SYSTEM DESIGN FOR DATA ENGINEERS

A Complete Interview & Architecture Playbook

By Manjinder Brar

Premium Edition - 2025

Version 1.0
Last Updated: *Dec 2025*

Contact
Email
Linkedin

FRAMEWORK

Foundations of Data Engineering System Design

Introduction

After going through countless interviews and researching how top data teams think about architecture, I realized something important: system design success has very little to do with memorizing tools and everything to do with how you think. This playbook captures what I learned through that journey. It teaches you how to approach system design the way experienced Data Engineers do, especially in interviews where clarity and reasoning matter more than anything else.

Throughout my preparation, I noticed that strong engineers always focus on fundamentals. They ask the right questions, clarify the problem, understand business constraints, evaluate real tradeoffs, and design with intention instead of copying patterns. This playbook is built on those principles.

You will learn how to handle ambiguity, structure your answers, explain your decisions, and communicate architectural choices with confidence. You will also understand how to move from simple ETL pipelines to more advanced systems like lakehouse architectures, real-time streaming pipelines, ML feature stores, identity resolution, and large-scale analytics.

Everything in this playbook is shaped by real interview experience, repeated feedback from hiring managers, and a lot of trial and error. My goal is to help you skip the confusion, avoid the mistakes I made in the beginning, and walk into any system design round with the mindset of a well-prepared, thoughtful engineer.

By the end, you will not just know system design. You will understand how to think about it the way senior engineers do.

Section I : Framework Overview

Mindset of a DE System Design Interview

System design for Data Engineering is fundamentally different from Software Engineering because the focus shifts from user-facing features to *data movement*, *scalability*, *reliability*, and *governance*.

A software engineer usually designs systems around request–response flows, APIs, microservices, and latency requirements, whereas a data engineer designs pipelines, storage layers, ingestion patterns, scheduling, and data transformations that must work reliably at scale.

In DE system design, questions revolve around how:

- data enters the system, how it is validated,
- to handle schema evolution,
- to handle late-arriving or corrupt data, and
- to guarantee accuracy across environments.

Instead of load balancing user traffic, we design for *volume spikes*, *streaming throughput*, *batch windows, backfill strategies, data quality checks, lineage,* and *auditability.*

The emphasis is not only on performance but also on cost optimization, governance, access control, and how downstream consumers like analytics, ML, and reporting will use the data.

Before diving into the steps, internalize this: the interviewer wants to see *how you think*. They are evaluating your ability to handle ambiguity, make trade-offs, communicate clearly, and justify your decisions. The Core Mindset: It's a Collaborative Discussion, Not a Test

- **Think Aloud**: Narrate your thought process. "Okay, we have data coming from mobile apps. My first thought is about potential connectivity issues, so I'm considering a client-side buffer..."

- **Be a Collaborator:** Treat the interviewer as a teammate. Ask questions like, "Does this assumption make sense?" or "What are your thoughts on using X for this component?"

- **Structure is Your Friend:** State your plan at the beginning. "First, I'd like to understand the requirements and constraints. Then, I'll sketch a high-level architecture, and after that, we can deep-dive into specific components. Finally, we can discuss scalability and monitoring. Does that sound good?"

The Detailed Framework: A 5-Step Approach

Follow these steps in order. Do not skip Step 1; it is the most critical.

Step 1: Clarify Requirements and Define the Scope (10-15% of time)

This is where you demonstrate your ability to understand a business problem before jumping to a technical solution. The goal is to transform a vague prompt (e.g., "Design a data pipeline for user analytics") into a concrete set of engineering requirements.

Ask *clarifying questions* about Functional and Non-Functional Requirements.

A. Functional Requirements (What does the system do?)

- Who are the users? (Data scientists, business analysts, machine learning models, external customers?)
- What are the use cases? (A daily sales dashboard, a real-time fraud detection model, a recommendation engine, ad-hoc exploratory analysis?)
- What are the key outputs? (Aggregated tables in a warehouse, a low-latency API, a feature store for ML?)

B. Non-Functional Requirements (The "-ilities" that drive design choices)

- *Data Sources:*
    - Where does the data come from? (Mobile/web events, relational databases, third-party APIs, IoT sensors, flat files?)
    - What is the data format? (JSON, Avro, Protobuf, CSV, database logs?)
    - Is it a push (sources send data) or pull (we fetch data) model?
- *Scale and Volume* (Quantify Everything!):
    - *Volume at Rest*: How much total data will be stored? (Terabytes, Petabytes?) This informs storage choices.
    - Volume in Motion: What is the ingestion rate? (1,000 events/sec, 10 GB/hour?) This informs ingestion and processing tech.
    - Growth: What is the expected data growth rate? (e.g., 20% year-over-year?)
- *Data Freshness (Latency)*:
    - How up-to-date does the final data need to be?
    - Batch: Daily, hourly? (e.g., daily sales reports)
    - Near Real-Time: Within seconds or minutes? (e.g., updating a user activity dashboard)

    - Real-Time: Within milliseconds? (e.g., fraud detection)

- *Consistency and Reliability:*

    - How critical is data loss? Can we tolerate losing a few events (e.g., ad clicks), or is every record critical (e.g., financial transactions)? This dictates choices like at-least-once vs. exactly-once processing.
    - What are the data quality requirements? Does the data need to be 100% accurate?

- *Security and PII* (Personally Identifiable Information):
    - Does the data contain sensitive information? This will require considerations for encryption, masking, and access control.

C. **Scope and Assumptions** After asking questions, explicitly state your assumptions and the scope.

- Example: *"Okay, based on our discussion, we're designing a system to process 10 million events per day from mobile clients. The main output will be a daily aggregated dashboard for business analysts, so a 24-hour data freshness is acceptable. We'll assume for now that we don't need to handle PII data, but we can discuss that later if time permits. The focus will be on a scalable and cost-effective batch processing pipeline."*

Step 2: High-Level Architecture (20-25% of time)

Draw a high-level block diagram. Don't get bogged down in specific technologies yet. Focus on the logical components and the flow of data. A great starting point is a simple, generic pipeline:

Ingest -> Store -> Process/Transform -> Serve

Draw the Blocks: On the whiteboard, draw boxes for each major component and arrows showing the data flow.

Identify Data Flow: Explain how data moves from one end to the other.

- "First, events from mobile and web clients will hit an Ingestion Service.
- This service will buffer the data and write it to a raw storage layer, our Data Lake.
- From there, a daily Transformation Job will run, which reads the raw data, cleans and aggregates it, and writes the results to a structured Data Warehouse.
- Finally, our BI tools and dashboards will query the Data Warehouse."

Justify the Components (The "Why"): Briefly explain the purpose of each component.

- Why a Data Lake? "We need a cheap, scalable place to store raw data in its original format. This allows us to reprocess data if our business logic changes and enables data scientists to do ad-hoc exploration."
- Why a separate Data Warehouse? "We need an optimized storage layer for fast analytical queries. A columnar format here will make our dashboard queries much faster than querying raw JSON files in the lake."

Step 3: Deep Dive into Components and Technology Choices (40-50% of time)

This is the core of the interview. The interviewer will likely guide you to focus on specific areas. Be prepared to go deep on any part of your diagram. For each choice, always discuss trade-offs.

1. Ingestion Layer

- *Batch:* SFTP, AWS DataSync, database dumps, tools like Fivetran/Stitch.

- *Streaming:*
  - Message Queue: Kafka, Kinesis, Google Pub/Sub.
    - Why? Decoupling producers from consumers, durability (data is saved to disk), backpressure handling.
    - Trade-offs: *Kafka* offers high throughput and flexibility but requires self-management.
    - *Kinesis/Pub/Sub* are managed services, which are easier to operate but might be less flexible or more expensive.
  - API Gateway/Load Balancer: For receiving HTTP requests from clients.

2. Storage Layer

- *Data Lake (Raw/Staging):* Amazon S3, Google Cloud Storage (GCS), HDFS.
  - File Formats: Talk about Parquet or ORC. Why? Columnar storage, good compression, predicate pushdown. This is a key detail that shows expertise.
  - Partitioning Strategy: "I would partition the data in the lake by date (e.g., `s3://bucket/raw/event_name/year=2023/month=10/day=25/`). This drastically improves query performance by allowing processing jobs to skip reading irrelevant data."

- *Data Warehouse (Processed/Structured):* Snowflake, BigQuery, Redshift, Databricks (Delta Lake).

    - Why? Managed, scalable, SQL interface, optimized for analytics.
    - Trade-offs: *Snowflake/BigQuery* have decoupled storage/compute, great for variable workloads. *Redshift* can be faster for specific, stable workloads but is less flexible.

- *Serving Layer (Low-Latency Access):*

    - If the use case is a user-facing dashboard, a warehouse might be too slow.
    - Options: Druid, Pinot (OLAP databases), or pre-aggregated tables in Postgres/MySQL, or a Key-Value store like Redis/Cassandra for single lookups.

3. **Processing/Transformation Layer**

- *Orchestration*: How will you schedule and manage dependencies?

    - Tools: Airflow, Dagster, Prefect, Step Functions, ADF
    - Why? To define workflows as DAGs (Directed Acyclic Graphs), handle retries, manage dependencies, and monitor job failures.

- *Transformation Engine:*

    - Batch:
        - Spark: The industry standard for large-scale distributed processing.
        - dbt (Data Build Tool): Excellent for transformations that can be expressed in SQL, runs directly inside your data warehouse. Great for analytics engineering.

    - Streaming:
        - Spark Streaming or Apache Flink.
        - Key Concepts to mention: Windowing (tumbling, sliding), state management, watermarking for handling late data.

Step 4: Discuss Cross-Cutting Concerns (10-15% of time)

This is where you graduate from a good candidate to a great one. It shows you think about production-ready systems.

- **Data Quality and Monitoring-** "How do we know the pipeline is healthy?" :
    - *Monitoring*: Track metrics like data volume, data freshness, job completion times. Use tools like Prometheus/Grafana or Datadog.
    - *Data Quality*: Implement data contracts. Use tools like Great Expectations or dbt tests to run assertions on your data (e.g., `user_id` should never be null, `revenue` should be positive).
    - *Alerting*: Set up alerts for pipeline failures or data quality test failures.
- **Security and Governance:**
    - How will you manage access control? (e.g., Role-Based Access Control - RBAC).
    - How will you handle PII? (Masking, tokenization, separate access policies).
    - How will you track data lineage? (What data was used to produce this report? Tools like Collibra, Alation, or open-source solutions).
- **Scalability:**
    - "How would this design handle 10x the data?"
    - Talk about the elasticity of your chosen components (e.g., auto-scaling Spark clusters, serverless warehouses like BigQuery).
    - Mention that the partitioned data lake and columnar file formats are designed for scale from the start.
- **Cost Management:**
    - Briefly touch on cost optimization strategies (e.g., using spot instances for batch jobs, data lifecycle policies to move old data to cheaper storage like S3 Glacier, choosing serverless tools to pay only for usage).

Step 5: Summary and Future Improvements

Conclude by summarizing your design and the key trade-offs you made.

- Recap: "So, to summarize, we've designed a batch data pipeline using Kafka for decoupled ingestion, S3 with Parquet files for our data lake, Spark managed by Airflow for daily transformations, and Snowflake as our data warehouse for analysts. This design prioritizes scalability and cost-effectiveness for the daily reporting use case."

- Identify Bottlenecks: "The main bottleneck here might be the 24-hour batch interval. If the business needs faster insights,..."

- Suggest Future Improvements: "...we could evolve this architecture. We could introduce a parallel streaming path using Flink to provide a near-real-time dashboard for key metrics, supplementing our daily batch reports. We could also build a data catalog to improve data discovery for our analysts."

Trade offs

In real-world data engineering, there is *no perfect architecture*, only deliberate choices made under constraints.

Great engineers separate themselves not by memorizing tools, but by demonstrating the ability to evaluate competing priorities: latency vs. cost, batch vs. streaming, managed vs. self-hosted, SQL vs. Spark, warehouse vs. lake, and more.

This section captures the *core trade-offs* that senior engineers naturally think about when designing systems at scale. In interviews, articulating these trade-offs clearly shows maturity, ownership, and practical experience.

The goal is not to pick the “right” answer but to explain why this solution fits these requirements, for this business, at this scale. If you can communicate trade-offs confidently, you instantly signal that you understand real production systems, not just textbook diagrams.

Use this section to anchor your thinking, justify your decisions, and stand out as a true data engineering architect.

The magic phrase is: "I'm choosing X over Y because for this specific problem, we need to prioritize [A] over [B]. For example..."

Let’s now look at some common trade-offs that could come up in an interview.

1. Managed Service vs. Self-Hosted

The Trade-off: Using a cloud provider's service (e.g., AWS Kinesis, Google BigQuery) vs. deploying and managing open-source software yourself (e.g., Kafka, Presto on EC2/Kubernetes).

Core Principle: Operational Simplicity vs. Control & Cost.

When to Choose a Managed Service:

- Small team, limited DevOps/SRE resources.
- Need to move fast and deliver business value quickly.
- Workload is "spiky" or unpredictable (leverages serverless/auto-scaling).
- The problem is standard and fits the service's capabilities well.

When to Choose Self-Hosted:

- Need for deep customization or specific configurations not offered by the managed service.
- Operating at a massive, predictable scale where you can optimize infrastructure for lower costs.
- Avoiding vendor lock-in is a strategic priority.
- Strict data residency or security requirements that a managed service can't meet.

Interview Talking Points:

"For this stage of the company, I'd recommend a managed service like BigQuery. It reduces our operational overhead and lets the team focus on building the data logic, not managing clusters."

"At this scale, self-hosting Kafka on Kubernetes could be more cost-effective. It would also give us fine-grained control over retention policies and replication factors."

2. Batch Processing vs. Stream Processing

The Trade-off: Processing data in large, discrete chunks (batch) vs. processing data event-by-event as it arrives (streaming).

Core Principle: Latency vs. Cost & Throughput.

When to Choose Batch (e.g., Spark, dbt):

- Use cases: Daily/hourly reports, BI dashboards, training ML models.
- Data freshness requirements are measured in hours or days.
- The goal is high throughput and cost-efficiency for large volumes of data.
- The logic is complex and requires analyzing the entire dataset at once.

When to Choose Streaming (e.g., Flink, Spark Streaming):

- Use cases: Real-time fraud detection, live user activity dashboards, alerting systems.
- Data freshness requirements are measured in seconds or milliseconds.
- The business needs to react to events as they happen.

Interview Talking Points:

"Since the requirement is for a daily sales report, a batch processing approach using Spark is the most appropriate. It's cost-effective and simpler to implement than a streaming solution."

"For the fraud detection component, we have a strict low-latency requirement. Therefore, I would implement a streaming pipeline using Flink to analyze transactions in real-time."

3. SQL-based (dbt) vs. General-Purpose Framework (Spark, Flink)

The Trade-off: Using a SQL-centric tool for transformation vs. a programming framework that supports Python, Scala, or Java.

Core Principle: Accessibility & Simplicity vs. Power & Flexibility.

When to Choose SQL-based (dbt):

- Transformations are primarily happening inside the data warehouse.
- The primary users are data analysts or analytics engineers who are strong in SQL.
- The logic involves structured data and can be expressed cleanly in SQL (joins, aggregations, filtering).
- You want to prioritize speed of development for analytics.

When to Choose Spark/Flink:

- Processing happens before the data warehouse (e.g., in a data lake).
- Dealing with unstructured or semi-structured data (JSON, text, images).
- The logic is too complex for SQL (e.g., iterative algorithms, machine learning feature engineering, sessionization).
- High-performance streaming state management is required.

Interview Talking Points:

"Once the data is in Snowflake, I'd use dbt for our modeling layer. It empowers analysts to build their own data marts and provides built-in testing and documentation, which is a huge win for data quality."

"To process the raw JSON logs in our S3 data lake, I'll use Spark. It gives us the flexibility to parse complex nested structures and apply custom Python business logic that would be impossible in pure SQL."

4. Data Warehouse vs. Data Lake

The Trade-off: Storing curated, structured data vs. storing raw, multi-format data.

Core Principle: Performance & Structure vs. Flexibility & Cost. (Note: Most modern systems use BOTH!)

When to Use a Data Lake (S3, GCS):

- To store a cheap, raw, immutable copy of all source data ("single source of truth").
- To enable ad-hoc exploration by data scientists who need the original, untransformed data.
- To land data from diverse sources in various formats (JSON, logs, CSV, etc.).

When to Use a Data Warehouse (Snowflake, BigQuery):

- To serve cleaned, modeled, and structured data to business users for BI and reporting.
- To provide fast query performance for analytical workloads.
- To enforce schemas and ensure data quality for critical business metrics.

Interview Talking Points:

"I propose a two-tiered storage architecture. We'll land all raw data into an S3 data lake for cost-effective, long-term storage. From there, a curated, aggregated subset will be loaded into BigQuery to power our executive dashboards, ensuring fast query performance."

5. Columnar (Parquet, ORC) vs. Row-based (Avro, JSON, CSV) File Formats

The Trade-off: How data is physically laid out on disk.

Core Principle: Analytical Query Speed vs. Full Record Retrieval Speed.

When to Choose Columnar (Parquet, ORC):

- Almost always for analytics and data lake storage.
- Queries typically select a few columns from a large dataset (e.g., SELECT user_id, SUM(purchase_amount) ...).
- High compression is needed to save costs.

When to Choose Row-based (Avro):

- When passing individual events through a messaging system like Kafka.
- When the use case is retrieving the entire record for a given key.
- Schema evolution is a critical feature. (Note: Avro is the best row-based format; avoid JSON/CSV for serious pipelines due to lack of typing and schema enforcement).

Interview Talking Points:

"For our data lake files, I would mandate using Parquet. Its columnar nature means our analytical queries in Spark or Athena will only read the columns they need, drastically reducing I/O and improving performance."

"The events flowing through our Kafka topics should be serialized using Avro. It enforces a schema, which prevents data quality issues upstream, and it's very efficient for serializing/deserializing entire records."

Some Great Sample Answers:

Design the extraction process

Example

"For our extraction process, I propose the following approaches:

For the MySQL database, we'll use AWS Database Migration Service (DMS) for the initial full load. This service is designed for efficient and reliable data migration. For subsequent incremental updates, we'll implement a custom solution that reads the binary log (binlog) to capture changes. This approach ensures we only extract modified data, reducing load on the source system.

To extract data from the REST API, we'll develop a custom Python script using the requests library. This script will handle authentication, pagination, and error retries. We'll schedule this script to run at regular intervals, adjusting the frequency based on the API's rate limits and our data freshness requirements.

For the CSV files, we'll leverage AWS S3 events to trigger our processing. Whenever a new file is uploaded or an existing file is updated, an AWS Lambda function will be invoked to initiate the extraction process. This event-driven approach ensures we process new data promptly without unnecessary polling.

These methods provide a balance of efficiency, reliability, and scalability. However, I'm open to adjusting this approach based on any specific constraints or preferences you might have."

Plan the transformation logic

"For our transformation logic, I propose the following steps:

First, we'll focus on data cleansing. This includes removing duplicate customer records based on a unique identifier like email address. We'll also standardize address formats to ensure consistency, which will be crucial for our geocoding step.

Next, we'll enrich the data by geocoding customer addresses to add latitude and longitude. This will enable spatial analysis in our data warehouse. We can use a service like Google's Geocoding API for this step.

For aggregations, we'll calculate daily order totals by product category. This pre-aggregation will speed up common analytical queries. We'll store both the raw data and these aggregates to allow for flexible analysis.

To handle schema changes, particularly for the nested JSON structures from our API, we'll implement a flexible transformation step. We'll use a library like JSONata to flatten the nested structures into a tabular format suitable for our data warehouse.

Throughout these transformations, we'll implement data validation checks to ensure data quality. This includes checking for null values, ensuring data types are correct, and validating that calculated fields are within expected ranges.

We'll implement these transformations using Apache Spark, which provides a scalable framework for complex data processing. Spark's ability to handle both batch and streaming data will give us flexibility as our needs evolve.

Determine the loading strategy

Example

"For our loading strategy, I propose a combination of approaches to balance efficiency, data freshness, and system load:

We'll start with a batch load for our initial historical data. This will allow us to efficiently populate our data warehouse with existing data. We'll use Redshift's COPY command to load data from our S3 staging area, as it's optimized for bulk loading.

For ongoing updates, we'll implement a micro-batch strategy, processing and loading data every 15 minutes. This strikes a balance between near-real-time data availability and system efficiency. Again, we'll use the COPY command for these loads.

To handle upserts (updates and inserts), we'll implement a staging table approach. New data will be loaded into a staging table, and then we'll use a merge operation to update existing records and insert new ones. This approach allows us to efficiently handle both new and updated data.

For late-arriving data, our staging table approach will naturally handle this. If data arrives out of order, it will simply be included in the next micro-batch and merged appropriately.

We'll also implement error handling and logging in our loading process. If a load fails, the system will retry a configurable number of times before alerting our operations team.

Ensuring Pipeline is performant, scalable and fault-tolerant

Example

"To ensure our ETL pipeline is performant, scalable, and fault-tolerant, I propose the following strategies:

For performance optimization, we'll implement partitioning in our Redshift tables based on frequently used query patterns. For example, we might partition order data by date to speed up time-based queries. We'll also use distribution keys to minimize data movement during joins. Additionally, we'll implement a caching layer using Redis for frequently accessed, slowly changing data like product information.

To address scalability, we'll use auto-scaling for our EC2 instances running Spark jobs. This will allow us to dynamically adjust our processing capacity based on the current workload. We'll also implement data partitioning in our S3 staging area, allowing for parallel processing of data. This will be particularly useful as our data volume grows.

For fault tolerance, we'll implement checkpointing in our Spark jobs. This will allow us to resume processing from the last successful checkpoint in case of failures. We'll use AWS Step Functions to orchestrate our ETL workflow, which provides built-in error handling and automatic retries.

We'll also implement comprehensive logging and monitoring using AWS CloudWatch. This will help us quickly identify and respond to any issues in the pipeline. We'll set up alerts for key metrics like job failure rates, processing times, and data quality issues.

Section II : Cross-Cutting Architecture Patterns in Data Engineering

Modern data systems are never built from a single technology or a single design choice.
They are built from **patterns**, repeatable architectural principles that appear across companies, industries, and use cases. Whether you are designing a streaming pipeline, a batch ingestion framework, a feature store, or a lakehouse, these patterns quietly shape every major decision.

Understanding them is what separates an *implementer* from an *architect*.

Most engineers learn tools first.
Great engineers learn patterns first because patterns survive tools.

In interviews, senior engineers are expected to explain **why a particular pattern fits the problem**, not just draw a diagram. In real projects, these patterns help teams reason about scale, correctness, latency, and cost long before a single line of code is written.

