AMAZON KINESIS DATA ANALYTICS SCENARIO BASED Q&A

BY - SHUBHAM WADEKAR

1. All IoT sensors from the same region use the same partition key, causing hot shards. How would you redesign the partitioning strategy?

I would stop keying by region and instead key by something that spreads load while keeping only the ordering I truly need.

- Decide what must stay ordered. Usually we only need “events from the same device in order,” not “entire region in order.”
    - Use partition key = device_id. This spreads events across many keys (one per device) and preserves per-device ordering.
- If some devices are still very hot, shard their traffic further without losing useful ordering:
    - Use partition key = device_id#b{00..15} where the bucket is hash(event_id) % 16. Readers can merge per-device buckets if they need a single timeline.
- If you don’t need any per-device ordering at all (rare for telemetry), use a fully randomized partition key (for example, a hash of event_id). This maximizes distribution.
- Avoid region as a key. If you must group by region for downstream storage, pass region in the payload; partition key should be for load distribution, not grouping.
- For very skewed tenants, allocate “virtual shards” per tenant: tenant_id#b{00..N}. Heavy tenants get more buckets.
- Optional: use ExplicitHashKey only when you purposely want to route certain keys to specific hash ranges; otherwise rely on a good partition key for natural spread.

Result: traffic fans out across shards, per-device order (if required) is preserved, and the “all to one region key” hotspot disappears.

2. Your stream has 10 shards, but some are overloaded while others are idle. How would you analyze and fix the partition key design?

I would measure key distribution, confirm which shards are hot, and then fix the key so records spread evenly; I'd also reshard as a short-term relief.

Analysis

- Enable enhanced monitoring (shard-level CloudWatch). Check, per shard: IncomingRecords/IncomingBytes, WriteProvisionedThroughputExceeded, and IteratorAgeMilliseconds. Hot shards will show high traffic and throttles; idle shards show near zero.
- Sample records from a consumer and compute a frequency histogram of partition keys (or prefixes like tenant_id). If a few keys dominate, the key design is skewed.
- Compare shard hash key ranges vs observed keys. If a few logical keys map to a narrow range, you'll see consistent hotspots.

Fix

- Redesign the partition key to increase cardinality where ordering isn't required:
    - Prefer device_id / user_id over coarse keys like region or product.
    - For heavy keys, add buckets: key = logical_key#b{00..15} (or more for hotter keys). Readers merge results from the small bucket set.
    - If ordering isn't needed, use a random/hash-based key (for example, hash(event_id)).
- Short-term operational relief: reshard.
    - Split hot shards (increase their count) and merge idle ones.
    - Or switch the stream to on-demand mode so Kinesis auto-scales shards during bursts while you roll out the new keying scheme.
- Validate after change: watch shard-level metrics again; the goal is similar IncomingRecords across shards and near-zero throttles.
- Keep a canary: continuously track top partition keys and alert if any single key exceeds, say, 5–10% of traffic so you can add more buckets for that key.

Result: you fix the root cause (skewed keys), use resharding/on-demand for immediate stability, and continuously monitor to prevent new hotspots.

3. A financial pipeline needs strict ordering per customer but also high throughput. How would you design the partition key and shards?

I keep ordering at the customer level and scale by spreading different customers across many shards.

- Partition key = customer_id so all of a customer's events land in order on the same shard. Kinesis guarantees ordering per partition key.
- Size shards for aggregate throughput: each shard supports ~1 MB/sec or ~1,000 records/sec writes. Estimate fleet-wide writes and provision enough shards so many different customer_ids hash across them.
- Hot customers: if one customer alone can exceed a shard's limits, you can't split their events across shards without losing order. Options are to:
  • batch/aggregate their small events with KPL so you approach the 1 MB/sec ceiling efficiently,
  • throttle that customer (queue on the producer) to the shard's max, or
  • give extremely hot customers a dedicated stream (still one key → one shard path) so they don't starve others.
- Consumers: use Enhanced Fan-Out so each consumer gets dedicated 2 MB/sec per shard without competing; checkpoint with KCL so resharding is seamless.
- Headroom: start with on-demand mode or slightly overprovisioned shards; monitor IncomingBytes, IncomingRecords, and WriteProvisionedThroughputExceeded and adjust.

Result: strict per-customer order is preserved, and overall throughput scales by adding shards for the population of customers.

4. Traffic suddenly doubles, and resharding causes downtime. How would you design a shard strategy to handle spikes without disruption?

I’d combine an auto-scaling capacity mode with shard-aware practices so scaling is seamless.

- Use on-demand capacity for unpredictable spikes. Kinesis on-demand auto-adds shard capacity behind the scenes, avoiding manual resharding during bursts.
- If you must stay on provisioned:
  • Keep headroom (pre-split to a higher shard count before expected peaks).
  • Automate scaling: CloudWatch alarms → Lambda → UpdateShardCount to split/merge proactively.
  • Make consumers resilient: run via KCL, checkpoint frequently; KCL automatically picks up child shards during splits without downtime (expect a temporary iterator age bump, not an outage).
- Fix partition keys so load actually spreads: avoid coarse keys (like region). Use high-cardinality keys (device_id/user_id) and, for known hot keys, append small buckets (key#b00..15). Without good keys, extra shards won’t help.
- Buffer bursts upstream: put SQS/Kinesis Producer Library aggregation in front to smooth spikes while scaling completes.
- Watch the right metrics: shard-level IncomingBytes/Records, WriteProvisionedThroughputExceeded, GetRecords.IteratorAgeMilliseconds. Scale when any shard approaches limits, not just when table-wide metrics spike.

Result: with on-demand (or automated, proactive resharding) plus well-distributed keys and resilient consumers, spikes are absorbed without downtime.

5. Analysts say Athena queries on JSON data from Firehose are slow and costly. How would you redesign the pipeline?

I reduce scanned bytes and file count by landing columnar, well-partitioned data with bigger files.

1. Stop writing raw JSON for analytics
   Enable Firehose data format conversion and write Parquet (Snappy or ZSTD). Register the schema in Glue so Athena gets types instead of schemaless JSON.

2. Partition for common filters
   Write to S3 with dt=YYYY-MM-DD/hour=HH (and optionally region or tenant). Point the table at this layout and make analysts always filter on these columns.

3. Right-size files at write time
   Increase Firehose buffering hints so each object is 128 MB+ before upload, and set a sensible time buffer (for example, 1–5 minutes depending on latency SLA). Larger files mean fewer opens and better Parquet row-group pruning.

4. Keep partitions queryable instantly
   Prefer partition projection in Athena so new hourly partitions don't require a crawler pass. If you keep a crawler, set it to "new folders only."

5. Normalize upstream and prune columns
   Use a small transform (Firehose data transformation or Kinesis Data Analytics/Glue Streaming) to flatten nested JSON, drop unused fields, standardize types and timestamps, and optionally sort within partition by a frequent filter key to improve Parquet min/max stats.

6. Handle schema evolution safely
   Add new fields as nullable in the Glue schema; avoid type flipping. For fast-changing sources, land raw JSON to a separate archive prefix and keep curated Parquet as the analytics surface.

Result: Athena reads far fewer bytes from fewer, larger Parquet files in partitions analysts filter on, so queries become faster and cheaper.

6. Millions of small events per second create tiny files in S3. How would you optimize buffering, file format, or compaction?

I create fewer, larger columnar files at ingest and run lightweight compaction in the background.

1. Aggregate before S3
   If using Kinesis Producers, enable KPL record aggregation to pack many user records into one Kinesis record. If using Firehose, increase buffering size (aim for 64–128 MB) and buffering interval (seconds) to batch writes.
2. Write columnar, not JSON
   Turn on Firehose Parquet conversion (or write via Glue/EMR) so each object contains many records with row-group stats. Use Snappy or ZSTD.
3. Partition sanely
   Avoid minute-level partitions that explode folder counts; prefer hour. Keep one or two additional low-cardinality keys (for example, region or tenant) if they match query filters.
4. Async compaction
   Run a periodic Glue/EMR job to compact yesterdays or last hour's data: read small files within each partition and rewrite as 256–512 MB Parquet files. If you use Iceberg/Hudi/Delta, schedule built-in compaction/clustering and keep the streaming writer focused on appends.
5. Control downstream file creation
   In Spark/Glue writers, coalesce or repartition to target a specific number of output files per partition, and avoid too many small tasks.
6. Monitor and tune
   Track S3 object counts per partition, average object size, and Athena "bytes scanned." Alert when average file size drops below target so you can raise buffers or compaction frequency.

Result: ingestion produces bigger Parquet files and background compaction cleans up any residual small files, keeping S3 efficient and Athena fast.

7. Compliance needs raw JSON, but engineers need Parquet. How would you design Firehose/S3 to store both raw and curated data?

I keep two separate S3 paths fed from the same streamone for immutable raw JSON (for audits) and one for cleaned Parquet (for analytics).

- Producers write to a single Kinesis Data Stream.
- I attach two delivery paths:
    1. Raw path: Firehose with no transform writes exact JSON to s3://lake/raw/... (optionally gzip). Enable strong retention, access controls, and Object Lock if needed.
    2. Curated path: Firehose with data transformation and format conversion to Parquet writes to s3://lake/curated/.... I normalize fields (timestamps, types), drop junk, and keep only needed columns.
- I partition both by date/hour (and maybe region/tenant) so Athena reads only the slices it needs.
- I size files well by increasing Firehose buffering (target 128–512 MB objects) to avoid tiny files.
- I register only the curated Parquet table in Glue for analysts; the raw zone has its own table for occasional audits.
- I add lineage: each curated batch stores a pointer to the raw batch (manifest or object key) so we can trace every record back.
- I put lifecycle policies: raw moves to cheaper S3 tiers after N days; curated stays hot longer.

This keeps compliance happy with untouched JSON while engineers get fast, cheap queries on Parquetboth kept in sync automatically.

