Data Engineering Interview: Questions & Answers.

Data Engineering roles are in high demand globally, with hundreds of thousands of open positions. Employers consistently seek a mix of strong SQL and programming skills, big data pipeline experience, and the ability to design scalable systems. For example, an analysis of 2025 job postings found SQL mentioned in 61% of listings, Python in 56%, and cloud data warehouse or lake technologies in 78%. Core responsibilities include building reliable **ETL/ELT pipelines**, optimizing data storage (e.g. in **warehouses/lakehouses**), ensuring data quality, and collaborating across teams to deliver data for analytics and machine learning.

Streaming & Event-Driven (Kafka, Flink, Beam)

Focus: Real-time data processing architectures and tools. Covers messaging systems (Kafka, Kinesis) and stream processing frameworks (Apache Flink, Spark Structured Streaming, Apache Beam). Key concepts include event time vs processing time, watermarks, exactly-once processing, windowing, and differences between Lambda and Kappa architecture. Expect design questions about handling late data or scaling a stream.

Why It Matters: Streaming data pipelines are critical for use-cases needing low latency (e.g., user activity tracking, IoT sensor processing). Interviewers assess whether you grasp the challenges of unbounded data: out-of-order events, fault tolerance, and maintaining state in streams.

---

26. **(Easy)** What is Apache **Kafka** and what are its main components?

Answer:

- Apache **Kafka** is a distributed event streaming platform (essentially a publish-subscribe message broker) designed for high-throughput, fault-tolerant ingestion of streams of events (messages)datacamp.com. It's commonly used for building real-time data pipelines and streaming apps.
- Main components:
  - **Topic:** A category or feed name to which records are published. Topics are partitioned for scalability.
  - **Producer:** An application that publishes (writes) messages to Kafka topicsdatacamp.com.
  - **Consumer:** An application that subscribes to (reads) messages from Kafka topics360digitmg.com.
  - **Broker:** A Kafka server (node) that stores message data and serves producers/consumers. Kafka cluster consists of multiple brokers.
  - **Partition:** Each topic is split into partitions. A partition is an ordered, immutable sequence of messages (with offset numbers). Partitions allow parallelism and distribution across brokers.
  - **Consumer Group:** A set of consumers working together, where each message in a partition is consumed by exactly one consumer in the group (enables scalable and load-balanced consumption)github.com.
  - **Zookeeper** (in older Kafka, now optional with Kafka's own quorum): coordinates the cluster, keeps track of brokers, topics, and consumer group offsets.
- Data is retained on brokers for a configurable time or size (e.g., 7 days retention), not automatically deleted upon consumption (which is difference from traditional MQ).
- Kafka guarantees ordering of messages within a partition and allows horizontal scaling by increasing partitions. It's designed to handle high volume (millions of messages per second) with sequential disk writes.
- Use cases: log collection, metrics, user activity events, streaming ETL, decoupling services with event-driven patterns.

What interviewers look for: Basic understanding of Kafka's role and architecture. Listing components (topic, partition, broker, producer, consumer, consumer group) is key. They also expect that you know Kafka stores streams (not a traditional DB, not like HTTP API).

Common pitfalls:

- Calling Kafka a database (it’s not, though it persists logs; it’s more of a durable messaging system).
- Not mentioning partitions or consumer groups (which are central to Kafka’s design for scaling and parallel consumption).
- Confusing Kafka with Kafka Streams (Kafka Streams is a processing library *on top of* Kafka, but core Kafka is just the broker system).

Follow-up: How does Kafka ensure fault tolerance? (Answer: replication of partitions across brokers. Each partition has leader and followers. If leader broker dies, a follower takes over ensuring no data loss beyond un-replicated messages. Also commit log with offsets, etc.)

---

27. **(Easy)** What is the difference between **event time** and **processing time** in streaming?

