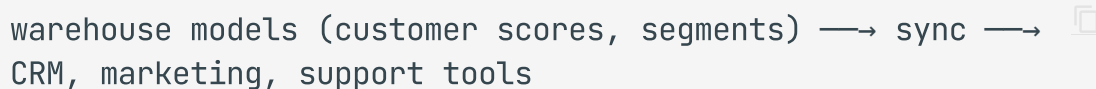
Weakness: two code paths that must agree — usually replaced today by streaming into table formats plus batch compaction.

CDC-based replication



Use for: analytics on operational data with minutes of latency and full delete/update fidelity.

Reverse ETL and operational analytics



Use for: putting modeled data back into business applications.

Data mesh (organizational pattern)

Domain teams own and publish their data as products (with contracts, SLAs, and documentation) on a shared self-service platform. **Use for:** large organizations where a central team is the bottleneck. It changes ownership, not just technology.

Batch or Streaming?

Question	Suggests batch	Suggests streaming
Required freshness	Hours or daily	Seconds to a few minutes
Consumer	Humans reading dashboards and reports	Machines acting on events (fraud checks, alerts, personalization)
Data arrival	Files, periodic exports	Continuous events
Transformations	Complex joins across many large tables	Per-event logic, windowed aggregations
Team experience	SQL, orchestration	Stream processing, stateful systems operations
Cost sensitivity	High	Business value justifies always-on compute

Middle ground: micro-batches every 5–15 minutes (incremental processing on a schedule, or streaming engines with `availableNow`-style triggers) satisfy most "near real time" requirements at a fraction of the complexity.

Storage and Serving Choices

Access pattern	Good fit
Large scans and aggregations, ad hoc SQL	Warehouse, or lakehouse tables with a SQL engine
Many engines sharing the same data	Open table format (Iceberg/Delta) in object storage
Sub-second dashboards over billions of events	OLAP store (ClickHouse, Druid, Pinot) or pre-aggregated tables
Point lookups by key at high QPS	Key-value / document store (e.g. DynamoDB, Bigtable, Cassandra, Redis)
Features for ML training and online inference	Feature store (offline tables + online key-value store)
Full-text or semantic search	Search engine or vector index

Access pattern	Good fit
Sharing data with external parties	Data sharing features, exports to object storage, or APIs

Modeling for serving: star schemas for BI; wide denormalized tables for specific dashboards; pre-aggregated rollups for high-concurrency, low-latency use cases.

Reliability and Correctness

Concern	Design choice
Retries and reruns	Every task idempotent: overwrite partitions or <code>MERGE</code> on keys; no blind appends
Duplicates	Stable event IDs; deduplicate in silver; exactly-once only where the sink supports it
Late data	Event-time processing with watermarks; reprocess affected partitions for a lookback window
Backfills	Parameterize by date range; same code path as daily runs; throttle to protect sources
Source outages	Retries with backoff; alert on freshness; don't publish partial data as complete
Bad data	Validate at each layer; quarantine; block publication of gold tables when critical tests fail
Schema changes	Contracts, schema registry, automatic handling of additive changes, alerts on breaking ones
Recovery	Immutable raw layer + time travel in table formats; documented runbooks

Publish atomically: consumers should see either yesterday's complete data or today's complete data — never a half-written table. Table formats (single commit), write-then-swap, or views pointing to the latest complete partition achieve this.

Scalability

- **Partition** large tables by the most common filter (usually a date); cluster or sort on secondary filters

- **Avoid skew:** hot keys in joins and aggregations (salting, broadcast joins, adaptive execution)
- **Keep files healthy:** 128 MB–1 GB files; compact small files from streaming writes
- **Process incrementally:** transform only new or changed partitions instead of full refreshes
- **Scale ingestion horizontally:** partitions in the event log, parallel extract tasks, consumer groups
- **Separate workloads:** isolated compute for ingestion, transformation, BI, and data science so they don't compete
- **Know the next bottleneck:** state what breaks at 10× — and how the design would change

Security, Privacy, and Compliance

Area	Practice
Access	Role-based access, least privilege, separate service identities per pipeline, no shared personal credentials
Sensitive data	Classify columns; mask, tokenize, or hash PII; restrict raw layers containing PII
Encryption	At rest (managed keys, or customer-managed for regulated data) and in transit (TLS)
Secrets	Secrets manager; short-lived credentials via workload identity
Right to erasure (e.g. GDPR)	Know where each person's data lives (lineage + catalog); delete with <code>MERGE / DELETE</code> in table formats, then expire old snapshots so deleted data is physically removed
Residency	Keep regulated data in its required region; replicate only derived or anonymized data
Audit	Log data access and changes; retain audit logs per policy

Cost

Lever	Effect
Right freshness	Hourly instead of real-time can cut compute cost by an order of magnitude
Incremental processing	Process only what changed
Storage tiers and lifecycle rules	Move old raw data to archive tiers; expire temporary data
Columnar formats and compression	Less storage and less data scanned per query
Partition pruning and clustering	Queries read only relevant data
Auto-suspend and autoscaling	Pay for compute only while it's working
Workload isolation and tagging	Attribute cost per team and pipeline; find the expensive jobs
Pre-aggregation for dashboards	Avoid re-scanning raw data for every dashboard view

State expected costs in the design, even roughly — it is part of the trade-off.

