LAMBDA SCENARIO BASED Q&A

BY - SHUBHAM WADEKAR

1. A 2GB file landed in S3, and your Lambda timed out while processing. How would you redesign the pipeline for large files?

I would stop doing heavy work inside one Lambda and split the flow into a small "router" and a durable backend that can handle big files without timeouts.

What I would build, step by step:

1. Keep the first Lambda tiny
   The S3 event triggers a small Lambda that only reads object metadata (bucket, key, size, versionId), writes an idempotency record, and decides the path. If the file is small, process inline. If it is large (like 2GB), hand it off to a state machine or a long-running service. This Lambda finishes in a few seconds, so no timeouts.

2. Use Step Functions to orchestrate big work
   I would start a Step Functions state machine with the object details. It gives me retries, backoff, error handling, and visibility.

3. Choose a processing pattern for large files
   Option A: Chunk in parallel with Lambdas
   • Plan chunk ranges using the S3 object size (for example 64–128 MB per chunk).
   • Map state fan-outs to parallel chunk Lambdas. Each chunk Lambda uses Range GET to read only its slice, processes it, and writes its output to a temporary S3 path.
   • A final "merge" task completes a multipart upload for the final file, or simply treats per-chunk outputs as independent rows and writes a manifest plus a _SUCCESS marker.

Option B: Offload to a longer-running engine
• Start an AWS Glue job, EMR Serverless job, or AWS Batch/ECS task with the S3 key as a parameter.
• These services have no 15-minute cap, can use more CPU/RAM, and are better for heavy parsing, compression, or ML steps.
• Results are written back to S3 in Parquet/CSV with a _SUCCESS marker.

4. Stream or filter to reduce bytes
   • Do not load the whole file into memory. Read and process in streams or blocks.
   • If the format is CSV/JSON/Parquet and I only need some columns/rows, I can use S3 Select to read just what's needed, reducing time and cost.

5. Make it idempotent and safe to retry
   • Use the object's bucket/key/versionId as the idempotency key in DynamoDB. If the same event comes again, skip reprocessing.
   • Every run writes to a unique temp prefix like s3://.../tmp/<versionId>/... and only promotes on success by writing a small _SUCCESS marker or updating a pointer/manifest.
   • Step Functions retries transient failures automatically; a DLQ path captures hard failures.

6. Secure and robust by default
   • Grant least-privilege IAM to read the source object and write only to target prefixes.
   • If objects are KMS-encrypted, give the role decrypt/encrypt on that key.
   • If VPC is used, add S3 and Step Functions endpoints to avoid network issues.

7. Monitor and control costs
   • Emit metrics for bytes processed, chunk counts, and duration.
   • Keep files right-sized (coalesce outputs to 128–512 MB) to avoid thousands of tiny objects later.

In simple words: the first Lambda just routes; big work runs in Step Functions using parallel chunk Lambdas or a Glue/Batch job; write outputs to a temp path and promote when done; use idempotency with versionId so retries don't duplicate; and stream data so nothing times out.

2. Lambda is triggered multiple times for the same S3 event, causing duplicate writes. How would you ensure idempotency?

I would design the pipeline so running the same event twice produces the same single output and does not double-write.

What I would put in place:

1. Use a deterministic idempotency key
   Take bucket + key + versionId from the S3 event and use that as the unique key. Store it in a DynamoDB table with a conditional write (PutItem with attribute_not_exists).
   • First attempt inserts the key and continues.
   • Any duplicate attempt sees the key already exists and exits early.

2. Write to a unique temp path, then "promote"
   Always write to a run-scoped prefix like s3://.../tmp/<versionId>/...
   • After a successful write, place a _SUCCESS marker or update a small pointer object (for example, latest.json or a manifest).
   • Readers and downstream jobs only look at paths with _SUCCESS.
   • If the same event runs again, the code sees _SUCCESS for that versionId and skips.

3. Add SQS between S3 and Lambda
   Point S3 notifications to an SQS FIFO queue (with content-based dedup using the same idempotency key). Let Lambda read from SQS.
   • FIFO dedup filters rapid duplicates.
   • SQS gives controlled retries and visibility timeout so only one consumer handles a message at a time.