This chapter introduces the **foundational architecture patterns** that sit underneath the case studies and system designs discussed throughout this guide. You will learn:

- *How batch and streaming pipelines are fundamentally organized*
- *Why some companies choose Lambda or Kappa architectures*
- *How medallion/lakehouse patterns ensure reliability and governance*
- *How event-driven designs differ from orchestrated workflows*
- *How modern platforms adopt data mesh principles for decentralization*
- *How state management, CDC, and upsert patterns influence correctness at scale*

These patterns are **cross-cutting** because they apply to every domain:
Analytics, machine learning, fraud detection, identity resolution, clickstream processing, and even traditional ETL.

Once you understand them deeply, you will be able to reverse-engineer any real-world data pipeline and redesign it with confidence. More importantly, you'll be able to walk into any system design interview and justify your architecture choices in a way that sounds both deliberate and senior.

The goal of this chapter is simple:

Give you the architectural vocabulary of a Data Engineering Architect.

From here onward, we will break down each pattern, explain why it exists, where it shines, where it breaks, and how to discuss it like a seasoned engineer.

Pattern 1 - Lambda Architecture

Lambda Architecture is one of the most influential patterns in large-scale data engineering.
It was designed to solve a fundamental challenge:

> **How do you support both real-time insights and long-term accuracy in the same system?**

Before modern lakehouse frameworks existed, companies struggled to balance **low-latency streaming** with **high-fidelity batch recomputation**. Lambda Architecture emerged as a hybrid model that combines the strengths of both worlds.

1. What Problem Does Lambda Solve?

In real-world systems, data arrives:

- Out of order
- Late by hours or days
- Sometimes duplicated
- With schema evolution
- From unreliable or noisy sources

A pure streaming pipeline often cannot guarantee **perfect correctness**, while a pure batch pipeline cannot meet **low-latency requirements**.

Lambda Architecture solves this tension by splitting the pipeline into **two parallel paths**:

Speed Layer - Real-Time View

Processes data within seconds or minutes.
Used for dashboards, monitoring, alerting, recommendation freshness, fraud signals.

Batch Layer - Source of Truth

Processes all data in large, deterministic batches.
Used for accurate historical tables, reconciliation, aggregation, and reporting.

Serving Layer - Combined Output

Merges the outputs of both layers to provide:

- Fast but approximate results
- Later corrected by accurate batch computations

This design ensures both **freshness** and **accuracy**, even at massive scale.

2. When to Use Lambda Architecture

Lambda Architecture shines when your system has:

Dual latency requirements

Example:
City ops need metrics within 5 minutes, but finance needs exact daily numbers.

High volume + messy data sources

When the system must handle out-of-order, late, or inconsistent events.

Strict accuracy requirements

When a single streaming pipeline cannot guarantee perfect historical correctness.

Machine learning use cases

When the model needs both:

- Fresh online features (from speed layer)
- Accurate historical training data (from batch layer)

Business workflows that evolve quickly

Batch reprocessing protects you when logic changes.

3. How to Explain Lambda Architecture in an Interview

A strong candidate never just defines Lambda; they explain when and why.

Here's an interview-quality summary:

> "Lambda Architecture solves the problem of balancing low latency with correctness.
> The speed layer provides fresh, approximate results for real-time use cases, while the batch layer recomputes accurate historical outputs.
> The serving layer merges both to ensure fast dashboards today and correct data tomorrow.
> I choose Lambda when the system has strict latency needs but the business also requires high historical accuracy, especially with late-arriving data."

4. When NOT to Use Lambda Architecture

Avoid Lambda when:

- Your latency requirements are > 5 minutes and can be met by micro-batched Spark
- You can adopt a modern lakehouse with streaming upserts (Delta / Hudi / Iceberg)
- Batch and streaming logic can be unified into a single engine
- The engineering team is small and cannot maintain dual pipelines

This naturally leads to *Kappa Architecture*, which simplifies the model.

Pattern 2 - Kappa Architecture

Kappa Architecture was introduced as a direct response to the complexity of Lambda.
The industry realized that maintaining two parallel pipelines - one for streaming and one for batch - creates overhead, duplication, and operational drag. Kappa simplifies this by saying:

> *"What if we process all data as a stream, even the historical data?"*

Instead of a batch layer and a speed layer, Kappa uses one unified streaming engine to handle both real-time events and reprocessing. This dramatically reduces code duplication and is far easier for teams to operate in modern data landscapes.

Why Kappa Exists

At scale, Lambda often becomes painful because:

- You write the same business logic twice (Spark batch + Flink/Spark Streaming)
- You maintain two storage paths
- You reconcile two outputs
- You fix bugs in two places
- You backfill in two different systems

Engineering teams wanted a simpler model - - one codepath, one compute engine, one set of semantics.

Kappa Architecture solves this by treating *all* data as a stream:

- Realtime data → processed continuously
- Historical data → replayed through the same engine when needed

For example, if a new feature needs backfilling, you simply replay events from Kafka, S3 logs, or your lakehouse files through the same streaming pipeline. No separate batch job required.

How Kappa Works (Conceptually)

Instead of a split-layer design, Kappa uses a single pipeline:

1. *Ingest everything as an event stream*
   Kafka, Kinesis, Pub/Sub act as the authoritative log.
   Even batch files can be replayed as streams.

2. *Use one streaming framework (Flink / Spark Structured Streaming)*
   The same transformation logic handles real-time and historical reprocessing.

3. *Persist results in a lakehouse or key-value store*
   Writes are typically idempotent, incremental, and append-only or MERGE-based.

This model assumes your streaming engine is powerful enough to handle joins, stateful processing, upserts, and late-arriving events which modern engines now support.

Strengths of Kappa Architecture

The biggest appeal is **simplicity**. Teams love it because:

- Only *one* pipeline to maintain
- Only *one* codepath for business logic
- Backfills are simply replays
- Production errors are easier to diagnose
- No more Lambda inconsistencies between speed and batch layers
- Perfect for event-driven companies like social networks, IoT, gaming, fraud detection

You also avoid the operational burden of large batch clusters, nightly jobs, and expensive warehouse recomputations.

Another strength is consistency:
The same processing semantics apply to both live and historical data, so you don't worry about discrepancies that come from different frameworks.

When Kappa Is a Great Fit

Kappa shines when:

- The business is primarily event-driven
- You already have Kafka as the backbone of the ecosystem
- Latency requirements are strict but accuracy still matters
- Data arrives continuously, not in large batch dumps
- You want a single, unified architecture for both serving and historical analytics

Streaming-first companies (TikTok, Netflix, Uber after 2021, Stripe, Airbnb) all lean toward Kappa-style architectures today, enhanced by lakehouse technologies.

Where Kappa Struggles

Kappa is powerful, but it's not always the right answer.

The main limitation:
Your streaming engine becomes responsible for everything.

That includes:

- Complex joins
- Heavy aggregations
- Multi-day windows
- Reprocessing terabytes or petabytes of history
- Stateful logic that may blow up memory
- High operational load on the state store (RocksDB, in-memory maps, etc.)

If the streaming engine is not well tuned, replaying large historical datasets can take hours or days.

Another issue is that some organizations still depend heavily on SQL, warehouses, and batch tools. For these teams, adopting Kappa requires a cultural and tooling shift.

How to Explain Kappa in an Interview

A strong, concise explanation might be:

> "Kappa Architecture eliminates the dual-pipeline problem in Lambda by treating all data - historical and real-time - as a stream. You use one codepath, one streaming engine, and replay historical events for backfills.
> It reduces complexity, ensures consistent logic, and is ideal for event-driven systems where streaming is the natural mode of operation."

This is crisp, senior, and highlights the reason the architecture exists.

Kappa vs. Lambda - The Real Tradeoff

The tradeoff is not about which is “better,” but which is more aligned with the system’s nature.

- Lambda → for systems with dual latency needs and very complex batch recomputation
- Kappa → for systems where streaming is the primary mode and batch adds redundancy

Modern lakehouse engines (Delta, Hudi, Iceberg) blur this line even further by enabling streaming upserts, change logs, and exactly-once semantics, making Kappa the natural fit for many next-generation platforms.

Pattern 3 - Medallion Architecture (Bronze → Silver → Gold)

The Medallion Architecture is the backbone of modern lakehouse design.
It brings order, reliability, and clarity to what would otherwise be a chaotic data lake filled with raw files, inconsistent schemas, and ad-hoc transformations.

Instead of treating the lake as a giant dumping ground, the Medallion model enforces *progressive data refinement* - each layer adds structure, quality, and meaning.

It solves a surprisingly common problem:
every team wants to use the data, but nobody agrees on which version is correct.
The Medallion tiers create a contract for trust.

Why This Pattern Matters

Unlike classical warehouse modeling, lakehouses deal with semi-structured data, schema drift, high-volume append-only logs, and event-driven pipelines. The Medallion pattern gives you a clean way to process messy data step by step.

The idea is simple:

- **Bronze:** land everything, keep it raw
- **Silver:** clean it, standardize it, dedupe it
- **Gold:** aggregate it, model it, serve it

This layered design aligns perfectly with the needs of analytics, ML, experimentation, and regulatory compliance.

Bronze - The Source of Truth Layer

Bronze is where raw data lands exactly as it arrives, with minimal transformation.

This layer is intentionally “ugly”:

- raw JSON, CSV, Avro, API responses
- logs from Kafka/Kinesis
- CDC data dumps
- 3rd-party vendor feeds

Bronze exists so you always have a replayable, immutable, auditable version of the source data.
If an upstream bug occurs, or a backfill is needed, or your logic changes later, Bronze saves you.

Think of it as the *black box recorder* of your data platform.

Silver - The Clean, Queryable Foundation

Silver is where the hard work happens.
This layer transforms raw data into usable, reliable, analytics-ready tables.

Typical operations include:

- deduplication
- schema alignment
- type casting
- handling late-arriving events
- SCD logic
- joining related datasets
- basic enrichment

Silver tables answer:
“What is the clean, trustworthy representation of this entity?”

For example, in a ride-sharing system, Bronze holds raw trip events; Silver constructs a clean `trips` table with one record per completed trip.

This layer dramatically reduces friction for all downstream teams.

Gold - The Business Layer

Gold tables represent *business-ready outputs.*
They are the curated, aggregated, often highly denormalized datasets consumed by:

- dashboards
- BI tools
- ML feature stores
- financial reporting
- operations teams

Gold is where you embed definitions like:

- daily active users
- customer funnels
- revenue metrics
- fraud scores
- user 360 profiles

Gold must be stable, well-documented, and contract-driven because it directly powers business decisions.

A key difference from Silver:
Gold reflects *business logic*, not just cleaned data.

Why Medallion Architecture Works So Well

A clean separation of responsibilities emerges naturally:

- Bronze = storage & lineage
- Silver = correctness & consistency
- Gold = business meaning & performance

It also enables teams to operate independently:

- Data engineering owns Bronze/Silver
- Analytics owns Gold
- ML teams read from whichever layer fits their needs
- Governance teams know exactly where sensitive data lives

This clarity is the main reason Medallion is now the default pattern across Databricks, Snowflake, and modern data platforms.

When Medallion Architecture is a Great Fit

It works best when:

- you ingest from many heterogeneous sources
- schema changes are common
- late or duplicate events arrive frequently
- multiple consumers (BI, ML, Product) share the same data
- you need strong audit trails
- your platform needs to scale independently at each stage

Most real-world systems, especially event-heavy or enterprise-grade platforms, fall into this category.

Where Medallion Falls Short

Despite its popularity, no pattern is perfect.

A few limitations appear in extremely large or high-latency environments:

- multiple layers can introduce extra compute cost
- transformations may duplicate storage across layers
- gold tables can proliferate if not governed
- if the Silver layer is poorly designed, the entire platform becomes rigid
- not ideal for ultra-low-latency < 5 second pipelines without hybrid streaming ingestion

Still, with proper governance, these issues are manageable.

How to Explain Medallion Architecture in an Interview

A polished answer might sound like this:

> "Medallion Architecture organizes lakehouse data into progressive layers.
> Bronze captures raw, immutable data for lineage and replay.
> Silver transforms and cleans it into trustworthy, analytics-ready tables.
> Gold aggregates and models it for dashboards and ML features.
> This pattern gives structure, quality, and clarity to lake environments and eliminates the 'multiple versions of truth' problem."

That's exactly how senior engineers speak about this pattern.

Pattern 4 - Event-Driven Architecture

Event-Driven Architecture (EDA) is the backbone of modern data platforms.
Almost every large-scale system today - ride sharing, payments, e-commerce, streaming analytics - emits a continuous stream of events describing what happened in the system.

Instead of polling databases or running heavy batch jobs, downstream pipelines react to these events as they occur.

In many ways, EDA shifts the mindset from “move data when *we* want to” to “move data when the *business* does something.”

This pattern is especially powerful because it creates real-time, decoupled, scalable systems that evolve naturally with the product.

Why Event-Driven Architecture Matters

Traditional batch pipelines struggle with latency, inefficiencies, heavy loads, and tight coupling to source systems. Event-driven systems solve these issues by delivering:

- real-time responsiveness
- loosely coupled producer/consumer relationships
- durable logs for replay and backfill
- scalable fan-out to multiple downstream systems

An event is the purest representation of activity in a business:
“A user clicked.”
“A transaction was created.”
“A driver accepted a ride.”

Capturing these events at the source gives the entire data platform a single, authoritative narrative of what happened.

Core Idea: Producers Emit, Consumers React

The architecture revolves around three concepts:

1. Producers generate events
    - applications
    - microservices
    - mobile devices
    - IoT sensors

2. Event Streams act as the backbone

    - Kafka, Kinesis, Pub/Sub
      These systems provide durability, ordering (within partitions), and replayability.

3. Consumers subscribe and process events independently

    - streaming jobs
    - alerting systems
    - fraud engines
    - ML feature pipelines
    - data lake ingestion flows

This decoupling is the secret sauce: producers don't need to know who is reading their events, how they're processed, or at what scale.

What Makes This Pattern So Powerful

Event-Driven Architecture solves many headaches that batch pipelines create.

- No more nightly batch delays - data appears instantly.
- No shared cron schedules - events trigger their own workflows.
- Backpressure-handling built-in - streams manage load gracefully.
- No tight coupling to source databases - reduces load on OLTP systems.
- Replayability for backfills - a huge win for system design interviews.

It also enables *multiple independent consumers* to evolve at their own pace.
A single "ride_completed" event can feed:

- pricing algorithms
- ML training pipelines
- fraud detection
- financial reconciliation
- reporting dashboards

All without modifying the app.

Where Event-Driven Architecture Fits Best

EDA thrives in environments where the business is naturally activity-driven.

You should consider it when:

- the system generates continuous actions (clicks, payments, trips, ad impressions)
- you need low-latency insights
- many downstream systems depend on the same source events
- you want replayability for debugging or reprocessing
- you're building ML features that depend on fresh signals

This pattern is why Kafka became the central nervous system at companies like Uber, Netflix, Airbnb, Stripe, and Shopify.

When Event-Driven Architecture Becomes Challenging

Despite its strengths, the pattern is not free of complexity.

Some challenges include:

- maintaining schema evolution across producers
- handling event duplication
- dealing with late-arriving events
- designing idempotent consumers
- ensuring exactly-once semantics (hard in distributed systems)
- managing state explosion in streaming jobs
- preventing hot partitions when keys are skewed

These issues require strong data engineering discipline and tooling.

But with lakehouse engines (Delta, Hudi, Iceberg) and schema registries, many of these challenges are easier to manage than before.

How to Explain Event-Driven Architecture in an Interview

A clean, senior-level explanation could sound like this:

> "Event-Driven Architecture captures business activity as a continuous stream of events.
> Producers emit events, a central log like Kafka stores them durably, and downstream consumers process them independently.
> This decouples systems, enables real-time analytics, supports replay and backfills, and scales naturally for ML, fraud, and operational use cases."

This summary shows you understand both the philosophy and the practical benefits.

Why This Pattern is Foundational

Nearly every other pattern in this chapter - Lambda, Kappa, Medallion, Feature Stores, CDC, real-time ML - builds on the concept of event streams.

In a world where systems emit billions of behavioral signals per day, EDA is the only scalable way to make sense of it all.

It's not just a pattern anymore.
It's how modern data ecosystems think.

Pattern 5 - Change Data Capture (CDC) Architecture

Change Data Capture (CDC) is the pattern that keeps modern data platforms *in sync* with operational systems. Almost every company today has a transactional database powering the product - orders, payments, rides, customers, bookings - and a separate analytical ecosystem that must reflect those updates quickly and accurately.

The problem is simple: you can't hammer your OLTP database with heavy queries, and nightly batch dumps can't keep up with real-time businesses.

CDC solves this by capturing *only the changes* from the source system - inserts, updates, deletes - and streaming them into the data platform with minimal load.

Instead of asking:
"What changed?"
CDC lets the source database *tell you* what changed.

Why CDC Exists

Data teams used to rely on full table snapshots, but these don't scale:

- they overload production databases
- they create long ETL windows
- they recreate the same data repeatedly
- they miss fast-changing records
- they break when schema evolves

CDC flips the model by continuously tailing the transaction logs of the database.
This gives you:

- real-time sync
- minimal overhead
- a complete audit of every change
- replayable historical logs
- exact reconstruction of state at any point in time

CDC essentially turns any database into an event stream.

How CDC Works (Conceptually)

Although implementations vary (Debezium, Fivetran, AWS DMS), the architecture follows a consistent pattern:

1. *The Source Database Writes to Its Transaction Log*
   Every insert/update/delete is recorded by MySQL binlog, Postgres WAL, SQL Server LSN, Oracle redo logs.

2. *A CDC Connector Reads This Log Non-Intrusively*
   No scanning, no SELECT *, no polling.
   The connector reads byte-level changes and converts them into structured events.

3. *Events Flow into a Streaming System*
   Usually Kafka, Kinesis, or directly into a lakehouse.

4. *Downstream Consumers Apply These Changes*

    - upsert into analytical tables
    - maintain dimension history (SCD2)
    - feed ML feature stores
    - update search indexes
    - replicate databases asynchronously

CDC makes every change in the business immediately visible to every system that needs it.

Where CDC Fits Beautifully

CDC is the go-to solution when:

- the application DB is the source of truth
- tables update frequently (orders, users, payments, rides)
- downstream systems need near-real-time freshness
- you want to avoid full-table extracts
- slowly changing dimensions matter
- you need traceability for compliance

It is the backbone of pipelines like:

- fraud detection (monitor transaction changes)
- Customer 360 (merge identity changes)
- real-time inventory
- financial systems (auditability is non-negotiable)
- replicating OLTP → OLAP environments

If your source of truth is a transactional database, CDC is almost always the right pattern.

The Strengths of CDC

CDC brings several advantages that batch ETL simply cannot match:

- Freshness - changes appear within seconds
- Efficiency - no full-table scans, minimal DB load
- Replayability - transaction logs act like Kafka streams
- Precision - you know *exactly* what changed
- Idempotency-friendly - each change includes unique identifiers
- Transformability - easy to build SCD1/SCD2 tables downstream

It's one of the rare patterns that supports both operational and analytical worlds without burdening either.

Common Challenges with CDC

CDC isn't magic; it has sharp edges that require care.

- Schema evolution must be tracked carefully
- Deletes and hard deletes can break consumers
- Repartitioning tables can cause historical inconsistencies
- Some OLTP databases don't expose full logs publicly (SaaS platforms)
- Merge/upsert operations in the warehouse can get expensive
- Ordering guarantees exist only within a partition or key
- Long outages may require replaying massive logs

And because CDC is continuous, errors propagate quickly if validation is weak.

But once a team builds proper guardrails - schema registry, DLQ handling, idempotent consumers - CDC becomes one of the most reliable ingestion patterns available.

How to Explain CDC in an Interview

A clear, senior-level answer might be:

> "CDC captures only the changes from a source database by reading its transaction logs and streaming those changes downstream.
> It enables low-latency replication, minimizes load on OLTP systems, and supports accurate upsert-based modeling in warehouses and lakehouses.
> CDC is essential when your analytical system must mirror production state without heavy batch jobs."

That's the type of explanation that signals deep practical understanding.

Why CDC Is a Cross-Cutting Pattern

CDC isn't just an ingestion technique - it shapes downstream architecture.

- Lakehouse tables must support MERGE and SCD patterns
- Feature stores rely on change-based updates
- ML freshness depends on event-level granularity
- Auditing requires reconstructing history from logs
- Backfills reuse CDC logs to rebuild tables

Once CDC enters your ecosystem, everything downstream becomes more reactive, more incremental, and more event-driven.
It pushes the whole platform closer to real-time architecture.

Pattern 6 - Upsert Patterns in Modern Data Platforms

Upserts - the ability to insert new records and update existing ones - are at the heart of every real-world data system.
Almost every pipeline eventually faces this question:

> *"How do we keep analytical tables in sync when source systems are constantly changing?"*

In theory, this sounds simple:
Just overwrite the changed rows.

In practice, updates arrive late, out of order, duplicated, or in massive bursts. Warehouses don't like row-by-row updates, data lakes historically couldn't update files at all, and streaming jobs complicate ordering even further.

This is why upsert patterns matter - they define how a modern lakehouse or warehouse maintains correctness without sacrificing performance.

Upserts are not just a database detail; they are an architectural pattern.

Why Upserts Are Hard in Distributed Systems

A single update in an operational database (e.g., "user changed email," "order status updated," "payment refunded") becomes complicated downstream because:

- data is stored in large files (Parquet/ORC), not rows
- updates require rewriting whole files, not single records
- streaming systems see events out of order
- duplicates and retries create ambiguity
- late-arriving data can overwrite more recent values if handled incorrectly
- batch and stream paths may disagree on the final state

Without a consistent upsert strategy, you end up with broken facts, inconsistent tables, and dashboards that change every day.

Upsert patterns are the glue that holds accuracy together.

The Three Core Upsert Patterns

Although implementations vary, almost every data platform uses one of these patterns depending on scale, latency, and cost.

1. MERGE-Based Upserts (Warehouse / Lakehouse)

This is the most common pattern in warehouses and lakehouses that support ACID operations.

A MERGE statement typically does:

- if key exists → UPDATE
- if key does not exist → INSERT

Snowflake, BigQuery, Databricks Delta, and Hudi all support MERGE natively.

This pattern works beautifully when:

- the volume of changes is manageable
- latency requirements are minutes → hours
- the underlying storage engine can rewrite files efficiently

It becomes the backbone of:

- CDC pipelines
- dimension updates
- gold business aggregates
- user profile tables
- fraud transaction states

MERGE is simple conceptually, but heavy operationally:
rewriting too many files can become costly unless clustering, Z-ordering, or compaction is managed correctly.

2. Log-Based Upserts (Append + Compaction)

This pattern is used heavily in streaming-first architectures like Flink, Kafka Streams, and Hudi's MOR (Merge-on-Read) mode.

Instead of rewriting entire files for every update, the system:

- Appends a new version of the row to a log
- Maintains an index to find the latest version
- Periodically runs a compaction job to merge logs into clean Parquet files

This avoids expensive file rewrites on every small change and handles:

- high-frequency updates
- out-of-order events
- late-arriving data

It fits perfectly with Kappa-style architecture where everything is a stream.