8. ML teams need real-time data, but Firehose delivers in batches. How would you design the architecture to give low-latency data?

I split the pipeline: keep Firehose for batch lake writes, and add a real-time branch off Kinesis for sub-second delivery to an online feature store.

- Producers send events to Kinesis Data Streams.
- Real-time branch (for ML/inference):
    - Consumers use Enhanced Fan-Out for low latency.
    - A Kinesis Data Analytics (Flink) app or a Lambda consumer compute features and writes to an online store (DynamoDB, Redis/ElastiCache, or SageMaker Feature Store Online). Reads from this store are ~millisecond for APIs.
    - Keep writes idempotent with a request_id so retries don't double-count.
- Batch branch (for analytics/training):
    - The same stream feeds Firehose → S3 Parquet (partitioned by date/hour). This gives durable history for training and BI.
- Keep schemas stable with Glue Schema Registry so downstream consumers don't break.
- Scale and reliability: use on-demand shards or proactive resharding; monitor iterator age and write throttles; add DLQs for bad records.
- Security: KMS encryption, private networking, and least-privilege IAM on both branches.

This gives ML teams fresh features in milliseconds without giving up the reliable, cost-efficient batch lake that Firehose provides.

9. Your Kinesis → Spark → S3 ETL pipeline fails due to uneven partitions and small S3 files. How would you fix it?

I fix two things at once: distribute load evenly from Kinesis into Spark, and make S3 writes produce fewer, larger files.

1. Spread data evenly from the source
   I check Kinesis shard hot spots. If one partition key is too hot, I add bucketing in the producer (for example, device_id#b00..15) so events land across many shards. That makes Spark micro-batches more balanced because each task reads from different shards.

2. Rebalance inside Spark before heavy operations
   Right after parsing, I narrow the columns I need and then rebalance:

- For uniform spread, I use repartition(N) to increase parallelism.
- For joins/aggregations, I use repartitionByRange or repartition(col("join_key")) to hash by the key that drives the shuffle.
- If there is key skew, I add "salting" (append a small random salt to the hot keys) so no single partition explodes.

3. Control micro-batch size
   In structured streaming, I keep a small, stable trigger (for example, 10–30 seconds) and cap intake per trigger if the source supports it. Smaller, predictable batches reduce skewed spikes and make file sizing more consistent.

4. Write fewer, larger files to S3
   Before writing, I coalesce to a target number of files per partition window. I also set writer options that limit file count, like maxRecordsPerFile or a format's target file size. My goal is 128–512 MB Parquet files, not thousands of tiny files.

5. Use a table format that can maintain layout
   If I write to Apache Hudi, Iceberg, or Delta Lake, I let the table handle clustering/compaction on a schedule. The streaming writer focuses on appends; a small background job compacts yesterday's or last hour's data into properly sized files.

6. Partition S3 the right way
   I partition by date/hour (and maybe region/tenant). I avoid minute-level partitions that explode the folder count. Analysts must always filter on these columns so Athena scans less.

7. Compact leftovers asynchronously
   If I inherit tiny files, I run a lightweight compaction job that reads small files within a partition and rewrites bigger Parquet files. I schedule it as a Flex/low-priority job so it's cheap and doesn't block streaming.

8. Validate and monitor
   I watch per-task input sizes, skew in shuffle metrics, S3 object counts, average object size, and downstream Athena "bytes scanned." I alert when average file size drops below target or when a single key dominates a batch.

Result: balanced partitions keep Spark stable, and S3 ends up with big Parquet files that are fast and cheap to query.

10. Raw JSON needs to be transformed into Parquet and enriched with reference data. Would you use Glue Streaming or Spark on EMR? Why?

I choose based on throughput, latency, and complexity. If the job is standard Kinesis-to-Parquet with small reference data and minute-level latency is fine, I use Glue Streaming (serverless, simpler). If throughput is very high, enrichment is complex or the reference data is large and frequently changing, I use Spark on EMR for more control.

When I pick Glue Streaming

- Throughput is moderate and predictable.
- Latency target is seconds to a couple of minutes.
- Reference data fits in memory and can be broadcast or refreshed periodically (for example, a small S3/Glue table).
- I want fewer moving parts: managed checkpoints, easy Kinesis connector, security configuration, and simple Parquet writes to S3.
- I can write to Hudi/Iceberg/Delta and schedule compaction separately.
  This gives me quick setup, serverless scaling, and lower ops overhead.

When I pick Spark on EMR

- Very high throughput or strict low latency that needs more tuning (custom batch intervals, fine control over executors, shuffle, and memory).
- Complex enrichment: large or SCD2 reference dimensions that don't fit in a broadcast join, requiring partitioned joins, Bloom joins, or point-in-time logic.
- Extra dependencies or native libraries (Geo, ML, specialized connectors) that are easier to manage on EMR.
- Need for autoscaling clusters, spot instances to cut cost, and tight control over compaction, clustering, and file layout at scale.
- Multi-sink fan-out (S3, Kafka, DynamoDB) with custom exactly-once semantics via foreachBatch.

How I'd decide in an interview scenario

- If the reference dataset is small/medium and updated hourly/daily, and we just need to parse JSON → Parquet with a few lookups, I choose Glue Streaming for simplicity and speed to production.
- If the enrichment table is big (millions of keys), changes frequently, and we need sub-second latency or complex joins, I choose Spark on EMR, repartition both sides by the join key, and manage memory and shuffles explicitly.

Common best practices for both

- Normalize schema early, cast types, and standardize timestamps.
- Partition S3 by date/hour and target 128–512 MB Parquet files.
- Use an ACID table format (Hudi/Iceberg/Delta) so late or corrected events can upsert cleanly.
- Keep checkpoints stable, and make the sink idempotent using unique event IDs and MERGE semantics.
- Monitor lag, batch duration, file counts/size, and downstream query cost.

11. A downstream team complains of high latency in your ETL output. How would you tune checkpointing, batch size, and parallelism?

I reduce how much each micro-batch does, make commits cheaper, and add the right amount of parallelism so batches finish well under the trigger interval.

checkpointing

- keep one stable checkpoint path in S3 for this job and never change it; slow/rotating paths make recovery and commits slow
- use a private S3 prefix with no lifecycle that deletes checkpoint files; enable SSE-KMS
- shrink state so checkpoint writes are small: add watermarks to stateful ops (aggregations, dropDuplicates) and keep watermark as low as business allows
- if using foreachBatch + upsert table formats (Delta/Hudi/Iceberg), commit once per batch, not per partition; avoid too many small commits

batch size / trigger

- pick a small, steady trigger (for example, 5–10 seconds) and then tune input per batch so processing time stays below that trigger
- for Kafka: set maxOffsetsPerTrigger (or maxBytesPerTrigger) so each batch is bite-sized
- for Kinesis: tune per-shard fetch limits (records per fetch, max fetch interval) so each batch pulls a stable amount
- keep parsing and projection early to reduce rows/columns before heavy work

parallelism

- increase cluster cores first; then set spark.sql.shuffle.partitions ≈ 2–3× total executor cores (avoid the default being too high/low)
- repartition by the join/aggregation key before big shuffles; if a key is skewed, add salting (key#b00..b07) so one reducer isn't overwhelmed
- avoid Python UDFs in hot paths; use Spark SQL or native functions for vectorized execution
- size output files sensibly (128–512 MB) so file commits don't dominate batch time

sinks and commits

- write to an upsert-capable table (Delta/Hudi/Iceberg) via foreachBatch; do one MERGE per batch per target table
- if writing plain Parquet, coalesce to a small, fixed number of files per partition to reduce commit overhead

quick validation loop

- measure per-batch "processing time" vs "trigger interval" in the streaming UI; the former must be well under the latter
- watch checkpoint write duration, shuffle read/write time, and any single task running much longer than others (skew)
- iterate: cap input → fix skew → right-size shuffles → confirm fewer, faster commits

result: smaller batches, cheaper checkpoints, and enough parallelism bring end-to-end latency down and keep it stable during normal peaks.

12. Spark Streaming consumers lag behind during spikes. How would you redesign the pipeline to handle backpressure and scaling?

I let the stream absorb spikes without choking, cap what each consumer takes, and scale both the source and the compute automatically.

control the firehose

- add a buffer in front if you don't have one: producers → Kinesis (or Kafka). the stream is the shock absorber
- for Kafka: use maxOffsetsPerTrigger (or maxBytesPerTrigger) so a spike doesn't produce giant batches
- for Kinesis: increase shard count or use on-demand mode; with Enhanced Fan-Out consumers get dedicated throughput per shard

auto-scale and parallelize

- add more executors/DPUs; use cluster auto scaling if available
- set spark.sql.shuffle.partitions ≈ 2–3× total cores; repartition on hot keys, and salt skewed keys so a few reducers don't stall the batch
- if a single tenant/device is hot, shard that key at the source (key#b00..b15) so events spread across shards and tasks

apply proper backpressure

- keep a short, steady trigger (for example, 5–10 seconds) so backlogs are visible quickly rather than building inside one huge batch
- bound state with event-time watermarks; large, unbounded state slows every spike
- for exactly-once sinks, use foreachBatch with idempotent upserts; if a batch has to retry, it won't duplicate data

make writes fast

- reduce tiny file creation; coalesce to target file sizes (128–512 MB)
- for Hudi/Delta/Iceberg, don't run heavy compaction/clustering inside the streaming jobschedule it separately so spikes don't fight maintenance

consumer resiliency

- use the KCL/Kafka source in structured streaming with frequent checkpoints; restarts pick up where they left off
- if you must reshard, KCL handles new child shards; expect a temporary iterator-age bump, not downtime

architecture options for big spikes

- split one heavy job into multiple pipelines by tenant/region or by topic to isolate hotspots
- fan-out from the stream: one branch writes Parquet to S3 (batch/BI), another branch serves low-latency use cases (Lambda/KDA → DynamoDB/Redis) so the analytics consumer isn't the only path

operate with guardrails

- alarm on iterator age, processedRowsPerSecond, and batch processing time; scale before iterator age climbs
- keep an SQS/Kinesis DLQ for poison messages so they don’t block the whole batch
- profile occasionally: remove unnecessary columns early, avoid wide rows, and replace UDFs with built-ins

result: the stream and consumers handle spikes gracefullybounded intake per batch, more shards/cores when needed, skew under control, and fast, idempotent writes prevent lag from snowballing.

