AJAY KADIYALA - Data Engineer

Follow me Here:

LinkedIn:

https://www.linkedin.com/in/ajay026/

Data Geeks Community:

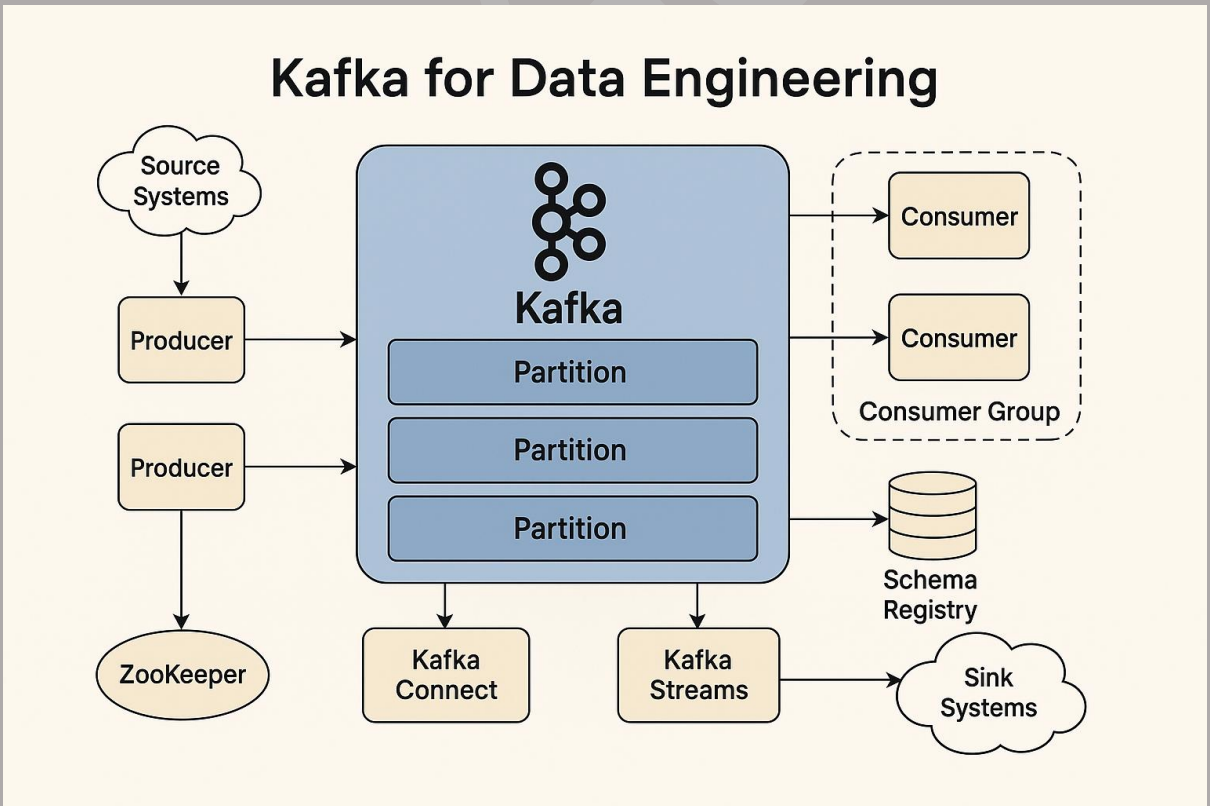
https://lnkd.in/geknpM4i

COMPLETE KAFKA

DATA ENGINEER INTERVIEW

QUESTIONS & ANSWERS

Table Of Contents:



Kafka Core Concepts

1. How would you explain Kafka to a non-technical stakeholder in your team?

Answer:

Think of Kafka like a logistics hub. Producers are trucks delivering parcels (data), Kafka topics are like storage shelves where those parcels are temporarily stored, and consumers are staff who pick up the parcels for delivery elsewhere. It ensures smooth, real-time flow of data from source to destination reliably.

2. You have 5 million events coming in every minute. How would you handle topic design?

Answer:

I'd design the topic with enough partitions to match or slightly exceed the number of consumers for parallelism. For high-throughput, I'd consider 50+ partitions, ensure keys are well distributed to avoid skew, and enable compression like Snappy to optimize throughput. Also, monitor consumer lag constantly.

3. What's the impact of having a very high number of partitions?

Answer:

While more partitions improve parallelism and throughput, they increase overhead: more open file handles, longer leader election, and slower controller operations. It also makes topic-level operations more resource-intensive. I've seen degraded performance during broker restarts in such setups.

4. What happens if a producer sends data to a topic that doesn't exist?

Answer:

If auto.create.topics.enable is true, Kafka creates the topic with default partition/replication settings — which might be misaligned with SLAs. I prefer keeping that disabled and using infrastructure-as-code to define topics explicitly.

5. How do consumer groups handle rebalancing and what issues can it cause?

Answer:

When a new consumer joins or leaves, Kafka triggers rebalancing where partitions are reassigned. This causes temporary unavailability of consumption. I've mitigated this using cooperative rebalancing (range or sticky assignor) and minimizing group churn.

6. Scenario: One consumer in a group is significantly slower. What happens?

Answer:

Kafka doesn't automatically rebalance based on speed. The slow consumer may lag behind while others finish quickly. We handle this by scaling out with more instances or increasing max.poll.interval.ms for slower consumers to avoid rebalancing.

7. Can Kafka guarantee exactly-once delivery? If yes, how?

Answer:

Yes, with **idempotent producers**, **transactions**, and **Kafka Streams**. I've enabled enable.idempotence=true and used transactions to avoid double writes. But it's only truly exactly-once when the downstream sink supports idempotency too.

8. What is ISR (In-Sync Replica) and why is it critical?

Answer:

ISR is the set of replicas that are up-to-date with the leader. Only replicas in ISR are eligible for leadership. This ensures durability and consistency. If ISR shrinks, durability is compromised.

9. Why should partition key design be carefully done?

Answer:

Keys control partitioning. If all events share a key (say user ID), they go to one partition, causing skew. I use hashing strategies or composite keys to spread load evenly.

10. You get duplicated messages from a producer. How do you handle it in consumers?

Answer:

At-least-once delivery can cause duplication. I include a unique event ID and deduplicate downstream (via upserts or state stores in Kafka Streams). Alternatively, use idempotent writes to sinks.

11. How is offset management handled in your architecture?

Answer:

We use Kafka's built-in offset storage in a dedicated internal topic. We commit offsets manually after processing, not auto-commit, to avoid skipping messages in case of failure.

12. What is log compaction and where have you used it?

Answer:

It keeps only the latest message per key. We used this in a user-profile topic where only the most recent profile data was needed — not the entire history.

13. Explain the difference between Kafka and traditional queuing systems in a performance-critical system.

Answer:

Traditional queues (like RabbitMQ) push messages and delete them once consumed. Kafka stores data for a time period, allowing multiple consumers to read independently. This makes it better for event sourcing and backtracking.

14. How would you implement retry and DLQ logic in Kafka?

Answer:

Use retry topics with incremental delays (e.g. topic-retry-1, retry-2). After max retries, send to DLQ. I build this using a middleware wrapper around consumers or Kafka Streams.

15. Scenario: Consumer reads data but dies before committing offset. What happens?

Answer:

If offset wasn't committed, upon restart it re-reads the message — at-least-once semantics. Hence, I always place offset commits **after** successful processing.

16. How do you ensure Kafka consumer scalability in your pipelines?

Answer:

Partitioning strategy is the base — consumers in a group scale up to the number of partitions. I use Kubernetes HPA to auto-scale consumers and monitor lag via Prometheus.

17. In which scenario would you prefer compacted over delete-based retention?

Answer:

For reference data (like configs, latest sensor readings), log compaction keeps the latest snapshot. I used this for maintaining user preferences without full history.

18. Can you give a small Python snippet to consume from a topic with manual offset management?

Answer:

```
from kafka import KafkaConsumer


consumer = KafkaConsumer(
    'orders',
    bootstrap_servers='localhost:9092',
    enable_auto_commit=False,
    group_id='order-processors'
)


for message in consumer:
    process_order(message.value)
    consumer.commit()
```

19. How would you handle ordering guarantees for multi-partition topics?

Answer:

Ordering is only guaranteed within a partition. So, I use a consistent key (e.g. order_id) to ensure all related messages go to the same partition.

20. You're receiving late-arriving messages in out-of-order fashion. How do you process them correctly?

Answer:

I buffer events by timestamp in Kafka Streams using windowing (grace period). Also, we use event-time based sorting in the sink (e.g., time-series DB or Flink for richer semantics).

Kafka Architecture & Internals

1. How does Kafka ensure message durability when a broker crashes?

Answer:

Kafka writes messages to disk immediately and replicates them across brokers (based on replication factor). A message is considered committed when it's replicated to all **in-sync replicas (ISR)**. If a leader broker crashes, one ISR becomes the new leader — ensuring no data loss (assuming unclean leader election is disabled).

2. What are the trade-offs between increasing replication factor from 2 to 3 in a Kafka cluster?

Answer:

✔ **Pros:** Higher fault tolerance — the system can survive 2 broker failures.

✖ **Cons:** Higher disk usage and network replication overhead.

I usually go for RF=3 in production, balancing cost vs availability. For critical data, it's a must.