Flink state stores and RocksDB also rely on a similar principle - the “last write wins” model with periodic cleanup.

3. Soft-Delete & Rebuild Pattern ("Immutable Log + Replay")

Some architectures avoid row-level updates entirely.

Instead, they:

- append a record describing the change
- mark the old record as invalid (soft delete)
- rebuild downstream tables periodically from scratch or via compaction

This pattern works when:

- updates are frequent
- full-file rewrites are unavoidable anyway
- historical accuracy matters
- the platform embraces immutability

Many large-scale identity-resolution and machine-learning feature pipelines follow this approach because it preserves lineage and reduces update conflicts.

It’s not the fastest, but it’s the cleanest for audit-heavy systems.

Choosing the Right Upsert Pattern

The choice depends on your requirements:

- *High throughput* + *streaming* → Log-Based Upserts
- *Analytical correctness* + *SQL simplicity* → MERGE Upserts
- *Strong lineage* + *periodic rebuilds* → Append + Soft Delete

Modern lakehouses (Delta, Iceberg, Hudi) blur the lines and support all three patterns depending on configuration.

A senior engineer doesn’t just say “we’ll MERGE.”
They explain *why* that model fits the workload.

Common Pitfalls With Upserts

Even though upserts appear straightforward, they introduce subtle problems:

- outdated data overwriting newer values
- key collisions caused by upstream bugs
- excessive file rewrites leading to high cloud costs
- small-file fragmentation reducing query performance
- compaction jobs running too slowly
- streaming systems losing ordering guarantees
- SCD logic implemented inconsistently across pipelines

The danger with upserts is not the SQL; it's the lack of discipline around them.

Great data teams implement:

- deterministic keys
- event timestamps
- idempotent logic
- schema versioning
- reproducible compaction models

This prevents "pipeline drift" where tables degrade silently over months.

How to Explain Upsert Patterns in an Interview

A strong, concise explanation might sound like this:

> "Upserts define how downstream tables stay in sync with source systems as records change.
> In warehouses, we use MERGE statements for accuracy.
> In streaming architectures, we use append-only logs with compaction to handle high throughput and out-of-order events.
> And in audit-heavy systems, we preserve history with soft deletes and periodic rebuilds.
> The pattern depends on latency needs, data volume, and storage engine capabilities."

This explanation sounds senior and demonstrates architectural awareness rather than tool knowledge.

Why Upsert Patterns Are Considered Cross-Cutting

Upserts aren't limited to one type of system.
They shape almost every domain:

- CDC pipelines depend on them
- Feature stores rely on them for freshness
- Gold-layer aggregates require them for accuracy
- Fraud engines apply state transitions via upsert
- SCD patterns are built on upsert semantics
- Lakehouse ACID engines exist to support them efficiently

Upsert patterns are the connective tissue of modern data architecture.
If you get them right, everything downstream becomes stable and predictable.
If you get them wrong, no amount of dashboards or ML modeling can save you.

Performance & Scaling Internals in Streaming Data Systems

Most system design discussions stop at the boxes and arrows:
“Kinesis here, Kafka there, Spark/Flink in the middle, warehouse at the end.”

That’s good enough for a junior engineer.

But once you start operating real systems - or interviewing for senior roles - people expect you to understand what happens when the system is under load:

- What if traffic spikes 10x?
- Why is one consumer lagging behind even though CPU is low?
- Why are some partitions red-hot while others are idle?
- Why did a simple “exactly-once” config double our latency?

These are no longer questions about tools.
They are questions about *throughput, backpressure, partitions, state,* and *skew*.

This section gives you the mental model to talk about scaling like a production engineer, not just a diagram artist.

Kafka / Kinesis Partitioning and Consumer Parallelism

A lot of streaming performance comes down to one simple relationship:

> Parallelism is limited by partitions.

For systems like Kafka or Kinesis:

- A partition is the unit of ordering and parallelism.
- A consumer in a consumer group can only read from a partition in order.
- If you have 12 partitions and 20 consumers, only 12 will actually be doing work.
- If you have 3 partitions and 20 consumers, you are throttled by 3.

So in an interview, when you talk about scaling a streaming system, it’s not enough to say “we’ll add more consumers.”
You should show that you understand:

- We scale throughput primarily by increasing partitions.
- The keying strategy (e.g., `user_id`, `merchant_id`, `region`) controls how evenly load is distributed.
- Too few partitions → not enough parallelism.
- Too many tiny partitions → overhead, management pain, and sometimes worse performance.

A senior answer sounds like:

> "We'd ensure enough Kafka partitions to match or exceed the number of consumer tasks, and we'd choose a partition key that avoids hot keys, for example `user_region + user_id` instead of just `user_id` in a highly concentrated market."

Backpressure: What Happens When Consumers Can't Keep Up

Backpressure is what happens when *data arrives faster than it can be processed.*

Different frameworks react differently, but the idea is the same:
something in the pipeline starts saying *"slow down."*

Typical symptoms:

- Kafka consumer lag grows steadily.
- Micro-batches take longer than the trigger interval.
- Checkpoint times keep increasing.
- Latency creeps from seconds to minutes.

Streaming engines like Spark Structured Streaming and Flink handle this by:

- reducing the rate at which they pull from Kafka/Kinesis,
- buffering more data in memory / state stores,
- sometimes triggering autoscaling if configured.

In a design interview, you don't need to go into framework internals, but it's powerful to say:

> "If the downstream Flink job can't keep up, Kafka lag will grow. I'll monitor lag, and either scale out the number of task slots / executors or increase partitions so that we can parallelize consumption."

That shows awareness of flow control, not just ingestion.

Data Skew and Hot Partitions

Skew happens when:

- too many events share the same key
- a few partitions get 80–90% of the traffic
- a small subset of users/merchants/content dominate the workload

Even if you have 100 partitions, if one “super user” or “super merchant” gets assigned to a single partition, that partition becomes a bottleneck. One task is overloaded while the rest are idle.

Common mitigation strategies:

- Composite keys: e.g., `merchant_id + random_suffix` to spread hot merchants across multiple partitions.
- Key salting: append a small random number to the key to distribute load.
- Special handling for whales: route “top N” heavy keys to separate pipelines or dedicated workers.
- Repartitioning in the stream: use a `repartition` / `keyBy` with a better distribution for downstream stages.

In an interview, mentioning skew and how you’d handle it is a big “this person has seen production” signal.

Parallelism vs Executors vs Tasks

In distributed processing you often have three knobs, and people confuse them:

- *Partitions (or shards)* → from Kafka/Kinesis, define max possible parallelism
- *Tasks / operators* → the number of parallel processing units in Flink/Spark
- *Executors / instances* → actual machines/containers running those tasks

When you say “we’ll scale the streaming job,” you should mentally check:

- Do we have enough partitions to feed extra tasks?
- Are we increasing task parallelism in the job config?
- Does the cluster have enough executors / pods / instances to host those tasks?

It’s not enough to throw “autoscaling” at the problem if partitions still limit throughput.

State Stores and How They Grow

Once you introduce:

- joins,
- aggregations over windows, or
- per-key state (fraud risk, last seen location, rolling averages),

your pipeline starts storing state.

That state typically lives in:

- in-memory structures, backed by
- RocksDB, local disk, or
- an external key-value store like Redis/DynamoDB.

If you don't think about state growth, you eventually hit:

- high checkpoint times,
- GC pressure,
- disk pressure,
- and in the worst case, job restarts.

You can show maturity by mentioning:

- TTL (time to live) on state - clear state for keys that haven't appeared in X days.
- Windowed state - only keep what's needed for the window, not all time.
- Sharding state across more task slots by increasing parallelism.
- Externalizing some state to DynamoDB/Redis when it's too large for local stores.

This is especially relevant in systems like fraud detection, recommendation engines, and sessionization.

Checkpointing and Exactly-Once Semantics

Exactly-once often sounds magical in marketing, but architecturally it has a price.

- Checkpoints take snapshots of state and offsets.
- The more state you have, the heavier the checkpoint.
- The more frequent they are, the more overhead.
- Remote checkpoint stores (S3, HDFS) add network and I/O cost.

So while "exactly-once" is attractive, a senior engineer will say:

> "For some parts, at-least-once with idempotent sinks is enough; we'll only pay for exactly-once where correctness is critical."

You can mention tuning:

- checkpoint frequency,
- incremental checkpoints (if supported),
- and balancing strict guarantees vs latency.

Scaling Rules Engines, State Stores, and ML Inference

As pipelines mature, the bottleneck often moves away from raw ingestion to business logic:

- complex fraud rules
- heavy joins
- ML inference calls
- feature computation

Scaling these parts usually involves:

- Isolating heavy logic: separate stream for fraud vs general analytics.
- Batching inference requests: scoring 100 events per call instead of 1.
- Caching recent results: avoid hitting the model for repeated queries.
- Precomputing hot features: push some logic to upstream or to scheduled batch jobs.

Mentioning this in a design interview shows you understand that scaling is not just about adding more CPU; it's about restructuring where and how the work is done.

Autoscaling: Useful but Not Magic

Autoscaling is often pitched as a silver bullet, but it only works properly if:

- your metrics reflect real pressure (lag, processing time, CPU, memory),
- your cooldown periods avoid flapping,
- and your min/max bounds are realistic.

Good talking points:

- Scale streaming consumers based on Kafka lag and processing time.
- Scale ML inference services based on QPS (queries per second) and p95 latency.
- Set upper limits to avoid surprise cloud bills.
- Remember that scaling out a job won't help if partitions are still the bottleneck.

In other words: autoscaling is a tool, not an excuse to avoid capacity planning.

Section III : Data Contracts - A Detailed, Industry-Realistic Deep Dive

In high-growth companies, data breaks for the same reason software breaks:
too many teams move too quickly, and everyone assumes someone else is handling stability.

A backend engineer renames a field because it “felt cleaner,” a microservice team migrates to a new framework that serializes timestamps differently, a partner API silently changes nullability rules - and suddenly an entire analytics organization is scrambling to fix dashboards and pipelines.

Each team did the “right thing” locally, but the ecosystem collapsed globally.

This is the world that gave birth to ***data contracts***.
They aren’t just technical documents - they are agreements that create social, operational, and architectural boundaries across fast-moving teams.

How Real Companies Discover They Need Data Contracts

If you talk to engineers at Uber, Airbnb, Shopify, or Doordash, the story is always the same:

> “We scaled our event systems faster than we scaled our governance.”

At first, events are free-form JSON blobs shipped by one team.
Then the company grows, and suddenly:

- 20 services write to the same event stream
- 15 teams consume that stream differently
- 5 ML models depend on subtle field semantics
- BI definitions drift because fields mean different things per department

By the time the company hits a billion events per day, a single schema change can cost millions in failed jobs, fraud false positives, customer reimbursements, or just wasted engineer-hours.

Contracts emerge as a survival mechanism.

What a Contract Actually Represents Internally

In real systems, a contract is not just a schema.
It describes three things:

1. Structure - the fields, types, nullability, nesting
2. Semantics - what the field *means*, not just what it looks like
3. Expectations - freshness, completeness, ordering guarantees

For example, a field named `payment_status` is more than an enum.
Finance depends on it to determine revenue recognition.
Fraud models depend on it to track transaction lifecycle.
Customer support uses it to resolve disputes.

A contract ensures that if this field changes meaning, 20 teams are not blindsided simultaneously.

The Real Lifecycle of Data Contracts

Let's walk through how a contract actually flows at a company like Uber or DoorDash.

1. Definition (Producer Side)

The service team defines the schema - often in Protobuf or Avro - *before* writing code.
Instead of hand-writing DTOs, the schema generates:

- producer-side classes
- validation logic
- serialization code

Engineers are not allowed to bypass this generation step.

A contract review often happens at the pull-request level, where data platform engineers comment on semantics:

> "Is `trip_dropoff_lat` always expected to be non-null?"
> "Will the meaning of `order_state` remain consistent across regions?"
> "Should timestamp fields be seconds, milliseconds, or RFC3339?"

The goal is not to block development but to prevent ambiguity from leaking downstream.

2. Publishing to Schema Registry (The Gatekeeper)

Once approved, the schema is published to a Schema Registry with strict compatibility rules.

At companies with strong governance (e.g., Shopify), the registry rejects incompatible schemas automatically.

A producer cannot publish new events if the schema breaks backward compatibility without creating a new version.

This shifts responsibility left:
producers see the error before data corrupts the platform.

3. Ingestion Layer Validation

Even if schemas are compatible, real data may violate contracts.

Common real-world issues:

- A field marked as “non-nullable” comes as null during a partial outage
- A service deploys early and writes a field before consumers are ready
- A sudden spike in invalid currency codes appears due to vendor-side bugs

Bronze ingestion jobs validate all of this.
Any breach routes events to DLQs with metadata:

- contract version
- violating field
- expected vs actual type
- producer service ID
- deployment hash

This makes debugging contract violations a *traceable, measurable process*, not tribal knowledge.

4. Downstream Enforcement in Silver/Gold

This is where contracts integrate with dbt, Great Expectations, Soda, etc.

Consumer teams embed business rules that rely on the contract:

- `order_id` must be globally unique
- `price` must be ≥ 0
- `order_state` must belong to a known lifecycle
- `event_timestamp` must be within 24h of ingestion

When a contract guarantees something, downstream tests stop being defensive; they become assertive.

That's how companies maintain high trust in their Silver layer.

How Breaking Changes Are Managed in Reality

In theory, breaking changes involve ceremonies.
In practice, they involve:

- Slack channels
- Jira tickets
- Change windows
- Deprecation timelines
- Dual-writing periods
- Migration playbooks
- A data PM or architect mediating between teams

For example:

A product team wants to rename `user_city` to `city_name`.
If it were an API, they'd version it.
For data, they must:

- dual-write both fields for one release
- communicate the plan ahead of time
- coordinate with ML teams (feature stores often break on missing fields)
- coordinate with BI teams (semantic layers may need remapping)
- ensure warehouses/lakes handle the transition gracefully

This is the invisible work of scaling data.

Without contracts, downstream teams only learn about the change when their pipeline fails.

How Contracts Prevent Real Incidents

Examples from industry war rooms:

Payment Systems

A backend update changed `amount` from integer cents to float dollars.
Without a contract, dashboards doubled revenue overnight.

Ride-Sharing

A new driver app release sent empty `route_polyline` fields for 6 hours.
Pricing models miscalculated detours, triggering driver overpayments.

E-commerce

A vendor started sending null `currency` values for international orders.
Currency conversion pipelines silently defaulted to USD, skewing revenue.

Contracts do not eliminate failure, but they contain it.
They turn silent corruption into visible violations.

Contract-First Design Is a Strategic Advantage

Companies that adopt contract-first data culture gain several benefits:

- upstream teams think about downstream impact
- platform teams catch changes early
- analysts stop firefighting schema drift
- ML systems become more stable
- governance becomes simpler
- time-to-debug shrinks from days to minutes

In interviews, saying “contract-first design” shows you operate like a senior engineer who thinks structurally rather than reactively.

How To Talk About This in a System Design Interview

A strong answer ties together the technical and cultural elements:

> “Data contracts define the structure, semantics, and evolution rules of events and tables.
> They flow from the API layer into Kafka via Schema Registry, and downstream into Bronze/Silver/Gold.
> They prevent breaking changes by enforcing compatibility, validating events at ingestion, and aligning teams during schema evolution.
> In large companies, contracts aren’t just schemas-they are governance mechanisms that keep dozens of teams aligned as systems evolve.”

This sounds like someone who has actually lived in a production environment.

Section IV : Data Orchestration Patterns - A Deep, Practical Chapter

In every real data platform, the “architecture diagram” tells only half the story.
The other half lives in the orchestration layer - the quiet machinery responsible for making things run *in the right order, at the right time, with the right guarantees*.

People often underestimate orchestration because it looks simple on paper:
draw a few arrows, set a few schedules, add a few retries.

But in production, orchestration becomes the *nervous system* of the entire platform.
It is where dependencies choke throughput, where backfills turn into multi-hour war rooms, where SLAs are missed, and where every subtle design flaw gets exposed under load.

This chapter is about the patterns that mature data teams rely on to keep pipelines predictable, debuggable, and resilient - whether they run on Airflow, Dagster, Prefect, Step Functions, or a custom orchestrator.

The Myth of “Just Use Airflow”

In many companies, orchestration starts with one simple Airflow DAG.
Then another.
Then fifty more.
Before long, the DAGs form a tangled web of dependencies that no one fully understands.

The anti-pattern is always the same:

- too many tasks
- too many cross-DAG triggers
- too many implicit dependencies
- too many retries masking real issues
- no distinction between critical paths and side pipelines

The orchestration layer becomes a fragile central brain - one bad task and the whole thing collapses.

This is why orchestration needs its own design patterns, not just an Airflow tutorial.

The Core Orchestration Patterns

Different companies converge on the same fundamental orchestration flows because real systems force them there.

1. Fan-In / Fan-Out Dependencies

This is the backbone of how data actually moves.

- A fan-out stage takes one raw dataset and branches it into many downstream transformations.

- A fan-in stage collects outputs from multiple upstream tasks before moving forward.

The challenge is not drawing the shape - it's designing it safely.

For example, a fan-in step waiting on 12 upstream datasets means:

- the slowest dataset controls your latency,
- one badly formatted file blocks an entire dashboard,
- testing becomes harder because partial DAG runs don't reflect real conditions.

Senior engineers often break fan-in into multiple layers or add checksums to ensure upstream tables are complete before joining them.

This is orchestration as *coordination*, not scheduling.

2. Idempotent Task Design - The Golden Rule

Every orchestrator eventually fails.
Every pipeline will be retried.
Every backfill will re-run tasks from days or months ago.

This is why idempotency is non-negotiable.

An idempotent task always produces the same result given the same input, no matter how many times it runs.
It doesn't append duplicate rows, doesn't break ordering, doesn't corrupt state.
It treats retries as normal behavior, not exceptional behavior.

In mature companies, platform teams often embed idempotency into the orchestration contract:

- tasks MUST write to temporary locations
- publish results only after success
- generate outputs that can safely overwrite prior versions

If a system design candidate mentions idempotency, interviewers instantly know they “get it.”

3. Event-Driven vs Schedule-Driven Orchestration

Two very different models exist.

Schedule-driven orchestration (like a daily warehouse load) is predictable and simple.
But it also introduces:

- unnecessary delays,
- wasted compute,
- needless coupling to cron-driven timing,
- and heavy backfills when things go wrong.

Event-driven orchestration triggers pipelines based on actual business activity:

- a file lands in S3
- a Kafka topic crosses a threshold
- a CDC stream emits a new batch
- a completion signal from an upstream service arrives

This reduces latency and compute cost, but increases coordination complexity.

Great platforms use both:

- *Schedules* for batch deadlines and SLA-dependent tasks.
- *Events* for freshness-critical, real-time sensitive flows.

A hybrid orchestrator (like Dagster or Step Functions) handles this beautifully.

4. Backfills - The Real Test of Your Orchestration Design

A pipeline that works today is easy.
A pipeline that can reprocess last year’s data is engineering.

The moment something needs to be backfilled - logic bug, schema change, historical correction - orchestration becomes the bottleneck.

Patterns you must account for:

- being able to run only a subset of the DAG
- running tasks parameterized by date ranges
- isolating backfill output from production output
- detecting and merging duplicates safely
- avoiding flooding the warehouse with 10,000 runs at once

Most companies have horror stories of Airflow clusters collapsing during backfills because tasks were not built for high-volume replay.

Mature systems treat backfills as *first-class operations*, not exceptions.

5. Retry and Failure Semantics

Retries are deceptively simple:
retry on failure, right?

In reality, retry semantics differ depending on the failure type:

- transient network issues
- data not yet available
- partial writes
- upstream system outages
- internal data corruption

In financial or healthcare systems, retries may also violate compliance rules if they trigger external writes twice.

This is why careful retry policies matter:

- exponential backoff,
- max retry limits,
- fallback to DLQ or manual review,
- marking upstream steps as “incomplete” instead of failed.

Retries are not blindly applied; they reflect business cost.

6. SLAs, SLOs, and SLTs for Data Pipelines

Once companies scale, orchestration is no longer a technical artifact - it becomes a business contract.

- *SLA (Service Level Agreement):*
  The *promise* to the business ("daily revenue dashboard ready by 7 AM").

- *SLO (Service Level Objective):*
  The *target* the pipeline aims for ("99% of the time before 7:00 AM").

- *SLT (Service Level Trigger):*
  The *alert level* that notifies engineers ("if lag > 30 minutes, wake someone up").

Pipelines must encode these expectations:

- critical path marking
- deadline-aware scheduling
- alert routing
- prioritization queues

This is the difference between a hobby pipeline and a production-grade orchestration system.

7. Orchestrating Hybrid Batch + Streaming Systems

This is where most senior engineers struggle.
A system may use:

- Streaming for real-time features and freshness
- Batch for accuracy, reprocessing, and financial reports

The orchestrator's job is to keep these worlds consistent.

Patterns include:

- micro-batch checkpoints that align with warehouse batch windows
- workflows that trigger streaming backfills before batch merges
- end-of-day cutoffs that coordinate reconciliation jobs
- event signals that notify batch pipelines when streams have caught up

You want to show in interviews that you understand the orchestration of hybrid ecosystems is not separate - it is integrated.

How to Summarize Orchestration in an Interview

A polished, senior-sounding statement would be:

> “Orchestration isn’t just about scheduling.
> It’s about dependency design, idempotent task execution, handling backfills safely, managing retries intelligently, and coordinating both batch and streaming workloads while meeting SLAs.
> Tools like Airflow and Dagster help, but the real orchestration complexity comes from designing workflows that behave well under failures, reprocessing, and large-scale growth.”

This is the kind of statement that convinces interviewers you’ve lived through real production issues.

Section V : Cost Architecture & Optimization Principles

When companies operate at small scale, cost feels like an afterthought.
A few extra compute nodes here… a larger warehouse cluster there… an inefficient join that scans a terabyte each day - it all seems harmless.

But at FAANG scale, everything becomes a cost problem:

- A daily job scanning 2 TB becomes 60 TB per month
- A warehouse cluster running 24/7 becomes a million-dollar line item
- A streaming pipeline with 50 shards becomes 500 shards after a product launch
- A single poorly partitioned table leads to 80% wasted compute
- A simple “retry storm” costs more in one night than an engineer’s salary

This is why senior engineers are obsessed with cost architecture.
It’s not “optimization” - it’s responsible engineering.

And every system design interviewer will eventually ask:

> *“How does this solution scale economically?”*

So this chapter is about the principles that keep pipelines fast *and* financially healthy.

1. Hot vs Cold Data - The First (and Most Important) Cost Decision

Every large-scale data platform eventually discovers the same truth:

> *Most data is stored as if it's valuable, but queried as if it's disposable.*

In almost every company - from Netflix to Stripe to Uber - only a very small percentage of datasets are queried frequently. Yet, without deliberate design, organizations end up keeping everything in the most expensive storage layers:
high-performance warehouse tables, low-latency indexes, and fast compute clusters.