13. For fraud detection, you need 30-sec customer windows. How would you design the windowing query for accuracy and low latency?

I process in event time (not processing time), maintain small 30-second windows per customer, allow a little lateness, and upsert results so late events can correct the window without double counting.

- Event time, not processing time
  I parse the event timestamp from the payload (normalize to UTC), store it as event_time, and drive all windows with it. This keeps windows accurate even if messages arrive late or out of order.
- Window + watermark
  I use a 30-second hopping window with a short slide (for example, 5 seconds) so the model gets fresh scores every few seconds:
  window(event_time, “30 seconds”, “5 seconds”) by customer_id.
  I add a watermark slightly larger than the expected delay (for example, 2 minutes): withWatermark("event_time", "2 minutes"). That bounds state and lets late events still update open windows.
- Dedup before aggregating
  If producers send a unique event_id, I dropDuplicates("event_id", "customer_id") within the watermark to avoid replays or at-least-once deliveries inflating counts.
- Output mode and sink semantics
  I use outputMode("update") and write via foreachBatch to an upsert-capable table (Delta/Hudi/Iceberg). I MERGE by a composite key (customer_id, window_start, window_end). If late events change a window, the MERGE updates the same row; no duplicates are appended.
- Features in the window
  I compute the features the fraud model needs (count, sum, distinct merchants, avg amount, max in last N seconds, etc.) inside the windowed aggregation per customer. Keep only necessary columns to reduce shuffle.
- Trigger/latency
  I use a short trigger (e.g., Trigger.ProcessingTime("5 seconds")). Batch processing time must stay < 5s so end-to-end latency ~ a few seconds while windows still cover 30s of activity.
- Guardrails
  Monitor state store size and batch duration. If state grows, tighten the watermark or reduce slide frequency. Keep shuffle partitions near 2–3× total executor cores and salt any skewed customer_ids.

Result: per-customer 30-second windows updated every few seconds, deduped and corrected for late data, with upserted outputs so downstream scoring sees accurate, low-latency features.

14. Sliding windows are giving duplicate counts. How would you fix the windowing strategy?

Sliding windows overlap by design, so "the same event appears in multiple windows." That's expected for sliding analytics. Duplicate counts usually come from two separate issues: replayed events and append-only sinks. I fix both.

- Remove event duplicates first
  Drop duplicates by a stable event_id within a watermark (withWatermark + dropDuplicates). This prevents replays or job restarts from double-counting the same event inside a window.
- Use update semantics, not append
  In sliding windows, each micro-batch refines existing windows. If you append results, you'll get multiple rows for the same (key, window). Instead:
  • Use outputMode("update")
  • Write with foreachBatch and MERGE by (key, window_start, window_end) into Delta/Hudi/Iceberg.
  Late or reprocessed batches update the same window row rather than inserting a new one.
- Ensure a consistent window key
  Always persist window_start and window_end (not just a label) so the MERGE key is deterministic. Align windows to UTC boundaries to avoid drift (e.g., 12:00:00–12:00:30, 12:00:05–12:00:35...).
- Pick the right window type for the metric
  If you truly need "one count per period without overlap," switch to tumbling windows (window("30 seconds") without slide). If you need sliding behavior for model features, overlapping counts are correctjust don't double-store them due to append.
- Watermark large enough, but not too large
  If the watermark is too short, late events get dropped (under-count). If too long, state grows and re-emits many updates. Set it to realistic lateness (for example, 2–5 minutes) and monitor.
- Handle skew and file churn
  If one key/window pair keeps showing multiple output rows, check for skewed keys causing partial aggregations to finish at different times. Repartition by key and salt hot keys. Coalesce output to avoid many tiny upserts that look like duplicates.
- End-to-end idempotency
  Use a batchId ledger (optional) in the sink to ignore accidental re-writes of the same micro-batch. Keep checkpointLocation stable so the job resumes from the last offsets and doesn't replay whole ranges.

Result: deduped inputs + upserted window rows eliminate duplicate records in storage. If you need non-overlapping counts, use tumbling windows; if you keep sliding windows, overlapping membership is expected but each (key, window) appears exactly once in the sink.

15. IoT data arrives late and out-of-order. How would you use event-time + watermarks to process it correctly?

I treat the device timestamp as the single source of truth (event-time), add a watermark to tolerate late arrivals, and make the sink idempotent so late records can update previous results.

What I do step by step

- Parse and normalize timestamps early
  I extract device_time from each message, convert to UTC, and store it as event_time. All windows, joins, and aggregations use this columnnot processing time.
- Add a realistic watermark
  I set a watermark equal to the maximum lateness I can tolerate (for example, 5 minutes if field data is usually within 2–3 minutes late). This tells the engine: "keep state for this long; accept late records until then; after that, close the window."
  Example concept: withWatermark(event_time, "5 minutes").
- Choose the right window type
  Tumbling windows for non-overlapping counts/sums per minute/5 minutes. Sliding windows if I need rolling features (like fraud). Session windows if data is bursty per device.
- Deduplicate within the watermark
  If messages can be retried, I drop duplicates using a stable event_id (or deterministic hash of device_id + event_time + payload) together with the watermark. This stops replays from inflating metrics.
- Use update semantics at the sink
  I write via foreachBatch (or upsert) into a table that supports MERGE (Delta/Hudi/Iceberg). The key is (device_id, window_start, window_end). If a late event arrives before the watermark closes the window, I update that row instead of appending a duplicate.
- Handle extremely late data
  Anything later than the watermark goes to a quarantine path. If needed, I run an off-hours backfill job that reprocesses those rare records and reconciles aggregates.
- Monitor and tune
  I watch state store size and lateness distribution. If I see frequent drops as "too late," I increase the watermark a bit; if memory/latency grows, I tighten it. I also verify windows align to fixed UTC boundaries so results are deterministic.

Result
Late and out-of-order IoT events are included correctly up to the watermark, windows finalize on time, and reports remain accurate without double counting.

16. Multiple window queries in Kinesis Data Analytics cost too much. How would you optimize query design and resources?

I reduce the number of passes over the stream, combine windows where possible, pre-aggregate early, and right-size the application so it does less work per record.

What I change in the SQL/Flink job

- Combine windows in one pass
  Instead of running separate queries for 1-min, 5-min, and 15-min, I compute the smallest granularity once (for example, 1-min per key), write that as an intermediate stream/table, and roll up to 5-min and 15-min by summing those 1-min buckets. This cuts CPU and state.
- Use keyed streams wisely
  Key by the true business key (for example, customer_id, device_id). Avoid unnecessary repartitions between operators. Fewer shuffles = less cost.
- Pick the lightest window that meets the need
  Tumbling is cheaper than sliding for the same span. If sliding is required, increase the slide (for example, slide every 10s instead of 1s) or switch to hop windows that approximate the same behavior with fewer firings.
- Pre-aggregate early (map-side combine)
  If multiple fields need sums/counts, compute them together in one aggregate per key/window instead of multiple distinct aggregates. Also project only columns you need to reduce serialization costs.
- Watermark and state TTL
  Set the watermark only as large as real lateness and configure state TTL/retention so old window state is cleaned promptly. Smaller state = lower memory and checkpoint I/O.
- Deduplicate before heavy work
  Drop duplicates up front using a compact keyed state (Bloom filter or a short TTL). You avoid counting the same event across multiple windows.
- Optimize joins
  If enriching with a static or slowly changing dimension, use a side input/broadcast (or reference table lookup) instead of a streaming join, so you don't maintain large join state. For time-bounded stream–stream joins, keep the join window tight.
- Batch outputs
  Sink results in controlled batches or upserts rather than many tiny writes. If landing to S3, target 128–512 MB Parquet files and avoid per-record writes.
- Right-size KDA resources
  Reduce parallelism that doesn't improve throughput, or increase it if operators are back-pressured. Tune checkpoint interval (e.g., 60–120s) for stability vs overhead. Disable exactly-once sinks where at-least-once is acceptable and you have idempotent upserts downstream.
- Measure and iterate
  Track operator CPU, busy time, records per second, checkpoint duration, backpressure, and state size. Drop or merge any operator that doesn't move the business metric.

Result:
By consolidating window logic, limiting state and shuffles, and tuning parallelism/checkpointing, the Kinesis Data Analytics app does far less work per record and costs drop, while latency remains within SLA.

17. Athena queries are costly because Firehose writes raw JSON to S3. How would you optimize the pipeline for faster, cheaper queries?

I stop landing raw JSON for analytics and switch to a columnar, partitioned layout with bigger files. I still keep a separate raw zone for compliance.

- Keep two S3 paths from the same stream
  raw zone for exact JSON (gzip) and curated zone for analytics in Parquet (or Iceberg/Hudi). Raw is for audits; curated is what Athena queries.
- Turn on Firehose format conversion
  enable conversion to Parquet (Snappy or ZSTD). This alone cuts scanned bytes dramatically because Athena reads only needed columns and uses min/max row-group stats.
- Add a lightweight transform before conversion
  flatten nested JSON, cast types, normalize timestamps to UTC, drop unused fields. Less width = fewer bytes scanned.
- Partition by common filters
  dt=YYYY-MM-DD/hour=HH (and optionally region or tenant). Teach analysts to always filter on these columns so Athena prunes partitions.
- Right-size files at ingest
  increase Firehose buffering (size + interval) so each Parquet file is ~128–512 MB, not thousands of tiny objects. Bigger files reduce open/seek overhead and improve row-group pruning.
- Register curated tables once
  create a Glue table over the curated path. Prefer partition projection in Athena so new hourly partitions are queryable immediately without extra crawler runs.
- Compact if needed
  if earlier data has many small files, run a periodic Glue/EMR compaction job that rewrites each partition into fewer large Parquet files.
- Optional lakehouse table
  if you need upserts/ACID and long-term maintenance, write to Iceberg/Hudi and run scheduled clustering/compaction; Athena reads the latest snapshot efficiently.