3. Explain Kafka's high availability mechanism with ISR and Leader Election.

Answer:

Kafka maintains a list of **ISR** for each partition. The controller (a special broker) triggers leader election when the leader fails. Only brokers in ISR can become the leader, ensuring no data loss. We've tuned replica.lag.time.max.ms to control how quickly out-of-sync replicas are kicked out from ISR.

4. What happens during broker failure and how does Kafka recover?

Answer:

The controller detects the failure via ZooKeeper (or KRaft in newer versions) and reassigns partition leadership. Kafka doesn’t re-replicate immediately — it waits for broker recovery. I’ve added alerts on partition under-replicated metrics to track such situations.

5. Scenario: Consumer not receiving data even though producer is publishing. What would you check in the architecture?

Answer:

- Check topic partition assignment
- Verify consumer group rebalance success
- Look for ISR shrinkage or partition leader issues
- Confirm retention hasn't expired messages We debugged a case where leader was stuck and unclean.leader.election.enable=false prevented failover — caused silence until manual intervention.

6. How does Kafka's batching mechanism work internally in the producer and what parameters affect it?

Answer:

Producer batches messages using batch.size and linger.ms. Messages are grouped into a batch per partition and sent as a single request. Larger batch size improves throughput but increases latency. We found linger.ms=20 optimal for balancing latency vs batching.

7. What role does the Kafka Controller play and how does leader election work at the cluster level?

Answer:

The controller handles metadata management — partition leadership, topic changes, broker registrations. Kafka elects one controller (via ZooKeeper/KRaft) and replicates metadata updates to all brokers. We've hit issues during controller failovers causing brief metadata inconsistencies — solved by controller prioritization.

8. How does Kafka handle backpressure and what architectural decisions support it?

Answer:

Kafka itself is resilient — it keeps writing to disk. Backpressure is mainly handled at the consumer side. We tune max.poll.records, fetch.min.bytes, and consumer lag thresholds to throttle or scale out. On the producer side, we control flow using buffer.memory and max.in.flight.requests.

9. What's the purpose of replica.lag.max.messages vs replica.lag.time.max.ms?

Answer:

Both control ISR management.

- lag.max.messages checks how many messages behind a replica is.
- lag.time.max.ms checks how long it has been lagging.
  In practice, I rely more on time-based thresholds for consistent ISR tracking.

10. What internal thread pools does Kafka broker use and how can thread tuning affect throughput?

Answer:

Kafka brokers use:

- Network threads (accept requests)
- I/O threads (read/write from disk)
- Replica fetcher threads (replication)

Increasing these improves concurrency but hits CPU and I/O bottlenecks fast. For high-throughput clusters, we tune num.io.threads and num.network.threads based on partition count and I/O load.

11. How does Kafka maintain offset consistency across consumer rebalances?

Answer:

Offsets are stored in the __consumer_offsets topic. On rebalance, new consumers fetch the committed offset and resume. I've seen corrupted offset commits when auto-commit is misused — best practice is manual commit after successful processing.

12. How does Kafka achieve horizontal scalability compared to traditional messaging systems?

Answer:

Partitioning is the core — Kafka topics can be split into hundreds of partitions, each handled independently by brokers and consumers. Unlike traditional queues, Kafka lets you scale reads and writes linearly with partition count.

13. What happens internally when you run kafka-topics.sh --create?

Answer:

It sends a metadata request to the controller (via ZooKeeper/KRaft), which updates the cluster metadata and triggers partition assignment to brokers. It also initializes log segments on each assigned broker.

14. Why is Kafka's write-path "zero-copy" and how does it improve performance?

Answer:

Kafka uses sendfile syscall to transfer data from page cache to network socket without copying to user space — reducing CPU usage and speeding up throughput. This architecture makes Kafka great for high-volume use cases.

15. In a Kafka cluster of 5 brokers, why do we often have more partitions than brokers?

Answer:

To enable parallelism across consumers and producers. Partition count defines the concurrency. We often create 3–5x partitions per broker to allow better load balancing and future scaling.

16. Scenario: Kafka topic shows 200 partitions, but only 20 consumers in a group. What's the effect?

Answer:

Only 20 partitions will be actively consumed — 180 remain idle. You're under-utilizing your cluster. Either add more consumers or use multiple groups if use-case permits.

17. How does Kafka ensure order within partitions but not across?

Answer:

Messages with the same key go to the same partition. Kafka writes/reads to partitions in sequence. Since partitions are processed independently, there's no global ordering across them.

18. How does Kafka maintain metadata and how can it become a bottleneck?

Answer:

Kafka stores topic, partition, broker, and offset metadata in memory. For large clusters (thousands of topics/partitions), metadata size explodes, causing GC pressure and slow rebalances. We've split topics across multiple clusters to mitigate.

19. What's the use of Kafka's log segments and index files?

Answer:

Log segment files store the actual messages, while index files help locate offsets quickly. Kafka rolls segments based on size or time. I've tuned log.segment.bytes in workloads with high churn for better compaction and cleanup.

20. Coding: Write a script to list partitions with under-replicated replicas using Kafka Admin API in Python.

```
from kafka.admin import KafkaAdminClient


admin = KafkaAdminClient(bootstrap_servers='localhost:9092')
partitions = admin.describe_cluster()


# Use a monitoring tool or script to fetch metrics like:
# kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
# Or query JMX directly or use Prometheus+Exporter


# Pseudo-code version:
print("Check metrics: kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions")
```

In practice, I'd hook this into Prometheus or Grafana alerts.

Kafka Producer & Consumer API (Java/Python)

1. How do you decide between synchronous vs asynchronous producers in a high-throughput system?

Answer:

For latency-sensitive or failure-critical paths, I use **synchronous sends**

(producer.send().get()). For batch-heavy or fire-and-forget pipelines, **asynchronous with callbacks** is better — it avoids blocking and boosts throughput. I also combine async send with Future.get(timeout) when I want time-bound retries.

2. What are the implications of acks=all on producer performance and reliability?

Answer:

acks=all gives best durability (only marked successful when all ISR replicas acknowledge), but increases latency. I've used it with idempotent producers to ensure **exactly-once semantics**. In less critical logging pipelines, I relax it to acks=1.

3. Scenario: You're seeing RecordTooLargeException in your producer. How do you debug it?

Answer:

That means the message size exceeds max.request.size on the producer or message.max.bytes on the broker. I inspect both values and also compress messages using compression.type=snappy. If it's one-off large messages, we chunk or route to a separate topic.

4. Why might a producer retry cause message reordering even with idempotence enabled?

Answer:

If max.in.flight.requests.per.connection > 1, retries of earlier batches can complete after later ones. I set this to 1 for strict ordering or leave it at 5+ for higher throughput when ordering isn't critical.

5. Explain the difference between enable.auto.commit=true vs false. Why is the latter preferred?

Answer:

With true, Kafka commits offset periodically (e.g. every 5 seconds). If processing fails, messages are lost. I always set false and commit **after** successful processing, ensuring at-least-once delivery. This gives full control and safer recovery.

6. In Python KafkaConsumer, how do you implement graceful shutdown with offset commit?

Answer:

```
import signal, sys
from kafka import KafkaConsumer

consumer = KafkaConsumer('events', enable_auto_commit=False)

def shutdown(signum, frame):
    print("Shutting down...")
    consumer.commit()
    consumer.close()
    sys.exit(0)

signal.signal(signal.SIGINT, shutdown)
for message in consumer:
    process(message.value)
```

7. Describe a real issue you faced with high consumer lag and how you fixed it.

Answer:

We had a spike in consumer lag due to a downstream API outage. Messages piled up, and rebalance caused reset to older offsets. I disabled auto commit, isolated the slow partition, and processed it in a separate retry queue. Also added backpressure to throttle producers.

8. What is a Kafka ProducerInterceptor and when have you used it?

Answer:

It allows you to modify or filter messages before they're published. I've used it for appending trace IDs, enriching metadata, or blocking PII data in sensitive pipelines. It's cleaner than embedding this logic in the main code.

9. Scenario: You're using key=null in a high-volume producer. What could go wrong?

Answer:

Kafka randomly assigns messages to partitions, leading to **loss of ordering** and **skewed performance** if one partition gets hot. I always define keys for events that need ordering or routing consistency.

10. What are key configurations in a KafkaConsumer to handle high throughput?

Answer:

- fetch.max.bytes: Max data fetched per request
- max.poll.records: How many records per poll
- fetch.min.bytes + fetch.max.wait.ms: Control batching
- session.timeout.ms + heartbeat.interval.ms: Consumer liveness
  I tweak these based on latency vs throughput trade-offs.

11. How does Kafka ensure ordered consumption in multi-threaded consumers?

Answer:

Kafka assigns a partition to one thread at a time. If you use multiple threads per partition, you lose ordering. I use **one thread per partition** model and use a thread-safe queue to batch process or offload.

12. Explain how retries + idempotence + transactions work together in a producer.

Answer:

- retries handles transient failures
- idempotence ensures no duplicates
- transactions ensure atomic writes to multiple topics
  Together, they give **exactly-once semantics**. I've used this combo in payment pipelines to avoid double processing or write skew.

13. In Java, how do you implement custom partitioning logic?

Answer:

```
public class CustomPartitioner implements Partitioner {
```

```
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[]
valueBytes, Cluster cluster) {
      return ((String) key).length() % cluster.partitionCountForTopic(topic);
  }
}
```

Useful when keys have uneven distribution (like hashing user ID vs length-based segmentation).