This is where the idea of tiered data architecture becomes the foundation for cost efficiency.

It recognizes a simple pattern of human behavior: data is hottest right after it is created and becomes increasingly irrelevant with time.

A tiered system reflects this natural decay.

Hot Tier: Fast, Expensive, Query-Optimized Data

Hot storage is where performance matters more than cost:

- most recent transactions
- dashboard queries that run every morning
- ML features requiring fresh inputs
- event streams for fraud detection or personalization
- high-frequency metrics (clicks, impressions, orders, rides)

This tier usually lives in:

- Snowflake (highest cost per TB)
- Databricks Delta (optimized + cached)
- BigQuery hot storage
- DynamoDB/Redis for fast operational lookups

Hot tier data is:

- quickly accessible
- optimized with clustering, Z-ordering, stats
- cached in memory
- often replicated for higher availability

But it's expensive - and letting old data accumulate here is the fastest way to double your cloud bill.

Warm Tier: “Good Enough” Storage for Medium-Frequency Analytics

Most companies have a middle layer - not blazing fast but not archival.

Warm tiers include:

- Parquet/ORC on S3 or ADLS with basic partitioning
- Compact Delta tables (larger files, fewer partitions)
- Warehouses configured with reduced compute
- Medium retention streaming topics

Warm-tier data is ideal for:

- monthly or quarterly reporting
- historical ML feature generation
- backfills
- data quality investigations
- internal audits
- slow dashboards

These datasets don't need millisecond performance.
They need cheap, reliable access.

Cold Tier: The Long-Term, Low-Cost Deep Archive

This is where data goes to retire.
Not deleted - just moved to the cheapest possible form:

- S3 Glacier
- GCS Archive
- Azure Archive
- compressed Parquet/Avro bundles
- event snapshots packed by week or month

Cold tier data is:

- rarely queried
- stored at 10–50× cheaper cost
- potentially slower to retrieve (minutes to hours)
- kept for compliance, audits, or rare investigations

FAANG companies use cold tiers aggressively:

- Netflix stores raw viewing logs in extremely cheap archive formats
- Uber stores historical trip records in monthly Parquet bundles
- Meta dumps certain ad impression logs into Glacier after aggregation
- Financial institutions compress and archive raw ledger data by quarter

This strategy reduces massive recurring cost without sacrificing traceability.

Real Industry Lifecycle Patterns

Here's how a real company lifecycles data - not "theoretical patterns," but what teams such as Netflix Data Platform actually do:

Hot (0–7 days)

Kept in the most expensive tier:
Delta tables, Snowflake, BI semantic layers.
Why? Because freshness and low-latency matter most here.

Warm (7–90 days)

Data is still relevant but no longer queried constantly.
Teams run daily or weekly compaction jobs to merge small files, rebuild stats, and reduce file counts.
This shrinks compute cost dramatically.

Cold (90 days → 1 year → multi-year)

Data becomes "inactive."
So companies:

- re-partition by month instead of day
- compress files aggressively
- move them to S3/GCS archive tiers
- drop secondary indexes
- store only essential summaries in the warehouse

Deep Archive (multi-year)

Cold data goes even deeper:

- fully frozen S3 Glacier Deep Archive
- no direct querying
- only retrieved during audits or regulatory requests
- retrieval may take hours (acceptable because it's rare)

This lifecycle alone often reduces storage + compute + query cost by 40–70%.

Why This Matters in System Design Interviews

Most candidates talk about compute optimization, parallelism, partitioning…
Few talk about the economics of data retention.

Interviewers especially at FAANG love when a candidate says:

> "Not all data deserves to stay hot.
> We'll keep only the recent N days hot, compact older data into warm tier Parquet, and then bucket and archive the rest to cold storage with reduced indexing.
> This reduces warehouse and compute costs dramatically without losing analytical capability."

This shows:

- architectural maturity
- cost awareness
- operational realism
- understanding of how enterprises manage petabytes

It's the difference between "I can build pipelines" and "I can run a platform."

2. Compaction & File Size Optimization - The Hidden Engine of Cost Efficiency

In every large data platform, small files are the silent killers.

They don't look dangerous; after all, a 5 MB Parquet file and a 500 MB Parquet file both "work." But under the hood, they behave completely differently.

One triggers thousands of tiny I/O operations, repeated metadata reads, excessive partition scanning, and expensive shuffles.
The other gives you a predictable, efficient, well-behaved data workload.

This is why companies like Databricks, Netflix, Airbnb, Stripe, and Shopify treat file size optimization as a first-class architectural responsibility, not an afterthought.

If you ask any experienced data platform engineer:
"What single fix cut your compute bill the most?"
They will say some version of:

> *"Compaction."*

Why Small Files Are So Expensive

A compute engine doesn't just "scan data."
It scans *files*.
For every file, it must:

- open a handle
- read the metadata
- materialize row groups
- schedule a task
- often shuffle partial results

This overhead is constant, regardless of file size.

So 10,000 small files create:

- 10,000 task schedulings
- 10,000 metadata reads
- 10,000 I/O fetch operations

Your job becomes compute-bound not because of size but because of file count.

This is why a dataset with 2 TB in 5,000 files performs better than a dataset with 500 GB in 500,000 files.

The second one is vastly more expensive and slower - despite being 75% smaller.

This is the paradox of small files.

Where Small Files Come From (Real-Life Causes)

Small files are not created intentionally.
They accumulate as a side-effect of how data arrives and is processed:

- micro-batch streaming (Spark Structured Streaming creates tiny checkpoints)
- CDC ingestion (thousands of incremental batches)
- concurrent writers producing independent partitions
- event streams flushed every minute
- overwrite jobs producing fragmented output
- backfills writing thousands of partitions
- upsert operations rewriting scattered portions of the table

Over weeks or months, a healthy table turns into a fragmented mess.

What Compaction Actually Does

Compaction is simply rewriting many small files into fewer, larger, well-optimized files.

But the value lies in *how* it does it:

- merges files within the same partition
- rebuilds statistics for pruning
- reorders data for clustering (Z-order in Databricks, clustering keys in Snowflake)
- removes tombstones or delete markers (especially for Delta Lake, Hudi, Iceberg)
- refreshes metadata so scans skip entire chunks of data
- rebalances partitions so one does not contain 800 files while another contains 12

A well-compacted partition typically has:

- file sizes between 128 MB and 1 GB
- evenly distributed row groups
- correct compression across row groups
- clean column statistics
- minimized number of files per partition

This makes everything downstream cheaper.

The Hidden Benefits of Compaction (Beyond Cost)

People think compaction is only about performance.
In reality, it improves *quality, reliability, and durability*.

When files are compact:

1. Query engines prune better

Statistics (min/max values) are more meaningful when they span large, coherent data ranges.

2. Cloud warehouses reduce scan cost

Snowflake, Databricks, Athena, BigQuery - all charge by data scanned.
Compacted files reduce scanning drastically.

3. Shuffles are smaller

Data is more evenly distributed across partitions.

4. Caching works properly

A single large file is easier to cache than 600 tiny ones.

5. Metadata overhead shrinks

Delta, Hudi, Iceberg rely heavily on metadata.
Small files explode metadata size, slowing down schema reads and time-travel operations.

6. Streaming jobs stabilize

Streaming sinks create thousands of small files; periodic compaction prevents runaway growth.

Industry-Standard Compaction Strategies

Different companies use different patterns, but they converge on a few common rhythms:

Daily Micro-Compaction

Merge files generated in the last 24 hours.
Used by streaming-intensive platforms like Uber or DoorDash.

Weekly Partition Compaction

Rebalance partitions for the previous week.
Great for systems with hourly ingestion.

Rolling Compaction

Continuously monitor partitions and compact based on rules:

- file count threshold
- median file size
- partition skew
- time since last compaction

This is common in Iceberg and Delta Lake deployments.

Adaptive Compaction (Databricks AUTOTUNE)

Databricks runtime automatically compacts files based on table health.
You simply enable it; the engine handles the rest.

Compaction + Z-Ordering

After compaction, Databricks teams often run Z-order to cluster data by high-cardinality fields (e.g., `user_id`, `trip_id`, `merchant_id`).

This can improve selective query performance by 3× to 20×.

Real-World Example: How Compaction Saves Money

Stripe engineers shared that one of their major cost optimizations came from:

- compacting Parquet files to ~512 MB
- reducing file count by 90%
- improving scan pruning by 40%

Result:

- 60% reduction in Snowflake compute
- 45% reduction in Databricks cluster time
- 70% faster ML feature lookups

No code change. Just smarter file layout.

Netflix also reported massive savings by aggressively managing file size and partition balance in their big data platform.

Compaction in System Design Interviews

If you really want to impress an interviewer, say something like:

> "The ingestion layer will generate small files through micro-batching.
> To keep compute low, we'll run daily compaction to merge them into 256–512 MB Parquet files, refreshing statistics and pruning indexes.
> This dramatically reduces scan cost, improves shuffle performance, and avoids the small-file problem that often cripples large Delta/Iceberg tables."

This shows:

- deep understanding of storage internals
- awareness of hidden cost drivers
- practical experience with real production systems
- ability to keep a platform stable at scale

Most candidates never mention compaction - so when you do, it stands out immediately.

3. Spot vs On-Demand Compute - The Economics of Reliability

When data platforms begin to scale, compute becomes the largest cost driver.
Not storage.
Not data transfer.
Compute.

And the largest mistake junior teams make is treating all compute the same - as if every workload deserves the same reliability, the same uptime guarantees, and the same price tag.

In reality, cloud compute pricing is a market, and just like all markets, the best strategy isn't to buy everything at retail.
It's to understand risk vs reliability and use the right pricing model for each kind of workload.

This is why mature engineering organizations - Netflix, Lyft, Stripe, Shopify - build compute strategies around two fundamental options:

- On-Demand instances → stable, predictable, always available
- Spot instances → up to 90% cheaper, but can disappear at any moment

A cost-efficient architecture is not one or the other - it's a deliberate combination of both.

What Spot Instances Really Represent

Spot instances are cloud providers selling their *unused capacity*.
Since capacity fluctuates constantly, spot instances come at a steep discount:

- AWS Spot: 70–90% cheaper
- GCP Preemptibles: ~60–80% cheaper
- Azure Spot VMs: often 70–90% cheaper

But here's the catch:
they can be reclaimed at any time with little notice.

To a batch job, that's an inconvenience.
To a streaming job, it might be fatal.
To a mission-critical job, it's unacceptable.

This is why real companies don't treat spot vs on-demand as a binary.
They treat it as a risk budget.

How Mature Companies Use Spot Compute (The Real Playbook)

In every well-run data platform, workloads fall into categories:

Fault-tolerant, Non-urgent Workloads → Mostly Spot

- backfills
- large-scale ETL transformations
- historical ML feature re-computation
- periodic compaction jobs
- ad-hoc analytics
- training ML models
- experiments, prototypes

If the job fails and restarts, nothing terrible happens.
These jobs often run 70–100% on spot.

Netflix's internal data platform routinely runs massive Spark jobs on spot clusters with graceful failover.
Airbnb, Stripe, and Lyft do the same.

Spot failures aren't catastrophic - they're expected.

Low-Latency, Customer-Facing, or Financially Sensitive Workloads → Mostly On-Demand

These workloads cannot be interrupted:

- real-time pricing engines
- fraud-scoring pipelines
- event ingestion consumers
- streaming join operators
- metadata/catalog services
- OLTP-proximal workloads
- Airflow/Dagster schedulers and control planes
- production ML model inference

If these stop, the business stops.

For these, on-demand instances remain the backbone.
Spot may supplement, but never replace.

Mixed Workloads → Baseline On-Demand + Burst Spot

This is the sweet spot for most data engineering systems.

Example:
Your streaming job requires 5 nodes to keep up with traffic at baseline, but needs 20 nodes during peak traffic.

The industry pattern is:

- 5 on-demand nodes → guaranteed availability
- 15 spot nodes → scale when capacity is cheap

If spot capacity disappears, the job slows down but doesn't collapse.

This hybrid strategy delivers the best price/performance ratio in almost every large organization.

Spot Failures Are a Feature, Not a Bug

A surprising truth:
Spot reclaimations pressure engineers to build more resilient pipelines.

They expose:

- brittle retry logic
- stateful operators that can't recover
- poorly designed checkpoints
- tasks that write partially and corrupt outputs
- streaming workers that can't resume from offsets
- compaction jobs that assume uninterrupted execution

Because spot failures are random, they act as natural chaos testing.
Teams that survive on spot instances end up with stronger systems, not weaker ones.

This is one reason Netflix famously embraced spot early - it forced robustness.

How Spot vs On-Demand Decisions Are Made in Architecture Reviews

In FAANG-level design reviews, engineers always ask:

> "What happens when this workload gets reclaimed?"
> "Can it be restarted safely?"
> "Does it write atomically?"
> "How is progress stored?"
> "Does this job have an SLA?"
> "Does it hold state? Can that state be recreated?"

These questions determine whether a workload is spot-safe.

A great answer sounds like this:

> "The pipeline writes outputs atomically, checkpoints progress every 5 minutes, and can resume from object storage.
> Failures don't cause data loss, so we'll run it 80% on spot and keep a minimal on-demand baseline to meet SLAs."

This demonstrates cost awareness *and* architectural maturity.

The Hidden Savings of Spot Compute

Companies often save millions annually by shifting the right workloads to spot.

Real examples:

- Airbnb: moved large-scale Spark workloads from on-demand → spot, saving ~60% compute cost

- Lyft: re-engineered ETL pipelines to be spot-reliable, reducing nightly batch compute cost by 70%
- Netflix: built their entire video-encoding pipeline on spot fleets
- Stripe: uses spot for model retraining and data compaction
- Databricks customers**: routinely cut cluster cost by 50–80% using spot workers

This isn't theoretical - this is industry reality.

The Senior-Level Explanation for Interviews

If an interviewer asks about compute cost, you can respond like this:

> "We'll run fault-tolerant workloads - ETL, backfills, compaction, ML training - primarily on spot instances because they tolerate interruptions.
> For latency-sensitive or stateful workloads, we'll keep a minimum on-demand baseline and use spot nodes only for horizontal scaling.
> This hybrid strategy combines high reliability with 60–80% cost savings."

4. Serverless vs Cluster-Based Compute - Choosing the Right Economic Model

One of the first questions any mature data team must answer is deceptively simple:

> *"Do we want the cloud to manage compute for us, or do we want to manage it ourselves?"*

The answer is never binary.
The real world forces nuance.

Serverless systems promise simplicity: no provisioning, no cluster babysitting, instant scale.
Cluster-based systems promise control: stable performance, predictable cost, long-running stateful workloads.

Both approaches are powerful. Both are flawed.
And senior engineers know the value is not in picking one - but in knowing *when* each is appropriate.

This is why companies like Netflix, Stripe, DoorDash, Shopify, and Instacart often run hybrid architectures with both serverless jobs and persistent compute clusters co-existing in the same pipeline.

Let's explore how real engineering orgs make this decision.

What Serverless Is Really Good At

Serverless shines when workloads are:

- bursty (e.g., occasional heavy load, then silence)
- unpredictable (e.g., streaming bursts, event triggers)
- short-lived (executions measured in minutes, not hours)
- embarrassingly parallel (thousands of small tasks)
- orchestration-driven (trigger → compute → finish)

Examples:

- AWS Lambda for file ingestion
- Glue serverless for nightly ELT jobs
- BigQuery on-demand SQL
- Snowflake virtual warehouses that auto-suspend
- Databricks Serverless SQL
- Cloud Functions triggering downstream ETL

In these situations, serverless lets you:

- pay only for what you use
- avoid cluster management
- eliminate idle compute
- auto-scale to huge concurrency
- deploy faster with fewer infra decisions

This is why small teams and early-stage systems gravitate toward serverless.

But senior engineers know serverless has sharp edges.

The Hidden Costs of Serverless

Serverless systems look cheap when usage is small.
But at scale, they accumulate costs in ways cluster-based compute does not:

1. Per-execution pricing

Lambdas charging per millisecond and per GB-sound harmless until you run millions of them per day.

2. Always-on metadata operations

Serverless SQL engines charge for *data scanned*, not *data returned* - punishing poorly partitioned tables.

3. High concurrency costs

Advanced serverless tiers charge premium fees to handle thousands of parallel executions.

4. No locality or caching benefits

Long-running clusters build warm caches.
Serverless jobs start cold every time.

5. Cold start latency

For latency-sensitive jobs, cold starts can be unacceptable.

Serverless is cheap at small scale, but extremely expensive when misused at large scale.

Where Cluster-Based Compute Still Dominates

Cluster-based systems remain the backbone of mature data platforms because they handle the workloads serverless cannot:

Long-running jobs

ML training, compaction, heavy joins, massive window operations.

Stateful streaming

Spark, Flink, Kafka Streams - these maintain in-memory state that cannot be restarted every minute.

Predictable traffic

If your pipeline runs 24/7 at stable throughput, paying per-second on a serverless model is often more expensive.

Workloads where caching matters

Clusters keep data warm in memory and SSD layers.
Serverless does not.

Workloads requiring fine-grained optimization

Executor sizing, shuffle tuning, JVM configuration - serverless hides these knobs.

Cluster-based compute is not “old school.”
It’s the only viable choice for many enterprise-scale data platforms.

How Mature Companies Decide Between the Two

The smartest organizations don’t ask:

> “Serverless or clusters?”

They ask:

> “Where does elasticity help us, and where does longevity help us?”

Let’s explore both sides with concrete examples.

When Companies Choose Serverless

(Because elasticity beats control)

- A file lands in S3 → trigger a Lambda → extract metadata
- A small CDC batch arrives → run a quick Glue job
- A microservice needs occasional SQL → Snowflake warehouse that auto-suspends
- An hourly analytics workflow → BigQuery on-demand
- A real-time notification → Cloud Function mapping to Slack, Kafka, etc.

These are low-overhead, bursty, and unpredictable workloads - perfectly aligned with serverless.

Cost savings come from not paying for idle time.

When Companies Choose Clusters

(Because control beats elasticity)

- Huge daily ETL joins involving terabytes
- Stateful streaming (session windows, keyed aggregations)
- Machine learning training loops
- Feature engineering pipelines requiring caching
- Compaction jobs rewriting thousands of partitions
- Kafka or ClickHouse clusters where disk locality matters

These workloads have long lifetimes, complex dependencies, and high statefulness.

Clustering reduces cost *because* you are not constantly spinning up new containers.

The Hybrid Pattern: The True Real-World Architecture

This is what almost every FAANG company ends up doing:

- Serverless for orchestration, ingestion, triggers, notifications, micro-tasks
- Clusters for deep, heavy computation
- Serverless SQL for ad-hoc analytics
- Clusters for production-grade batch + streaming pipelines
- Serverless autoscaling for unpredictable workloads
- Reserved/spot clusters for predictable workloads

The hybrid model is the real-world solution - not either/or.

How to Explain This in a System Design Interview

Here's the kind of senior-level statement that interviewers *love*:

> "For event-driven, bursty workloads with unpredictable concurrency, we'll use serverless compute because it gives us elasticity and eliminates idle cost.
>
> For long-running, stateful, or high-throughput ETL and streaming jobs, we'll rely on cluster-based compute where caching, locality, and predictable pricing matter.
>
> In practice, platforms use a hybrid approach: serverless for ingestion and orchestration, and clusters for continuous pipelines. This offers the best balance of cost, reliability, and operational control."

This shows you understand:

- economics
- operational constraints
- architectural tradeoffs
- real-world deployment patterns

It separates you from candidates who only "use whatever tool they know."

Kafka & Kinesis Throughput & Cost Math - The Economics of Streaming at Scale

Streaming looks cheap when you're handling a few thousand events per second.
It becomes shockingly expensive when you cross millions.

Every real streaming platform - Uber, Lyft, Netflix, DoorDash, Stripe - eventually realizes that Kafka/Kinesis cost is driven by one thing:

> You don't pay for messages.
> You pay for *throughput, partitions, retention, and replication*.

And most teams drastically over-provision because they never learned the actual math behind streaming.

This chapter explains that math.

1. Kafka: You Pay for Brokers, Storage, and Replication

Kafka itself is "free," but running Kafka is not.

When you run Kafka self-managed (or via MSK), your cost comes from:

1. *Broker EC2 Instances*
   CPU, memory, SSDs - the hardware footprint.

2. *Replication Factor (RF)*
   RF=3 means every message is stored 3 times.

3. *Storage (EBS/Gp3)*
   Kafka storage grows linearly with:

   - retention duration
   - message size
   - replication factor
   - number of partitions

4. *Network Traffic*
   Kafka replicates data across AZs (a silent but huge cost).
   Consumers in different AZs amplify this.

In short:

> Kafka cost = brokers × storage × replication × traffic.

This is why mature companies obsess over partition count, retention, and payload size - not event frequency.

2. How Kafka Throughput Math Actually Works

Kafka partitions determine:

- maximum parallelism
- maximum throughput
- consumer scalability
- producer write distribution
- broker load balance

But partitions also determine cost.

Each partition:

- uses file handles
- uses memory for indexes
- uses CPU for compaction
- adds metadata overhead
- increases controller workload

Rule of thumb from industry:

> **Each Kafka broker comfortably handles ~2,000–4,000 partitions.**

Push beyond that, and the cluster becomes unstable.

This means:

- 10 topics × 100 partitions each = 1,000 total - OK
- 100 topics × 1,000 partitions each = 100,000 total - costly & unstable

Partition count is the #1 cause of runaway Kafka cost.

People think “just increase partitions,” but in industry:

> Adding partitions is an operational commitment, not a tuning knob.

3. Why Message Size Is More Important Than Message Count

Kafka performs best with messages in the 1–10 KB range.

Small messages (<1 KB) increase overhead per message.
Huge messages (>1 MB) slow down replication and cause broker churn.

But cost-wise:

- Message size × retention × replication × partitions
  determines storage cost.

Example:

- message size = 4 KB
- throughput = 100K messages/s
- retention = 7 days
- replication = 3

Storage =
100,000 msg/s × 4 KB × 604,800 seconds × 3 =
~700 TB per week

This is why Uber compresses almost every Kafka payload (Snappy or gzip).

4. Kinesis Cost: Shard Math Changes Everything

Kinesis is easier to operate than Kafka but much trickier economically because:

You pay per shard, not per message.

Each shard provides:

- 1 MB/s write
- 2 MB/s read
- 1,000 records/s write limit
- Unlimited partitions (Kinesis handles hashing)
- Retention up to 365 days (extra cost)

So if your throughput is:

- 5 MB/s → you need 5 shards
- 20 MB/s → 20 shards
- 80 MB/s → 80 shards

Shard cost =
(~$0.015/hr × #shards)
≈ $11 per shard per month

But with 500 shards?
→ $5,500 per month
With replication in multiple regions?
→ $11,000 per month
With enhanced fan-out consumers?
→ cost grows further

This is why companies fear shard explosions during peak events.