Result: Athena now scans a tiny fraction of the bytes (columnar + partition pruning + larger files) and queries get faster and cheaper, while raw JSON is still preserved separately for compliance.

18. Your Firehose → S3 → Redshift COPY pipeline has minutes of delay. How would you reduce latency while staying cost-effective?

I shorten buffers end-to-end, keep files COPY-friendly, and trigger ingest immediately when files landonly moving to streaming ingestion if we truly need sub-minute latency.

- Tuning inside Firehose
  reduce buffering interval (for example, from 300s to 30–60s) and set a moderate buffer size (for example, 16–64 MB) so objects land more frequently but are still big enough for efficient COPY. Keep Parquet+Snappy/ZSTD to minimize load time.
- Auto-ingest into Redshift as soon as files arrive
  use Redshift Auto-Copy (S3 event → Redshift) or an EventBridge/Lambda trigger that runs COPY against only the new keys (or a manifest). This removes cron/polling gaps.
- COPY for speed and idempotency
  stage into a narrow staging table, then MERGE into the target. Use COMPUPDATE OFF STATUPDATE OFF during load (analyze later), and compress columnar input (Parquet) so COPY is fast. Keep transactions short: commit every batch.
- Balance file size vs latency
  aim for per-batch file sets totaling ~64–256 MB per COPY. Too small = many COPY overheads; too big = longer wait to fill buffers. Measure and adjust to hit your SLA.
- Isolate load resources
  dedicate a WLM (workload management) queue or a separate Redshift workload group to COPY jobs so they don't queue behind BI queries. Consider short statement_timeout to kill stuck loads quickly.
- Parallelism
  if you partition S3 by dt/hour/part=NN, run multiple COPY commands in parallel (up to a sensible limit) to saturate slices. Make sure you're not flooding the cluster with hundreds of tiny COPYs.
- Schema + sort/distribution
  load into sort-key-aligned tables. If the table is heavy on upserts, load to staging then MERGE; run ANALYZE (and VACUUM if needed) off-peak.
- If you truly need sub-minute latency
  switch the path delivering to Redshift to Redshift Streaming Ingestion (from Kinesis Data Streams/MSK) or Kinesis Data Analytics → materialized view in Redshift. Keep Firehose → S3 for the data lake. This costs more than batched COPY but gives seconds-level freshness.
- Guardrails
  add dead-letter S3 for bad records; make COPY idempotent with manifests; monitor end-to-end lag (event time → S3 time → COPY start → rows visible). Alert if lag exceeds threshold.

Result: by shrinking Firehose buffers to sensible sizes, triggering COPY on arrival, and loading columnar batches in parallel, latency drops from minutes to well under a minutewithout paying for always-on streaming unless you actually need it.

19. Analysts query by date and region in Athena, but S3 isn't partitioned. How would you redesign the S3 structure?

I would convert the dataset to columnar Parquet and reorganize it into Hive-style partitions that match how analysts filter: date first, then region. Then I'd expose it via Glue Catalog with partition projection so queries prune folders without waiting on crawlers.

I would do this:

- Choose a partition layout that mirrors filters: s3://lake/curated/<dataset>/dt=YYYY-MM-DD/region=<REGION>/ and keep everything in UTC. If queries always start by date, putting dt first maximizes pruning.
- Rewrite data to Parquet with Snappy or ZSTD, selecting only needed columns and normalizing types. Target 256–512 MB files per partition to reduce file-open overhead.
- Register a single authoritative Glue table over the curated prefix. Enable Athena partition projection (define dt range and region values), so new daily/hourly partitions are queryable immediately without a crawler.
- Keep the old non-partitioned data as raw/archive; point analysts to the curated table. Backfill recent history into the new layout, then run a one-time CTAS or Glue job to migrate older ranges as needed.
- Teach usage: every Athena query should include dt BETWEEN … AND … and region IN (…). Provide example queries and a view that exposes only approved columns to keep scans small.
- Ongoing hygiene: compact tiny files that slip through; watch "bytes scanned" in Athena and average S3 object size per partition; add lifecycle rules to transition cold partitions to cheaper S3 tiers.

Result: Athena can skip entire folders for dates/regions not requested, scans far fewer bytes due to Parquet and partition pruning, and queries become both faster and cheaper.

20. During heavy campaigns, Firehose → Redshift queries slow down due to skew. How would you optimize integration for spikes?

I would make batches more COPY-friendly, eliminate slice skew in Redshift with better distribution/sort choices, separate load from query workloads, and scale the right pieces only during spikes.

I would implement:

- Land COPY-friendly files: keep Firehose conversion to Parquet; set buffering to produce 64–256 MB per batch and at least 1–2 files per Redshift slice per load. Many tiny files or single giant files create skew and slow COPY.
- Auto-ingest quickly but in parallel: use Redshift Auto-Copy or EventBridge to trigger COPY as soon as new objects land. If you write N part files, run up to a sensible number of concurrent COPY commands (or one COPY over a manifest that lists many files) so all slices work.
- Use a staging → merge pattern: load into a narrow staging table with COPY options like COMPUPDATE OFF STATUPDATE OFF (analyze later), then MERGE into the target. Keep each transaction small to avoid table locks during spikes.
- Fix distribution and sort so slices do equal work: prefer DISTKEY on a high-cardinality column used in joins (or DISTSTYLE AUTO on RA3); avoid ALL/EVEN that create skew with hot keys. Choose a SORTKEY that matches ingestion or query predicates so new data co-locates and scans fewer blocks.
- Isolate resources: dedicate a WLM/workload group or queue for COPY jobs so they don't queue behind BI queries. Enable Concurrency Scaling for the BI queue so user queries don't pile up when loads are busy. Consider short statement timeouts on COPY to shed stuck loads.
- Smooth ingress when traffic explodes: lower Firehose buffering interval during spikes only if you still meet efficient file sizes; otherwise keep buffers steady and let Redshift ingest continuously. If input is extremely bursty, buffer through Kinesis Data Streams and scale consumers.
- Handle data skew in the batch itself: if one campaign/tenant dominates, shard output keys (e.g., add hash bucket to S3 object names and to a staging column) so COPY distributes evenly; avoid partitions that funnel all hot rows to a few slices.
- Post-load maintenance off the hot path: run ANALYZE after large loads (and VACUUM only when necessary) during off-peak. Don't block streaming loads with heavy maintenance.
- Measure and iterate: watch COPY throughput per slice, skew (max vs min rows per slice), WLM queue wait, and commit time. Adjust file count per load to keep all slices equally busy.

Result: Firehose delivers balanced Parquet batches, COPY runs in parallel across slices without skew, loads no longer starve queries, and brief spikes are absorbed by better batching plus isolated, scalable Redshift resources.

21. IoT sensors sometimes send events late. How would you design your pipeline to still do correct time-based aggregations?

I treat the device timestamp as the source of truth (event time), allow a controlled amount of lateness with watermarks, and make my sink upsert-friendly so late events can fix earlier results without double counting.

- Read and normalize timestamps early
  Parse device_time from each record, convert to UTC, store as event_time. All windows and joins run on event_time, not processing time.
- Add a realistic watermark to tolerate lateness
  Choose a watermark slightly larger than typical delay (for example, 5–10 minutes if most sensors are <3 minutes late). This keeps window state open long enough to incorporate late arrivals but prevents unbounded memory growth.
- Use the right windowing
  Tumbling (non-overlapping) for "per minute/hour" KPIs, or sliding/hopping for rolling metrics. Key by device_id or the business key you aggregate on.
- De-duplicate within the watermark
  If records can replay, drop duplicates by a stable event_id (or a hash of device_id+event_time+payload) so retries don't inflate counts.
- Upsert results, don't append
  Write aggregates via an upsert-capable table (Delta/Hudi/Iceberg). Use a composite key like (device_id, window_start, window_end). When a late event arrives before the watermark closes, MERGE updates the same row.
- Handle very late data safely
  Route "later than watermark" records to a quarantine S3 path with reason=too_late. If needed, run an off-hours batch backfill that recomputes the affected windows and re-MERGEs them.
- Keep latency low
  Short, steady triggers (for example, 5–10 seconds) and limited per-batch intake (maxOffsetsPerTrigger / Kinesis fetch caps) so processing time stays well under the trigger interval.
- Operability
  Monitor state store size, dropped-as-late counts, and batch duration. Tune watermark up/down based on real lateness distribution.

Result: aggregates stay correct as late events arrive, windows finalize predictably, and dashboards don't double-count.

22. Fraud detection suffers from out-of-order events. How would you fix user-level aggregations with watermarks or logic?

I aggregate in event time per user, allow bounded disorder with a watermark, and make outputs idempotent so out-of-order arrivals re-shape features without duplicates.