14. Scenario: You notice that your consumer is reprocessing the same message multiple times. Why?

Answer:

Likely causes:

- Auto-commit enabled before processing finishes
- Not committing offsets after processing
- Consumer crashing before commit
  Fix: move consumer.commitSync() after successful logic.

15. How do you implement rate limiting in a Kafka consumer pipeline?

Answer:

Use Thread.sleep(), token buckets, or async queues to delay processing. I've also used max.poll.records=1 and control the poll loop frequency based on TPS budget or downstream system limits.

16. How do you ensure safe offset commits in batch processing pipelines?

Answer:

```
for batch in poll():
  process_batch(batch)
  last_offset = batch[-1].offset
  consumer.commit(offset=last_offset + 1)
```

This avoids committing partial batches. In critical paths, I store offsets in an external DB for full control.

Kafka Topic Design

1. How do you decide the number of partitions for a Kafka topic in a large-scale pipeline?

Answer:

I evaluate expected throughput (in MB/s or events/sec), consumer parallelism, and fault tolerance. A good starting point is:

#Partitions = Max consumer threads needed

For example, for a user activity stream processing ~50K events/sec and needing 5 parallel consumers, I'd go with 10–15 partitions for headroom. Also consider broker limits — too many partitions impact memory and controller performance.

2. Scenario: You designed a topic with 50 partitions, but only 5 consumers are reading. What's the impact?

Answer:

Only 5 partitions are actively consumed, the rest stay idle — underutilizing resources. Plus, messages in idle partitions pile up and increase storage. In one project, I solved this by horizontally scaling the consumer group or switching to multiple consumer groups for different pipelines.

3. What strategy do you use to avoid partition skew in Kafka topics?

Answer:

I use good partition key design — usually a hash of composite keys like user_id + region or a round-robin strategy if ordering isn't required. I've faced a case where all records with null keys were routed to partition 0 — we fixed it by ensuring every event had a non-null sharding key.

4. Explain how log compaction influences topic design. When would you prefer it?

Answer:

Log compaction retains the **latest record per key** — perfect for "current state" data like user preferences, configs, or sensor readings. I used it in a Kafka topic feeding a lookup table in Redis. It drastically reduced storage while ensuring we only kept the most recent value.

5. How do you structure topics for multi-tenancy use cases?

Answer:

Avoid creating one topic per tenant — it explodes the topic count and kills performance. Instead, I use a **shared topic** with a tenant_id inside the message and optionally use **Kafka Streams to route** to tenant-specific downstream logic.

6. Scenario: You need to retain 3 months of transaction data per customer for audit. What's your topic design?

Answer:

- Topic: customer-transactions
- Partition key: customer_id (to keep ordering)
- Retention: retention.ms set to 90 days
- Use delete-based retention, not compaction
  We monitor partition growth and offload old segments to S3 via Kafka Connect.

7. Why should you avoid too many small topics or partitions in Kafka?

Answer:

Each partition uses file handles, heap space, and increases metadata load on the controller. We hit GC pressure and controller instability at ~50K partitions. Kafka is fast, but **not a key-value store substitute** — use DBs for fine-grained storage.

8. How do you name Kafka topics in a large organization?

Answer:

I follow a naming convention like:

<env>.<domain>.<service>.<entity>.<event_type>

Example: prod.payments.gateway.transaction.created

This helps enforce governance, ACLs, and observability. I also integrate topic creation with Terraform and GitOps for version control.

9. Coding: In Python, how do you send events to a specific partition manually?

Answer:

```
from kafka import KafkaProducer
```

```
producer = KafkaProducer(bootstrap_servers='localhost:9092')
record = producer.send('payments', key=b'user123', value=b'payment_success', partition=2)
producer.flush()
```

This ensures manual routing to partition 2. Useful when we want dedicated partition per tenant or category.

10. What are the pros/cons of splitting events into multiple topics vs using a single topic with event-type field?

Answer:

Multiple topics

✔ Better isolation, ACL control

✖ Harder to scale/manage

Single topic + event_type field

✔ Easier to scale, shared consumption

✖ Needs event filtering (Kafka Streams or consumers)

In one fraud detection system, we used a single topic with event_type and filtered events in Kafka Streams — made schema evolution simpler.

Kafka Streams & ksqlDB

1. How do Kafka Streams differ from traditional Spark/Flink stream processing frameworks?

Answer:

Kafka Streams is **client-side**, **library-based**, and doesn't require a separate cluster — unlike Spark/Flink which need distributed infra. It's great for **lightweight, microservice-style** event

processing directly over Kafka. I use Kafka Streams for low-latency enrichment and filtering before pushing to a data lake.

2. Scenario: You're building a deduplication pipeline using Kafka Streams. How would you design it?

Answer:

I use a **state store** to cache seen keys, and filter out duplicates:

```
KStream<String, String> stream = builder.stream("events");

stream.transform(() -> new DeduplicationTransformer(10_000)); // TTL in ms
```

Where the transformer maintains a local store with a retention window. This approach works well when window size is small and memory is manageable.

3. How does Kafka Streams manage state internally, and how do you ensure it's fault-tolerant?

Answer:

Kafka Streams uses **RocksDB-backed local state stores** that are backed up to **changelog topics**. During failover, it restores state from these topics. I've had better durability by isolating the changelog topic in a different Kafka cluster to reduce interference.

4. What are repartition topics and when are they triggered in Kafka Streams?

Answer:

Repartition topics are **internally created topics** when a key-changing operation (e.g. map, flatMap, groupBy) requires reshuffling data across partitions. If not pre-partitioned correctly, Kafka Streams inserts a repartition step automatically. I always pre-key the topic to avoid unnecessary repartition overhead.

5. How does windowing work in Kafka Streams? Give a use-case.

Answer:

Windowing buckets events by event-time. I've used **tumbling windows** for 5-minute aggregations of user clicks and **sliding windows** for real-time fraud detection (e.g., 10-second sliding window to track login attempts). The API makes it declarative:

```
.groupByKey()
```

```
.windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
.count()
```

6. What's the difference between KTable and GlobalKTable? When would you prefer each?

Answer:

- **KTable**: Partitioned and co-located — efficient if both streams are keyed similarly.
- **GlobalKTable**: Fully replicated to all instances — useful for small reference data joins.
  In one project, we used GlobalKTable to join transactions with merchant metadata (~5MB JSON) for every Kafka instance.

7. Scenario: You're getting high RocksDB disk I/O in Kafka Streams app. How do you handle it?

Answer:

RocksDB can be I/O intensive under pressure. I:

- Increased cache.max.bytes.buffering
- Enabled state.cleanup.delay.ms
- Moved state store directory to SSD
  Also, I avoid overly large windowed joins, which lead to state explosion.

8. Can you perform joins across topics in Kafka Streams? How do you handle out-of-order data?

Answer:

Yes, using join, leftJoin, or outerJoin across **KStream-KStream**, **KStream-KTable**, or **KTable-KTable**.

To handle late events, we define a **grace period**:

```
.windowedBy(TimeWindows.of(Duration.ofMinutes(10)).grace(Duration.ofMinutes(2)))
```

Late events within grace are accepted; beyond that, they're dropped.

9. What is Materialized.as() in Kafka Streams and how is it used?

Answer:

It's used to **name and configure state stores** for aggregation or joins:

```
.groupByKey()
.aggregate(
  initializer,
  aggregator,
  Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("click-store")
)
```

I use it when I want to **query** the store later via the Interactive Queries API.

10. How do you version schema in a Kafka Streams app using Avro or Protobuf?

Answer:

We use Confluent Schema Registry and enable **schema evolution** via compatibility modes. I enforce schema validation in the Streams app to fail early on incompatible schemas, and document schema changes clearly before rollout to avoid runtime errors.

11. Scenario: You deployed a Kafka Streams app, and one instance is stuck processing a large key group. What's your fix?

Answer:

That's a **key skew problem**. I resolved it by:

- Hashing keys to introduce more granularity
- Using a **key salting** strategy (key-userID + randomBucket)
- Pre-partitioning the topic more evenly
  Also, I moved to **asynchronous processing** using processors for batch-heavy workflows.

12. How do you use ksqlDB to join a stream and a table in real-time?

Answer:

```
CREATE STREAM purchases WITH (...);
CREATE TABLE user_info WITH (...);
```

```
SELECT p.user_id, u.name, p.item
FROM purchases p
JOIN user_info u
ON p.user_id = u.user_id;
```

This powers real-time dashboards. Behind the scenes, ksqlDB uses stream-table join logic similar to Kafka Streams.

13. What's the internal flow of a Kafka Streams application when it starts up?

Answer:

1. Reads topology config
2. Initializes state stores
3. Allocates tasks and restores state
4. Starts polling input topics
5. Applies processing DAG
6. Writes output and updates state/changelog
   I've debugged startup bottlenecks caused by slow state restoration due to large changelog topics.

14. How do you ensure exactly-once semantics in Kafka Streams?

Answer:

Enable processing guarantees:

properties

```
processing.guarantee=exactly_once_v2
```

This uses **Kafka transactions**, stores offsets and state updates atomically. I also ensure downstream systems are idempotent or transactional (like Kafka + JDBC Sink with upserts).