5. Kinesis "Peak Provisioning" Problem

Kinesis charges by max required throughput, not average.

So if your:

- normal traffic = 5 MB/s
- peak for 30 seconds = 30 MB/s

You must provision for 30 shards, not 5.

This is notoriously expensive.

Many companies move high-throughput workloads off Kinesis for this reason.

AWS On-Demand Mode helps but becomes expensive above ~20 MB/s.

6. Practical Cost Optimization Techniques (Real Companies Use These)

No robotic lists - here's how real systems approach it.

A. Reduce Payload Size

Stripe moved many JSON payloads to Avro/Protobuf.
Result:

- ~50% smaller messages
- ~2× lower throughput usage
- lower storage & network cost

Every FAANG company eventually converts JSON → binary.

B. Adjust Retention Window Intelligently

Not all topics need 7 days of retention.

At Uber:

- operational topics = 24h retention
- analytical topics = 3–7 days
- audit/compliance topics = stored elsewhere, NOT in Kafka
- ML features = streamed → lake → compacted (not retained indefinitely)

Kafka is not a storage system.
That's what S3 is for.

C. Drop Unnecessary Partitions

Companies audit partition counts regularly.

Airbnb decreased partition count by 40% with:

- better hashing
- merging low-volume topics
- consolidating event categories

Partition cleanup often stabilizes clusters immediately.

D. Use Compression Everywhere

Snappy is the sweet spot:
fast + good reduction + low CPU overhead.

Real savings:

- 40–80% reduction in payload size
- lower network replication cost
- smaller storage footprint

Compression is the cheapest optimization you can make.

E. Separate Hot Path from Bulk Data Path

At Netflix:

- high-frequency events → Kafka
- heavy logs → S3 firehose
- near-real-time analytics → Flink on top of S3-backed streams

Kafka handles *business events*, not everything.

F. Use Multi-Level Streaming

Many companies use:

- Kafka/Kinesis → for real-time events
- S3 → for historical backfill
- Pub/Sub or other bus → for notifications

Streaming is expensive - not everything needs to go through it.

7. Senior-Level Explanation for Interviews

Interviewers often ask:

> "How would you scale Kafka/Kinesis economically?"

A polished answer:

> "I'd calculate throughput requirements, choose partition/shard counts based on peak traffic, compress payloads, minimize retention windows, and move cold data to S3.
>
> In Kafka, I'd limit partition counts to what brokers can handle, balance partitions evenly, and avoid storing large payloads.
>
> In Kinesis, I'd use shard-on-demand or autoscaling to avoid over-provisioning, and process heavy events using low-cost S3 pipelines instead of high-throughput shards."

This shows you understand the economic side of streaming - the part most engineers ignore.

Section VI : Introduction to Case Studies

This section takes everything you've learned so far and puts it into practice through *progressively challenging*, interview-style case studies. Each scenario is designed to mirror how top tech companies evaluate real system design maturity starting with foundational ETL patterns and scaling all the way to complex, production-grade architectures.

You'll begin with simple batch pipelines, then advance to multi-layer lakehouse designs, near–real-time streaming systems, identity resolution frameworks, fraud detection engines, feature generation pipelines for recommendation systems, and large-scale clickstream analytics.

Every case intentionally introduces new constraints, new trade-offs, and deeper architectural reasoning. The goal is not just to give you the "right" solution, but to train your instincts on how to ask clarifying questions, how to structure your approach, how to justify decisions, and how to think like a senior data engineer.

By the end of this section, you'll be able to handle any open-ended system design prompt with clarity, confidence, and a level of depth that stands out in FAANG-level interviews.

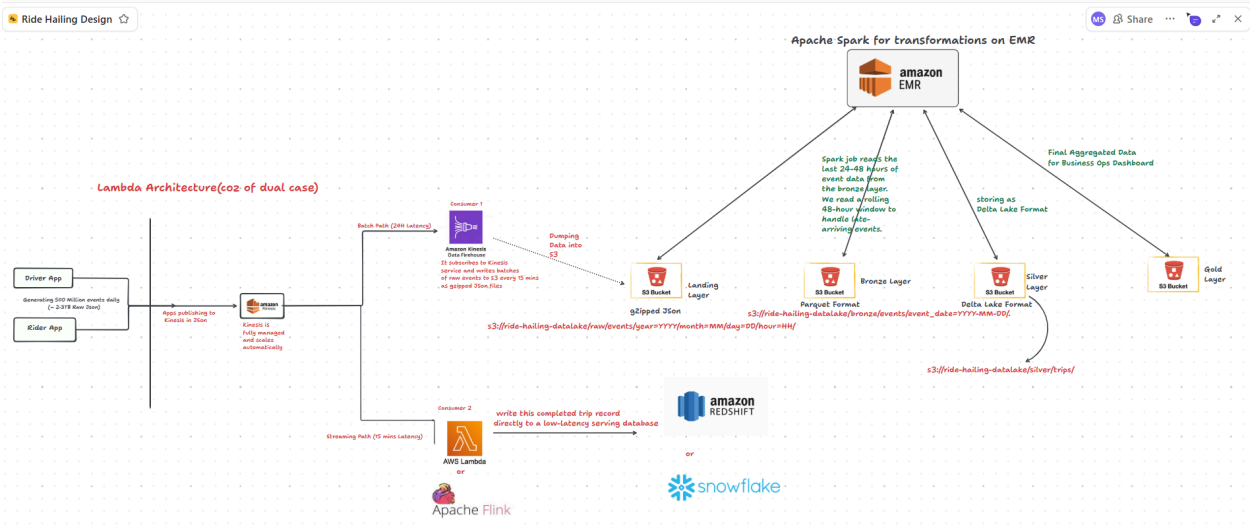
A Note Before You Begin

I strongly encourage you to attempt each case study on your own before reading the solution. Start by sketching the workflow or architecture using tools like draw.io, Whimsical, Figma, or any diagramming tool you prefer. Once you've created your version, go through the provided breakdown and draw a second, refined diagram.

This simple habit dramatically strengthens understanding. Our brains retain visual information far more effectively than text, and the act of drawing systems forces deeper reasoning, stronger memory, and real architectural intuition.

Treat each case study as a real design interview; think, draw, refine, and compare. That's how you level up your system design skills like an experienced engineer.

Case Study 1: Design a Ride Sharing App



Problem Statement

Interviewer:
Let's start the interview. Here's **Case Study**:

We are a large ride-hailing company, operating in hundreds of cities globally. We need to design a core data platform to process and analyze trip data. This platform's primary goal is to generate key business metrics for city operations teams, such as

- trips per hour
- driver utilization
- surge pricing effectiveness

The raw data comes from our driver and rider apps as events. A single trip generates multiple events:

- trip_requested
- driver_assigned
- driver_arrived
- trip_started
- trip_completed
- payment_processed.

The daily volume is approximately **500 million events**, totaling around **2-3 terabytes of raw JSON data per day**.

Design the end-to-end data pipeline, from ingestion to the final data mart that powers the business dashboards. Focus on scalability, reliability, and data quality.

Clarifying Questions

Interviewee:
That's a great problem. It touches on many core data engineering principles. Before I propose a solution, I need to ask a few clarifying questions to better define the scope.

1. **Latency:** How fresh does the data on these dashboards need to be? Are the city ops teams okay with data that is 24 hours old, or do they need near real-time visibility, say, within 5-15 minutes?

2. **Consumers:** Are the primary consumers BI analysts running ad-hoc queries and viewing dashboards (e.g., in Tableau or Looker), or are there also machine learning teams that need this data for model training?

3. **Data Schema:** You mentioned JSON events. Is the schema stable, or does it evolve frequently as the app teams add new features? Is there a central schema registry?

4. **Historical Data:** How far back do we need to store queryable data? A year? Five years? Indefinitely? This will impact storage choices.

5. **Technology Stack:** Is this a greenfield project, or are we building on an existing cloud platform like AWS, GCP, or Azure? Are there any preferred technologies I should consider?

Interviewer:
Excellent questions.

1. **Latency:** Let's aim for a dual-use case. The executive-level "state of the business" dashboards are fine with a 24-hour delay. However, the city operations teams need to react to live events, like a concert ending or a subway line shutting down. For them, a 15-minute latency is the target.

2. **Consumers:** Primarily BI and Operations. However, a fraud detection ML team has expressed interest in using this data eventually. Let's keep that as a future requirement.

3. **Schema:** The schema evolves, but not chaotically. Expect new fields to be added quarterly. Let's assume there isn't a mature schema registry in place today, so the pipeline should be resilient to these changes.

4. **Historical Data:** We need to keep raw data forever for compliance. The queryable "hot" data for dashboards should cover at least the last 13 months.

5. **Technology Stack:** Assume we are on AWS and prefer using managed services where it makes sense to reduce operational overhead.

Interviewee:
Thank you, that provides a much clearer picture. The 15-minute latency requirement for city ops alongside the 24-hour requirement for execs suggests a hybrid, or **Lambda, architecture** is a good fit.

I'll structure my proposed solution into the following stages:

1. **Ingestion:** Getting the events from the apps into our data ecosystem.
2. **Storage (Data Lake):** Persisting the raw and processed data.
3. **Transformation:** Cleaning, structuring, and modeling the data.
4. **Serving Layer:** The final data marts for consumption.
5. **Orchestration & Quality:** How we manage the pipeline and ensure trust in the data.

Ingestion Layer

Given the high volume and streaming nature of the events, I would use **Amazon Kinesis Data Streams** or a managed Kafka service like **MSK**. Kinesis is often simpler to manage. App backends would publish the JSON events to a Kinesis stream.

- **Trade-off:** Kinesis is fully managed and scales automatically, which fits our "managed services" preference. Self-hosting Kafka on EC2 would give us more control but adds significant operational burden. MSK is a middle ground. I'll stick with Kinesis for simplicity.

To handle schema evolution, I'd wrap the JSON payload within a standard envelope containing metadata like event_name, event_timestamp, and schema_version.

Interviewer:
Okay, Kinesis sounds reasonable. What happens right after the data lands in the Kinesis stream? Where does it go?

Interviewee:

From Kinesis, I'd set up two parallel consumers for our dual-latency requirements.

1. **Batch Path (for 24h latency):** An **Amazon Kinesis Data Firehose** delivery stream would be subscribed to the Kinesis stream. Firehose is great because it handles micro-batching, compression, and delivery to S3 out of the box. It would write batches of raw events to an S3 "landing zone" every 15 minutes as gzipped JSON files. This creates our immutable raw data source of record.

2. **Streaming Path (for 15m latency):** A separate consumer, likely an **AWS Lambda function** or a **Flink application running on Kinesis Data Analytics**, would process events in near real-time.

Let's focus on the **Storage (Data Lake)** on S3 for the batch path first.

I would structure the S3 bucket with a clear hierarchy:
s3://ride-hailing-datalake/raw/events/year=YYYY/month=MM/day=DD/hour=HH/.
This partitioning scheme is fundamental for efficient querying later on.

Interviewer:
You mentioned Firehose writes gzipped JSON. Why JSON? What are the implications of that choice for the downstream transformation jobs?

Interviewee:
That's a critical point. Storing raw data as JSON is good for fidelity and debugging, as it's exactly what we received. However, it's terrible for analytical query performance because it's row-based and not splittable.

Therefore, the very first transformation job in our batch process will be to convert this raw JSON into a columnar format. I would choose **Apache Parquet**.

- **Why Parquet?** It offers excellent compression and columnar storage, meaning analytical queries that only need a few columns don't have to scan the entire row. This drastically improves performance and reduces cost in query engines like AWS Athena, Redshift Spectrum, and Spark.

This first job would run hourly, converting the raw JSON into a partitioned Parquet dataset in a bronze layer of our data lake:
s3://ride-hailing-datalake/bronze/events/event_date=YYYY-MM-DD/.

Transformation Layer

Interviewer:
Okay, so we have raw data in a bronze Parquet format. Now, walk me through the **Transformation** stage. How do you get from individual events like trip_started and trip_completed to a single, clean "trips" table that an analyst can use? This is the core modeling problem.

Interviewee:
Absolutely. This is where we build our silver (cleaned, integrated) and gold (business-aggregated) tables. I would use **Apache Spark** running on **AWS EMR** or **AWS Glue** for these transformations.

The main challenge is stateful processing: stitching together multiple events that belong to the same trip.

My proposed model for the silver layer would be a trips table, with trip_id as the primary key. The process would be:

1. **Read Bronze Data:** A daily Spark job reads the last 24-48 hours of event data from the bronze layer. We read a rolling 48-hour window to *handle late-arriving events.*

2. **Deduplication:** We'd use a ROW_NUMBER() window function partitioned by event_id to ensure we don't process duplicate events.

3. **Sessionization/Stitching:** We'll group events by trip_id. Within each group, we can pivot the data, taking the timestamp from trip_started as start_time, the timestamp from trip_completed as end_time, the fare amount from payment_processed, etc. This collapses multiple event rows into a single row representing a complete trip.

4. **Data Cleansing:** Apply business rules, handle nulls, cast data types, and cleanse location data.

5. **Write to Silver Layer:** The resulting trips table is written back to S3, again as Parquet, into a silver path: s3://ride-hailing-datalake/silver/trips/. This table represents our clean, trip-level source of truth.

To make this robust, I'd use a modern data lake table format like **Apache Hudi** or **Delta Lake** on top of Parquet. Let's say Delta Lake.

- **Why Delta Lake?** It gives us ACID transactions on S3, which prevents partial writes if a job fails. It also simplifies handling updates and deletes (e.g., for GDPR) and provides time travel, which is great for debugging.

Handling Late Arriving Data

Interviewer:
You mentioned handling late-arriving data by processing a 48-hour window. What if an event arrives 3 days late? How does your design handle that, and what is the impact on the "gold" tables and the dashboards?

Interviewee:
That's a classic and important edge case.

My approach would be:

1. **Watermarking:** The daily Spark job would define a "lateness threshold," say 48 hours. Any data older than that is considered truly late.

2. **Updates via Delta Lake:** When the main daily job runs, it creates the silver.trips table. If a 3-day-old event comes in, a subsequent run will process it. Because we are using Delta Lake, we can issue an MERGE (UPSERT) operation to update the existing trip record with the newly arrived information, rather than just appending.

3. **Downstream Impact:** This means our gold tables, which are aggregations on top of silver, will also need to be recomputed for the affected period (e.g., the day the trip actually occurred). The orchestrator needs to be smart enough to trigger these backfills. For the dashboard user, this means a metric for a past day might change slightly. We must communicate this "**eventual consistency**" model to our stakeholders. It's a trade-off between accuracy and the complexity of holding back processing until all data arrives, which is impossible.

Interviewer:
Good. Now let's talk about the streaming path you mentioned for the 15-minute latency requirement. How does that work, and how do you ensure consistency with the batch pipeline?

Interviewee:
For the streaming path, I'd use **Flink on Kinesis Data Analytics**.

1. **Stateful Streaming:** The Flink job would read from the Kinesis stream and maintain an in-memory state for each active trip_id. As events for a trip arrive (driver_assigned, trip_started), it would update the state for that trip.

2. **Windowing:** When a trip_completed event arrives, or after a timeout window (e.g., 2 hours) passes without updates, the Flink job considers the trip record complete.

3. **Micro-batch to Serving Layer:** The job would then write this completed trip record directly to a low-latency serving database. A good choice here would be **Amazon Redshift** or even **Snowflake**. We can use a micro-batch COPY command to load data into Redshift every 5-10 minutes.

Consistency Challenge (Lambda Architecture): The biggest challenge is that we now have two different codebases (Spark for batch, Flink for streaming) creating trip data. This can lead to discrepancies.

- **Solution:** The batch pipeline is the ultimate source of truth. The streaming pipeline provides fast, provisional data. Every 24 hours, the batch job's output can be used to overwrite the last 24 hours of data in the serving layer table populated by the streaming job. This reconciliation process ensures long-term accuracy while providing short-term speed. The gold tables that execs see would only be built from the batch source of truth.

Data Quality and Governance

Interviewer:
Let's pivot to data quality and governance. A city operations manager calls you, furious, saying the "driver utilization" metric for San Francisco dropped by 50% overnight, and they think the data is wrong. How do you debug this? What tools or processes should be in place?

Interviewee:
A critical scenario. Trust is paramount. My approach would be multi-layered:

1. **Proactive Prevention (In-Pipeline Checks):**

    - **Data Contracts:** Work with the app teams to establish schemas using something like Avro or Protobuf and enforce them at the ingestion point using a Schema Registry (e.g., Confluent Schema Registry or AWS Glue Schema Registry). This prevents malformed data from entering the pipeline in the first place.

    - **Automated Data Quality Tests:** I'd embed a framework like **Great Expectations** or use dbt tests within our transformation jobs. After creating the silver.trips table, we'd run checks like:

        - expect_column_values_to_not_be_null for trip_id, start_time.
        - expect_column_value_to_be_between for fare_amount (e.g., not negative, not > $1000).
        - Freshness checks to ensure we've received recent data.

    - If these tests fail, the orchestrator (**Airflow** or **Dagster**) should halt the pipeline downstream and send an alert to the on-call data engineer.

2. **Reactive Debugging (When an issue is reported):**

    - **Data Lineage:** I would use a tool that provides lineage, so I can immediately see that the "driver utilization" metric in the gold table is derived from the silver.trips table, which comes from the bronze.events table. This narrows the search space.

    - **Dashboarding & Monitoring:** The pipeline itself should be monitored. I'd check our Airflow DAG runs for failures. I'd look at CloudWatch metrics for Kinesis (e.g., IncomingBytes) to see if there was a drop in event volume from the SF region.

    - **Querying the Lake:** I would query the bronze layer directly for the last 48 hours of data for San Francisco to see if there's an anomaly in the raw data itself (e.g., a new app version is sending null driver_ids). The time-travel capability of Delta Lake would also be invaluable here to compare snapshots of the silver table before and after the issue appeared.

By having these proactive checks and reactive tools, we can quickly pinpoint whether the issue was a genuine business anomaly, a bug in the source app, or a bug in our data pipeline code.

Case Study 2: Design a Customer 360 pipeline

Problem Statement

Interviewer (FAANG Principal Data Engineer**):**

Here's **Case Study** : Imagine you're tasked with building a **Customer 360 pipeline** at scale. We want a unified customer profile by merging multiple systems-CRM, web, mobile app, and call center.

The pipeline should:

1. Handle **identity resolution** (e.g., email, phone, device ID → unified user_id).
2. Manage **PII masking** and GDPR "forget me" requests.
3. Be highly scalable, fault-tolerant, and support both batch and near real-time updates.

Walk me through how you'd design this pipeline, step by step.

Clarifying Questions

Interviewee (Senior Data Engineer, Candidate**):**

Great, thanks for the problem statement. Before I propose a solution, I'd like to clarify a few things:

1. **Volume & Velocity:** Roughly how many customers are we dealing with? Millions? Hundreds of millions? How frequent are updates across these systems?

2. **Latency Expectations:** Do business users expect the Customer 360 view to update in near real time (say <5 minutes) or is daily/ hourly batch refresh acceptable?

3. **Source System Access:** Do we get direct DB access (CRM SQL database, call center system) or are we pulling through APIs/events?

4. **Downstream Consumers:** Will this serve BI dashboards only, or also downstream ML models (e.g., churn prediction, recommendation)?

5. **PII Governance:** Do we need encryption at rest + column-level masking? Or tokenization?

Interviewer:

Good questions. Let's say:

- We have about **200M customer records** across sources.
- Web and mobile send **real-time clickstream events**, CRM updates are **hourly**, call center is **daily batch**.
- Latency target: **<1 hour** for the unified profile.
- Consumers: **dashboards, ML models, and marketing automation tools**.
- For PII: Yes, **encryption at rest + masking for analysts**; also must handle **GDPR forget requests within 24 hours**.

Interviewee :

Perfect, let me propose a solution. I'll break it into **data ingestion, identity resolution, storage/processing, governance, and serving**.

1. Ingestion Layer

- Use **Kafka/Kinesis** for real-time clickstream (web, mobile).
- For CRM & call center (batch systems), use **CDC (Change Data Capture)** or API-based ingestion via **Airflow/ADF** orchestrating jobs into the lake.
- Store raw data in a **Data Lake (S3/ADLS)** in partitioned Parquet/Delta format for durability and replay.

2. Identity Resolution

- Build a **Master Data Management (MDM)-like service** in Spark/Databricks.
- Deterministic matching (exact email, phone) + probabilistic matching (fuzzy name, address, device graph).
- Assign a persistent unified_user_id stored in a **Delta Lake Gold table**.
- Maintain a mapping table: {source_id → unified_user_id} for traceability.

3. Storage & Processing

- **Bronze/Silver/Gold Medallion architecture**:
    - Bronze: Raw events, partitioned by source + ingestion_date.

    - Silver: Cleaned/standardized PII with schema evolution handling.
    - Gold: Unified customer profile table with latest attributes and history (SCD2 for attribute changes).

- Use **Delta Lake** for ACID guarantees and time travel (important for GDPR rollback).

4. Governance & GDPR

- Store PII encrypted with **KMS keys**. Analysts see only masked columns unless explicitly whitelisted.
- Maintain a **GDPR service**: when a "forget me" request arrives, trigger a pipeline that:
    - Deletes customer from Gold + Silver layers.
    - Issues a delete transaction on Delta tables (vacuum after retention period).
    - Pushes delete requests downstream (e.g., marketing systems).

5. Serving Layer

- Expose unified profile as:
    - **Warehouse tables (Snowflake/BigQuery/Redshift)** for BI.
    - **Feature Store** for ML teams.
    - **REST/GraphQL API** for external marketing tools.

6. Orchestration & Monitoring

- Use **Airflow** (batch) + **Kafka Streams/Spark Structured Streaming** (real-time).
- Add **data quality checks** (e.g., Great Expectations): no duplicate unified IDs, schema drift alerts.
- Monitor SLA breaches with **Prometheus/Grafana** dashboards.

Scaling up / Sudden Volume Spike

Interviewer:

So now imagine it's **Black Friday**. Your web and mobile clickstream volume spikes 10x. We're talking about **tens of billions of events per day**.

- How would you ensure your pipeline **scales** without blowing up compute costs or breaching the 1-hour SLA for the unified customer profile?

Interviewee :

That's a great scaling challenge. I'd approach it on three fronts: **stream ingestion, storage optimization, and processing efficiency**.

1. **Stream Ingestion**
    - Kafka/Kinesis: Scale out partitions ahead of the event. For billions/day, I'd aim for ~10K partitions, balanced across brokers.
    - Use compression (Snappy, LZ4) to reduce payload size.
    - Decouple raw ingestion from enrichment by writing directly to a **bronze Delta table** in append-only mode.
2. **Storage Optimization**
    - Partition clickstream data by event_date + event_hour.
    - Use **auto-optimized files / compaction** to avoid small-file problems (Databricks OPTIMIZE with Z-Order on user_id).
    - Store raw data in Parquet + Delta logs for ACID.
3. **Processing Efficiency**
    - Use **Structured Streaming with micro-batches** sized dynamically (say 30s–2m).
    - Scale Spark clusters with autoscaling (compute only when backlog > threshold).
    - Implement **state store pruning**: keep only relevant keys in memory for identity resolution.