4. Make the destination idempotent too
   If writing to a database, use upsert keyed by the idempotency key. If writing files, include the key in the output filename so a second run overwrites the same object rather than creating a second one.

5. Always use S3 versioning
   Process using the event's versionId so a new upload creates a new idempotency record. You never mix two different versions of the same key.

6. Operational hygiene
   Log the idempotency key in every message and metric. Route failures to a DLQ. Keep Lambda timeouts reasonable so partial work doesn't trigger many retries.

In simple words: record each object version in DynamoDB before work starts, write outputs to a temp path and promote only once, put SQS FIFO in front to dedup triggers, and design the sink to accept the same record twice without creating duplicates.

3. A Lambda consuming from Kinesis lags during traffic spikes. How would you fix scaling and consumer lag?

I fix this by attacking both sides: make the consumer (Lambda) read faster and make the stream handle higher throughput without creating backlogs.

Step 1: measure the problem
I look at two metrics: IteratorAge (how far behind each shard is) and the Lambda duration/concurrency. If IteratorAge climbs during spikes, the consumer cannot keep up.

Step 2: speed up Lambda consumption

- Increase batch size and batching window so each poll returns more data per invoke without extra overhead.
- Turn up parallelization factor so Lambda processes multiple batches per shard in parallel. This lifts the one-invoke-per-shard ceiling and helps catch up quickly.
- Raise the event source's max concurrency so all shards can run in parallel at your chosen parallelization factor.
- Make handler work per-record efficiently: avoid heavy per-record network calls; use bulk operations, connection reuse, and async I/O.
- Right-size memory for Lambda; more memory also increases CPU/IO and can lower duration.
- Use provisioned concurrency if cold starts during spikes add noticeable lag.
- Add failure controls: enable bisect-on-error and a small max retry age so a few bad records don't stall the shard.

Step 3: raise Kinesis capacity when needed

- If shards are at their write or read limits, scale shards up (split shards) or switch the stream to on-demand capacity so it elastically adds shards during spikes.
- If you have multiple consumers, consider enhanced fan-out so each consumer gets dedicated read throughput and lower latency.
- Use CloudWatch alarms on incoming bytes/records and IteratorAge to trigger resharding automatically (via a small Lambda).

Step 4: make production traffic easier to consume

- Compress and batch at the producer so each record carries more payload and the consumer does fewer per-record ops.
- Keep partition keys diverse so spikes spread across shards (avoid hot shardssee next answer).

Step 5: verify end-to-end

- After changes, check IteratorAge drops to near zero during peaks and that throttles/ProvisionedThroughputExceeded disappear.
- Load test with a replay or synthetic spike to validate headroom.

In simple words: let Lambda read more in parallel per shard, give it more CPU, reduce per-record overhead, and scale the stream's shards or use on-demand. Watch IteratorAge to confirm the lag is gone.

4. Your Kinesis → Lambda pipeline has hot shards from poor partitioning. How would you redesign keys?

Hot shards happen when many records share partition keys that hash into the same shard. I redesign keys so traffic spreads evenly, while preserving ordering only where it truly matters.

Step 1: decide what must stay ordered

- If you must keep per-user or per-order ordering, use that entity as the base key.
- If you don't need strict ordering, you can add randomness more freely.

Step 2: add entropy to the partition key

- Use a composite key: <entityId>#<salt> where salt is a small range like 0–7 derived from a stable hash of the entity and time bucket. This spreads load across shards but keeps stable mapping per entity+bucket.
- Or use a pure hash: md5(entityId + timeBucket) so high-traffic entities don't all hash into one range.
- If ordering is required per entity, do not vary the salt within that entity's ordering window; instead, give very hot entities their own dedicated key or even their own stream.

Step 3: control time-based bursts

- Include a coarse time bucket in the key (for example minute) only if you do not need ordering across those buckets. This prevents all events at the same instant from landing in one shard.
- If you need strict global ordering per entity, keep the entity key stable and increase shard count instead.