Answer:

- **Event time** is the timestamp when an event actually occurred (at the source/origin). It’s an inherent property of the event, often part of the event data (e.g., a sensor reading’s timestamp, or a user click time).
- **Processing time** is the time at which the event is observed/processed in the streaming pipeline (e.g., when it arrives at the stream processor or when it’s processed by a certain operator). This depends on system characteristics and delays.
- In an ideal world without delays, event time and processing time might align, but in practice events can arrive late or out-of-order, so processing time could be much later than event time.
- Example: A mobile app user action happens at 10:00 (event time), but due to network lag the log arrives at server at 10:05 (processing time for streaming job).
- Why it matters: Many streaming computations (like windowing) can be done based on event time (to get correct results aligned with when events happened) vs processing time (simpler but potentially inaccurate if events are late).
- **Event-time processing** requires handling out-of-order data using watermarks and triggers (to decide when you’ve seen “enough” of the late arrivals for a time window). It allows correct aggregates based on the actual occurrence time.
- **Processing-time windows** just use the clock of the machine to cut windows, which is simpler (no need for watermarks) but if events arrive late, they either miss their intended window or skew results.
- Modern stream frameworks (Flink, Beam, Spark Structured Streaming) support event-time semantics extensively, because it yields more accurate results in presence of late data.

What interviewers look for: Understanding that streaming isn’t as simple as real-time arrival – distinguishing the two timestamps is foundational for reasoning about correctness. If candidate mentions watermarks/late data handling, that's a plus (shows deeper knowledge).

Common pitfalls:

- Saying processing time is when event "enters Kafka" or similar – processing time can refer to any stage's local clock, but generally the concept is clear as above.
- Not articulating why event time is needed (e.g., use case: if you want daily user counts by actual day of action vs by when data arrived).
- Thinking event time is always present – sometimes events might not have a timestamp, then you might only have processing time.

Follow-up: What is a **watermark** in streaming and how is it used? (Answer: A watermark is a mechanism to track progress of event time, indicating that no events older than a certain event-time timestamp are expected (or any that arrive will be considered late). It's used to know when to close event-time windows and emit results with completeness.)

---

28. **(Medium)** Define **exactly-once processing** in the context of streaming. How do streaming frameworks achieve it?

Answer:

- **Exactly-once processing** means that each event in the source will **affect the final results exactly once**, despite potential retries, failures, or duplicates during processing. It's an assurance about the end-to-end outcome, not that the event is literally only touched once internally.
- This is in contrast to at-least-once (where duplicates could appear in output) or at-most-once (where some events might be lost).
- Streaming frameworks achieve exactly-once typically via a combination of **idempotent operations**, **transactional writes**, and **checkpointing with state rollback**:
    - **Checkpointing & State management:** Frameworks like Apache Flink maintain state and periodically checkpoint it (including progress of reading from source). If a failure happens, it rolls back to the last checkpoint and resumes, avoiding processing events twice erroneously.
    - **Kafka transactions / Source offsets:** For example, Spark Structured Streaming + Kafka can use Kafka's transactions to commit offsets and results in an atomic way. Or frameworks track source offsets in checkpoint; on restart they know where to continue without reprocessing what was already processed.
    - **Idempotent sinks:** Ensuring the sink (output) can handle replays or duplicates such that the final outcome is like each event once. For example, writing to a database using primary key deduplication, or using upsert operations rather than blind inserts.
    - **Two-phase commit protocols:** Some frameworks (Flink's two-phase commit sinks) use a prepare/commit so that if a checkpoint is successful, the output is committed, otherwise not, ensuring no partial commits on failurereddit.com.

- It’s important to note exactly-once in streaming is about effect, often achieved by **at-least-once delivery + deduplication** or transactional handling to avoid double-count.
- Example: Flink with checkpointing can guarantee that if an operator had partially processed some events and failed, after recovery it will reprocess them but with state rewound, so the final count doesn’t double count.
- Kafka Streams (the library) also has exactly-once semantics (EOS) by managing commit and idempotent producers.

What interviewers look for: Understanding that exactly-once is tricky but solvable. It’s more nuanced than batch (where you just re-run a job from scratch). Should mention checkpointing or transaction as key techniques.

Common pitfalls:

- Confusing exactly-once with literally not reading twice – some say “only processed one time” which might imply at-most-once incorrectly. Need to articulate it’s outcome that counts.
- Not differentiating between messaging guarantee vs processing guarantee. Kafka can deliver at-least-once, but using offset management you get exactly-once in processing pipeline.
- Claiming a system is exactly-once without qualifiers – e.g., vanilla Kafka without special handling is at-least-once to consumers, but using Kafka Streams or transactions can reach exactly-once processing.

Follow-up: Can Kafka on its own (as a broker) guarantee exactly-once delivery to consumers? (Answer: Not by itself; Kafka guarantees at-least-once delivery. Exactly-once requires cooperation of consumer or use of the Kafka Streams API or transactions with consumer groups.)

---

29. **(Medium)** What is a **watermark** in stream processing and how is it used for late data handling?