Worked Design: Clickstream Analytics

Requirements: 500M events/day from web and mobile apps; product dashboards within 5 minutes; analysts need 2 years of history; sessionization; PII (IP address, user ID) must be protected.

```
apps —HTTPS—> collector service —> event log (24 partitions, 7-day retention)
                                     |
                                     stream processor: validate, enrich
                                     (geo from IP, then drop IP),
                                     deduplicate by event_id, event-time
                                     windows with 10-min watermark
                                     |
                                     |
                                     bronze_events (Iceberg/Delta, append,
                                     rollups_5m (pre-aggregated by page,
```

```

        partitioned by event_date)
country, device) → real-time dashboard
|
    nightly batch: sessionize (30-min inactivity),
build fct_sessions, fct_page_views,
    dim_user (pseudonymous ID) → warehouse/BI for
analysts

```

Key decisions and trade-offs - Streaming for the 5-minute dashboards; batch for sessionization and history (cheaper, simpler, easy to backfill) - Raw IP used only for enrichment, then dropped; user IDs pseudonymized in silver - Dashboard reads small pre-aggregated tables, not raw events - Late events (mobile apps offline) handled by the watermark for real-time views and by the nightly batch for complete history - At 10×: more partitions and processor instances; compaction jobs for streaming output; OLAP store for dashboards if concurrency grows

Worked Design: Operational Database to Lakehouse

Requirements: replicate 40 tables from a production Postgres database for analytics; latency under 15 minutes; deletes must be reflected; no noticeable load on production; point-in-time history for customers.

```

Postgres (logical replication) → CDC connector → change topics (one per table, Avro + schema registry)
|
    bronze_<table>_changes:
append-only change log (op, before, after, LSN)
|
every 10 minutes
    silver_<table>: MERGE latest
change per key by LSN (deletes applied)
|
    dim_customer (SCD Type 2
built from the change log) • fact tables • marts

```

Key decisions and trade-offs - Log-based CDC: captures deletes, preserves order, minimal source load - Change log kept append-only: supports audits, SCD Type 2, and rebuilding silver from scratch - 10-minute micro-batch MERGE instead of per-event updates: meets the SLA with far less cost than continuous upserts - Monitoring: replication

slot lag, connector status, end-to-end freshness per table, daily row-count reconciliation with the source - Schema changes: additive columns evolve automatically; breaking changes pause the affected table and alert

Worked Design: Daily Business Metrics with an SLA

Requirements: finance and leadership dashboards with revenue, orders, and customer metrics; data complete by 07:00 local time; numbers must match the billing system to the cent; 12 sources including SaaS APIs and two databases.

```
02:00 ingestion tasks (managed connectors + custom API  
extracts) → bronze  
03:00 sensors confirm every source delivered yesterday's  
data (freshness + row-count checks)  
03:15 transformations: staging → intermediate → marts, run  
incrementally, tests after each layer  
05:30 reconciliation: revenue vs billing system totals  
(tolerance: 0.01)  
05:45 publish: swap views to the new partition only if  
every critical test passes  
06:00 buffer for retries • 07:00 SLA • alerts to on-  
call from 06:15 if not published
```

Key decisions and trade-offs - Batch is sufficient: daily consumers; simplicity and cost win
- Explicit dependency on source freshness — never publish metrics built on missing data -
Reconciliation against the system of record is a blocking test, because trust is the product
- Atomic publish: dashboards show yesterday's complete numbers until today's pass every
check - Metric definitions in one place (a semantic layer or governed gold tables) so every
dashboard agrees

Common Pitfalls

Pitfall	Symptom	Fix
Designing before clarifying requirements	Wrong freshness, wrong grain, rework	Spend the first minutes on consumers, SLAs, volume, and constraints