Step 4: shard strategy and growth

- Pre-split shards when you expect a surge, or use on-demand mode so Kinesis scales shard count automatically.
- Monitor partition key distribution (sample keys in producers) and shard metrics; adjust the salt range N if a few shards still run hot.

Step 5: producer hygiene

- Avoid predictable low-entropy keys like "US", "mobile", or a small set of tenant IDs.
- If a single tenant is extremely hot, route that tenant to its own stream or allocate multiple shards and keys just for that tenant, then fan-in downstream.

Step 6: consumer considerations

- After key redesign, verify that shards are balanced and IteratorAge stays low.
- Keep Lambda parallelization factor enabled so each shard's workload can process faster during spikes.

In simple words: keep ordering only where you need it, add a small, deterministic salt or hash to spread keys, scale shards (or use on-demand), and monitor distribution so no single shard carries the spike.

5. A downstream team needs strict event ordering per user. How would you design the Lambda consumer?

I make sure all events for the same user arrive on one ordered lane and that my consumer processes them one-at-a-time for that user, with retries that don't break order.

What I do, step by step

- Keep per-user ordering at the source: choose a partition key that is exactly the userId (for Kinesis or Kafka). That guarantees all of a user's events land on the same shard/partition in order.
- Add a buffer that preserves order per user: fan the stream into an SQS FIFO queue using a small Lambda or Kinesis Firehose. Set MessageGroupId = userId. FIFO guarantees in-order delivery within each group.
- Consume with Lambda from the FIFO queue: set batch size to 1–10 and a maximum batching window (e.g., 1–2 seconds) so you get small ordered batches per user group. Lambda processes one message group at a time in order.
- Make the handler idempotent: use a deterministic idempotency key (stream sequence number or eventId) and upsert downstream so retries don't duplicate writes.
- Commit only after success: for Kinesis sources, don't checkpoint a batch until all records in it succeed; enable bisect-on-error so a bad record doesn't block everything forever. For SQS FIFO, let a failure keep the message in-flight and retry; if it's poison, move it to a DLQ dedicated to that user group.
- Control parallelism safely: raise concurrency/parallelization factor to speed up different users in parallel, but keep ordering because each userId is its own FIFO message group. Use reserved concurrency so the consumer never overwhelms the downstream system.
- Backpressure and visibility: set a visibility timeout comfortably larger than the max processing time; add CloudWatch alarms on age-of-oldest-message per queue and per message group.

Simple picture: shard by userId → SQS FIFO with MessageGroupId=userId → Lambda with small batches, idempotent writes, and retries → downstream. Different users run in parallel; each user's events are strictly ordered.

6. Your DynamoDB → Lambda processor sends too many writes downstream. How would you throttle or batch them?

I insert a buffering layer and control concurrency so the downstream only sees manageable, batched requests.

What I put in place

- Buffer with SQS: route DynamoDB Streams into SQS (standard or FIFO). Lambda then pulls from SQS instead of Streams directly. SQS lets me batch, delay, and smooth bursts.
- Batch in the consumer: in the Lambda that reads SQS, collect up to N records or up to M seconds (maximum batching window) and write them downstream in one call:
    - Databases: group rows into INSERT … VALUES … or COPY-like bulk operations.
    - APIs: use bulk endpoints when available; otherwise send fixed-size chunks.
    - DynamoDB targets: use BatchWriteItem with retries on unprocessed items.
- Throttle with concurrency controls: set reserved concurrency on the Lambda so only K workers can run; that caps downstream QPS. If needed, implement a token-bucket limiter in code to pace external API calls.
- Tune event source parameters: increase batch size from SQS (e.g., 10–50) and set a small batching window (e.g., 1–5 seconds) to accumulate micro-batches without too much latency. For FIFO, keep MessageGroupId semantics if you need per-key order.
- Handle partial failures cleanly: when a batched write partially fails, retry only failed items (don't re-send the whole batch). Use idempotency keys so a retried item does not duplicate.
- Protect the downstream: add circuit-breaker logicon repeated 429/5xx from the downstream, slow down (exponential backoff), park messages with a short DelaySeconds, or temporarily route to a "parking" SQS queue. Use DLQ for poison messages.
- Monitor and autoscale the buffer: alarm on ApproximateNumberOfMessagesVisible and AgeOfOldestMessage. If backlog grows but downstream is healthy, you can safely raise Lambda reserved concurrency to drain faster.