- Key by user (or card/account) and use event time
  All fraud features (counts, sums, distinct merchants) are computed in windows keyed by user_id on event_time, not arrival time.
- Add a watermark to bound reordering
  Set withWatermark(event_time, "X minutes") where X > typical disorder. This lets late but in-order events update the correct user window while keeping state bounded.
- Choose windowing that matches features
  Tumbling for "last 5 min" non-overlapping signals; hopping/sliding for rolling signals; session windows for bursty behavior. Persist window_start and window_end so the same window can be updated deterministically.
- Dedup and sequence checks
  DropDuplicates on a stable event_id within the watermark. If producers can send a per-user sequence number, reject or log any hard regressions (seq less than last seen) to catch device clock issues.
- Upsert window rows (exactly-once effect)
  Write via foreachBatch and MERGE by (user_id, window_start, window_end) into Delta/Hudi/Iceberg. Late events update the same feature row instead of inserting another copy.
- Side outputs for extreme cases
  Events later than the watermark go to a "too_late" stream/path. Optionally run an hourly/day-end reconciliation job that re-computes features for impacted users and windows.
- Joins and enrichments
  For device or merchant metadata, use broadcast/static tables (side inputs) versioned by effective_date if you need point-in-time correctness; avoid wide stream-stream joins unless strictly time-bounded.
- Low latency and backpressure control
  Keep short triggers (e.g., 5s), cap intake per batch, and salt any skewed user keys to avoid single-partition hotspots. Monitor iterator age/lag and scale shards/executors before the watermark starts dropping useful events.
- Validation and monitoring
  Track "updates per window," "late but accepted," and "dropped as late." Alert if dropped-as-late rises or if skewed users dominate processing time.

Result: user-level features are accurate despite out-of-order arrivals, late events fix the correct windows via upserts, and only truly too-late events are quarantined for optional backfill.

23. In ETL, data sometimes arrives out of sequence, but strict per-customer ordering is needed. How would you redesign keys/consumers?

I make the stream partition by customer and force a single, ordered lane per customer on the consumer side. I also add a tiny reorder buffer so small out-of-order bursts don't break correctness.

- Partition by customer so the stream gives per-customer order guarantees
  Use customer_id as the partition key (Kinesis/Kafka). This ensures all of a customer's events land in the same shard/partition and therefore have a defined order.
- Add a producer-side sequence or monotonic timestamp
  Include a per-customer sequence_number (ideal) or at least an event_time with millisecond precision. This lets the consumer detect and correct minor disorder.
- Use keyed, single-thread processing per customer in the consumer
  In Spark/Flink/KDA, keyBy(customer_id). This routes all events for a customer to the same task, preserving order during processing. Avoid operations that would re-parallelize by a different key before you finish the ordered logic.
- Add a small, bounded reorder buffer
  Maintain a per-customer buffer (e.g., last 30–120 seconds) keyed by sequence_number/event_time. Hold back just enough to wait for stragglers; flush in order. Anything older than the buffer window is treated as late and handled separately.
- Exactly-once and idempotency at the sink
  Use an upsert sink (Delta/Hudi/Iceberg/DB MERGE) keyed by (customer_id, business_id). Include request_id or sequence_number to deduplicate replays. If an event shows up "late but in sequence," it updates the same record/window instead of creating a duplicate.
- Scale by number of customers, not by reordering the same customer
  Increase shards/executors so many customers run in parallel. Do not split a single hot customer across multiple shards/tasksyou'd lose order. If one customer is extremely hot, rate-limit that customer, batch their events, or isolate them to a dedicated stream/consumer.
- Backpressure and recovery
  Keep stable checkpoints. Cap batch intake (maxOffsetsPerTrigger / Kinesis fetch caps) so a spike for one customer doesn't stall everyone. On restart, the per-customer buffer + idempotent sink avoid double-processing.

Result: each customer's events are processed by exactly one ordered lane with a tiny reorder buffer; outputs are idempotent, so correctness holds even with minor disorder or retries.

24. Late events are being dropped in tumbling windows. How would you adjust queries (e.g., grace periods, session windows)?

I let windows accept late events for a short “grace” period, process in event time, and write results as upserts so late arrivals fix windows instead of being dropped.

- Switch to event-time with a watermark and grace
  Define windows on event_time (not processing time). Set a watermark that reflects real lateness (for example, 5 minutes). Add an allowed lateness/grace period so tumbling windows stay “open” for late arrivals within that bound.
- Upsert window results instead of append
  Persist window_start and window_end and write via MERGE (Delta/Hudi/Iceberg) using (key, window_start, window_end). When a late event arrives within grace, re-emit the aggregate and update the same row. No duplicates.
- Tune window type if needed
  If you need strict, non-overlapping counts: keep tumbling but use grace. If users generate bursts with idle gaps, consider session windows with a small session timeout (for example, 5 minutes) so late events that belong to the same burst still attach to that session. If you need frequent refreshes, use hopping/sliding windows with update semantics.
- Deduplicate before the window
  DropDuplicates by a stable event_id within the watermark to prevent at-least-once delivery from inflating counts when windows are re-fired.
- Set realistic boundaries
  Grace too short → under-counts. Grace too long → bigger state, higher latency. Start slightly above the 95–99th percentile of observed lateness and adjust from metrics (accepted_late vs dropped_as_late, state size).
- Side output for “too late” events
  Route events later than the grace to a quarantine path for optional nightly backfill, so you never silently lose data.
- Operational guardrails
  Keep batch time < trigger interval, monitor state store growth, and watch the ratio of updates per window. Salt skewed keys if one user/window dominates processing.

Result: tumbling windows accept late arrivals within a clear grace period and correct the stored aggregates via upsert. Truly too-late events are isolated, not dropped, and your counts stop duplicating or under-reporting.

25. A consumer missed a day of data due to default 24-hour retention. How would you redesign for replayability?

I give myself a safe replay window on the stream and also keep a cheap, durable archive in S3 so I can reprocess any range even months later. I also make the sink idempotent so replays don't create duplicates.

1. extend retention on the stream
   Set Kinesis Data Streams retention to more than 24 hours (for example, 7–14 days, or longer if business justifies it). That gives me time to recover a stuck consumer without data loss.
2. add an always-on S3 archive
   Fan out the same stream to S3 (Kinesis Firehose or a Lambda consumer) writing compressed, partitioned objects by date/hour. This becomes the long-term "source of truth" for replays and audits.
3. make consumers checkpoint and be able to rewind
   Use KCL/Enhanced Fan-Out consumers with checkpoints in DynamoDB. If I must replay, I can:
   • reset the consumer to an earlier sequence number, or
   • start a temporary replay consumer with "AT_TIMESTAMP" pointing to the outage start.
   Keep the normal consumer paused until the replay catches up.
4. keep downstream writes idempotent
   Write to a sink that supports upsert/merge (Delta/Hudi/Iceberg, or database MERGE) keyed by a stable event_id. Reprocessing the same records will update the same rows instead of duplicating.
5. handle poison messages safely
   Send bad records to a DLQ (SQS/S3) so they don't block progress. Fix and replay DLQ later.
6. operational guardrails
   Alarms on iterator age, shard throttles, and consumer errors. Runbook includes exact steps to: pick timestamp → reset checkpoint → replay → verify counts → switch back.

Result: if a consumer goes down, I either rewind from the extended stream retention or reprocess from the S3 archive, and because the sink is idempotent I never double-count.

26. Compliance requires 6-month data retention. How would you use Kinesis + S3 to balance cost and replay?

I keep Kinesis retention only as large as I need for short-term recovery, and I store the full 6 months in S3 with the right layout, security, and lifecycle so it's cheap, queryable, and replayable.

1. dual-path design
   • real-time path: producers → Kinesis Data Streams (retention 7–30 days depending on recovery needs).
   • archive path: the same stream → Firehose (or Lambda) → S3 "raw" bucket, gzip/Parquet, partitioned by dt=YYYY-MM-DD/hour=HH.

2. compliance on S3
   Enable SSE-KMS, bucket policies, and least-privilege IAM. If required, turn on Object Lock (WORM) with legal hold/retention policies. Keep access logs and inventory.

3. cheap tiers and lifecycle
   Store recent months in S3 Standard or Intelligent-Tiering. Transition older partitions to Glacier Instant/Flexible Retrieval with sensible restore settings. This gives 6-month retention at low cost.

4. replay from S3 when needed
   If I must reprocess:
   • launch a one-off Glue/EMR job that reads the S3 partitions for the time window and writes to the target sink with upserts, or
   • rehydrate a "replay" Kinesis stream by reading S3 files and PutRecords back into Kinesis, then let normal consumers process.
   Because sinks are idempotent, replays don't duplicate data.

5. keep the archive analytics-ready
   Prefer Parquet for the curated archive so Athena/EMR can query directly for investigations. For strict "raw," land JSON gzip in a raw prefix and keep a curated Parquet prefix for analysis.

6. scale for spikes without cost runaway
   Use on-demand Kinesis or proactive resharding during peaks. Firehose buffers to produce 128–512 MB objects so S3/Athena are efficient. Compaction jobs run as low-priority (Flex) to keep cost down.

7. governance and auditing
   Track schema with Glue Schema Registry, write lineage pointers from curated objects back to raw batches, and keep CloudTrail/CloudWatch metrics to prove retention and access controls.

Result: Kinesis gives fast streaming with a short recovery window, S3 provides the 6-month compliant archive at low cost, and I can replay any time range safely and idempotently.

27. A schema error caused bad transformations for hours. How would you replay historical Kinesis data into S3/Redshift?

I fix the code, then reprocess the affected time window from a reliable source (stream retention or S3 archive), and write results idempotently so no duplicates appear.