Cost Control

- Offload cold storage to Glacier/Archive tiers.
- For hot compute, use **spot instances/preemptible VMs** with checkpointing to survive terminations.

So even if traffic spikes, my system will elastically scale without breaking SLA.

Ensuring Deduplication and Avoiding State Explosion

Interviewer :

Okay, but identity resolution is stateful. You're joining billions of new device IDs against 200M existing customers. How do you handle **state explosion** and ensure deduplication doesn't overwhelm memory?

Interviewee :
Yes - naive stateful joins can overwhelm memory. My approach:

- Use a **two-tier strategy**:

    1. **Deterministic matches** (exact email, phone) are handled with **Bloom filters** or **RocksDB-backed state stores** in streaming jobs.

    2. **Probabilistic/fuzzy matches** (name/address similarity) run in **offline batch jobs** (hourly or nightly).

- The streaming layer assigns a **temporary ID → unified_user_id** when a deterministic match succeeds. If no match, mark as "orphan candidate" and later resolve in batch.

- To reduce memory, maintain state in **external key-value stores** like Redis/Delta Live Tables instead of Spark in-memory only. This allows scaling horizontally.

GDPR Compliance

Interviewer :

Good. Now let's throw a wrench:

A customer submits a **GDPR forget-me request** while you're processing billions of events. Some of their events are already in the system; some are in-flight in Kafka; some are being enriched.

How do you **guarantee compliance** without corrupting your unified profile or losing system consistency?

Interviewee :

This is a tough but real-world case. My strategy would be:

1. **Central GDPR Service**
    - All “forget me” requests are published as events themselves (into a gdpr_delete Kafka topic).
    - Every consumer pipeline listens to this and applies deletes.
2. **Delta Lake Deletes**
    - When delete arrives:
        - Delete user from Gold/Silver tables using Delta DELETE transactions.
        - Add their user_id to a **blacklist table** (fast lookup).
3. **In-Flight Data**
    - Streaming jobs check against the blacklist table in every micro-batch.
    - If a record belongs to a deleted user, it’s dropped immediately (never lands in Gold).
4. **Auditability**
    - Maintain immutable logs of delete requests + actions taken.
    - Run periodic sweeps to guarantee no residual traces.

This way, even if events were in-flight, they get quarantined at the micro-batch stage. And downstream systems consuming Gold tables will never see deleted IDs.

Schema Drift/ Schema Evolution

Interviewer:

Alright. CRM adds a new column preferred_language. Meanwhile, the mobile team drops device_id entirely due to privacy rules. Schema drift is real. How do you keep your Customer 360 pipeline stable without breaking downstream dashboards and ML models?

Interviewee :
I’d handle schema evolution with a **layered + schema-aware design**:

1. **Ingestion (Bronze Layer)**
    - Use **schema-on-read** for raw storage: store the exact JSON/Avro payload with all fields as-is.

- Maintain ingestion metadata (schema registry + schema version) in **Confluent Schema Registry or Glue Schema Registry**.

2. **Processing (Silver Layer)**
    - Use a **schema evolution–aware format** like **Delta Lake** (or Iceberg/Hudi). These handle:
        - Column addition → auto-evolved with default null.
        - Column drop → retained in older partitions but simply absent in new writes.
    - Transformations always include a **schema alignment step**: map incoming fields to a canonical schema, fill missing fields with defaults.
3. **Gold Layer Stability**
    - Unified Customer Profile schema is **contract-driven**. It evolves in a controlled way through **Data Contracts**.
    - If CRM adds preferred_language, we:
        - Backfill for historical users as null.
        - Expose it downstream only after contract update + version bump.
    - If mobile drops device_id:
        - Continue using historical values but stop expecting it for new data.
        - Update identity resolution logic to fall back on other identifiers.
4. **Resilience**
    - Build **schema drift alerts**: if a column is added/dropped, Airflow task fails gracefully with an alert rather than silently corrupting data.
    - Analysts/ML consumers always query via **Gold tables**, never raw Silver → ensures consistency.

This way, schema drift doesn't break pipelines. Instead, it triggers a controlled schema evolution process.

Interviewer :

Good. But what if a **late-arriving batch** from the call center comes in with an old schema that doesn't have preferred_language - two weeks after we've already added the field in Gold?

Wouldn't that cause conflicts or overwrite issues?

Interviewee :
Yes, late data with old schema can cause mismatched writes. Here's how I'd address it:

- **Schema Versioning in Metadata**
    - Each record is tagged with schema_version at ingestion.
    - Transformations are aware of version → they align fields accordingly.
- **Backward Compatibility**
    - In Delta Lake, missing fields are filled with null automatically.
    - Our ETL logic ensures no overwrite of newer attributes: for example, if preferred_language is null from late data, it doesn't overwrite an existing non-null value in Gold.
- **Merge Strategy (SCD2)**
    - Use **MERGE INTO** with conditional updates:
        - Only update attributes if source has a non-null, more recent value.
        - This prevents regressions from late data.

So the system is resilient to late + outdated schema data without corrupting Gold.

Interviewer :

Excellent. Let's talk about serving **and performance**. BI dashboards and ML models need fast access to the unified profile, but the Gold table is 200M+ customers and growing.

How do you **serve data efficiently** for both BI analysts and ML training jobs without duplicating too much?

Interviewee :

I'd split serving into **two optimized paths**:

1. **BI/Analytics (ad-hoc queries, dashboards)**
    - Expose Gold table via **cloud warehouse** (Snowflake/BigQuery/Redshift Spectrum).
    - Partition + cluster on user_id and last_updated_date.
    - Build **materialized aggregates** (active users, churn rate, segment counts) so dashboards don't scan billions of rows.
    - Add **row-level security policies** so analysts see only compliant views (masked PII).
2. **ML/AI (training & real-time inference)**
    - Push Gold attributes into a **Feature Store** (e.g., Databricks Feature Store, Feast).
    - Offline store → Parquet/Delta, optimized for Spark training.
    - Online store → Redis/DynamoDB for <10ms lookup latency (used by recommender systems).
3. **API Access**
    - Wrap Gold data with a **GraphQL/REST API** that allows external marketing systems to query user attributes in real time.
    - API enforces GDPR blacklists + PII masking at query time.

This way, BI gets query flexibility, ML gets high throughput, and external systems get fast lookups - all without duplicating full datasets unnecessarily.

Resilience and Disaster Recovery

Interviewer :

Alright. Imagine this pipeline is running in production for a year. Suddenly:

- Kafka brokers crash and you lose 3 hours of web events.
- A Spark job corrupts 5% of Silver tables.
- A GDPR audit finds that 200 users were not deleted correctly.

How do you design for **resilience + disaster recovery** so these incidents don't destroy trust in your Customer 360?

Interviewee :

Here's how I'd cover resilience end-to-end:

1. **Kafka Loss (3h downtime)**
    - Enable **Kafka topic replication** across multiple brokers + AZs.
    - Use **log compaction + retention** → replay from offsets after brokers recover.
    - Worst case: fall back to **raw event replays** from CDN logs or client-side buffering.
2. **Silver Table Corruption**
    - Delta Lake provides **time travel**. I'd roll back to the last valid checkpoint and reprocess from Bronze.
    - Add **data quality gates** (row count checks, null ratio checks) before promoting Silver → Gold.
3. **GDPR Audit Failures**
    - Implement **two-step deletion**:
        - First soft-delete (mark as "to be purged").
        - Then physical delete after verifying deletion propagated downstream.
    - Maintain **GDPR audit trail table**: every request + delete action is logged immutably.
    - Run **daily reconciliation jobs** to ensure no residual users exist.

So even if incidents occur, we can recover, reprocess, and demonstrate compliance without system-wide failure.

Interviewer:

Solid answers. You covered ingestion, scaling, identity resolution, GDPR, schema evolution, serving, and recovery.

If I had to press you more, I'd ask about **cost governance** (warehouse query costs vs. ML feature store duplication) and **multi-region setups** for global compliance. But overall, you're demonstrating strong system design maturity.

Case Study 3: Design a Realtime Fraud Detection System

Problem Statement

Interviewer

Here's **Case Study** : You need to design a **real-time fraud detection system**.

Requirements:

1. Ingest financial transactions in a **streaming pipeline**.
2. Join each transaction with **historical aggregates** (e.g., user's avg spend, past locations).
3. Trigger fraud alerts **within seconds**.
4. Balance **latency vs. accuracy** - can't miss too much, but can't have too many false positives either.

Walk me through your design.

Clarifying Questions

Interviewee :

Great, thank you. Before jumping into a solution, I'd like to clarify a few things:

1. **Volume:** How many transactions per second globally are we handling?

2. **Latency:** Is "seconds" literally <2s end-to-end, or can alerts be near real time (~10–30s)?

3. **Accuracy vs Recall:** Is false negative (missed fraud) worse than false positive (flagging a legit user)?

4. **Serving:** Should alerts integrate with a **real-time service** (e.g., customer blocked at POS) or just **notify analysts** for investigation?

5. **Historical Context:** Do we have a centralized store for past user behaviors (e.g., last 90 days transactions), or do we compute aggregates on the fly?

Interviewer :

Good questions. Let's assume:

- **Volume:** 50K transactions/sec peak, ~3B/day globally.

- **Latency:** Alerts must be generated within **2–3 seconds**.
- **Accuracy:** False negatives are worse - fraud must be caught even at cost of extra alerts.
- **Serving:** Alerts must be **real-time** (block transaction at POS/online checkout).
- **History:** We have 90 days of transaction history stored in a data lake, but we also need **hot aggregates** for quick lookups.

Solution

Interviewee :

Got it. I'll structure my solution into **ingestion, stream processing, historical join, scoring, serving, and monitoring**.

---

1. Ingestion Layer

- **Kafka** (or Kinesis) as transaction backbone. Each transaction: {user_id, amount, merchant_id, device_id, location, timestamp}.
- Kafka partitioned by user_id (to keep ordering for the same user).
- Raw data written to **Bronze Delta tables** for replay + audit.

2. Stream Processing

- Use **Flink or Spark Structured Streaming** for sub-second event processing.
- Micro-batch size: 1–2s max (or continuous processing mode in Flink).
- Implement **deduplication** (idempotency key) to handle retries.

3. Historical Aggregates Join

- Maintain a **state store** of rolling aggregates:
    - Avg transaction amount (last 30 days).
    - Last known location(s).
    - Device fingerprint.
- State store backed by **RocksDB (Flink)** or **Delta Live Tables + Redis**.
- For large history (90 days):
    - Store in Delta Lake (batch).

    - Periodically pre-aggregate into **hot window tables** (hour/day/week) pushed to Redis/DynamoDB.

- Streaming job joins new transaction with these aggregates (low-latency lookup).

4. Scoring & Rules Engine

- Each transaction evaluated against:

    1. **Rules:**
        - If spend > 5x avg → suspicious.
        - If location ≠ last known + >500 miles away within 5 minutes → suspicious.
        - If device_id unseen → suspicious.

    2. **ML Model:**
        - Gradient Boosted Trees / Logistic Regression pre-trained on labeled fraud data.
        - Deployed in **low-latency inference service** (e.g., TensorFlow Serving, ONNX, or SageMaker endpoints).
        - Streaming job calls inference API synchronously.

- Combine scores → fraud probability.

5. Serving Layer

- If fraud probability > threshold (say 0.85):

    - Trigger **alert event** → fraud_alerts Kafka topic.
    - Send to **fraud decisioning service** that blocks card at POS in real time.
    - Persist decision in **Gold Delta table** for analyst dashboards.

- Analysts can review in **Snowflake/BigQuery dashboards**.

6. Latency vs Accuracy Balance

- Keep **rules simple in stream** (sub-second evaluation).
- More complex ML + graph analysis runs **asynchronously** in batch (e.g., flag suspicious clusters of merchants overnight).

- This way:
    - Immediate alerts stop 90% fraud in seconds.
    - Deeper insights catch 10% long-tail fraud later.

7. Monitoring & Governance

- **SLA Monitoring:** End-to-end latency tracked in Prometheus/Grafana.
- **Model Drift Detection:** Check feature distribution shift vs training set.
- **Explainability:** Log rules triggered + model features used for compliance.
- **GDPR:** PII encrypted; user can request "forget me" → remove from history + retrain models excluding them.

Scalability

Interviewer :

Good. Now let's stress test. Suppose you're seeing **huge traffic spikes** (e.g., holiday shopping season) → 200K transactions/sec. How does your design **scale**?

Interviewee :

For scale-up:

- **Kafka**: Increase partitions, autoscale brokers. Use compression.
- **Streaming Layer**: Horizontal scale via Kubernetes auto-scaling. Stateful processing stores (RocksDB/Redis) sharded by user_id.
- **Inference Service**: Deploy ML model in GPU-backed inference servers with autoscaling. Use **batch inference micro-batches** (score 1000 transactions per request) to cut API latency.
- **Storage**: Compact Parquet/Delta files (OPTIMIZE every 5 min). Hot aggregates remain in-memory/Redis.

This ensures we stay under 3s SLA even under peak.

Common problem with Historical Join

Interviewer :

Okay, but what if the **historical join** becomes a bottleneck - say Redis can't handle 200K lookups/sec?

Interviewee :

I'd optimize by:

1. **Sharding:** Redis/DynamoDB sharded by user_id hash → distributed load.
2. **Pre-loading hot users:** Keep high-frequency users' aggregates cached in-memory on stream workers.
3. **Approximation:** Use **Bloom filters or sketches** (e.g., HyperLogLog) to approximate some aggregates under heavy load.
4. **Async Enrichment:** If lookup takes >50ms, transaction flagged with "needs enrichment" → default conservative rules applied until async join completes.

This way, we never block the fraud alert pipeline on slow historical joins.

Backup Mechanism

Interviewer :

Good. But latency is critical. What happens if the ML inference service goes down? Do we stall fraud detection?

Interviewee :

No, never stall. I'd use **graceful degradation**:

- If ML service down → fall back to **rules-only detection** (slightly higher false positives but zero downtime).
- Maintain **circuit breaker**: if >5% inference calls fail in 1 minute, cut traffic to ML, switch to rules-only.
- Once ML back, reprocess buffered transactions (async) to label fraud risk properly.

Governance and Privacy

Interviewer :

Nice. Now compliance. Fraud detection touches **PII (location, device ID, merchant info)**. How do you ensure **governance and privacy** in this real-time system?

Interviewee :

Governance strategy:

- **PII Encryption:** All PII stored encrypted with KMS keys; decrypted only in secure stream workers.
- **Masking:** Analysts see masked values (hashed emails, truncated device IDs).
- **GDPR Forget Requests:** When user requests deletion → remove from history store + mask in logs. New transactions from that user can still be processed (for fraud detection), but logs exclude personal identifiers.
- **Access Control:** Role-based access → fraud analysts vs engineers. Logs scrubbed before retention.

Fault Tolerance and Recovery

Interviewer :

Alright, last stress test. Imagine:

1. Kafka topic corrupted → you lose 30 minutes of incoming transactions.
2. Fraud alert SLA breached during that time.
3. Regulators demand an explanation.

How do you **recover + ensure trust**?

Interviewee :

Recovery strategy:

1. **Kafka Corruption**
    - Multi-AZ replication + mirror topics. Replay from replicas.

- If raw events are unrecoverable, fall back to merchant system replay (they batch-settle transactions later).

2. **SLA Breach**

   - During outage, system auto-switches to **fallback mode**:
     - Block high-risk transactions (rules like “amount > $10K, overseas”).
     - Allow low-risk ones.

   - Post-outage → replay missed transactions from Bronze storage → run fraud scoring retroactively.

3. **Regulatory Audit**

   - Immutable logs of incident timeline.
   - Explain fallback measures + show no high-value fraud was missed.
   - Demonstrate RTO/RPO compliance: Recovery Time Objective < 1 hour, Recovery Point Objective = zero data loss via replay.

This shows resilience + accountability.

Interviewer :

Excellent. You covered ingestion, joins, ML integration, scale, governance, SLA, and failure recovery. You balanced **latency vs accuracy** well.

If I pushed further, I’d ask about **multi-region active-active setups** (fraud detection must work globally) and **online learning models** (drift adaptation). But overall, this was a strong, real-world system design answer.

Case Study 4: Design a Recommendation Engine Pipeline

Problem Statement

Interviewer : Thanks for joining today. Let's dive right in.

Here's the scenario :

Imagine you're a data engineer on our social media app's Feed Ranking team. The feed is powered by a machine learning model that ranks content (videos, posts, images) for each user. To do its job, this model needs a constant supply of fresh features.

Your task is to design the data pipeline that computes and serves features about content engagement. These features include things like:

- Number of likes, shares, and comments in the last hour.
- Click-through rate (clicks / impressions) over the last 24 hours.
- The number of times a creator's new posts have been liked in the first 10 minutes of being published.

The pipeline must be able to handle a massive volume of *real-time engagement events* (likes, shares, comments, follows, impressions) and ensure that the features available to the ranking model are very fresh.

How would you approach this?

Clarifying Questions

Interviewee: This is a fascinating and challenging problem that sits at the intersection of large-scale data processing and machine learning. Before I design the system, I have a few questions to clarify the scope.

1. **Scale:** Could you quantify the volume of events? Are we talking millions, or billions of engagement events per day? What's the number of unique content items we need to maintain features for?

2. **Latency / Freshness:** What is the specific SLA for "very fresh"? Are we talking about features being updated within 5 minutes, 1 minute, or sub-second? This will be the single biggest factor in my architectural choices.

3. **Consumers:** Who consumes these features, and how? Is it an online ML model that needs sub-millisecond lookups for a single content ID? Is it an offline model training job that needs a full historical snapshot? Or both?

4. **Feature Complexity:** The examples are great. Are there more complex features, like time-decayed counts (where a recent like is worth more than a like from an hour ago)?

5. **Consistency:** What are the consistency requirements between the online and offline feature values? Is it okay if they are slightly different due to processing times?

Interviewer : Great questions. Let's get specific.

1. **Scale:** Expect around 50 billion engagement events per day (a peak of ~1 million events/sec). We have about 1 billion active content items to track features for.

2. **Latency:** For the online ranking model, features should reflect events that happened no more than 5 minutes ago.

3. **Consumers:** Both. An online service needs key-value lookups for features (`GetFeatures(content_id)`) with a p99 latency of <10ms. A weekly offline training job needs a point-in-time correct snapshot of all features for all content.

4. **Feature Complexity:** For V1, let's stick to the windowed aggregations you mentioned (counts, ratios). We can ignore time-decay for now.

5. **Consistency:** Eventual consistency is acceptable. The online and offline stores don't need to be perfectly in sync down to the microsecond.

Given this, lay out your V1 architecture.

Interviewee: Thank you, that's very clear. The 5-minute latency requirement rules out a traditional daily batch approach. This calls for a streaming architecture. My design will be centered around a Feature Store concept, which serves both online and offline consumers.

I'll break it down into Ingestion, Transformation, and Serving.

Proposed Solution

Stage 1: Ingestion

- All engagement events (likes, comments, impressions, clicks) are published to dedicated Kafka topics. This is the standard for high-volume, real-time event ingestion. It's scalable and provides a durable buffer.

- We'll use Avro for serialization and a Schema Registry to manage schemas. This allows for schema evolution without breaking consumers.

Stage 2: Transformation (The Streaming Engine)

- Given the 5-minute latency SLA and the simplicity of the V1 features (windowed aggregations), I'd propose using Spark Structured Streaming with a micro-batch trigger of 1 minute.

Why Spark Streaming?

It has a mature ecosystem, a unified API for batch and stream processing (which is excellent for backfills), and its micro-batch model is operationally simpler to reason about for this type of aggregation compared to a pure event-time engine, while still easily meeting the 5-minute SLA.

The streaming job would:

- ❖ Read from all the relevant Kafka topics.
- ❖ Perform stateful aggregations over tumbling windows. For example, to calculate likes in the last hour, it would aggregate counts over 1-hour windows, sliding every minute.
- ❖ Calculate ratios like CTR by joining impression counts with click counts on `content_id` within the stream.
- ❖ The output of this job would be a stream of (`content_id`, `feature_name`, `feature_value`, `timestamp`).

Stage 3: Serving (The Online/Offline Feature Store)

This is a dual-path destination. The output of our Spark job needs to go to two places:

- ***Offline Store:***

  The computed features will be continuously written to a Delta Lake or Apache Iceberg table in S3.

Why?

These formats provide ACID transactions, snapshot isolation, and time travel. This is *perfect* for the offline training job. It can query the feature table as of a specific point in time (`AS OF TIMESTAMP '...'`) to get a consistent snapshot for training, preventing data leakage from the

future. This table is the source of truth for historical features.

- ***Online Store:***

  The features also need to be written to a low-latency key-value store. I would choose Redis or Amazon DynamoDB.

Why?

They are designed for the <10ms lookup latency required by the online ranking model. The key would be `content_id`, and the value would be a hashmap or JSON object of all its features.

The Spark Streaming job would use a `forEachBatch` sink to write the latest feature values from each micro-batch directly to Redis/DynamoDB. This operation must be an `UPSERT`._

This architecture creates a robust system that meets the dual requirements of serving fast online predictions and enabling consistent offline training.

Spark Streaming vs Flink : Trade Off

Interviewer : That's a solid and practical V1. You chose Spark Structured Streaming in a micro-batch mode. A pure streaming engine like Apache Flink is often praised for its low latency and superior event-time handling. Why didn't you choose Flink for this use case? What are the specific trade-offs you made?

Interviewee: That's an excellent question, as it gets to the heart of the design choice. The choice between Spark Streaming and Flink is a classic trade-off between latency, operational simplicity, and ecosystem.

Why I Chose Spark Streaming for V1:

SLA Tolerance: The 5-minute SLA is well within the capabilities of a 1-minute micro-batch. If the requirement had been sub-second, Flink would have been the undisputed choice. For this V1, Spark is "good enough" on latency and excels elsewhere.

Operational Simplicity: The `MERGE` and `UPSERT` logic needed to update features is often easier to express and manage in Spark's micro-batch `forEachBatch` paradigm against a Delta Lake table. You process a small batch of data and merge it into the main table. Flink's continuous operator model requires more careful state management for the

same task.

Unified API for Backfills: Inevitably, we will need to backfill features for all 1 billion content items, either due to a bug fix or a new feature request. Spark's unified API means I can write one piece of logic and run it in batch mode over historical data and in streaming mode for new data. This significantly reduces development and maintenance overhead.

Trade-offs I Made:

- *Higher Latency:* I accepted a slightly higher end-to-end latency (1-2 minutes) compared to what Flink could offer (sub-second). This was a deliberate trade-off because the business requirement allowed for it.

- *Potential for Stragglers:* Micro-batching can sometimes be less efficient at handling late-arriving events across batches compared to Flink's sophisticated watermarking and event-time mechanisms, though Spark's watermarking is quite capable.