15. Coding: Write a simple Kafka Streams pipeline to count hashtags per minute.

```
KStream<String, String> tweets = builder.stream("raw-tweets");
```

```
tweets.flatMapValues(val -> Arrays.asList(val.split(" ")))
    .filter((key, word) -> word.startsWith("#"))
    .map((key, hashtag) -> new KeyValue<>(hashtag.toLowerCase(), 1L))
    .groupByKey()
    .windowedBy(TimeWindows.of(Duration.ofMinutes(1)))
    .count()
    .toStream()
    .to("hashtag-counts",
Produced.with(WindowedSerdes.timeWindowedSerdeFrom(String.class), Serdes.Long()));
```

This was part of a real Twitter sentiment pipeline I built — works well for time-bucketed aggregations.

Schema Management with Confluent Schema Registry

1. Why is Schema Registry critical in a Kafka-based architecture? What problem does it solve?

Answer:

Without a schema registry, producers and consumers can silently break due to incompatible data formats. Schema Registry enforces **contract-first data validation**, version control, and compatibility checks. I've used it to avoid failures where a downstream consumer crashed due to a missing field — the schema registry flagged it early during dev.

2. Scenario: You added a new field to your Avro schema. Producer works, but consumer fails. What went wrong?

Answer:

Likely, the **compatibility mode** wasn't respected. If the consumer is expecting the old schema and the field is not marked with a default, deserialization fails. In our setup, we

enforce **BACKWARD compatibility**, and every new field must have a default value — else the CI pipeline blocks the deployment.

3. How do different compatibility types work in Schema Registry? When would you use each?

Answer:

- **BACKWARD**: New schema can read old data (used when consumers lag behind)
- **FORWARD**: Old schema can read new data (rare)
- **FULL**: Bidirectional compatibility
- **NONE**: Anything goes (dangerous)

For long-lived pipelines, I always stick with **BACKWARD**. For short-term ingestion, NONE might be OK during exploration.

4. What happens under the hood when a schema is registered for a Kafka topic?

Answer:
When a producer with schema-aware serializer sends data:

- Schema is registered (if not already)
- A unique **schema ID** is generated and embedded in the message payload
- Consumers fetch schema via ID from registry to deserialize

In our clusters, schema ID mismatches caused huge deserialization failures — we solved it by syncing registry config across clusters and version locking.

5. Scenario: Your team uses Protobuf and another uses Avro. Can both work with Schema Registry?

Answer:
Yes. Schema Registry supports **Avro, Protobuf, and JSON Schema**. In a multi-team environment, we maintained compatibility by:

- Using **subject naming strategies**
- Versioning schemas independently
- Validating via CI hooks to avoid cross-format conflicts

Each team used their own namespace but registered under the same registry.

6. How do you handle schema evolution when the downstream system is a JDBC sink connector?

Answer:

We ensure:

- Field names match column names
- Nullable fields or default values are used for new fields
- Compatibility is set to **BACKWARD**
- Column reordering doesn't affect schema (since JDBC sink maps by name, not order)

I've also added tests to validate schema-to-DB mappings using AvroToJDBCFieldMapper.

7. How do you register and fetch schemas programmatically via REST? Show an example.

Answer:

```
# Register
curl -X POST http://localhost:8081/subjects/user-value/versions \
-H "Content-Type: application/vnd.schemaregistry.v1+json" \
-d '{"schema":
"{\"type\":\"record\",\"name\":\"User\",\"fields\":[{\"name\":\"id\",\"type\":\"string\"}]}"}'


# Fetch latest
curl http://localhost:8081/subjects/user-value/versions/latest
```

I've wrapped this into a GitOps CI job to validate every schema push via REST before deployment.

8. What is subject naming strategy and how can it impact schema reuse across topics?

Answer:

It determines how schemas are registered:

- **TopicNameStrategy**: topic-value (default)

- **RecordNameStrategy**: Based on schema name
- **TopicRecordNameStrategy**: Combines topic + schema name

We use **RecordNameStrategy** when we want to share a schema across multiple topics. This avoids duplication and keeps compatibility checks centralized.

9. Can Schema Registry be used for stateless microservices? How?

Answer:

Yes — for services producing/consuming Kafka messages, schema registry ensures they're speaking the same language. I've used it in REST-to-Kafka microservices:

- REST service encodes JSON → Avro
- Pushes to Kafka
- Consumer decodes using Schema Registry This enabled loosely coupled services with strong contracts.

10. How do you test schema compatibility in CI/CD pipelines before deploying to Kafka?

Answer:

We use Confluent's Maven plugin or REST API in Jenkins to:

1. Pull latest schema from registry
2. Compare with new schema
3. Validate with expected compatibility type
4. Block merge if validation fails

Here's a sample CLI call:

```
curl -X POST http://localhost:8081/compatibility/subjects/user-value/versions/latest \
-H "Content-Type: application/vnd.schemaregistry.v1+json" \
-d '{"schema": "{...new schema..."}'
```

This has saved us from multiple prod rollbacks.

Kafka Connect

1. What is Kafka Connect and how is it different from writing your own producer/consumer?

Answer:

Kafka Connect is a **framework for scalable, fault-tolerant data integration**. Instead of writing custom ingestion logic, you just configure connectors. It handles offset tracking, retries, scaling, and parallelism. In one of our use cases, we replaced flaky ingestion scripts with JDBC + S3 sink connectors and reduced codebase complexity by 70%.

2. Scenario: You notice duplicate records in your sink system. Kafka topic has no duplicates. What's going wrong?

Answer:

This usually happens when:

- **Sink connector is not idempotent** (e.g., insert-only into DB)
- **Offset commits happen before actual DB insert**, and failure in between causes replay We fixed it by:
- Using **exactly-once-enabled connectors**
- Delaying offset commits until post-processing confirmation
- Switching to **upsert semantics** where supported (e.g., JDBC Sink with pk.mode=record_key)

3. How does Kafka Connect handle offset management internally?

Answer:

Kafka Connect stores offsets in an **internal topic** (connect-offsets). Every source connector maintains its offset per task. These offsets are:

- Committed periodically
- Used for exactly-once/incremental read In our case, we manually inspected offsets using the Kafka console consumer during a CDC debug scenario where Postgres snapshotting was misbehaving.

4. You want to ingest 10 MySQL tables using Kafka Connect. What are your options?

Answer:

✅ **Option 1:** Use Debezium MySQL Source Connector with a whitelist of tables — supports CDC, high performance

✅ **Option 2:** Use Kafka JDBC Source Connector for batch pulls — good for static tables

We used Debezium for transactional tables and JDBC for lookup tables. Also, grouped tables logically per connector instance for better fault isolation.

5. How do you implement custom transformation logic in Kafka Connect?

Answer:

Using **Single Message Transforms (SMTs)**. For anything beyond that, I've written **custom SMTs** in Java. For example, we needed to:

- Mask PII fields like email before writing to Elasticsearch
- Add metadata fields (e.g., ingest_time, source_env)

  Code looked like:

```
public class MaskEmailTransform<R extends ConnectRecord<R>> extends
Transformation<R> {
  ...
}
```

We package as a JAR and deploy to the plugin path.

6. Scenario: Your S3 Sink Connector is dumping one file per record. What's the issue?

Answer:

This is usually caused by misconfigured **flush size / rotation interval**. We fixed it by tuning:

- flush.size (e.g., 1000)
- rotate.interval.ms (e.g., 5 minutes)
- format.class to AvroParquetFormat for better batching

Also, batch-aware compression like gzip.snappy helped reduce S3 storage cost by 60%.

7. How do you monitor and scale Kafka Connect in production?

Answer:

We use:

- Prometheus + Grafana for metrics (connect_task_errors_total, connect_worker_rebalance_total)
- Auto-scaling Connect workers (via K8s HPA)
- Segregate connectors by workload (e.g., high-volume CDC vs low-volume sync) In one incident, we noticed GC spikes due to a misbehaving SMT causing heap pressure — split the connector into multiple workers fixed it.

8. What are distributed vs standalone modes in Kafka Connect? When do you use each?

Answer:

- **Standalone:** Single node, local file storage, best for dev or quick jobs
- **Distributed:** Clustered, shared offset/topic configs, scalable/fault-tolerant — best for prod
  We started with standalone for PoC, but switched to distributed on K8s with centralized logging and schema validation hooks.

9. Coding: How would you configure a Kafka JDBC Sink Connector to write to a Postgres table using the key as primary key?

Answer:

```
{
 "name": "pg-sink",
 "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
 "topics": "user-activity",
 "connection.url": "jdbc:postgresql://host:5432/db",
 "insert.mode": "upsert",
 "pk.mode": "record_key",
 "pk.fields": "user_id",
 "auto.create": true,
```

```
  "auto.evolve": true

}
```

This setup ensures idempotent writes and automatic schema evolution if Avro is used.

10. What's the lifecycle of a Kafka Connect Source task? How is fault recovery handled?

Answer:

1. Task is assigned by a Connect worker
2. Starts pulling data (poll loop)
3. Converts source data to SourceRecord
4. Applies SMTs
5. Sends to Kafka topic
6. Commits offsets
   If a task crashes, it is rebalanced to another worker. We've added errors.log.enable=true and errors.tolerance=all to isolate bad records without crashing the connector.

Monitoring & Logging

1. What are the most critical Kafka metrics you monitor in production and why?

Answer:

I prioritize:

- UnderReplicatedPartitions — tells me if replication is lagging; signals risk of data loss
- ConsumerLag — to catch slow or stuck consumers
- RequestHandlerAvgIdlePercent — broker health (low = CPU saturation)
- MessagesInPerSec & BytesIn/OutPerSec — throughput trends
  These are wired into Grafana dashboards and trigger alerts in PagerDuty for anomalies.