1. stop the bad writer and cordon off bad data
   Pause the ETL job that writes to S3 or Redshift. Tag or move the bad S3 partitions to a quarantine prefix so analysts don't query them. In Redshift, stop downstream loads and, if needed, mark the impacted rows with a revision flag.
2. choose the replay source
   If the window is still inside Kinesis retention, use the stream directly. Otherwise, use the S3 raw archive written by Firehose or a Lambda archiver.
3. fix and version the schema/transform
   Patch the transformation to handle the schema correctly. Bump a pipeline version so runs are traceable. Add a unit test with the offending records.
4. reprocess into S3 (curated Parquet)
   Run a backfill job (Glue or EMR) limited to the dt/hour range. Read raw JSON from S3 (or from Kinesis with AT_TIMESTAMP), parse with the fixed schema, drop bad rows to a quarantine path with reason, and write Parquet to a staging prefix with big files (128–512 MB) and the same partition layout. When validated, atomically swap the staging prefix to the official curated prefix.
5. reload Redshift safely (no duplicates)
   Load curated data into a Redshift staging table with COPY (Parquet, COMPUPDATE OFF STATUPDATE OFF), then MERGE into the target using a unique event_id or natural key+timestamp so rows are updated or inserted exactly once. If the bad rows already exist, MERGE overwrites them; no delete-insert needed.
6. replay from Kinesis when within retention
   Start a temporary consumer application with a new consumer group and set starting position AT_TIMESTAMP to the beginning of the bad window. Run it with the fixed code, write to the same S3 staging prefix or directly to Redshift staging, then MERGE as above. When the replay catches up, stop the temporary consumer.
7. validate and reopen
   Compare counts and checksums between raw and curated for the window. In Redshift, run row-count and spot-value checks. Re-enable regular scheduled loads and point dashboards back to the curated table.
8. prevent a repeat
   Add schema validation (Glue Schema Registry or JSON schema), fail fast on incompatible changes, and add a canary that compares a sample of new output to the previous schema before promoting.

Result: the broken hours are reprocessed from a trustworthy source, S3 and Redshift are corrected via idempotent upserts, and analysts see clean data again without double-counting.

28. One consumer lags behind and misses SLAs. How would you use retention + checkpointing so it can catch up safely?

I make sure the stream retains data long enough, the consumer checkpoints correctly, and it can replay from where it left off without duplicating results.

1. extend retention to create a safety buffer
   Increase Kinesis retention from 24 hours to something safer (for example, 7–14 days). This gives the lagging consumer time to recover without data loss.
2. use proper checkpointing
   Run the consumer with KCL or Structured Streaming and store checkpoints in DynamoDB or S3. Each shard has its own checkpoint, so on restart it resumes exactly from the last processed sequence number.
3. let it replay from the checkpoint
   Do not reset to LATEST. Start the same consumer group; it will read the backlog until caught up. If you need a one-off backfill without disturbing the main consumer, spin up a temporary consumer in a separate group with starting position AT_TIMESTAMP for the missed window.
4. make the sink idempotent
   Write to a sink that can upsert by a unique event_id (Delta/Hudi/Iceberg on S3, or MERGE in Redshift). If the consumer retried or replayed, the same key updates the same row and does not create duplicates.
5. scale throughput while catching up
   Add shards (or switch the stream to on-demand) to increase read bandwidth. On the consumer side, add parallelism/executors; use Enhanced Fan-Out so other consumers aren't impacted. Tune batch size and fetch settings to maximize processing while keeping latency stable.
6. handle poison messages and backpressure
   Send bad records to a DLQ so they don't block the shard. Cap per-batch intake so processing time stays below the trigger interval. Monitor iterator age; if it keeps growing, scale further or split the workload by tenant/region.
7. guardrails and alerts
   Set alarms on iterator age, throttles, and consumer errors. Document a runbook: extend retention if needed, check checkpoint table, start catch-up consumer, monitor progress, then switch back to normal.

Result: with longer retention, correct checkpoints, and idempotent writes, the lagging consumer can safely replay and catch up without losing or duplicating data, and you have a clear path to scale during the catch-up period.

29. Your stream has 100 shards, but only 40% utilization. How would you reduce costs without losing throughput or order?

I right-size shard count based on real peak throughput, keep per-key ordering intact, and make scaling automatic so we don't pay for idle capacity.

- Measure what we truly need. I look at peak IncomingBytes/IncomingRecords per second across a recent busy period. Required shards ≈ max(write MBps)/1 MBps or max(records/sec)/1,000 (whichever is higher), plus 20–30% headroom.
- Merge shards safely. I use UpdateShardCount to decrease shard count during a quiet window. KCL/EFO consumers handle resharding automatically; ordering per partition key is still preserved because Kinesis ordering is by partition key, not by shard count.
- Fix key distribution first. If skewed keys keep a few shards hot, I add a small bucket to the partition key (key#b00..15). That lets me merge more shards without creating hotspots.
- Consider on-demand mode. If traffic is unpredictable, I switch to on-demand so capacity scales with usage and I pay only for actual throughput. Ordering per key is unchanged.
- Producer-side efficiency. I enable KPL aggregation and compression to push utilization up (fewer records/MB per event), so fewer shards are needed.
- Scale by schedule. For predictable spikes, I schedule pre-scaling (temporary shard increases) before a campaign, then merge back afterward.
- Watch consumers and EFO. I keep only the EFO consumers we need. Extra EFO consumers cost more; consolidating can save money without affecting order.
- Continuous guardrails. I alert when average utilization drops below, say, 50% for 24 hours or when any shard exceeds 70% so we can merge or split proactively.

End result: fewer shards (or on-demand) with even key spread keeps the same throughput and per-key ordering while cutting the monthly bill.

30. Firehose → S3 logs in JSON lead to high Athena costs. How would you redesign to save costs but keep analytics-friendly?

I keep raw JSON for compliance but make analysts query a curated Parquet table that's partitioned and written in larger files.

- Two-zone layout. Raw: gzip JSON to s3://lake/raw/... for audit only. Curated: Parquet (Snappy or ZSTD) to s3://lake/curated/..., which is what Athena queries.
- Firehose format conversion. I enable Firehose conversion to Parquet and add a lightweight transform to flatten nested fields, cast types, drop unused columns, and normalize timestamps to UTC.
- Partitioning that matches filters. I write dt=YYYY-MM-DD/hour=HH and optionally region or tenant. Analysts must filter on these columns so Athena prunes partitions.
- Bigger, fewer files. I raise Firehose buffering size/time so each Parquet object is ~128–512 MB. Fewer files = fewer opens and better row-group pruning.
- Glue Catalog + projection. I register one curated Glue table and turn on partition projection so new hours are queryable immediately without crawler runs.
- Compaction for legacy data. I run a small Glue job to compact tiny historical files into big Parquet files per partition.
- Optional lakehouse table. If we need upserts or late-arrival corrections, I land to Iceberg/Hudi and schedule clustering/compaction; Athena reads the latest snapshot efficiently.
- Cost guardrails. I publish sample queries, enforce "SELECT only needed columns" in views, and monitor Athena bytes scanned per query. I also add lifecycle policies: raw to Glacier after N days; curated older than M months to Intelligent-Tiering.

Result: analysts scan a fraction of the data thanks to Parquet + partition pruning + larger files, so queries are much faster and cheaper, while raw JSON remains available for audits.

31. Enhanced Fan-Out consumers are very costly. When would you choose standard consumers instead?

I pick standard consumers when my latency needs are modest and total read throughput per shard is not huge. Standard consumers share the 2 MB/s (or 5 transactions/sec) per shard, so they're cheaper when a few consumers can coexist without starving each other.

When I choose standard consumers

- I can tolerate extra latency (tens to a few hundreds of milliseconds) and occasional bursts in iterator age during peaks.
- I have only 1–3 consumers per stream (for example, ETL → S3, a metrics consumer, and maybe one more). Sharing the 2 MB/s works fine.
- Per-shard read throughput is low to medium (my consumers aren't trying to read at max rate simultaneously).
- My consumers can batch reads and checkpoint efficiently (KCL/SDK), and I'm okay tuning prefetch/batch size instead of paying for dedicated read pipes.
- Cost matters more than ultra-low latency; I'd rather scale shards slightly (or optimize processing) than pay for EFO.
- I don't need strict isolation across many teams/consumers. Coordination is simple.

When I still need EFO

- Many independent consumers must read the same shard at high rate, and I can't risk them throttling each other.
- Latency must be consistently low (EFO pushes records to each consumer with dedicated 2 MB/s).
- Multi-tenant analytics where each team runs its own consumer and we want isolation.

Practical approach

- Start with standard consumers; measure IteratorAgeMilliseconds and GetRecords throttles.
- If a specific consumer needs tighter SLOs, enable EFO only for that one, not for everyone.
- Keep consumer count low by doing fan-out inside one consumer (for example, one process reads the stream and publishes to multiple downstreams) instead of every downstream being its own consumer.

Result: standard consumers cut costs whenever latency and concurrency needs are moderate; I reserve EFO for the few consumers that genuinely need dedicated, low-latency throughput.

32. An ML team needs real-time features but a dedicated stream is costly. How would you reuse existing pipelines or cheaper options?

I branch from what we already have and add a low-latency path for features, without creating a brand-new stream per team.

What I would do

- Reuse the existing Kinesis Data Stream as the single source of truth. Add one additional consumer just for ML features instead of a new stream.
- If other consumers already use EFO, keep the ML consumer as a standard consumer to save costonly switch it to EFO if its latency SLOs truly require it.
- Compute features in-stream using a lightweight tool:
    - Kinesis Data Analytics (Flink) if I need windowed aggregations; write features to a fast online store (DynamoDB/ElastiCache/SageMaker Feature Store Online).
    - Or a small Lambda consumer for simple transforms (idempotent, batch writes to DynamoDB/Redis).
- Keep the batch lake intact: the existing Firehose → S3 path continues for offline training. Use the curated S3 data to backfill or validate features (offline/online parity).

Cheaper alternatives and patterns

- If the raw events already land in DynamoDB (or are stored there later), use DynamoDB Streams to derive featuresno extra Kinesis consumer cost.
- If you only need a subset of events, filter early (Lambda filter patterns or KDA SQL) so the ML consumer processes less volume.
- Use Enhanced Fan-Out only for the hot partitions: often unnecessaryoptimize the ML consumer (batch gets, efficient serialization) before paying for EFO.
- If latency can be ~100–300 ms, standard consumer + DynamoDB/Redis is typically cheap and good enough.
- For very small teams or prototypes, consider SQS as a tee: one existing consumer republishes the minimal feature payloads to SQS, and the ML service reads from SQS (cheap, simple).
- If your warehouse is Redshift with streaming ingestion already enabled, publish a materialized view in Redshift for near-real-time features and cache them at the app layer no extra stream.