In short, for this specific V1, I prioritized operational simplicity and the power of a unified batch/streaming API over achieving the lowest possible latency, as the SLA didn't demand it. If the freshness requirement were to shrink to seconds, I would absolutely re-evaluate and likely pivot this core transformation job to Flink.

Weighted Feature Requirement

Interviewer : Good justification. Now, let's evolve the problem. The ML team is thrilled, but they have a new feature request. They want a time-decayed count for each piece of content. A like from 1 minute ago should have a weight of 1.0, a like from 12 hours ago should have a weight of 0.5, and a like from 24 hours ago should have a weight of 0.25.

How does this new, more complex stateful feature impact your design? Would you stick with your Spark micro-batch architecture?

Interviewee: This is a fantastic question because it moves from simple windowed aggregates to a more complex, continuously evolving state. This requirement significantly strains the simplicity of the micro-batch model and pushes the design firmly into the territory of a pure stream processor.

Impact and Proposed Change:

My V1 Spark micro-batch approach would be inefficient here. To calculate a decayed score, I would have to:

- read the previous state (the score from the last batch),

- decay it based on the time elapsed, and
- then add the new likes.

This is possible, but it's not what Spark's declarative API is optimized for.

This is where I would pivot the core of my transformation engine to Apache Flink.

Evolved Architecture using Flink:

1. **Ingestion:** Remains the same (Kafka).

2. **Transformation (The Core Change):**

I would replace the Spark Structured Streaming job with a Flink DataStream job.

I'd use a `KeyedProcessFunction` in Flink, keyed by `content_id`. This function gives me fine-grained control over state and time.

➢ State: For each `content_id`, the function would store the last calculated decayed score and its timestamp in Flink's managed state backend (e.g., RocksDB).

➢ Processing: When a new `like` event arrives for a `content_id`, the function does the following:

- Retrieves the current state (`last_score`, `last_timestamp`).
- Calculates the time delta: `delta_t = current_event_time - last_timestamp`.
- Applies the decay formula to the old score: `decayed_score = last_score * decay_function(delta_t)`.
- Adds the new like: `new_score = decayed_score + 1`.
- Updates the state with `(new_score, current_event_time)`.
- Emits the `(content_id, new_score)` downstream.

3. **Serving: The sinks remain the same.**

   The Flink job would have two sinks:

   - An `IcebergSink` or `DeltaSink` to write the updated features to the offline data lakehouse table.

   - A `RedisSink` or `DynamoDBSink` to push the updated feature value to the online KV store with very low latency.*

Why Flink is the Right Tool Now:

- *Low Latency:* Flink processes events one by one, allowing it to update and emit the decayed score to Redis in milliseconds, far exceeding the old 5-minute SLA.
- *Efficient State Management:* Flink's state backends are highly optimized for this kind of per-key read/update/write pattern.
- *Expressiveness:* The `ProcessFunction` provides the imperative control needed to implement custom logic like exponential decay, which is cumbersome in a purely declarative, micro-batch framework.

First Run Problem

Interviewer : That's a strong pivot. Let's address a real-world problem. The "cold start" problem. A new video is uploaded. It has no likes, no shares, no impressions. Its features are all zero. The ranking model will likely ignore it. How can your *data pipeline* help solve this, so new content gets a fair chance in the feed?

Interviewee: That's a critical product question that directly influences the data architecture. The pipeline can't just process engagement data; it must also provide the features that help the model make intelligent decisions when engagement data is sparse.

My solution is to enrich the feature set with content-based and author-based features that are available at upload time.

1. *New Event Source:* The content upload service would publish an event to a new Kafka topic, `content_creation_events`, every time a new video is successfully uploaded. This event would contain the `content_id` and `author_id`.
2. *Asynchronous Enrichment Pipeline:* I would create a separate pipeline (it could be another Flink job or an event-driven system using something like AWS Lambda) that listens to this `content_creation_events` topic.
3. *Content Feature Generation*: For each new `content_id`, this pipeline would perform several enrichment steps:
    - Call a video analysis service to extract topics, objects, or concepts from the video frames.
    - Call a text analysis service to extract entities and sentiment from the title and description.

- Extract basic metadata like video duration, resolution, etc.

4. *Author Feature Generation:* The pipeline would then use the `author_id` to look up pre-computed features about the author from another feature table. These features might include:

    - `author_follower_count`
    - `author_historical_avg_engagement_rate`
    - `Author_account_age`

5. Feature Store Integration:

    - Once all these cold-start features are generated, the enrichment pipeline's final step is to write them to the *same online and offline feature stores* we've been discussing, using the `content_id` as the key.

How this Solves the Problem

When the ranking model sees a new `content_id` and queries the online feature store, it won't find zero features.

Instead, it will find a rich set of content and author features. The ML model can then be trained with a rule like:

"If engagement features are below a certain threshold, give more weight to author and content features."

This allows the model to intelligently "bootstrap" new content into the feed, giving it a chance to accumulate the engagement features that will determine its long-term ranking. This turns the data pipeline into an active participant in solving a core product problem.

Case Study 5: Design a Clickstream Analytics Pipeline

Problem Statement

Interviewer: Thanks for joining. Imagine we're a Netflix/YouTube-scale platform. Clients on web, mobile, and smart TVs emit interaction events:

- play,
- pause,
- seek,
- impression renders,
- recommendation clicks.

Peak traffic hits a few million events per second globally; we need

- near-real-time session analytics (<5 minutes lag)
- a reliable daily cut for partner dashboards by T+30 minutes.
- Late and out-of-order events are common
- schemas evolve.

Design the end-to-end system.

Clarifying Questions

Interviewee: Great-before jumping in, I'd like to clarify a few constraints so I don't optimize the wrong thing. What are your guarantees and constraints around identity, privacy, and tooling?

Interviewer:

- Use device_id and session_id everywhere;
- user_id is present for signed-in clients, but TV often lacks it.
- GDPR/CCPA apply: EU data must remain in-region; deletes must be honored within 30 days.
- We prefer a lakehouse on object storage with Parquet/Delta,
- Spark as the main compute.
- We also run Kafka regionally for ingestion.
- For schema, we use Protobuf with a registry, but we've had the occasional breaking change slip through.

Interviewee: Understood. Final two:

- Do you operate multi-region active/active? And beyond session analytics and DAU/watch-time,
- Do we need experiment reads-e.g., A/B cohorts-same day?

Interviewer: Yes, multi-region. Same-day experiment reads are in scope; joins need to be correct, not eventually consistent guesses.

Proposed Solution

Interviewee: Perfect. I'll walk through the design as a narrative: how an event is produced, lands in the lake, transforms into analytics, and is served, while calling out evolution, late data, and governance.

1. Edge & Event Production

- On the edge, clients produce *Protobuf events* with a *unique event_id*, *session_id*, *device_id*, and *monotonic client_ts*.
- We enforce **idempotent**, exactly-once-capable producers (Kafka idempotent producer enabled, retries with the same event_id).
- Events go to regional Kafka clusters, partitioned primarily by *session_id* to keep sessionization friendly.
- We keep partitions generously **sharded** to avoid hot partitions during spikes; MirrorMaker or managed cross-region replication lets us aggregate to a central or per-region lake as needed.

2. Bronze Layer: Raw Ingestion

- From Kafka, streaming ingestion jobs (Spark Structured Streaming or a lightweight autoloader) land raw events into Bronze Delta tables in object storage.
- I write larger files on purpose-tuning trigger intervals and coalescing-to avoid small-file pathologies.
- Bronze is append-only, partitioned by ingestion date/hour, and carries both client_ts and ingest_ts.
- I keep 14–30 days in Bronze for replays.

3. Silver Layer: Normalization & Quality

- A Silver streaming job handles:
    - schema evolution
    - deduplication
    - basic enrichment

4. Schema Evolution

- Schema evolution is constrained by the registry set to *BACKWARD_TRANSITIVE.*
- If a producer pushes a breaking change, the registry blocks it.
- If something slips, the reader detects an unexpected required field and **quarantines that batch** (quarantine = side Delta table with offending partition + error payload) and **pages the on-call**.

5. Deduplication

- For dedupe, I **MERGE** on (event_id) with the earliest seen timestamp-this gives me idempotency across retries and replays.

7. Late & Out-of-Order Events

- Silver also handles late and out-of-order events.
- I set a **24-hour watermark** on client_ts and allow reordering within that window.
- Anything arriving after the watermark goes to a late-events DLQ.
- We run a **nightly backfill** that:
    - re-processes the DLQ
    - rewrites **only the impacted partitions** downstream
- This keeps the real-time path fresh and improves daily cut correctness.

8. Sessionization Logic

- Sessionization = turning a stream of events into coherent sessions.
- It's a stateful streaming step keyed by session_id with a 30-minute gap rule.
- I size the state store (RocksDB) carefully and shard sessions across many partitions.
- State TTL = ~36 hours to account for long TV sessions.

- For heavy features (session boundaries, completion rates), I pre-compute and persist a Gold Session table with one record per session:
    - user/device keys
    - content IDs watched
    - watch_time_seconds
    - completion ratios
    - seek patterns
    - buffering metrics

9. Experimentation / A/B Testing

- There's a daily table of experiment assignments (user_id or device_id → experiment, variant, validity window).
- I make this small and fast: bucket by the key, and Z-order if using Delta.
- The streaming job performs a temporal join - "what assignment was valid at event time?"
- This aligns real-time metrics with the daily cuts.
- If we lack user_id, we fall back to device_id deterministically to avoid variant contamination.

10. Serving Layer

Real-Time Analytics

- Real-time reads query the Gold tables directly via the lakehouse engine.
- I maintain minute-level aggregates (watch time, CTR by content/region/device) as rolling materialized views.

Daily T+30 Partner Dashboards

- For the daily dashboards:
    - freeze the previous day's partitions after the last DLQ backfill
    - produce immutable daily aggregates
    - tag them with the cut's version and lineage

Scale Pain and Skewness

Interviewer: You've covered the happy path. We do run into scale pain. Imagine prime time: *three million events per second globally*. A handful of blockbuster titles create **key skew**; your sessionization state grows hot; and writers spill thousands of tiny files. How do you keep this upright without doubling the bill?

Interviewee: I'd address it in four layers-Kafka, writes, compute skew, and storage maintenance.

Kafka

At Kafka, I'd overshard partitions with headroom, enable compression (lz4 to keep CPU reasonable), and tune linger/batch.size so producers batch efficiently.

For extremely hot titles, I add a light **salting** scheme to partition keys-e.g., session_id hash with a small salt range-so a single hot title can't pin one partition.

Writes

On writes, I increase micro-batch intervals slightly (say, from 5s to 30–60s) to emit bigger files, and I coalesce to target **128–512 MB** file sizes.

In Delta, we schedule frequent *OPTIMIZE* passes with bin-packing to clean up stragglers.

If the platform supports it, I turn on **liquid clustering** or background clustering keyed by content_id and event date. That keeps read performance stable and reduces aggressive Z-ORDER runs.

Compute Skew

For compute skew, sessionization by session_id is usually okay, but blockbuster live streams can create long, chatty sessions.

I shard the sessionization state by (session_id, shard) where shard is derived from a secondary hash of device_id.

Spark's AQE with skew join hints helps downstream aggregations.

Where the sessionization state becomes a true bottleneck, we can offload that step to **Flink**, which handles very large keyed state with better predictability, and keep Spark for the enrichment and batch-ish summarization.

Storage Maintenance

Finally, cost controls: tier older Bronze/Silver to cheaper storage classes; pre-aggregate high-cardinality metrics (e.g., minute-by-minute watch time per title/region) to avoid recomputing from raw during adhoc queries; and cap Z-ORDER cadence to only after heavy backfills.

Schema Drift

Interviewer: Good. Now let's break something. A mobile client deploy accidentally makes a previously optional field-say, bitrate_kbps-required in the Protobuf. Older TVs still send the optional version. Some consumers crash because the reader expects the new shape. Walk me through the blast radius and your recovery plan. I care about both real-time and the daily cut.

Interviewee: The registry is the first line: the new schema with "required" should have been rejected under backward-transitive compatibility. If not, the Bronze reader sees two shapes: one with required bitrate and one without.

My *Bronze job* reads at the **union schema** level-compatible fields unioned-and writes both variants into Bronze, tagged with a schema_version in metadata.

The *Silver job* is stricter; it expects the optional form and has explicit parsing. When it sees the "required" variant from mobile, it doesn't try to coerce; it **routes those micro-batches to quarantine** with metadata: producer app version, topic, partition, offsets, first/last timestamps.

That alert pages the on-call and posts to the schema-change Slack channel. Real-time dashboards degrade gracefully: fields tied to bitrate-dependent metrics are marked as **unknown/partial coverage**, and we annotate the UI with a banner and coverage percentage.

Meanwhile, we hotfix the Silver reader to handle both shapes-treat missing bitrate as null with a default semantics-and **replay from Kafka offsets** that were quarantined.

Because Kafka is the source of truth and our Silver writer is idempotent (MERGE on event_id), the replay corrects Gold tables without duplication.

For the daily cut, we delay the finalize step if necessary or publish the partition with a **"corrected later"** tag and run the DLQ backfill before T+30 if we can. If we miss T+30, we publish the known-good subset and a corrections job updates partner aggregates with a change log-transparency beats silent drift.

Exactly Once Guarantee

Interviewer: Let's talk **exactly-once semantics**. Lots of teams claim it. What is your precise contract end-to-end?

Interviewee:

I define **exactly-once processing** as:

No double-counting and no missing events anywhere from the producer all the way to the Gold layer - even if there are retries or replays.

How the Pipeline Guarantees This

1. Producers → Kafka

- Producers send events to Kafka using **idempotent writes**.
- Each event has a **stable event_id**, so retries produce the **same identifier**, not a new event.
- This prevents duplicates at the ingestion point.

2. Bronze Layer (Raw Storage)

- Bronze is **append-only**, so duplicates can appear.
- This is expected - we don't enforce uniqueness here.
- Bronze simply captures **everything exactly as it arrived**.

3. Silver Layer (Deduplication & Normalization)

- Silver performs a **MERGE on event_id** using "**first event wins**" logic.
- All duplicate copies of the same event_id collapse into **one clean record**.
- This ensures **unique events** before any downstream processing.

4. Gold Layer (Aggregates & Derived Tables)

- When computing sessions or metrics, each **event_id only participates once**, because Silver already guarantees uniqueness.
- For derived tables, I make the writes **idempotent as well** by using a MERGE key such as:
    - **(session_id, sequence_no)**

    - **(video_id, minute_bucket)**
    - or a **checksum** of contributing events to detect changes

This ensures that if we recompute or replay data, Gold tables do not double-count or create inconsistent records.

5. Safe Replays & Backfills

- Because both Silver and Gold are written with **idempotent MERGE patterns**, we can safely:
    - replay data from old Kafka offsets
    - backfill from the DLQ
    - reprocess historical partitions

…and the output will stay consistent.

6. What I Avoid

- I avoid **at-least-once sinks** where duplicates would be impossible to clean later - e.g., appending blindly to Parquet without a unique key.

Late Arriving Data

Interviewer: And **late data**? Your watermark is 24 hours. A TV batch comes five days late because someone unplugged the router. How do you avoid corrupting partner numbers?

Interviewee:

If an event arrives **after the 24-hour watermark**, it is sent straight to the **DLQ (dead-letter queue)** instead of entering the real-time pipeline.

To keep reporting consistent:

1. Daily Reports Are Versioned

- The initial report is the **T+0 cut** (whatever arrived on time).
- Any later corrections are reflected in a **T+N corrected** version.

This makes data lineage transparent.

2. Nightly Rolling Backfills

Every night, we run a **rolling 7-day backfill** job that:

- reads the DLQ
- finds which partitions are impacted
- uses Delta's file-level statistics to narrow the work
- **recomputes only the affected partitions**, not the whole dataset
- writes **compensating deltas** to downstream aggregates

Example:
➡️ "+1200 watch seconds for content X on 2025-09-03 (EU region)."

This ensures accuracy without rewriting everything.

3. Partner Dashboards Use the Corrected View

- Dashboards default to the **updated T+N corrected dataset**.
- But we still **preserve the original T+0 snapshot** for auditing and compliance.

4. Outcome

This approach provides:

- **Fresh numbers in real time**, without constant fluctuation
- **Eventual accuracy** as late data is reconciled
- **Stable partner dashboards** that don't whiplash during the day

Iceberg vs Delta: Trade off

Interviewer: Suppose we pick **Iceberg** instead of Delta. What changes?

Interviewee: Conceptually, the pipeline stays the same - **Bronze/Silver/Gold**, compaction, schema evolution, GDPR deletes, and metadata-driven reads all still apply.
But a few implementation details differ with Iceberg.

1. GDPR Deletes & Rewrites

With Iceberg, I rely on:

- **Position deletes** – remove specific rows using file + row position
- **Equality deletes** – delete rows matching a predicate (e.g., user_id = X)

Both enable GDPR compliance and incremental rewrites without rewriting entire partitions.

2. Partitioning & Hidden Partitions

- Iceberg uses **hidden partitioning**, which avoids “partition spec drift.”
- This means partition evolution is cleaner and you don't have to expose partition columns in queries.
- It handles time-based partitions more gracefully than manual partition columns.

3. File Sorting & Clustering

- Z-ORDER is a Delta-specific optimization, so we don’t use it here.
- In Iceberg, I’d choose:
    - sorted writes
    - clustering strategies
    - distribution + ordering hints in Spark

This improves scan performance and filtering.

4. Query Engine Behavior

Iceberg works best with engines that support strong predicate pushdown, such as:

- Trino / StarRocks
- Spark (EMR/Synapse/Databricks)
- Athena with Iceberg connector

These engines prune files aggressively using Iceberg metadata.

5. MERGE & Idempotency Discipline

- Iceberg supports MERGE operations, but performance varies across engines.
- The real key is **idempotent MERGE logic**, compaction cadence, and consistent data contracts.
- As long as those are enforced, the table format is secondary.

So to sum it up, Switching to Iceberg changes how we optimize storage (deletes, partitions, sorting), but the core architecture and data engineering principles remain identical.

Case Study 6: Design Add to Purchase Conversion

Problem Statement

Interviewer : Thanks for joining today. Let's dive right in.

Here's the scenario:

We are a large social media company. We serve billions of ad impressions daily, and we need to attribute user conversions (like a purchase or an app install) back to the ads they've seen or clicked.

Your task is to design the data pipeline and model that powers our core attribution reporting dashboard. This dashboard needs to show, for any given ad campaign,:

- the number of impression
- Clicks
- attributed conversions,

updated on a daily basis.

Let's start with your high-level approach.

Clarifying Questions

Interviewee: Great, this is a classic and interesting problem. Before I propose a solution, I need to ask a few clarifying questions to understand the constraints and requirements better.

1. **Volume & Velocity:** You mentioned "billions" of impressions. Could you be more specific? Are we talking 1-10 billion per day, or closer to 100 billion? Also, what's the rough volume of clicks and conversions? This will inform my choice of tooling and scaling strategy.

2. **Latency:** The requirement is a "daily" updated dashboard. Does this mean a 24-hour latency is acceptable? For example, are we okay with seeing yesterday's data this morning?

3. **Data Sources:** Where do the impression/click logs and conversion events originate? Are impressions and clicks coming from a real-time stream (e.g., Kafka from ad servers)? Are conversions coming from a different system, perhaps with a delay (e.g., a batch feed from partners)?

4. **Attribution Logic:** What's the attribution window? For example, is a conversion attributed to a click if it happens within 7 days? And what's the model? Last-click attribution?

5. **Data Schema:** Do the events have a stable schema? Do we need to plan for schema evolution? Are there any nested fields, like JSON blobs, that we need to parse?

Interviewer: Excellent questions. Let's scope this.

1. **Volume**: Let's say 10 billion impression events, 100 million click events, and 1 million conversion events per day.

2. **Latency**: Yes, for this V1, a 24-hour latency is acceptable. The executive dashboard is reviewed each morning.

3. **Data Sources**: Impressions and clicks are streamed in real-time from our ad servers via a Kafka topic. Conversion events come from a separate system. They are uploaded as hourly batch files (e.g., CSVs) to a cloud storage bucket.

4. **Attribution Logic**: Let's use a simple 7-day last-click attribution model. A conversion is attributed to the most recent click that occurred within the 7 days prior to the conversion.

5. **Schema**: The Kafka stream schema is relatively stable but new fields can be added by the ad server team with a week's notice. The conversion files have a fixed schema. Both contain user identifiers.

Given this information, walk me through your proposed V1 architecture.

Proposed Solution

Interviewee: Perfect, that provides a clear picture. I'll structure my proposed solution into four stages: **Ingestion, Storage, Transformation, and Serving.**

My core assumption is that we're operating in a major cloud environment (AWS, GCP, Azure) and can leverage its managed services for efficiency and scalability.

Stage 1: Ingestion

Impressions & Clicks (Streaming):

We have a **Kafka stream**. I would use a managed ingestion service like **AWS Kinesis Data Firehose** or a **simple Spark Streaming/Flink job** that reads from the Kafka topic.

The primary goal here is to land the raw data reliably into our Data Lake. This job will batch the streaming data into small files and write them to cloud storage (e.g., S3) in a raw data zone.

Conversions (Batch):

Since these are hourly files dropped into a bucket, we can use an event-driven trigger (like an **S3 Event Notification** triggering a **Lambda function**) to log the file's arrival in a metadata database and trigger our downstream processing.

Stage 2: Storage (The Data Lake)

I'd structure our S3 bucket into zones:

- raw,
- processed
- aggregated

File Format: I'll land the raw data as is, but for the processed layer, I will convert and store all data in **Apache Parquet format**.

Why Parquet? It's a columnar format, which is highly efficient for the kind of analytical queries we'll be running (selecting specific columns like user_id, timestamp, campaign_id). It also offers excellent compression, reducing storage costs.

Partitioning: I will partition the data heavily to optimize query performance.

For impression/click data, I'd partition by date:

s3://our-bucket/processed/impressions/dt=YYYY-MM-DD/.

For conversions, the same:

s3://our-bucket/processed/conversions/dt=YYYY-MM-DD/.

This is crucial for the transformation job, as it will allow us to avoid full table scans.

Stage 3: Transformation (The Core ETL/ELT Job)

This is where the attribution logic happens. I would use **Apache Spark** running on a managed cluster (like EMR, Databricks, or Dataproc) for this large-scale data join.

Orchestration: An orchestrator like Apache Airflow or Dagster would trigger this Spark job once a day, after we're confident all of the previous day's data has landed.

The Job Logic:

- ***Read Data:*** The job will read the last 8 days of impression/click data (to cover the 7-day lookback window for conversions that happened on D-1) and the last 1 day of conversion data from the processed zone. Partitioning makes this read very efficient.

- ***Join:*** It will perform a left join from the conversions dataset to the clicks dataset on user_id.