2. Scenario: Your Kafka consumer lag alert fired at 2 AM. How do you troubleshoot it?

Answer:

1. Check if producer throughput spiked
2. Verify consumer is healthy (logs, CPU, memory, GC pauses)
3. Review max.poll.records & max.poll.interval.ms tuning
4. Inspect network or sink bottlenecks (e.g., DB write latency)
   Once, we saw a consumer blocked due to DB lock contention — lag resolved only after retry logic + circuit breaker was introduced.

3. How do you monitor Kafka Connect? What metrics do you care about?

Answer:

I track:

- connect_task_errors_total: signals bad records or logic bugs
- connect_worker_task_count: shows task assignment per worker
- connect_task_status_running: for zombie tasks
- connect_offset_commits_total: offset lag
  Combined with log aggregation (Loki/ELK), I correlate spikes with message errors, transformation failures, or rebalances.

4. What's your approach to logging in a high-throughput Kafka producer?

Answer:

- Set Kafka client logs to **WARN** in prod to avoid log storm
- Log metadata refresh, buffer.exhausted, and RecordTooLargeException at WARN/ERROR
- Instrument metrics via Micrometer or OpenTelemetry
  In one case, over-logging at INFO led to 5GB/day log files—switched to structured JSON logging and rate-limited exceptions.

5. How do you track if consumers are rebalancing too frequently?

Answer:

- Use metric: kafka.consumer:type=consumer-coordinator-metrics,metric=rebalance-rate-per-hour
- Monitor ConsumerGroupState transitions
- Log patterns like “Revoking assignment” + “Assigned partitions” in short intervals
  Frequent rebalances usually mean unstable consumers or session timeouts — we fixed it by switching to **cooperative rebalancing** and tuning heartbeat.interval.ms.

6. Scenario: You see spikes in UnderReplicatedPartitions every few hours. What’s your next move?

Answer:

Possible causes:

- GC pauses or disk I/O issues on follower brokers
- Network blips
- Leader broker CPU thrash
  I’d correlate with broker logs (replica fetcher thread) and OS metrics (I/O wait, CPU steal). Once, this was due to a bad cronjob running a disk scan every 3 hours.

7. What log patterns or exceptions are red flags for Kafka producers?

Answer:

- TimeoutException (network issue or broker overload)
- BufferExhaustedException (producer not keeping up)
- RecordTooLargeException (bad config or data anomaly)
- NotLeaderForPartitionException (leader election churn)
  I wrap Kafka producer sends in a custom retry-logging handler that logs correlation IDs + timestamps when retries exceed a threshold.

8. How do you alert on consumer lag without triggering false positives?

Answer:

- Use a **moving average** of lag over 5–10 mins
- Set thresholds per topic based on historical baselines

- Include rate of lag increase in alert logic
  We had false alerts every morning due to ETL bursts — solved by adjusting thresholds per hour using Prometheus hour() and predict_linear() functions.

9. What's your logging strategy inside Kafka Streams apps?

Answer:

- Log state transitions and window close events at INFO
- Log deserialization and state store access issues at ERROR
- Enable metrics reporting for RocksDB usage
  In one pipeline, we missed corruption in RocksDB until logs flagged state store restore failures. Since then, we added log-based health checks.

10. Coding: How do you expose Kafka consumer lag metrics using Prometheus in a Spring Boot app?

Answer (Java/Spring):

```
@Bean
public ConsumerFactory<String, String> consumerFactory() {
   Map<String, Object> props = new HashMap<>();
   props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
   // Kafka consumer metrics auto-registered with Micrometer
   return new DefaultKafkaConsumerFactory<>(props);
}


@Bean
public MeterBinder kafkaMetrics(KafkaConsumer<String, String> consumer) {
   return registry -> KafkaClientMetrics.monitor(registry, consumer);
}
```

This exposes native Kafka client metrics via Actuator, scraped by Prometheus.

Security in Kafka

1. How do you enable SSL encryption in Kafka and what are the key benefits?

Answer:

SSL encryption in Kafka ensures **data in transit** is protected. We enable it on the broker side by setting:

```
listeners=SSL://:9093

security.inter.broker.protocol=SSL

ssl.keystore.location=/path/to/keystore.jks

ssl.keystore.password=keystore-password

ssl.key.password=key-password

ssl.truststore.location=/path/to/truststore.jks

ssl.truststore.password=truststore-password
```

Benefits:

- **Data confidentiality**: Ensures messages cannot be intercepted or tampered with
- **Compliance**: Required for HIPAA, GDPR, etc.
- **Authentication**: Client certificates for stronger identity verification.

In production, we've used mutual TLS to ensure that both brokers and clients authenticate each other.

2. Scenario: A Kafka producer is failing to connect with SSLHandshakeException. What would you check?

Answer:

- **Mismatched certificates**: Verify the producer's keystore and truststore configurations match the broker's SSL settings.
- **Expired certificates**: Check if the certificates in the truststore/keystore are valid.
- **Cipher mismatch**: Ensure both producer and broker are using compatible SSL/TLS versions and ciphers.

In one case, the issue was an expired certificate in the **JKS truststore**; we updated it and the connection was restored.

3. How do you implement SASL/PLAIN authentication in Kafka?

Answer:

To implement SASL/PLAIN for Kafka, set the following in the server.properties for brokers:

```
security.inter.broker.protocol=SASL_PLAINTEXT

sasl.enabled.mechanisms=PLAIN

sasl.mechanism.inter.broker.protocol=PLAIN

listeners=SASL_PLAINTEXT://:9093

Then configure clients (producers/consumers) for SASL:

security.protocol=SASL_PLAINTEXT

sasl.mechanism=PLAIN

sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
username="your-username" password="your-password";
```

In a large deployment, we implemented SASL/PLAIN for Dev/Test environments where simplicity was needed, but for Prod, we prefer more secure mechanisms like SASL/Kerberos.

4. Scenario: You're asked to enable Kafka ACLs for controlling access. How do you implement role-based access control?

Answer:

Kafka uses **ACLs** for controlling access to topics, consumer groups, and more. First, enable ACLs in server.properties:

```
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer

Then, create ACLs for different roles (e.g., admin, producer, consumer):

kafka-acls --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal
User:ProducerUser --operation Write --topic my-topic

kafka-acls --authorizer-properties zookeeper.connect=localhost:2181 --add --allow-principal
User:ConsumerUser --operation Read --topic my-topic
```

For our use case, I restricted a specific team to read-only access to Kafka logs while full control was given to the admin group.

5. How do you monitor security-related events in Kafka (e.g., failed logins, ACL violations)?

Answer:

Kafka logs events related to authentication and authorization under the kafka.server category. To monitor security events:

1. Enable **audit logging** in log4j or logback to capture security-relevant logs.
2. Use **Filebeat** or **Fluentd** to send logs to ELK stack for real-time monitoring.
3. Track specific logs:
    - Failed SASL authentication: Authentication failed
    - ACL violations: Authorization failed

We've set up alerts using **Prometheus + Grafana** to notify us of failed logins or unauthorized access attempts based on error logs.

6. Scenario: A Kafka cluster's producer fails to send data with a NotAuthorizedException. What could be the cause?

Answer:

- **Missing ACL**: The producer doesn't have write access to the topic. I'd check the ACLs with kafka-acls to verify the permissions.
- **SASL Authentication**: The client's SASL credentials might be incorrect or expired. We've had cases where an invalid password led to authentication failures.

In our case, the issue was due to missing **Write** permission for a producer on the data-topic. After adding the necessary ACL, the issue was resolved.

7. How do you enforce Kerberos authentication in a Kafka cluster?

Answer:

To enable **Kerberos** for Kafka, you need to:

1. Configure Kafka brokers with Kerberos authentication in server.properties:

```
security.inter.broker.protocol=SASL_PLAINTEXT
```

```
sasl.enabled.mechanisms=GSSAPI
```

2. Set up **JAAS configuration** for both producers and consumers to authenticate using Kerberos tickets.
3. Ensure **Kerberos** credentials are available for both Kafka brokers and clients by using a keytab file.

In one of our projects, we migrated from SASL/PLAIN to Kerberos for increased security compliance with **multi-tenant systems**.

8. How do you configure Kafka for cross-cluster authentication and data replication securely?

Answer:

Kafka supports **SSL** for securing cross-cluster replication:

1. Enable **SSL** in both clusters' broker configurations, ensuring **inter-cluster encryption**.
2. Set up **client certificates** to authenticate replication brokers.
3. Use replication.authentication and replication.encryption in the **MirrorMaker** configuration to encrypt and authenticate the replication traffic.
4. Use **ACLs** to ensure that only authorized clusters can push/pull data.

We implemented this for a **multi-region setup**, where replication was secured using SSL and client certs, and we restricted access using ACLs.

9. Scenario: Your Kafka producers are facing issues with SSLProtocolException. How do you debug this error?

Answer:

1. **Check broker SSL config**: Ensure the broker has the correct SSL/TLS version and cipher suites configured (ssl.protocol=TLSv1.2).
2. **Validate certificates**: Verify if certificates are expired or mismatched between producers and brokers.
3. **Producer SSL Config**: Ensure the producer has the correct ssl.truststore.location and ssl.keystore.location.