Operational safeguards

- Make feature writes idempotent with a request_id to avoid double counting during retries.
- Monitor end-to-end freshness (event time → feature store write time). Alert if freshness exceeds SLO.
- Keep schemas stable with a registry and contract tests so ML doesn't break on new fields.

Result: the ML team gets low-latency features by branching off the existing stream and writing to an online store, while we avoid the cost of duplicate streams and keep the batch lake for trainingpaying for EFO only when truly necessary.

33. A consumer app lags by 2 hours. How would you check if it's due to shards, consumer speed, or downstream bottlenecks?

I break the problem into three layers: the stream, the consumer, and the sink. I look for where time is piling up and fix that layer.

stream (shards and key spread)

- Check shard metrics (IncomingBytes/IncomingRecords, WriteProvisionedThroughputExceeded, GetRecords.IteratorAgeMilliseconds) per shard. If only a few shards have high iterator age, it's a hot-key problem; split those shards and/or fix partition keys (add buckets like key#b00..15). If all shards show low iterator age, the stream is healthy; the bottleneck is later.
- Confirm read capacity path. If using standard consumers, multiple readers may contend for 2 MB/s per shard; consider Enhanced Fan-Out for isolation if the reader is starved.
- If total traffic is higher than shard capacity, temporarily increase shard count or switch to on-demand mode.

consumer (speed and parallelism)

- For Lambda: check concurrent executions vs account/reserved limits, batch size/window, function duration, and error retries. If throttled, raise reserved concurrency, enable provisioned concurrency (to remove cold-start spikes), increase batch size (up to payload limits), and consider parallelization factor (with care for ordering). Watch IteratorAgeMilliseconds after changes.
- For Spark/KCL/KDA: check batch duration vs trigger interval, maxOffsetsPerTrigger (Kafka) or Kinesis fetch limits, number of executors/cores, and shuffle skew. Increase executors, cap per-batch intake, repartition on hot keys, and remove slow UDFs. If one stage drags, fix skew or salt keys.
- Verify checkpointing frequency and errors. Frequent failures force replays and inflate lag.

downstream sink (writes and locks)

- S3: too many tiny files or per-record writes slow batches; buffer and write 128–512 MB Parquet files.
- DynamoDB: look for throttles; increase RCU/WCU or add batching; make writes idempotent.
- Redshift: COPY queueing or WLM contention; give loads a dedicated queue, use MANIFEST/parallel COPY, and MERGE from staging.
- External APIs: rate limits or long latencies; batch and retry with backoff, or decouple via SQS.

triage checklist

- Is lag uniform across shards? yes → consumer/sink. no → shard/key skew.
- Is consumer CPU/memory maxed? yes → scale or optimize.
- Are sink retries/throttles high? yes → scale or batch the sink.
- After each change, watch iterator age trend; it should decrease steadily.

result: identify whether shards are underprovisioned/skewed, the consumer can't keep up, or the sink is slow, and apply the targeted fix so lag drains back to near real time.

34. During peak traffic, your Kinesis → Lambda pipeline throttles. How would you fix concurrency and scaling issues?

I tune Lambda's concurrency and batching for Kinesis, ensure shard-level parallelism is sufficient, and remove downstream bottlenecks so invocations finish fast.

make Lambda able to scale

- Increase account concurrency and set a higher reserved concurrency for the function so it isn't capped. With Kinesis as an event source, baseline concurrency is one concurrent per shard; with parallelization factor you can run up to 10 concurrent per shard (trades off strict per-shard ordering).
- Add provisioned concurrency if cold starts add latency during peaks.

optimize batch intake

- Raise batch size (records per batch) and/or batch window so each invocation processes more data (within 6 MB/10,000-record limits). This reduces invoke overhead and throttling pressure.
- Enable tumbling window (if suitable) to group records and cut invocations for aggregation use cases.

handle ordering vs parallelism

- If strict ordering per shard is required, keep parallelization factor = 1 and scale by adding shards. If ordering by partition key (not whole shard) is enough, you can increase parallelization factor but ensure your code and downstream can handle reordering across sub-batches.

reduce function time

- Increase memory (also increases CPU/network) to shorten duration. Use vectorized parsing, batch writes to sinks, and avoid per-record network calls. Make the handler idempotent so retries don't double-apply.
- Use async/bulk clients (DynamoDB BatchWrite/TransactWrite where appropriate, Redshift COPY via S3, S3 multipart uploads) instead of synchronous per-record writes.

remove downstream throttles

- DynamoDB: raise capacity or switch hot paths to on-demand; shard hot keys; batch writes.
- S3: write larger files, not per-record puts.
- Redshift: avoid per-record insertsbuffer to S3 and COPY in parallel; reserve a WLM queue for loads.

scale the stream correctly

- If producer spikes exceed current shard capacity, pre-scale shards for scheduled peaks or use on-demand streams. EFO can isolate reads if multiple consumers compete.

resilience and error handling

- Configure maximum retry attempts and failure destinations (DLQ/SNS) so poison batches don't stall the shard indefinitely; consider bisect on function error to isolate bad records.
- Monitor IteratorAgeMilliseconds, ConcurrentExecutions, Throttles, Duration, and Error metrics; alert before lag grows.

result: by giving Lambda adequate concurrency, increasing per-invoke work, accelerating the handler, and ensuring sinks can keep up, the pipeline absorbs peaks without throttling while preserving the ordering guarantees you need.

35. Spark jobs consuming Kinesis show backpressure. How would you fix it (scaling, checkpoints, batch tuning)?

I attack the three places that create backpressure: how fast we pull from Kinesis, how quickly a micro-batch finishes, and how safely we commit/checkpoint so retries don't snowball.

pull less per batch, more predictably
• Keep a short, steady trigger (for example, every 5–10 seconds) so batches are small and frequent.
• Cap intake per batch so a spike doesn't create a monster batch. For Kinesis connectors, set fetch limits (records/bytes and fetch time per shard) so each trigger reads a bounded amount.
• If the stream is under-sharded, add shards (or switch to on-demand) so each task reads less per shard. If multiple consumers compete, use Enhanced Fan-Out for isolation.

finish work faster (parallelism + less shuffle)
• Scale executors/DPUs up first; then set spark.sql.shuffle.partitions ≈ 2–3× total executor cores.
• Repartition by the key you join/aggregate on before the heavy step; salt skewed keys (append a small bucket 00..07) so one reducer isn't overloaded.
• Project and filter earlydrop unused columns and rows before joins/aggregates. Replace UDFs with built-ins to keep vectorized execution.
• Avoid per-record sinks: batch writes (DynamoDB BatchWrite, S3 large Parquet files, Redshift COPY via S3). For lake formats, write via foreachBatch and do one MERGE per batch.
• Increase executor memory + cores (more memory also speeds GC). Turn on adaptive query execution (AQE) if available to auto-fix skew/shuffle sizes.
• Keep output files big (target 128–512 MB Parquet). Too many small files lengthen commits.

checkpointing and state (don't let it balloon)
• Use one stable checkpoint location (S3) with SSE-KMS; never rotate paths.
• Add event-time watermarks to stateful ops and set them only as large as real lateness (for example, 2–5 minutes). Smaller state → smaller checkpoint I/O.
• Deduplicate within the watermark to avoid reprocessing storm from retries.
• Commit once per batch (not per partition), and avoid running compaction/clustering inside the streaming jobschedule those separately.

source and key design
• If a few partition keys are hot, fix at source (add buckets key#b00..15). More shards alone won't help a single hot key.
• For EFO consumers, verify each has capacity; for standard consumers, ensure they aren't contending for the 2 MB/s/shard read limit.

observe, then iterate
• Watch batch "processing time" vs trigger interval (must be comfortably lower), shuffle read time, the longest task in a stage (skew), checkpoint write time, and IteratorAgeMilliseconds.
• Tighten fetch caps until iterator age falls; if it still grows, add executors/shards or reduce per-batch work.

Result: bounded intake per batch, more parallel work with less skew, and lightweight checkpointingiterator age drops and the stream keeps up during spikes.

36. CloudWatch alarms for shard use give false positives on short spikes. How would you refine monitoring?

I make alarms “trend-aware” and multi-signal so brief blips don’t page us, but sustained stress or real errors do.

use longer windows + percentiles
• Alarm on a rolling p95/p99 over 5–15 minutes, not 1-minute averages. Short bursts won’t trip it; real pressure will.
• Track both IncomingBytes and IncomingRecords utilization (divide by per-shard limits) with metric math; alert if either stays >70–80% for N consecutive periods.

combine signals (composite alarms)
• Create a composite alarm that requires BOTH sustained high utilization AND a symptom:
– producers: WriteProvisionedThroughputExceeded > 0 or put failures;
– consumers: GetRecords.IteratorAgeMilliseconds above threshold for ≥2–3 datapoints.
This suppresses “busy but healthy” noise.

per-shard visibility, but group alerts
• Enable enhanced shard-level metrics. Alarm at the stream level only if X% of shards breach (for example, ≥20%) to avoid one noisy shard waking you up. Keep a separate, low-urgency alarm that pinpoints which shards are hot.

anomaly detection for seasonality
• Use CloudWatch Anomaly Detection on IncomingBytes with a band trained over 7–14 days. Alarm when utilization breaks out of the band AND lasts longer than, say, 10 minutesgood for campaign traffic patterns.

debounce and maintenance windows
• Require “M out of N” datapoints (for example, 3 of 5) and add a short evaluation period delay so single-minute spikes don’t alert.
• Suppress alarms during planned resharding or load tests with maintenance windows.

watch the right causes, not just effects
• Producers: alert on increased PutRecords.FailedRecords and high PutLatency.
• Consumers: alert on Lambda throttles/concurrency, function duration, and KCL error rates alongside iterator age.
• Downstream: add alarms on sink throttles (DynamoDB WCU/RCU, Redshift WLM queue wait) so you can see when “slow downstream” is the root cause.

actionable runbooks and autoscaling
• Tie alarms to an autoscaling action (increase shard count or switch to on-demand) only when the composite alarm fires for sustained periods.
• Page humans only if autoscaling didn’t resolve or if errors are present.