- ***Filter & Logic:*** The join condition will also filter for clicks that happened before the conversion but within the 7-day window (click_timestamp <= conversion_timestamp and click_timestamp >= conversion_timestamp - 7 days).

- ***Attribution***: We'll use a window function (ROW_NUMBER() OVER (PARTITION BY conversion_id ORDER BY click_timestamp DESC)) to find the last click for each conversion and filter for it (where row_number = 1).

- ***Aggregation***: Finally, the job will aggregate the results by campaign_id, date, counting impressions, clicks, and attributed conversions.

Stage 4: Serving

The final aggregated data from the Spark job will be written to an aggregated zone in **S3 (again, as Parquet).**

From there, we'll load it into a columnar data warehouse like Snowflake, BigQuery, or Redshift. This warehouse is optimized for the low-latency, high-concurrency queries generated by BI tools and dashboards.

Our BI tool (e.g., Tableau, Looker) will connect to this data warehouse to power the daily attribution report.

This V1 architecture is robust, scalable, and uses industry-standard components for a daily batch workload.

S3 Dump vs Direct Ingestion : Trade Off

Interviewer : That's a solid, well-structured V1. Let's dig deeper.

You chose to land the streaming data as small files in S3 first. Why not stream it directly into a system like Delta Lake or a Kafka-enabled database to simplify the process? What are the trade-offs of your approach?

Interviewee: That's a great point. The trade-off is between simplicity/cost and latency/complexity.

My approach of landing raw files first (often called the "landing zone" or "raw zone") has a few key advantages for a V1:

- **Decoupling & Resilience:** Ingestion is decoupled from transformation. If the downstream Spark job fails, the raw data is still safe and sound in S3. We can simply re-run the job. We never lose the source of truth. Streaming directly into a structured table means an ingestion failure could be more complex to debug and replay.

- **Cost-Effectiveness:** S3 is the cheapest form of storage. For 10 billion events per day, storing the raw, compressed source data is very cost-effective for long-term audit and reprocessing needs.

- **Simplicity of Ingestion:** Services like Kinesis Firehose are fully managed and incredibly simple to set up. They handle batching, compression, and delivery to S3 out-of-the-box, which is a low-engineering-overhead way to get started.

The primary disadvantage, as you hinted, is the introduction of micro-batch latency. We're not truly real-time. But since the business requirement is a daily dashboard, this trade-off is perfectly acceptable for V1.

Using a technology like Delta Lake with streaming sinks is a more advanced pattern. It would offer ACID transactions on the data lake and simplify updates, which becomes very relevant when we talk about late-arriving data or GDPR, but it adds complexity to the ingestion part of the pipeline. For V1, I'd stick with the simpler, more resilient pattern.

Changing Latency Requirements

Interviewer: Okay, that makes sense. Now, let's stress-test your design. The marketing team loves the dashboard, but now they want to see attribution results with a maximum latency of one hour. The daily batch job is no longer acceptable.

How would you evolve your architecture to meet this new requirement? What are the biggest challenges you foresee?

Interviewee: An excellent evolution of the problem. This fundamentally shifts the architecture from batch to streaming. Here's how I'd adapt the design.

The biggest challenge is performing the stateful join (the 7-day lookback) in a streaming context.

Evolved Architecture:

Ingestion & Storage:

We can no longer just dump files into S3 and wait. I would replace the Parquet files in the processed zone with Delta Lake or Apache Hudi tables.

Why Delta/Hudi? They provide ACID transactions and snapshot isolation on top of the data lake. More importantly, they have excellent support for streaming sources and sinks with Spark Structured Streaming, which is what I'll use for transformation. This allows us to continuously write to and read from these tables.

The Kafka ingestion process will now be a continuous Spark Structured Streaming job that reads from the impression/click Kafka topic and writes to a clicks_delta table.

The conversion file processing will also become a streaming read. We can use Spark's cloudFiles source, which can incrementally process new files as they land in S3 and append them to a conversions_delta table.

Transformation (The Core Change):

Spark Structured Streaming : I would replace the daily batch Spark job with a long-running Spark Structured Streaming job.

The Join Logic: The core challenge is the join. A conversion event arriving now needs to be joined with clicks from up to 7 days ago. This is a classic stream-stream join with a time window. Spark Structured Streaming supports this directly using stream-stream joins with watermarking.

State Management: Spark will have to maintain the state of clicks for the last 7 days to perform this join. This is a significant operational consideration. The state will be stored in a fault-tolerant way (e.g., on HDFS or S3) using checkpoints. Managing this state's size is the primary challenge. We'd need to monitor it closely and ensure our cluster has enough resources.

Serving:

The output of the streaming join job would be another Delta table (hourly_attributions_delta).

From here, we have two options for the dashboard:

Option A (Push):

A micro-batch job runs every hour, reads the newly aggregated data from the Delta table, and MERGEs it into the data warehouse (Snowflake/BigQuery).

Option B (Direct Query):

If the data warehouse supports it (like Databricks SQL or BigQuery), we could potentially point the dashboard directly at the final Delta table, eliminating another data movement step.

Biggest Challenges:

State Size & Cost:

Maintaining a 7-day state of clicks for billions of users is memory and I/O intensive. The cost of the streaming cluster will be significantly higher than the daily batch job.

Late-Arriving Data:

Watermarking helps manage late data to a degree, but if a conversion event arrives an hour late (past the watermark), it might be dropped. We'd need a strategy for handling this, perhaps a separate, less frequent batch job to "correct" any missed attributions.

Operational Complexity:

A long-running streaming job is operationally more complex than a daily batch job. We need robust monitoring, alerting, and automated recovery mechanisms.

GDPR: Right to be Forgotten

Interviewer: Good. You mentioned state size as a major challenge. Let's talk about GDPR. A user in the EU invokes their "right to be forgotten." We need to delete all of their impression, click, and user data from our entire system.

How does this request impact your batch V1 design versus your streaming V2 design? How would you implement this deletion?

Interviewee: This is a critical governance requirement and it significantly highlights the limitations of using immutable Parquet files, which was my V1 choice.

Impact on V1 (Immutable Parquet):

Deleting a specific user's data from a giant collection of Parquet files is extremely difficult and computationally expensive. Parquet files are immutable. To delete data, you have to:

1. Read all the partitions that might contain the user's data.
2. Filter out the records for that user_id.

3. Rewrite the entire partition as new Parquet files.
4. Delete the old files.

Implementation:

This would be a painful, expensive batch job. We would need to maintain a list of users to be deleted and run a periodic "compaction/deletion" Spark job that rewrites terabytes of data. It's slow, costly, and error-prone. It's a major weakness of that simple design.

Impact on V2 (Delta Lake / Hudi):

This is precisely where formats like Delta Lake, Hudi, or Iceberg shine. They are designed for this. They provide DELETE, UPDATE, and MERGE capabilities on data lake storage.

Implementation: The process becomes a simple DML statement.

```
-- For Delta Lake
DELETE FROM impressions_delta WHERE user_id = 'user_to_be_deleted';
DELETE FROM clicks_delta WHERE user_id = 'user_to_be_deleted';
```

Behind the scenes, Delta Lake handles the file management. It identifies the files containing the user's data, rewrites them without the user's records, and updates its transaction log to point to the new files, atomically. The old files are later cleaned up by a VACUUM command.

This is far more efficient, reliable, and auditable. The streaming architecture using these file formats is inherently better suited to handle GDPR and other data privacy requirements. This requirement alone is often a strong justification for moving away from plain Parquet to a transactional table format on the lake.

Interviewer: Your answers on V2 and GDPR are very clear. It seems you've dealt with these issues before. Let's continue to add some real-world complexity.

Data Quality Checks

1. On Data Quality: Your pipeline design assumes the incoming data is clean. What happens if a batch of conversion files arrives with 50% of the `user_id` values being `null` or malformed? How would you design a data quality framework to automatically detect this, prevent bad data from polluting your attribution results, and alert the source team?

Interviewee: That's a critical operational concern. A pipeline without data quality checks is a liability. I would implement a multi-layered Data Quality (DQ) framework.

My approach would be to Detect, Quarantine, and Alert.

Detection:

I'd integrate a DQ tool directly into my pipeline, likely as a step in my Airflow DAG right after ingestion. I'd use a library like Great Expectations or Deequ (for Spark). I would define a suite of data quality checks as code, including:

- `expect_column_values_to_not_be_null('user_id', mostly=0.98)`: This would check that at least 98% of `user_ids` are not null. A 50% null rate would immediately fail this check.
- `expect_column_values_to_match_regex('user_id', '^[0-9a-fA-F]{8}-...$')`: To ensure the ID format is correct.
- `expect_table_row_count_to_be_between(min, max)`: To detect empty or unusually small files.

Quarantine:

If a batch fails these critical DQ checks, the pipeline must not process it further into the main `processed` zone. Instead, the Airflow task would be configured to divert the data. The failed batch of files would be moved to a separate "quarantine" or "bad_data" location in S3 (e.g., `s3://our-bucket/quarantine/conversions/dt=YYYY-MM-DD/`). This prevents pollution of our core dataset while preserving the bad data for debugging.

Alerting:

The Airflow DAG would immediately fail the task and trigger an alert via an `on_failure_callback`. This alert would go to two places:

- The on-call data engineering PagerDuty/Opsgenie for immediate investigation.
- A high-priority Slack channel that includes the product manager and engineers from the source team that produces the conversion data. The alert must contain the file path, the specific DQ check that failed, and a summary statistic (e.g., "52% null user_ids found in file X, expected <2%").

Ultimately, this is a defensive measure. The long-term solution is establishing a Data Contract with the source team, enforcing these rules at the point of data creation, not just at ingestion.

Scalability and Performance

Interviewer:

2. On Scalability & Performance: Your core attribution logic joins conversions against a 7-day window of clicks. A small number of hyper-active users or bots might generate millions of clicks,

leading to severe data skew on the user_id join key. How would this skew manifest as a problem in your Spark job, and what specific techniques would you use to mitigate it?

Interviewee: Data skew is a classic Spark performance killer.

Manifestation of the Problem:

In the Spark UI, I would see the job stuck on a single task for a very long time, while all other tasks in the stage have completed. One executor's CPU would be pegged at 100%, processing the massive partition associated with the skewed user_id key, while other executors sit idle. This leads to unpredictable and often failed job runs._

Mitigation Technique:

Salting the "Big" Table:

On the large clicks dataset, I would transform the user_id by appending a random integer within a certain range (e.g., 0-9). So, user_id_123 becomes user_id_123_0, user_id_123_1, ..., user_id_123_9. This effectively splits the skewed key into 10 distinct, smaller sub-keys.

Exploding the "Small" Table:

On the smaller conversions dataset, I would duplicate each row for every possible salt value. I'd create an array of salt keys (from 0 to 9) and explode the DataFrame, so each conversion row now has 10 rows, one for each salted key.

The Join:

Now, I can join conversions_df and clicks_df on salted_user_id. The workload for the previously skewed user_id is now distributed across 10 different keys, allowing Spark to process them in parallel across multiple tasks and executors.

This adds a little complexity but provides massive performance gains and predictability for skewed joins. Additionally, modern Spark versions with Adaptive Query Execution (AQE) can sometimes handle moderate skew automatically, but for severe, known skew like this, manual salting is the most robust solution.
Handling Late Arriving Data

Interviewer:

3. On Late Arriving Data (Deeper Dive): What if a whole batch of conversion data from a partner is delayed by 36 hours? It's far too late for the watermark. How do you design a robust and automated reconciliation process?

Interviewee: You're right, watermarking protects against minor delays but not systemic, multi-day outages. For this, I'd design a complementary batch reconciliation job.

Architectural Design: The streaming pipeline remains the primary path for low-latency results. The *reconciliation job* is a separate Airflow DAG scheduled to run once daily (e.g., at midnight).

Job Logic: This batch Spark job's logic is designed to "correct history."

1. It ignores the stream's watermark and looks at a wider, fixed window. For example, it would re-process all conversions that arrived in the last 3 days (`now() - 3 days`).

2. It performs the same core attribution logic: joining these recent conversions against the full 7-day click history.

3. The crucial step is how it writes the data. It would use a `MERGE` statement (supported by Delta Lake, Hudi, and data warehouses) against the final attribution table.

Idempotency & Cost: This `MERGE` operation is idempotent. Re-running it on the same late data won't create duplicates. It's also efficient because it only re-processes a small slice of recent data (3 days, not the entire history), keeping costs manageable. This dual-pipeline (streaming + batch reconciliation) approach provides both speed and accuracy, which is a common pattern in robust data systems.

Backfilling Pipelines

Interviewer:

4. On Business Logic Changes & Backfills: The marketing team now wants to test a new attribution model: first-click attribution with a 30-day window. How would you support both models and run a 6-month historical backfill efficiently?

Interviewee: An excellent question that gets at the heart of pipeline agility. My strategy would be to decouple the expensive join from the business logic application.

Decoupling Logic:

Instead of one monolithic job, I would break the transformation into two stages:

- **Stage 1** (Pre-computation/Join): This job's only purpose is to find all *potential* click-to-conversion matches within the largest possible window (30 days). It joins conversions with clicks on `user_id` and the time window. The output is a wide, denormalized, user-level table: (`conversion_id`, `user_id`, `conversion_time`, `click_id`, `click_time`, `campaign_id`, ...). This is the most computationally expensive step.

- **Stage 2** (Logic Application): I would have separate, lightweight jobs that run downstream from Stage 1.

    - last_click_model_job: Reads the pre-computed data and applies the 7-day last-click window function.
    - first_click_model_job: Reads the same pre-computed data and applies the 30-day first-click window function. This is far more efficient as we perform the expensive join only once.

Efficient Backfill:

- I would create a dedicated, parameterized Airflow DAG for the backfill. It would take start_date, end_date, and model_name as parameters.
- To avoid disrupting the production pipeline, this DAG would run on a separate, auto-scaling compute cluster (e.g., a temporary EMR cluster or Databricks job cluster) that spins up for the job and shuts down after.
- Crucially, the backfill job would write its output to a *new, separate table* (e.g., attribution_model_comparison_2023_h1), not the production dashboarding table. This isolates the backfill, allowing the data science and marketing teams to validate the results in a sandbox environment before deciding to productionize the new model.

Cost Management

Interviewer:

5. On Cost Management: Your streaming solution with a 7-day stateful join will be expensive. How would you cut cloud compute costs by 30%?

Interviewee: A 30% cost reduction requires a multi-pronged approach, moving from simple tuning to architectural changes.

Compute & Configuration Tuning (Low Hanging Fruit):

- Use Spot/Preemptible Instances: For a streaming job, we can't use 100% Spot instances due to the risk of interruption. However, we can use a mix, like 70% Spot and 30% On-Demand, for the worker nodes. This can yield significant savings.

- Instance Right-Sizing: Profile the job to see if it's CPU- or memory-bound. Stateful streaming is often memory-bound. I'd switch from general-purpose instances to memory-optimized instances (like AWS R-series), which have a

better price-to-RAM ratio.

- Shuffle & State Optimization: I'd tune Spark's shuffle partitions (spark.sql.shuffle.partitions) and investigate RocksDB for state management, which can spill state to disk more efficiently than the default in-memory store, allowing us to use instances with less RAM.

Data Layout Optimization:

- Z-Ordering/Clustering: On the Delta Lake tables for clicks, I would apply Z-Ordering on user_id and click_timestamp. This co-locates related data on disk, dramatically improving data skipping and reducing the amount of data that needs to be read and held in the state store.

Architectural Change (Most Impactful):

- Hybrid State Management: Instead of keeping all 7 days of click state in Spark's memory, I would implement a hybrid approach. The most recent 24 hours of click data-which is the hottest and most likely to be needed for attribution-would be maintained in the Spark state. For clicks older than 24 hours, the job would perform a targeted lookup against the Delta Lake table, which is highly optimized for this with Z-Ordering. This dramatically reduces the size of the in-memory state, which is the primary driver of cost in stateful streaming jobs.

Idempotency

Interviewer:

6. On Idempotency: How do you design your transformation logic to be idempotent, ensuring that a duplicated event doesn't lead to double-counting?

Interviewee: Idempotency is crucial for data accuracy. My strategy relies on having a unique business key for each event and using a transactional sink.

Unique Event ID: I will assume (and enforce via data contracts) that every impression, click, and conversion event has a globally unique event_id.

Batch Deduplication: In the batch V1 architecture, before processing the data, I would add a simple dropDuplicates(['event_id']) step in the Spark job. This is a straightforward way to handle duplicates within a given batch.

Streaming Deduplication with Watermarking: In the streaming V2 architecture, Spark Structured Streaming provides built-in deduplication. This tells Spark to use the

watermark to manage the state required for deduplication. It will remember `event_id`s it has seen within the time window and discard any duplicates that arrive.

Transactional Sink (The Ultimate Guarantee): The most robust method, especially for the final table, is to leverage the `MERGE` capability of the sink (Delta Lake, Hudi, or a Data Warehouse). When writing the final attributed conversions, I would use the `conversion_event_id` as the merge key. If a conversion is processed a second time, the `MERGE` operation will see that the key already exists and will perform an `UPDATE` rather than an `INSERT`, preventing any double counting.

Access to Raw Data

Interviewer:

7. On Data Modeling for Analytics: A data scientist wants to build a model and needs access to the raw, joined, user-level data, not the aggregated campaign data. How would you modify your serving layer?

Interviewee: This is a common and important requirement. A data platform should serve multiple personas. My design already accounts for this by materializing the intermediate data.

Expose the Intermediate Table: The aggregation step that creates the campaign dashboard data is the *last* step in my transformation. The table immediately before that step-the result of the expensive join and attribution logic-is exactly what the data scientist needs. It contains one row per attributed conversion, with all the user, click, and conversion features. I would name this table `attributed_conversions_user_level` and materialize it as a first-class citizen in our Data Lakehouse (e.g., as a Delta table).

Data Cataloging & Access: I would ensure this new table is properly registered in our data catalog (e.g., AWS Glue Data Catalog, Unity Catalog) with a clear description, column definitions, and ownership information.

Serving Layer:

- For BI Analysts: They continue to query the `campaign_attribution_daily` aggregated table in the data warehouse. It's fast, simple, and serves their needs.
- For Data Scientists: They are given read access directly to the `attributed_conversions_user_level` Delta table. They can use Spark, Python libraries (like DuckDB with Arrow), or Presto/Trino to query this granular data for feature engineering and model training.

This layered approach serves both analytics and data science use cases efficiently without duplicating computation or storage.

Schema Evolution

Interviewer:

8. On Schema Evolution: The click event Kafka topic adds a new optional field: `user_geo_location`. Walk me through the exact steps to make this field available for analysis without downtime.

Interviewee: Absolutely. Here is the end-to-end process, which should be seamless with the right tooling.

> *Schema Registry:* The upstream team registers the new Avro schema for the Kafka topic. Since `user_geo_location` is an optional field (i.e., it has a default value, like `null`), this is a backward-compatible change. This is the most critical step. As long as compatibility is maintained, consumers won't break.
>
> *Spark Streaming Job:* My Spark Structured Streaming job, which reads from Kafka, uses an Avro deserializer that integrates with the Schema Registry. On its next micro-batch, it will fetch the latest schema, see the new field, and automatically add it as a new column to the DataFrame it's processing. No code changes are required in the Spark job itself.
>
> *Delta Lake Sink:* The streaming job writes to our `clicks_delta` table. To allow the table schema to be updated automatically, I must have the write stream configured with the option `.option("mergeSchema", "true")`. When the first batch of data with the new field arrives, Delta Lake will see the new column in the DataFrame and automatically add it to the table's schema in its transaction log.
>
> *Downstream Consumers:*
>
> - The downstream attribution job will now see the `user_geo_location` column in the `clicks_delta` table and can be modified to include it in the final `attributed_conversions_user_level` table. This is the only point where a minor code change might be needed to propagate the field.
> - The BI tool's data source view on the data warehouse will need to be refreshed to pick up the new column after it has been added to the final table.

The entire process, from registry to data lake, is zero-downtime and fully automated, which is a key benefit of using a schema-aware ecosystem like Avro, Schema Registry, and Delta Lake.

Monitoring & Observability

Interviewer:

9. On Monitoring & Observability: What are the three most critical metrics you would monitor to ensure the health and accuracy of this pipeline?

Interviewee: To ensure operational excellence, I'd focus on three pillars of observability: freshness, quality, and performance.

Freshness (Data Latency):

- Metric: The delta between the current wall-clock time and the latest event timestamp processed from the source (`max(event_timestamp)`).
- Tooling: I would have my Spark job calculate this metric for every micro-batch and push it to a time-series database like Prometheus. A Grafana dashboard would visualize this latency, and an alert in Alertmanager would fire if the latency exceeds a threshold (e.g., "Data is more than 15 minutes stale").

Quality (Record Validity):

- Metric: The number of records dropped or quarantined per batch due to failing data quality checks (like `null user_ids`).
- Tooling: Great Expectations automatically generates Data Docs (HTML reports) on every run. For alerting, I would configure the Airflow DAG to fail if the validation step fails, triggering a PagerDuty alert. I would also log the count of bad records to a monitoring system like Datadog to track trends over time.

Performance (Job Throughput & Duration):

- Metric: For the streaming job, it would be `inputRowsPerSecond` and `processingTimePerBatch`. For the batch reconciliation job, it would be `job_run_duration`.
- Tooling: Spark's native web UI is great for live debugging. For long-term monitoring, I would configure Spark's metrics system to push these metrics to a service like Datadog or Prometheus. An alert would be set to detect significant regressions, for example, if batch processing time suddenly doubles, indicating a performance issue or data skew.

Security and PII

Interviewer:

10. On Security & PII: The click and conversion events contain user_id, which is PII. How would you design access control to allow analysts to see campaign performance but restrict access to user-level data?

Interviewee: This requires a defense-in-depth strategy using Role-Based Access Control (RBAC) at both the storage and compute layers.

Storage Layer (S3 / Data Lake):

- I would use IAM roles and bucket policies to enforce access.
- analyst_role: This role would have List and Read access *only* to the aggregated prefix in S3 (e.g., s3://our-bucket/aggregated/). It would be explicitly denied access to the raw and processed zones containing PII.
- data_scientist_role: This role would have read access to the processed and aggregated zones, as they need the granular data for modeling.
- pipeline_role: The service role used by the Spark jobs would have read access to the source zones and write access to the destination zones.

Warehouse / Query Layer (Snowflake / BigQuery / Databricks SQL):

- This is where we can implement more granular controls.
- Role-Based Views: I would create a database analyst role. This role would not have SELECT permission on the base tables (attributed_conversions_user_level). Instead, I would create a database view, campaign_attribution_vw, on top of the aggregated data and grant the analyst role SELECT permissions *only on this view*. This ensures they can never accidentally query the PII.
- Column-Level Security: For more advanced use cases, I would use the warehouse's built-in column-level security or dynamic data masking features. I could create a policy that says: "If the user's current role is analyst, return a hashed or null value when they select the user_id column. If the role is data_scientist, return the actual value."