In simple words: put SQS between Streams and the writer, batch records for bulk writes, cap Lambda concurrency to a safe level, retry only the failures with idempotency, and use alarms to keep backlog and throughput in balance.

7. Your Lambda writes many small Parquet files to S3, hurting Athena queries. How would you optimize the pipeline?

I would change the flow so data is batched before writing and files land in S3 as fewer, larger Parquet files. That makes Athena much faster and cheaper.

What I do step by step

- Add a buffer before S3. Instead of letting Lambda write every single record, I send events to Kinesis or SQS. Then a "writer" Lambda (or Firehose) reads in batches.
- Use Kinesis Data Firehose if possible. Firehose can batch automatically, convert to Parquet, compress, and write to S3 with target sizes (for example 128–512 MB). I set buffering by size/time so I get larger files without too much latency.
- If I must stay with Lambda-only, I micro-batch. The writer Lambda reads a batch from SQS, accumulates records in memory or in /tmp, writes one Parquet file per batch, and names it with a date folder and a run id. I keep file sizes around 128–512 MB.
- Partition the S3 layout well. I write to Hive-style paths like table/dt=YYYY-MM-DD/hr=HH/. That lets Athena read only what it needs.
- Compact small files later if needed. I run a periodic compaction job (EMR/Glue/EMR Serverless) that reads tiny files in a partition and rewrites them into larger Parquet files, then swaps the pointer and drops a _SUCCESS marker.
- Use the S3 optimized committer. That avoids slow rename patterns and reduces the chance of half-written partitions.
- Control concurrency so I do not flood S3 with thousands of small writes. I cap the writer Lambda's reserved concurrency and increase batch size first.
- Verify with Athena. After the change, I check scanned bytes and runtime. If still high, I trim columns at write time and use Snappy.

In simple words: stop writing per record, batch records, write bigger Parquet files into date partitions, and compact if needed. This cuts small-file overhead and speeds up Athena.

8. How would you use Lambda to mask PII (like emails) in real-time as data flows into the lake?

I would put Lambda in the ingestion path, detect PII fields, replace them with safe tokens or hashed values, and then write only the masked data to the analytics buckets.

How I build it

- Put an event stream in front. Producers send events to Kinesis or SQS. A Lambda subscriber processes each batch so I can scale and keep latency low.
- Detect PII reliably. I prefer schema-driven masking: the event schema tells me which fields are emails, phones, names. Where schema is not guaranteed, I add regex checks for emails and phones with a small allowlist/denylist to reduce false matches.
- Choose masking method per field.
    - Emails, phone numbers: tokenize or hash with a secret salt (for example HMAC-SHA256) so the same input maps to the same token. That keeps joinability without exposing the real value.
    - Names, free text: redact or replace with generic labels. Only keep what is needed for analytics.
    - If a business user needs reversible lookup (rare), use a token vault service so only a few secured apps can detokenize.
- Manage keys and salts securely. I store the salt or KMS data key in AWS Secrets Manager and read it with a cached client in Lambda. I rotate secrets on a schedule.
- Keep the data path clean. Lambda writes masked records to a “curated” S3 bucket/prefix. If raw copies are required for compliance, they go to a separate, restricted bucket with tighter IAM and KMS keys.
- Make it fast and cheap. Process in batches, reuse HTTP connections, and avoid heavy libraries. Increase Lambda memory to get more CPU for hashing. Use provisioned concurrency if cold starts affect latency.
- Prove it’s masked. I add unit tests on the Lambda, sample logs with masked outputs only, and a periodic scan job that checks the curated bucket for unmasked patterns. I also tag masked columns in the Glue Data Catalog.
- Enforce access control. Even after masking, I protect columns with Lake Formation. Only teams that need the tokenized fields can see them. Everyone else gets already-aggregated views.
- Be idempotent. Use a deterministic idempotency key per event so retries do not double-write. Write outputs to date-partitioned paths with a _SUCCESS marker.