Answer:

- A **watermark** is a mechanism that streaming systems use to track the progress of event time and to make decisions about how long to wait for late events. It’s essentially a timestamp (event-time) that the system believes all future incoming events will have timestamps **less than or equal to** (i.e., the stream is complete up to that time)docs.databricks.com.
- When a watermark for time T is emitted, it signals that the framework has seen (or assumes to have seen) all events up to time T. Events with timestamp <= T that arrive after this watermark are considered **late**.

- Watermarks allow windowing logic to close a window and emit results even if some events might still trickle in later – you decide a threshold for lateness. For instance, you might set watermark = event time - 5 minutes, meaning you're okay with up to 5 minutes late arrivals.
- Late data handling: if an event comes with timestamp older than the current watermark (meaning it fell outside the waiting threshold for its window), the system can either drop it, count it separately, or send it to a "late events" side output. Some frameworks allow updating previously emitted results with late data (though that complicates exactly-once, etc.).
- Example: In Flink, you might have a 1-minute event-time window with allowed lateness 2 minutes. Flink will watermark time moving forward as events come. If at 12:05 watermark passes beyond 12:00, the window [12:00-12:01) is triggered and output. If at 12:06 an event arrives with timestamp 12:00:30 (which is late beyond 2 min), it's late and Flink by default drops it or routes it to a side output if configured.
- Watermarks are typically generated either from timestamps in data (maybe taking max timestamp seen minus some bound) or by sources (like Kafka connectors that add watermarks as time progresses).

What interviewers look for: Familiarity with concept as used in Flink/Beam/Structured Streaming. It's critical for correct event-time windowing. If the candidate can articulate how watermarks prevent waiting indefinitely for late events, that's good.

Common pitfalls:

- Some confuse watermark with just latest event time seen – it's typically a bit behind to allow for stragglers.
- Not covering what happens with late events – just saying watermark closes windows but not mention late data disposal or processing.
- Using the term loosely. For credit, one should mention "late events" or "window completeness".

Follow-up: How would you set a watermark delay in a system where events can be delayed by variable amounts (some up to hours late)? (Possible answer: depends on application – you might set a watermark delay (allowed lateness) that balances result latency vs completeness. If some events can be hours late but are rare, maybe accept dropping or special-case them, or use very late watermarks and accept that results take longer to finalize.)

---

30. **(Hard)** Compare **Lambda Architecture** vs **Kappa Architecture** for data processing.

Answer:

- **Lambda Architecture:** A paradigm that uses two parallel processing paths for data: a **batch layer** and a **speed (real-time) layer**nexocode.com. The batch layer processes all data (typically with high throughput, e.g., nightly Hadoop jobs) to produce accurate results (gold standard), while the speed layer handles recent data with low latency (e.g., using streaming) to provide real-time updates. The outputs from both are merged (often in a serving layer) for queriesnexocode.com.
  - Pros: You get **accuracy and completeness** from batch (recomputes can correct errors, handle late data fully) and **low-latency** updates from streaming. Fault tolerance is simpler in batch, and streaming can focus on short-term approximate results.
  - Cons: It's complex – two code bases, two paths to maintain. Code duplication or logic divergence can happen (you need to implement same computations twice). Higher operational overhead.
- **Kappa Architecture:** Proposed as a simplification – treat all processing as a **single stream processing pipeline**medium.com. There is no separate batch layer; the streaming system handles both historical reprocessing and real-time data. If reprocessing is needed, you just feed data through the same streaming pipeline (e.g., replay from a log). The idea is using a stream processor that can also consume old data from storage if needed.
  - Pros: **Simpler design** – one pipeline to maintain (easier code management). Modern stream processors (like Apache Flink, Kafka Streams) can achieve high throughput and correctness, making separate batch less necessary. Uses event logs (like Kafka) to replay data for recomputation if needed.
  - Cons: Requires a powerful streaming engine that can handle large-scale reprocessing. Sometimes more challenging to do full historical recompute if data is huge (batch might still be faster for full recompute). Also, not every use-case easily fits pure streaming (some logic might be simpler in batch SQL).
- When to use which: Lambda came from limitations of older technology (batch MapReduce for correctness, Storm for speed). With newer frameworks that provide exactly-once and good windowing (Beam model), many organizations try Kappa (single pipeline on Kafka + Flink/Storm etc.). Kappa suits when you have a unified log (like Kafka) storing all data.
- Example: In Lambda, you might have Hadoop computing daily aggregates and Spark Streaming updating hourly deltas. In Kappa, you'd have just a Flink job reading from Kafka from beginning and continuously maintaining aggregates.