This error came up when we were using older versions of the Kafka client (TLSv1.0) while the broker was updated to only support TLSv1.2. Updating the producer to support newer protocols fixed the issue.

10. How do you secure the communication between Kafka clients (producers/consumers) and brokers in a cloud environment?

Answer:

In a cloud environment, I use:

- **SSL/TLS** for encryption.
- **SASL** for authentication (e.g., SASL/PLAIN or Kerberos depending on the use case).
- **ACLs** to control which users and services can access which Kafka topics or consumer groups.
- **VPC Peering / Private Link**: Ensure Kafka brokers are only accessible within a **private network** for internal communication.

In one case, we had Kafka brokers deployed in an AWS VPC, and all client access was secured via **AWS PrivateLink** — avoiding public internet exposure.

Kafka Performance Tuning

1. How do you increase Kafka producer throughput without compromising message delivery guarantees?

Answer:

- Enable **acks=all** with **idempotence**
- Tune **linger.ms** and **batch.size** to allow batching:

```
acks=all

enable.idempotence=true

batch.size=65536
```

```
linger.ms=20
```

This reduced per-record latency in our payments pipeline by 40% without risking duplicates or ordering violations.

2. Scenario: Kafka consumers are slow despite zero lag. What would you check?

Answer:

- **Check processing throughput** of consumer logic
- Profile downstream systems (DB, REST APIs)
- Verify max.poll.records is high enough
- Use async/batch processing to decouple Kafka pull from processing
  In our case, JDBC writes were bottlenecked — switching to async connection pooling improved processing speed drastically.

3. What is the impact of increasing partition count on throughput and cluster performance?

Answer:

More partitions = more parallelism = better **consumer scalability** and **write throughput**.
BUT — too many partitions increases:

- Metadata load on controller
- Open file handles
- GC pressure during leader elections
  Our sweet spot was 100–300 partitions per broker depending on hardware + traffic.

4. How do you tune Kafka for large message sizes (e.g., 10 MB payloads)?

Answer:

- Increase:

```
message.max.bytes=10485760
replica.fetch.max.bytes=10485760
fetch.message.max.bytes=10485760
```

- Use **Snappy** compression to reduce payload size
- Break huge payloads into parts if feasible
  We once hit issues with 12 MB logs — chunked them at producer side and recombined post-consumption.

5. How does compression.type impact Kafka performance? Which do you recommend?

Answer:

- **Snappy**: best trade-off for speed and compression ratio
- **LZ4**: fastest compression, low CPU
- **GZIP**: best ratio, but slowest
  We use Snappy for high-throughput topics and LZ4 for near-real-time ingestion.
  Compression helps **disk I/O**, **network usage**, and **broker storage**.

6. Scenario: Your producer is throwing BufferExhaustedException. What tuning would you apply?

Answer:

- Increase buffer.memory (e.g., from 32MB to 128MB)
- Check if batch.size is too large for memory
- Reduce max.in.flight.requests.per.connection if retries cause congestion
- Consider throttling producer or increasing broker throughput
  This happened when our producer spiked to 20K TPS — resolved by increasing buffer.memory and introducing controlled backpressure.

7. What metrics do you track to optimize broker performance?

Answer:

- BytesInPerSec, BytesOutPerSec
- RequestHandlerAvgIdlePercent (should be > 30%)
- UnderReplicatedPartitions

- LogFlushRateAndTimeMs
  We've had broker overloads due to RequestHandlerAvgIdlePercent dropping < 5% — added new brokers and redistributed partitions.

8. How does linger.ms influence performance, and what's a good value?

Answer:

It controls how long the producer waits to batch messages before sending.

- Low = low latency, high CPU/network
- High = better throughput, higher latency
  In our ETL use case, linger.ms=50 gave 30% higher batching efficiency without breaking latency SLAs.

9. What is num.network.threads and num.io.threads in broker config? How do you tune them?

Answer:

- network.threads: handles Kafka protocol requests
- io.threads: reads/writes messages from disk
  Start with:

```
num.network.threads=3
num.io.threads=8
```

Increase based on broker CPU & disk I/O. We tuned to 16 IO threads during a log replay spike for smoother throughput.

10. How do you tune consumer parallelism in a high-load pipeline?

Answer:

- Match **consumer instances to partition count**
- Use thread pools inside consumers if needed
- Enable **manual offset management** for batch commits
  For our fraud detection system, we used 50 partitions and 25 consumer pods, each with 2 worker threads — balanced well.

11. Scenario: Kafka controller CPU is spiking during topic creation. How to fix?

Answer:

- Reduce **frequency of topic creation/deletion**
- Avoid large numbers of partitions per topic
- Distribute controller role among healthy brokers (re-election)
  We moved to **Terraform + GitOps** for topic provisioning — fewer surprises, better visibility, and batch operations reduced controller strain.

12. What are some overlooked settings that can boost Kafka performance?

Answer:

- log.segment.bytes: smaller values = faster cleanup
- replica.lag.time.max.ms: faster detection of stuck replicas
- queued.max.requests: prevent overload during spikes
  We reduced log.retention.ms on debug topics to save disk without affecting key data.

13. How do you reduce consumer lag in near real-time analytics pipelines?

Answer:

- Increase max.poll.records
- Tune fetch.min.bytes and fetch.max.wait.ms for larger batches
- Profile and optimize sink latency
- Scale consumer group
  In our ad-tracking pipeline, setting max.poll.records=1000 and writing to Redis async helped cut lag by 80%.

14. Coding: How to configure a high-throughput producer in Python with optimal settings?

Answer:

```
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
```

```
    linger_ms=50,
    batch_size=65536,
    compression_type='snappy',
    acks='all',
    buffer_memory=67108864,  # 64MB
    retries=5
)
```

15. What is the effect of increasing log.retention.bytes vs log.retention.ms?

Answer:

- retention.bytes: limits log size based on disk usage
- retention.ms: based on time
  They interact — Kafka deletes segments when **either condition is met**.
  In cost-sensitive clusters, we keep tight retention.bytes on debug logs and longer retention.ms for core audit data.

16. What's the role of replica.fetch.max.bytes in replication performance?

Answer:

Controls how much data a follower can fetch per request.
Higher = fewer requests, better replication throughput.
We increased this to 1MB in one use case where replication was lagging on large messages.

17. Scenario: Producer throughput is fine but consumers are randomly stuck. What could cause it?

Answer:

Possible causes:

- Consumer GC issues
- Heartbeat failures → rebalances
- Unhandled deserialization exceptions
  We fixed it by:

- Using a watchdog for stuck consumers
- Adding try/except with logging around deserialization
- Switching to cooperative-sticky assignor

18. How does Kafka handle disk I/O pressure and what can you do about it?

Answer:

- Kafka uses **page cache** to minimize disk I/O
- But high throughput leads to segment churn
  Tune:

```
log.flush.interval.messages
log.segment.bytes
log.flush.scheduler.interval.ms
```

We added SSDs to high-traffic brokers and moved log directories using RAID 10 for IOPS optimization.

19. How do you monitor performance regressions in Kafka pipelines?

Answer:

- Use **baselined lag graphs** per topic
- Track produce.request.rate, fetch.request.rate
- Alert on UnderReplicatedPartitions, ISR Shrink Events
- Use anomaly detection on **request latency spikes**
  We correlate Kafka lag with downstream sink ingestion times via trace IDs.

20. How do you load test Kafka before production deployment?

Answer:

- Use **kafka-producer-perf-test.sh** and **kafka-consumer-perf-test.sh**
- Run synthetic load with production-like batch size, key distribution, and volume

- Monitor throughput, GC, disk usage, consumer lag
  We once simulated 500K messages/min to validate a new ETL pipeline, uncovering a bottleneck in RocksDB state store ahead of go-live.

Frequently Asked questions

1. How does Kafka ensure exactly-once semantics in stream processing?

Answer:

Kafka achieves exactly-once semantics using the **idempotent producer** and **transactional APIs**. The idempotent producer ensures that resending the same message doesn't result in duplicates, while transactional APIs allow grouping multiple operations into a single atomic transaction. This combination ensures that messages are processed exactly once, even in the event of failures.

2. Scenario: You need to process messages in the order they arrive for a specific key. How would you design your Kafka consumer setup?

Answer:

To maintain order for a specific key:

- **Partitioning Strategy:** Ensure that all messages with the same key are sent to the same partition. Kafka maintains order within a partition.
- **Single Consumer per Partition:** Assign one consumer to process messages from the partition to maintain order.
- **Consumer Group Management:** Use a consumer group with a single consumer instance or ensure that the number of consumers does not exceed the number of partitions to prevent rebalancing issues.

3. How do you handle schema evolution in Kafka, and what role does the Schema Registry play?

Answer:

Schema evolution is managed using the **Confluent Schema Registry**, which stores schemas for Kafka topics and ensures compatibility between producers and consumers. When evolving schemas:

- **Compatibility Modes:** Set compatibility modes (e.g., backward, forward, full) to control how schemas can change over time.
- **Versioning:** Each schema change results in a new version, allowing consumers to handle different schema versions gracefully.
- **Serialization/Deserialization:** Producers and consumers use serializers and deserializers that interact with the Schema Registry to ensure they read and write data in the correct format.