In simple words: events come in → Lambda identifies PII fields → replace with tokens or redactions using a secure salt → write masked data to S3 partitions → lock down raw data and keys. This gives real-time masking with low latency and safe analytics.

9. One Lambda does ingestion + transform + load into Redshift but hits limits. How would you refactor with Step Functions?

I would split the work into small, reliable steps and let Step Functions orchestrate them. Lambda stays lightweight, and heavy work moves to services built for it.

What I change, step by step

- Land first, then process. The event lands raw data in S3 (ingestion). A tiny Lambda validates metadata and writes a run record in DynamoDB with a unique run_id and the S3 object versionId for idempotency.
- Orchestrate with Step Functions. EventBridge triggers a state machine with inputs like run_id, date, and S3 paths. The state machine owns retries, timeouts, and error handling.
- Transform outside Lambda. Use Glue job or EMR Serverless for the transform. Step Functions starts the job (synchronous integration), waits for completion, and captures row counts and output paths. The job writes Parquet to s3://.../curated/dt=.../run_id=... and drops a _SUCCESS marker.
- Load Redshift the right way. A Lambda (or the Redshift Data API Task) runs COPY from the curated S3 path into a staging table, then an upsert/merge into the final table. This avoids slow row-by-row JDBC from Lambda.
- Make it idempotent. The loader checks for _SUCCESS and for a "loaded" flag in DynamoDB keyed by run_id. If set, it skips safely. Outputs always use unique run prefixes, never overwrite in place.
- Add guards and backoff. Every Task state has retry policies for throttling and transient errors, with exponential backoff and a max attempts limit. Hard failures go to a Catch path that logs, alerts via SNS, and marks the run as failed.
- Parallel where safe. If the raw file is huge, use a Map state to process chunks in parallel (S3 Range GET) and then merge. If there are multiple independent tables, run their branches in parallel.
- Observability. Each step writes structured logs with run_id, input path, rows_out, and durations. The state machine publishes metrics and pushes a short run summary to a "runs" DynamoDB table for audit. CloudWatch alarms fire on failed executions or long durations.
- Security. Least-privilege IAM per step, KMS on buckets and Redshift, and VPC endpoints where needed.

Simple outcome: Lambda just validates and coordinates; Glue/EMR Serverless does the heavy transform; Redshift loads via COPY; Step Functions glues it all together with retries, idempotency, and clear visibility. No more 15-minute or memory limits.

10. A Lambda triggers Glue jobs, but failures aren't visible. How would you redesign orchestration for better error handling?

I would move orchestration into Step Functions (or Glue Workflows) so job state is explicit, failures are caught, and alerts are automatic.

What I implement

- Use Step Functions service integration for Glue. The state machine calls StartJobRun and waits synchronously for the run to finish. Success, failure, and job error messages are captured in the execution history.
- Add retries and catches. For throttling or transient S3/KMS errors, retry with exponential backoff. On persistent failure, a Catch state logs details, updates a run record in DynamoDB, and sends an SNS/Slack alert with the Glue jobRunId and links to logs.
- Enforce idempotency. The workflow writes a "processing" marker at start and a "completed" marker on success. If a re-run starts with the same inputs, it checks the marker and either resumes or no-ops.
- Surface logs and lineage. Each Glue job writes to CloudWatch Logs and a known S3 log prefix; Step Functions includes the log URLs in its output. The state machine also records input → output mapping (S3 path, partition, row counts) for auditing.
- Partitioned outputs with markers. Glue writes to a run-scoped S3 prefix and drops _SUCCESS only on completion. Downstream steps (including Athena refresh or Redshift COPY) depend on that marker, so partial runs are never consumed.
- Optional EventBridge rule. Also subscribe to Glue "Job State Change" events to push real-time notifications and to reconcile any out-of-band runs with the orchestrator's run ledger.
- Clear timeouts and hearts. Set per-step timeouts in Step Functions; long-running Glue jobs can heartbeat via CloudWatch metrics. If a job hangs, the step times out, raises an alert, and moves to cleanup.
- Rollup dashboards. Build a simple dashboard showing last N executions, success/fail, durations, and links to the exact Glue run and logs. On-call has one place to look.