What interviewers look for: Understanding of architectural patterns for mixing batch & real-time. Knowing that Lambda = batch + speed layers; Kappa = single stream (no separate batch). Possibly expecting commentary that Kappa is feasible now with advanced stream processors.

Common pitfalls:

- Thinking Lambda is about function programming (due to the name Lambda) – here it's an architecture name, not related to lambda functions.
- Not mentioning the complexity downside of Lambda.
- Asserting one is always better – in practice, lambda still used if existing systems or if streaming tech not fully trusted for some part.

Follow-up: If you already have a Hadoop batch pipeline producing daily reports, how might you evolve it to a more real-time pipeline? (Could answer: add a speed layer to handle recent updates -> Lambda approach as intermediate step, or replace with a streaming solution that can do near-real-time but also possibly backfill via log replay -> Kappa style.)

---

31. **(Hard)** How does Apache **Flink** achieve exactly-once state consistency in streaming?

Answer:

- Apache Flink achieves exactly-once consistency primarily through its **checkpointing** mechanism and the **Changelog (Write-Ahead Log)** two-phase commit protocol for sinks:
  - Flink periodically creates **distributed snapshots** of the state of all operators (and their positions in the input streams) using a Chandy-Lamport algorithm (barrier). A **checkpoint barrier** flows through the streams; when an operator receives a barrier from all inputs, it snapshots its state (to durable storage like DFS)blog.devgenius.io.
  - These checkpoints include source offsets (like Kafka offsets) and any operator state (e.g., aggregations, windows content).
  - If a failure occurs, Flink will restore the entire job to the last successful checkpoint state and resume processing from there (reading from sources starting at the saved offsets).
  - For sinks, Flink uses a **two-phase commit** if needed: e.g., for exactly-once sinks, Flink can write data in a pre-commit stage and only commit once checkpoint is successful (so either the sink sees the whole transaction or none)medium.comdocs.databricks.com.
    - Many connectors (like to transactional databases or Kafka producer) implement a TwoPhaseCommitSinkFunction: on checkpoint, prepare transaction, and on checkpoint completion commit it, or abort on failure.
  - Flink's state backend (like RocksDB) combined with checkpointing ensures that any stateful operation (window counts, etc.) rolls back so no double counting happens.
- In summary: Flink's exactly-once is achieved by **replay + state snapshot** so events are effectively processed once in final outcome. It doesn't prevent multiple processing internally if needed (like if failure, some events might be reprocessed) but external observers or final results see each event's effect exactly once.
- Also mention: Flink supports **idempotent or transactional sinks**. If the sink can deduplicate (idempotent) or handle transactions, exactly-once can be maintained end-to-end. If a sink doesn't support it (e.g., plain file sink), one might only get at-least-once or have to rely on idempotence by design (like partitions files by batch).

What interviewers look for: Familiarity with Flink's core design for reliability – checkpoint barriers, state snapshots, two-phase commit. This is a deep question; expecting high-level mention of checkpointing at least.

Common pitfalls:

- Confusing exactly-once with one-time delivery – need to articulate the mechanism.
- Not knowing specifics like barriers or state snapshots – might just guess "via checkpoints" which is okay but better to elaborate a bit.
- Not acknowledging that exactly-once often depends also on sink capability.

Follow-up: How is Flink's approach different from Spark Streaming (old DStream) or Spark Structured Streaming? (Possible differences: Spark Structured Streaming uses a "micro-batch" or continuous processing that also uses checkpointing but at the level of batches; Flink is true event-at-a-time with barriers. Spark's exactly-once often relies on idempotent sink or transactional sink similar concept, but Flink's event-driven checkpointing is highly integrated.)

Sources (Streaming):

- **Confluent Kafka Intro** – components of Kafkadatacamp.comgithub.com
- **Data Engineer Academy blogs** – differences between Lambda vs Kappanexocode.commedium.com
- **Apache Beam Model (watermarks)** – via blog or Beam docs for watermark explanationdocs.databricks.com
- **Ververica Flink blogs** – checkpointing & two-phase commit in Flinkblog.devgenius.iomedium.com
- **Google Cloud Blog** – event-time vs processing-time concepts (not explicitly opened but standard definitions)
- **Confluent Blog** – exactly-once semantics with Kafka Streams (conceptually similar approach)