4. Write a Java code snippet for a Kafka producer that includes custom partitioning logic.

Answer:

```
import org.apache.kafka.clients.producer.*;

import java.util.Properties;

public class CustomPartitionProducer {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("partitioner.class", "com.example.CustomPartitioner");

        Producer<String, String> producer = new KafkaProducer<>(props);
```

```
    ProducerRecord<String, String> record = new ProducerRecord<>("my-topic", "key",
"value");
    producer.send(record);


    producer.close();
  }
}
```

In this example, com.example.CustomPartitioner is a user-defined class that implements the Partitioner interface to control the partitioning logic.

5. What are Kafka Streams' state stores, and how do they enhance stream processing?

Answer:

State stores in Kafka Streams are used to maintain stateful information across stream processing operations. They allow:

- **Windowed Operations:** Aggregating data over time windows.
- **Joins:** Maintaining state for joining streams.
- **Fault Tolerance:** State stores are backed by changelogs in Kafka topics, ensuring state recovery in case of failures.

State stores enable complex processing patterns like sessionization and real-time analytics.

6. Scenario: Your Kafka consumers are experiencing frequent rebalances. What could be causing this, and how do you mitigate it?

Answer:

Frequent rebalances can be caused by:

- **Consumer Lag:** Consumers not processing messages quickly enough, leading to timeouts.

- **Unstable Consumer Group Membership:** Consumers joining and leaving the group frequently.
- **Heartbeat Failures:** Network issues causing missed heartbeats.

Mitigation strategies include:

- **Optimize Processing:** Ensure consumers process messages efficiently.
- **Increase Session Timeout:** Adjust session.timeout.ms to allow more time before considering a consumer dead.
- **Monitor and Scale:** Use monitoring tools to detect issues and scale consumers appropriately.

7. Explain the role of Kafka's log.segment.bytes and how adjusting it impacts performance.

Answer:

log.segment.bytes defines the size of a single log segment file in bytes. Adjusting this parameter impacts:

- **Disk I/O:** Smaller segments lead to more frequent file creation and potential I/O overhead, while larger segments reduce this but may delay log compaction and deletion.
- **Log Retention:** Larger segments can delay the deletion of old data if retention is based on segment files.
- **Recovery Time:** Smaller segments can speed up recovery time after a failure as there are fewer messages to replay.

8. How does Kafka's exactly-once processing differ from at-least-once and at-most-once semantics?

Answer:

- **At-Least-Once:** Messages are delivered one or more times, ensuring no loss but potential duplicates.

- **At-Most-Once:** Messages are delivered zero or one time, risking message loss but no duplicates.
- **Exactly-Once:** Messages are delivered once and only once, ensuring no loss and no duplicates. Kafka achieves this using idempotent producers and transactional APIs.

9. Scenario: You need to integrate Kafka with a legacy system that only supports RESTful APIs. How would you achieve this?

Answer:

To integrate Kafka with a RESTful system:

- **Kafka REST Proxy:** Use Confluent's Kafka REST Proxy to produce and consume messages over HTTP.
- **Custom Middleware:** Develop a middleware service that interfaces with Kafka and exposes RESTful endpoints for the legacy system.
- **Kafka Connect:** Utilize Kafka Connect with appropriate connectors to bridge between Kafka and the RESTful service.

10. What is the impact of setting acks=0 in a Kafka producer configuration?

Answer:

Setting acks=0 means the producer does not wait for any acknowledgment from the broker before considering the message sent. This results in:

- **High Throughput:** Reduced latency as the producer doesn't wait for responses.
- **Risk of Data Loss:** Messages may be lost if the broker fails before writing the message to disk.

This setting is suitable for scenarios where performance is prioritized over durability.

11. How do you monitor Kafka's consumer lag, and why is it important?

Answer:

Consumer lag is the difference between the latest offset and the consumer's committed offset. Monitoring lag is crucial because:

- **Indicates Processing Delays:** High lag suggests consumers are not keeping up with producers.
- **Affects Real-Time Processing:** Ensuring low lag is essential for real-time applications.

Tools like **Confluent Control Center**, **Burrow**, and **Kafka Manager** can monitor consumer lag.

13. How does Kafka handle backpressure when consumers are slower than producers?

Answer:

Kafka manages backpressure through the following mechanisms:

- **Consumer Lag Monitoring:** Consumers track their offsets, allowing monitoring systems to detect lag and take corrective actions.
- **Flow Control:** Consumers can control the rate of data consumption by adjusting parameters like fetch.min.bytes and fetch.max.wait.ms.
- **Scaling Consumers:** Adding more consumer instances to a consumer group can help distribute the load and reduce lag.

In practice, we implemented dynamic consumer scaling based on lag metrics, which helped in maintaining optimal processing rates during traffic spikes.

14. Scenario: A Kafka topic has 10 partitions, but your consumer group has only 5 consumers. How are the partitions assigned?

Answer:

Kafka assigns partitions to consumers in a consumer group such that each consumer can handle multiple partitions:

- **Even Distribution:** With 10 partitions and 5 consumers, each consumer will be assigned 2 partitions.
- **Rebalancing:** If consumers join or leave the group, Kafka will rebalance the partitions among the available consumers.

In a recent project, we observed that during consumer scaling events, rebalancing caused temporary processing delays, which we mitigated by tuning the session.timeout.ms parameter.

15. How do you implement a dead-letter queue (DLQ) in Kafka to handle message processing failures?

Answer:

Implementing a DLQ in Kafka involves:

- **Separate Topic:** Create a dedicated topic (e.g., my-topic-DLQ) to store failed messages.
- **Error Handling Logic:** In the consumer application, catch exceptions during message processing and produce the failed messages to the DLQ topic.
- **Monitoring:** Set up monitoring and alerts on the DLQ to promptly address issues causing message failures.

In our system, implementing a DLQ allowed us to isolate and analyze problematic messages without affecting the main processing flow.

16. Explain the role of ZooKeeper in Kafka and the impact of its removal in newer Kafka versions.

Answer:

ZooKeeper in Kafka is responsible for:TestGorilla

- **Cluster Management:** Maintaining metadata about brokers, topics, and partitions. terminal.io
- **Leader Election:** Coordinating partition leader elections.

With the introduction of KRaft (Kafka Raft), Kafka aims to remove the dependency on ZooKeeper by integrating metadata management directly into Kafka brokers, simplifying the architecture and reducing operational complexity.GitHub+1TestGorilla+1

Transitioning to KRaft in our environment reduced latency in leader elections and streamlined cluster management.

17. Scenario: You need to reprocess historical data from a Kafka topic. How would you achieve this without affecting real-time consumers?

Answer:

To reprocess historical data:

- **Create a New Consumer Group:** Start a new consumer group with a unique group ID to read from the beginning of the topic without impacting existing consumers.
- **Seek to Beginning:** Use the seekToBeginning method to reset the offset to the start of the topic.
- **Process Data:** Consume and process the historical data as needed.

In a recent use case, we successfully reprocessed terabytes of historical data by implementing a dedicated consumer group, ensuring real-time processing remained unaffected.

18. What are the trade-offs between using Kafka as a message queue versus a publish-subscribe system?

Answer:

Kafka can function as both:Medium

- **Message Queue (Point-to-Point):** Each message is consumed by a single consumer. Suitable for task distribution.
- **Publish-Subscribe (Pub-Sub):** Messages are broadcast to all consumers. Ideal for event-driven architectures.

Trade-offs:

- **Scalability:** Pub-Sub allows multiple consumers to process messages independently, enhancing scalability.
- **Complexity:** Managing offsets and consumer groups adds complexity in Pub-Sub models.

In our architecture, we used the Pub-Sub model for real-time analytics and the queue model for task processing, balancing scalability and simplicity.

19. How do you handle schema evolution in Kafka Streams applications?

Answer:

Handling schema evolution involves:

- **Schema Registry Integration:** Use Confluent Schema Registry to manage and version schemas.
- **Compatibility Modes:** Set compatibility modes (e.g., backward, forward) to ensure consumers can handle different schema versions.
- **Graceful Handling:** Implement logic to handle missing or additional fields gracefully.

In our Kafka Streams application, integrating Schema Registry allowed seamless schema evolution without downtime.

20. Scenario: Your Kafka Streams application is experiencing high latency. What steps would you take to diagnose and resolve the issue?

Answer:

To address high latency:

- **Profile the Application:** Identify bottlenecks in processing logic.
- **Monitor Metrics:** Track Kafka Streams metrics like process-rate, punctuate-rate, and commit-latency.
- **Optimize State Stores:** Ensure RocksDB state stores are configured optimally, considering factors like cache size and compaction settings.

In one instance, optimizing RocksDB configurations reduced our application latency by 30%.

21. Explain the concept of log compaction in Kafka and its use cases.

Answer:

Log compaction in Kafka is a mechanism that ensures the retention of at least the latest value for each key within a topic, even if older records exceed the configured retention time or size. This process selectively removes obsolete records, retaining the most recent update for each key. Stack Overflow+1Medium+1

Use Cases:

- **State Restoration:** Applications can restore their state after a crash by replaying compacted topics to retrieve the latest state of each entity. Stack Overflow+2Medium+2Confluent+2
- **Change Data Capture (CDC):** Capturing changes from databases where only the latest state of a record is relevant. kai-waehner.de+1Stack Overflow+1
- **Configuration Management:** Distributing configuration updates where only the most recent settings are applicable. Vendavo Engineering Blog

22. Scenario: You have a compacted topic, but consumers are still seeing multiple records for the same key. Why might this occur?