Result: alarms trigger on real, sustained problems (or actual errors), not brief spikes. You see which layer is stressed, avoid alert fatigue, and scale only when it truly matters.

37. How would you design a real-time recommendation engine using Kinesis → ETL → S3 → Redshift/Athena for both real-time and batch?

I split the platform into two paths from the same stream: a low-latency path that serves recommendations within seconds, and a batch path that builds history for training and BI. Both paths share schemas and IDs so they stay consistent.

Real-time path (serving in seconds)
• Producers send clicks, views, cart events to Kinesis Data Streams with a stable event_id and event_time (UTC).
• A streaming ETL (Kinesis Data Analytics or Glue Streaming/Spark) reads Kinesis, parses JSON, validates with a schema registry, de-duplicates by event_id, and enriches with small reference data (catalog, user segments) via a broadcast map.
• It computes real-time features per user/item (recent views, dwell time, last N categories, trending score). Windows are event-time based with a short watermark (for example, 2–5 minutes) to handle late arrivals.
• Features are written to an online store that is fast to read: DynamoDB (keyed by user_id or user_id#feature_set) or Redis/ElastiCache. Writes are idempotent using event_id so retries don't double-apply.
• The recommendation service calls the online store to fetch features and the latest candidate lists, then scores with a lightweight model (for example, SageMaker endpoint or on-box model). Latency target is tens of milliseconds for reads and inference.

Batch path (history for training and BI)
• The same stream fans out to Firehose or a streaming job that lands raw JSON to S3 raw/ (gzip) and curated Parquet to S3 curated/ with partitions dt=YYYY-MM-DD/hour=HH (and region/tenant if needed).
• A daily Glue batch compacts small files to 256–512 MB, reconciles any too-late events, and maintains a lakehouse table (Iceberg/Hudi/Delta) so upserts are easy.
• Redshift uses COPY from curated S3 into staging, then MERGE into fact tables (fact_events, fact_user_item_features). Athena reads the same curated tables directly for ad-hoc analysis.
• Offline training jobs (SageMaker Processing or EMR) pull long windows of events and labels from the curated Parquet tables to train models, then register the model and export feature definitions to keep online/offline parity.

Keeping real-time and batch consistent
• One schema registry governs event schemas. New fields are added as nullable; breaking changes are blocked.
• Every online feature has an offline definition and backfill job so training uses the same logic as serving.
• Feature store pattern: optionally write features to both an online store (DynamoDB/Redis) and an offline S3 table so you can reproduce training exactly.

Reliability, security, and cost
• Use on-demand Kinesis or proactive resharding; Enhanced Fan-Out only for consumers that need it.
• Idempotent upserts everywhere; checkpoints in durable storage; DLQ for poison messages.
• Encrypt S3 and streams with KMS, least-privilege IAM, private networking.
• Monitor iterator age, batch duration, feature freshness, S3 file sizes, and Redshift load times; alert if freshness SLO is breached.

Result: the real-time branch keeps features fresh for recommendations within seconds, while the batch branch maintains a clean historical lake feeding Redshift and Athena for analytics and model training. Both paths stay consistent by sharing schemas, IDs, and feature definitions.

38. How would you integrate Kinesis with ML pipelines so models get real-time features but raw data is still archived?

I fan out from Kinesis into two destinations: an online feature store for low-latency inference and an S3 lake for long-term archival and training. I make the online/offline features use the same definitions so there is no training–serving skew.

Low-latency features for models
• Kinesis is the single ingress. A streaming job (Kinesis Data Analytics or Glue Streaming) reads events, validates schema, de-duplicates by event_id, and computes feature aggregates per user/item with event-time windows and a small watermark.
• The job writes features to an online store: DynamoDB (strong keys like user_id#feature_set) or SageMaker Feature Store Online. Updates are idempotent and small; hot users can be sharded with a short suffix if needed.
• The model endpoint (SageMaker Real-Time or a service container) fetches features from the online store in a few milliseconds, runs inference, and returns recommendations or fraud scores. Optional caching in the app layer reduces repeated reads.

Archival and offline training
• The stream also feeds Firehose to write raw JSON to S3 raw/ for compliance (gzip, immutable, long retention) and curated Parquet to S3 curated/ for analytics and offline features.
• A daily or hourly Glue job compacts Parquet, corrects late data, and maintains an offline feature table (Iceberg/Hudi/Delta).
• SageMaker training jobs pull the offline feature table and labels from the curated S3 data to train new models. After training, the model is registered and deployed.
• To ensure parity, feature logic is shared between streaming code and offline batch (library/package or SQL definitions), and offline tables are backfilled to match online definitions.

Orchestration and reliability
• Checkpoints and exactly-once effects by using idempotent writes and MERGE in the offline table.
• DLQ for bad records; monitoring on feature freshness (event_time to feature_write_time), stream lag, and model latency.
• Use on-demand Kinesis or resharding to handle spikes; Enhanced Fan-Out only if another consumer starves the ML consumer.

Security and governance
• KMS encryption for Kinesis, S3, DynamoDB; strict IAM per component.
• Schema registry to manage evolution; non-breaking fields added as nullable.
• Lineage from curated Parquet back to raw JSON via batch manifests so any feature can be traced.

Cost controls
• Keep features narrow; store large payloads only in S3.
• Batch updates to DynamoDB/Redis; avoid per-record external calls.
• Right-size Firehose buffers for 128–512 MB Parquet files to keep Athena/Redshift cheap.

Result: models get fresh, consistent features from an online store powered by Kinesis, while every raw event is archived to S3 for training and audits. The same feature definitions power online serving and offline training, keeping accuracy high and operations predictable.

39. A global e-commerce app ingests via Kinesis. Fraud, BI, and DS teams need different SLAs. How would you serve multiple consumers without duplicating streams?

I keep one Kinesis Data Stream as the single source of truth and add separate consumers per SLA, each optimized for its job. I isolate them using consumer settings and fan-out internally so teams don't need their own streams.

Design

- One stream, many consumers. Producers write once. I create separate consumer applications: fraud (low latency), BI (near-real-time to S3/Redshift), and DS (feature building or sampling).
- Pick consumer type per SLA. Fraud gets Enhanced Fan-Out if it truly needs consistently low latency; BI and DS use standard consumers to save cost unless they conflict.
- Branching inside a consumer. Where helpful, I run one shared consumer that publishes to multiple outputs (for example, a single streaming job reads Kinesis and writes both features to DynamoDB and curated Parquet to S3). This reduces the number of independent readers.
- Back-pressure control. Each consumer caps intake per batch and checkpoints separately, so slow BI doesn't slow fraud. Fraud runs short triggers and small state; BI prefers larger batches for S3 efficiency.
- Downstream stores per need. Fraud writes features to DynamoDB/Redis and serves a model endpoint. BI writes Parquet to S3 (partitioned) and kicks Redshift COPY. DS can read curated S3 or subscribe as a standard consumer for online experiments.
- Replay and retention. Stream retention is at least 7–14 days. All events also land in S3 raw via Firehose so any team can replay or backfill without touching production consumers.
- Governance and schema. A schema registry guards changes. Each consumer validates and routes bad records to a DLQ. All outputs carry event_id so upserts are idempotent.
- Cost control. Use EFO only where justified. Keep BI file sizes 128–512 MB to minimize Athena/Redshift cost. Share enrichment lookups via a small cache to avoid repeating external calls across consumers.

Result: one stream powers many independent consumers with different SLAs, no duplication of streams, low coupling, and clear cost boundaries.

40. Your Kinesis → Lambda → S3 → Athena pipeline is expensive and hard to manage. How would you redesign it for cost and scalability?

I simplify the path, batch more efficiently, and land analytics data in a columnar, partitioned lake. I keep Lambda only where it adds value.

What I change

- Remove per-record Lambda where possible. Instead of Lambda writing each event to S3, I use Kinesis Firehose to batch and deliver directly to S3. I enable Parquet conversion and a lightweight transform in Firehose (or a single Lambda transform inside Firehose, which is cheaper and batched).
- Land analytics-friendly data. Curated path writes Parquet with Snappy/ZSTD, partitioned by dt/hour (and region/tenant if needed). Buffering is tuned so each file is 128–512 MB. This alone cuts Athena bytes scanned and request count massively.
- Keep raw for audits. In parallel, I write gzip JSON to a raw prefix for compliance. Analysts query curated only.
- Register metadata once. I register a Glue table for curated, with partition projection so new hours are queryable immediately without a crawler run.
- Optional stream processing. If I need enrichments or dedupe that Firehose can't do cleanly, I replace the Lambda chain with a single Glue Streaming or Kinesis Data Analytics job that reads Kinesis, enriches, and writes Parquet to S3. One job, batched commits, better control.
- Cost knobs. I use on-demand Kinesis if traffic is spiky, or right-size shards with KPL aggregation. I turn off unnecessary EFO consumers. I size Firehose buffers to reduce S3 PUTs and file counts.
- Athena hygiene. I publish sample queries that always filter by dt and region and select only needed columns. I add views to hide heavy or sensitive columns by default.
- Operations. I monitor Firehose delivery success, S3 object size, Athena bytes scanned, and end-to-end freshness. I keep DLQs for bad records. Lifecycle policies move old raw to Glacier and old curated to Intelligent-Tiering.

Result: fewer moving parts, big Parquet files, partition pruning, and batched delivery drop costs sharply and make the pipeline simpler to run, while performance and scalability improve.