If staying with Lambda only, I would at least poll GetJobRun until SUCCEEDED/FAILED, implement retries and a max wait, and push structured failure details to SNS. But Step Functions gives this out of the box, plus visual traces and clean error handling.

11. Your Lambda pipeline fails on bad “poison-pill” events. How would you use DLQs or retries to handle them safely?

I make sure bad events don’t block the whole pipeline and that I can inspect and fix them later.

What I set up

- Use a dead-letter path. If the Lambda is triggered by SQS or EventBridge, I configure a DLQ (another SQS queue). For Kinesis, I add a failure sink (for example, an SQS queue or Firehose) in the error path of my code.
- Add controlled retries. I keep a small number of automatic retries with exponential backoff. This handles transient issues. If the event still fails, I send it to DLQ with the exact error, timestamp, and a correlation id.
- Make the handler defensive. I validate and parse early. If a record is malformed, I attach a clear error code and push only that record to DLQ, while letting the rest of the batch succeed. For Kinesis, I enable “bisect on error” so one bad record doesn’t stall the shard forever.
- Use idempotency for side effects. I generate a deterministic id for each event and upsert downstream. If a retry happens, it won’t double-write.
- Quarantine and notify. I publish an alert when DLQ receives messages beyond a threshold. The DLQ message includes original payload, headers, and stack trace so debugging is fast.
- Have a replay plan. A separate “DLQ reprocessor” Lambda can fix or drop known-bad events and then resend good ones to the main queue, with safeguards to avoid loops.

In simple words: retry briefly, send stubborn records to DLQ with full context, keep the rest flowing, and provide a safe replay tool for later.

12. Thousands of failed Lambda events are in DLQ. How would you reprocess them without breaking order or duplicating?

I replay in controlled batches with idempotency and, if needed, per-key ordering.

Steps I take

- Freeze and analyze first. I sample the DLQ to understand why events failed. If it's a fixable bug, I deploy the fix before replaying.
- Reprocess via a replay tool. I write a small "replayer" Lambda or Step Functions job that reads from the DLQ in batches and sends messages back to the original input queue or a special "retry" queue.
- Preserve order when required. If order matters per key (for example userId), I move DLQ messages into an SQS FIFO queue with MessageGroupId = that key. Lambda then processes one group at a time in order, while many groups run in parallel.
- Enforce idempotency. Downstream writes use a deterministic id (for example event id or Kinesis sequence number). Writes are upserts or overwrite-by-key so replays don't double-insert.
- Rate limit to protect downstream. I cap Lambda reserved concurrency and set a fixed batch size. If the downstream shows 429/5xx, the replayer backs off and retries failed items only.
- Mark and skip already-processed. The replayer checks a DynamoDB idempotency table (or destination state) before sending; if an event was processed, it's skipped and archived.
- Audit and finalize. Successful replays are removed from DLQ; poison records that still fail go to a "quarantine" queue with a final error tag for manual review.

In simple words: fix the root cause, replay in batches through a FIFO path when ordering matters, use idempotent writes to avoid duplicates, and rate limit so you don't overload systems.

13. Some Lambda events are silently dropped. How would you design monitoring to detect and fix these issues?

I add end-to-end, gap-detecting monitoring so I know when events go missing and where.