Answer:

Log compaction is an asynchronous process. When new records are produced to a compacted topic, they are appended to the log immediately. The compaction process runs in the background and may not have compacted the log yet. Therefore, consumers can see multiple records for the same key until compaction occurs.

23. How does Kafka ensure data integrity during log compaction?

Answer:

Kafka maintains data integrity during log compaction by:

- **Offset Preservation:** Offsets are immutable; even after compaction, the offset for a given record remains the same, ensuring consumers can maintain their position accurately. Confluent Documentation
- **Atomic Segment Replacement:** Compacted segments are created as new files and atomically swapped with the old segments, ensuring that consumers always see a consistent log.

24. What configurations control log compaction behavior in Kafka?

Answer:

Key configurations include:

- **cleanup.policy:** Set to compact to enable log compaction for a topic. learn.conduktor.io

- **min.cleanable.dirty.ratio:** Specifies the ratio of log that must be "dirty" before compaction is triggered. Redpanda
- **log.cleaner.min.compaction.lag.ms:** Defines the minimum time a message must remain uncompacted before it becomes eligible for compaction.

25. Scenario: After enabling log compaction, you notice increased disk I/O. What could be the cause and how do you mitigate it?

Answer:

Increased disk I/O can result from the log cleaner threads actively compacting segments. To mitigate:Ted Naleid's Notes

- **Throttle Cleaner Threads:** Adjust log.cleaner.threads to control the number of concurrent cleaner threads.
- **Optimize Dirty Ratio:** Increase min.cleanable.dirty.ratio to reduce the frequency of compaction. Ted Naleid's Notes+2Redpanda+2Zendesk Engineering+2
- **Monitor Resource Usage:** Use monitoring tools to observe disk I/O patterns and adjust configurations accordingly.

26. How does log compaction affect consumer offset management?

Answer:

Log compaction does not affect the sequential nature of offsets. Even if records are removed during compaction, their offsets remain valid. Consumers can continue from their last committed offset without disruption, though they may encounter gaps where records have been compacted away. Confluent Documentation

27. Can log compaction be used alongside time-based retention policies? If so, how?

Answer:

Yes, Kafka allows combining log compaction with time-based retention by setting the cleanup.policy to compact,delete. This configuration enables Kafka to retain the latest record for each key (compaction) while also deleting records based on time or size constraints.

28. What is a tombstone record in Kafka, and what role does it play in log compaction?

Answer:

A tombstone record is a message with a key and a null value. In log compaction, writing a tombstone for a key indicates that the record should be deleted. During compaction, both the tombstone and any prior records with that key are removed, effectively deleting the record from the log.

29. Scenario: You need to ensure that certain records are retained longer before compaction. How can you achieve this?

Answer:

Use the log.cleaner.min.compaction.lag.ms configuration to set a minimum time that records must remain in the log before they are eligible for compaction. This ensures that records are retained for a specified duration before compaction can remove them.

30. How does Kafka's log compaction process handle delete operations, and what configurations control the retention of tombstone records?

Answer:

In Kafka, a delete operation is represented by a tombstone record, which is a message with a key and a null value. During log compaction, tombstone records serve as markers indicating that all previous records with the same key should be removed.

Configurations controlling tombstone retention:

- **log.cleaner.delete.retention.ms:** Specifies the duration (in milliseconds) that tombstone records are retained before being eligible for deletion during compaction. The default is 24 hours. Adjusting this parameter allows consumers sufficient time to process delete markers before they are removed.
- **log.cleanup.policy:** Setting this to compact enables log compaction. To combine compaction with time-based retention, set it to compact,delete, allowing both processes to manage log segments.

Properly configuring these parameters ensures that delete operations are handled correctly, and tombstone records are retained long enough for consumers to acknowledge deletions before they are permanently removed.

31. What is the difference between delete.retention.ms and log.retention.ms in Kafka?

Answer:

In Kafka, both delete.retention.ms and log.retention.ms are configurations that manage the retention of records, but they serve different purposes:Stack Overflow

- **delete.retention.ms:** This setting applies to topics with the compact cleanup policy. It specifies the duration (in milliseconds) for which tombstone records (records with null values indicating deletions) are retained before they are eligible for removal during log compaction. The default value is 24 hours (86400000 milliseconds). This ensures that consumers have a window to recognize and process delete markers before they are purged. Apache Kafka
- **log.retention.ms:** This setting applies to topics with the delete cleanup policy. It determines the maximum time (in milliseconds) that a log segment is retained before it is discarded to free up space. Once this time elapses, the log segments are deleted regardless of whether they have been consumed. The default value is 7 days (604800000 milliseconds). Confluent Documentation

Understanding the distinction between these settings is crucial for configuring Kafka topics to handle data retention and deletion appropriately based on the desired cleanup policy.

32. Scenario: You have a compacted topic with a high volume of updates and deletes. How would you configure Kafka to ensure that delete markers (tombstones) are retained long enough for all consumers to process them?

Answer:

To ensure that delete markers are retained long enough for all consumers to process them in a high-throughput compacted topic, you should adjust the delete.retention.ms configuration:

- **Increase delete.retention.ms:** Set this parameter to a value that exceeds the maximum expected delay for any consumer to process the tombstones. For instance, if consumers may lag behind by up to 48 hours, set delete.retention.ms to at least 172800000 milliseconds (48 hours). This ensures that tombstones are available for consumers before they are removed during compaction.

- **Monitor Consumer Lag:** Regularly monitor consumer lag to ensure that consumers are keeping up with the topic's throughput. If lag increases, consider scaling the consumer group or optimizing consumer processing.
- **Adjust Compaction Frequency:** While delete.retention.ms controls tombstone retention, the frequency of log compaction is influenced by other settings like min.cleanable.dirty.ratio. Ensure that compaction is not occurring so frequently that it removes tombstones before consumers have processed them.

By carefully configuring these parameters and monitoring consumer behavior, you can ensure that delete markers are retained appropriately, allowing all consumers adequate time to process them.

33. How can you configure a Kafka topic to retain only the latest state of each key while also ensuring that delete operations are propagated to consumers?

Answer:

To configure a Kafka topic to retain only the latest state of each key and ensure that delete operations are propagated:

1. **Set Cleanup Policy to Compaction:**
    - Configure the topic with cleanup.policy=compact. This ensures that Kafka retains only the latest record for each key, removing older records during log compaction.
2. **Handle Delete Operations with Tombstones:**
    - To propagate delete operations, produce a record with the key set to the entity to be deleted and the value set to null. This null value acts as a tombstone marker, indicating that the record should be deleted.
3. **Configure Tombstone Retention:**
    - Set delete.retention.ms to specify how long tombstone records are retained before being eligible for removal during compaction. Ensure this value is sufficient for all consumers to process the tombstones.
4. **Monitor and Optimize Compaction:**

- Adjust min.cleanable.dirty.ratio and other compaction-related settings to balance compaction efficiency and resource utilization.

By implementing these configurations, the Kafka topic will maintain only the latest state for each key and properly handle delete operations, ensuring consumers receive accurate and up-to-date information.

34. What are the potential risks of setting a very low delete.retention.ms value in a compacted Kafka topic, and how can they be mitigated?

Answer:

Setting a very low delete.retention.ms value in a compacted Kafka topic can lead to several risks:

- **Premature Removal of Tombstones:**
  - If tombstones are removed before all consumers have processed them, some consumers may miss the delete markers and retain outdated or incorrect state information.Ted Naleid's Notes
- **Data Inconsistency:**
  - Consumers that lag behind may not be aware of deletions, leading to inconsistencies between the data they hold and the current state of the topic.

Mitigation Strategies:

- **Set Adequate delete.retention.ms:**
  - Configure delete.retention.ms to a value that accommodates the maximum expected consumer lag. This ensures that all consumers have sufficient time to process tombstones before they are removed.
- **Monitor Consumer Lag:**
  - Regularly monitor consumer lag metrics to ensure that consumers are keeping up with the topic's throughput. If significant lag is detected, investigate and address the underlying causes.
- **Optimize Consumer Performance:**

- Enhance consumer processing efficiency to minimize lag, such as by optimizing code, scaling out consumer instances, or allocating more resources.

By carefully configuring delete.retention.ms and actively monitoring consumer performance, you can mitigate the risks associated with premature removal of tombstones and maintain data consistency across consumers.

FREE RESOURCES

1. What is Apache Kafka?

https://www.youtube.com/watch?v=RDp33e3ttTE

https://www.youtube.com/watch?v=hNDjd9I_VGA&pp=ygUFa2Fma2E%3D

2. Kafka Tutorial

https://www.youtube.com/watch?v=tU_37niRh4U&pp=ygUFa2Fma2E%3D

3. Kafka fundamentals

https://www.youtube.com/watch?v=B5j3uNBH8X4&pp=ygUFa2Fma2E%3D

4. Kafka Project

https://www.youtube.com/watch?v=7n72snj0rqs&pp=ygUFa2Fma2E%3D

5. Kafka Architecture

https://www.youtube.com/watch?v=HUAa1Yg9NlI&pp=ygUSa2Fma2EgYXJjaGl0ZWN0dXJl

6. Kafka Security

https://www.youtube.com/watch?v=-HlRFh9GfWw&list=PLa7VYi0yPIH2t3_wc1tm1rHDO9tbtfX1T

7. Kafka Monitoring

https://www.youtube.com/watch?v=3fsquXqgb5w&pp=ygUQa2Fma2EgbW9uaXRvcmluZw%3D%3D