Pitfall	Symptom	Fix
Streaming by default	High cost and on-call burden for data read once a day	Match freshness to the actual decision latency; consider micro-batch
No estimates	Components wildly over- or under-sized	Back-of-the-envelope numbers for throughput, storage, and files
Too many tools	Integration and operational overhead dominates	Fewer components; use managed services where they fit
Non-idempotent pipelines	Duplicates and gaps after every retry or backfill	Overwrite partitions or <code>MERGE</code> ; parameterize by date
Publishing partial data	Dashboards show a revenue drop that isn't real	Freshness gates, blocking tests, atomic publish
Ignoring deletes and late data	Silent divergence from the source	<code>CDC</code> or reconciliation; watermarks and lookback reprocessing
Security added at the end	<code>PII</code> spread across every layer	Classify and protect at ingestion; least-privilege access by layer
No cost model	Surprise bills after launch	Estimate cost per component; tag and monitor spend
No evolution story	Redesign needed at the first growth spurt	State what changes at 10× and where the next bottleneck is

Cheat Sheet

Interview framework: clarify → estimate → high-level diagram → deep dive → reliability, quality, security, cost → trade-offs and 10× plan

Freshness → architecture

Freshness	Typical design
Daily	Scheduled batch, medallion layers, orchestrator
Hourly	Incremental batch (only new partitions)
5–15 minutes	Micro-batch or <code>CDC</code> + periodic <code>MERGE</code>

Freshness	Typical design
Seconds	Event log + stream processor + low-latency serving store

Estimation shortcuts: 1 day $\approx 10^5$ s · peak $\approx 3\text{--}5\times$ average · JSON $\rightarrow$ Parquet $\approx 5\text{--}10\times$ smaller · target files 128 MB–1 GB · 1M events/day ≈ 12 events/s

Checklist for every design: idempotency · late data · backfills · schema changes · data quality gates · atomic publish · monitoring (freshness, volume, errors, cost) · access control and PII · retention · disaster recovery · ownership and on-call

Interview Questions

Q: How do you approach a data system design question? A: Clarify requirements first — consumers, use cases, freshness, volume, correctness, SLAs, security, and budget — and write assumptions down. Estimate scale with rough numbers. Draw the high-level flow from sources through ingestion, storage, processing, and serving. Deep-dive into the hardest part (often the data model or correctness under failure). Then cover reliability, quality, security, and cost, and finish with trade-offs and what changes at $10\times$ scale.

Q: When is streaming worth it compared with batch? A: When a consumer acts on data within seconds or minutes — fraud detection, alerting, operational dashboards, personalization — and the value of that speed exceeds the added cost and complexity of always-on stateful processing. For humans reading reports, hourly or daily batch is usually enough, and micro-batching every few minutes covers most "near real time" needs at a fraction of the effort.

Q: How do you make sure dashboards never show incomplete data? A: Gate publication on completeness: check that every source delivered (freshness and row-count checks), run blocking data tests and reconciliations, and publish atomically — a single table-format commit, or a view swap to the new partition — only after everything passes. Until then, consumers keep seeing the previous complete version, and on-call is alerted if the SLA is at risk.

Q: Compare Lambda and Kappa architectures. A: Lambda runs a batch layer for complete, accurate results and a speed layer for low-latency approximations, merging them at query time — but it requires maintaining two implementations of the same logic. Kappa uses a single streaming path and reprocesses history by replaying the event log, which is simpler but needs long retention and a capable stream processor. Many modern designs stream into open table formats and use batch jobs only for compaction and corrections, getting most of the benefit of both.

Q: How would you design for a GDPR deletion request in a data lake? A: Know where the person's data lives through a catalog and lineage keyed by a stable identifier. Minimize the problem by pseudonymizing identifiers early and keeping direct PII in few, well-known tables. Execute deletes with `DELETE / MERGE` in the table format across affected tables, then expire old snapshots and vacuum files so the data is physically removed, and record the request's completion for audit. Raw immutable layers need either short retention or encryption per subject (crypto-shredding).

Q: What would change in your design at 10× the volume? A: Name the first bottleneck and its fix: more partitions and consumers for ingestion; incremental rather than full processing; stronger partitioning, clustering, and compaction; pre-aggregation for serving; separate compute per workload; and revisiting cost — for example tiered storage or shorter raw retention. The goal is to show you know where the design stops working, not to over-build for it on day one.

Further Reading

- *Designing Data-Intensive Applications* — Martin Kleppmann (O'Reilly)
 - *Fundamentals of Data Engineering* — Joe Reis & Matt Housley (O'Reilly)
 - *Streaming Systems* — Tyler Akidau, Slava Chernyak & Reuven Lax (O'Reilly)
 - *Data Mesh* — Zhamak Dehghani (O'Reilly)
 - [The Log: What every software engineer should know about real-time data](https://web.archive.org/web/20260612105408/https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying) (<https://web.archive.org/web/20260612105408/https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>) (archived — the original LinkedIn Engineering post has since been taken down) — Jay Kreps
 - [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (<https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/>) • [Data Ingestion & CDC](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/>) • [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (<https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/>)
-

Previous: [Data Ingestion & CDC](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/ingestion-cdc/>) • **Next:** [Choosing a Stack](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/) (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/>) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026