What I implement

- Source-to-sink counters. I emit metrics for count in at the source and count out at the sink per time window and key (for example per partition or tenant). A simple CloudWatch Metric Math alarm fires if the gap exceeds a threshold.
- Mandatory acknowledgements. Each successful write emits a “processed” metric and a structured log line with the event id. I also track failures and DLQ counts.
- Dead-letter alarms. I set alarms on DLQ depth and age-of-oldest-message. If either grows, I get alerted quickly.
- Error budgets and SLOs. I define a target success rate (for example 99.9 percent processed within 5 minutes) and alert when burn rates exceed limits. This catches partial outages fast.
- Traceability per event. I pass a correlation id through the pipeline (request id or event id) and include it in every log and metric. I enable X-Ray where possible to see latency and failures across services.
- Idempotent sinks with audit tables. Downstream, I keep an audit table with event id, processed_at, and status. A scheduled job compares expected ids versus processed ids to detect silent drops.
- Canary inputs. I inject periodic known-canary events. If a canary doesn’t arrive at the sink in time, an alarm fires regardless of traffic volume.
- Backpressure visibility. I monitor IteratorAge (Kinesis), ApproximateAgeOfOldestMessage (SQS), Lambda throttles, and concurrency. Spikes here often correlate with silent drops due to timeouts or exhaustions.
- Guardrails in code. I fail loud on parsing errors (send to DLQ) rather than swallowing exceptions. I avoid blanket catch without logging. I add a final “no-op success” log at the end of the handler so missing logs indicate an early exit.

In simple words: count events at ingress and egress, alarm on gaps and DLQ growth, tag every event with an id for tracing, use canaries, and never swallow errorsalways log and route to DLQ so nothing disappears silently.

14. A Lambda → Redshift pipeline floods Redshift with concurrent COPYs. How would you control and batch the load?

I would put a queue in front of Redshift, batch files on S3, and allow only a small, controlled number of COPY commands at a time.

What I would do

- Buffer every load request in SQS instead of calling COPY directly from producer Lambdas. This decouples producers from Redshift.
- Add a single "loader" Lambda (or a small ECS task) that drains SQS and aggregates files by table/partition. It issues COPY only when either a size threshold (for example 1–5 GB) or a time threshold (for example 2–5 minutes) is met.
- Use S3 manifests so one COPY ingests many files at once. This turns hundreds of tiny writes into a single bulk load, which is what Redshift is best at.
- Enforce a concurrency limit. Give the loader a reserved concurrency of 1 per target table (or use a DynamoDB "semaphore" item per table). This guarantees at most N concurrent COPYs systemwide.
- Load to staging, then merge. COPY into a staging table, then run an upsert/merge into the target. This keeps COPY short and avoids table locks during user queries.
- Right-size files for slices. Aim for tens to low hundreds of files per COPY, each 100–512 MB (Parquet if possible). This balances parallelism across slices without tiny-file overhead.
- Put loads in their own WLM/queue. Use a dedicated Redshift queue (or workload management rules) for COPY, separate from user queries. Enable Concurrency Scaling if you often have overlapping loads.
- Retry safely. If a COPY fails, retry the same manifest; the merge step is idempotent. Keep an audit table with run_id, file counts, and row counts.

In short: queue → batch → single controlled loader → COPY with manifest → staging → merge. This stops the flood and makes loads predictable and fast.

15. A Lambda reading from SQS is throttled in bursts. How would you tune batch size, batch window, and concurrency?

I would let each Lambda invocation do more useful work, smooth the spike, and cap concurrency so downstream systems aren't overwhelmed.

My tuning steps

- Increase BatchSize so each poll gets more messages (for example from 10 to 25–50 for standard queues; keep 10 for FIFO). This lowers invoke overhead.
- Set MaximumBatchingWindowInSeconds to a small value (1–5 seconds). This allows Lambda to wait briefly and form fuller batches during spikes, without adding much latency.
- Raise function memory. More memory gives more CPU and network, reducing per-batch duration (higher throughput per concurrent worker).
- Control parallelism with MaximumConcurrency on the event source mapping and reserved concurrency on the function. This caps how many workers run in parallel during bursts.
- Use partial batch response. Acknowledge only the successfully processed messages and immediately retry just the failed ones instead of redelivering the entire batch.
- Tune visibility timeout to be greater than (max function time × retries). This prevents premature redelivery and duplicate processing during bursts.
- Enable long polling on the queue (ReceiveMessageWaitTimeSeconds 10–20) to reduce empty polls and throttling churn.
- If one queue can't keep up, shard by key across multiple queues (or add routing attributes) and run one event source mapping per shard.
- Protect downstreams. Implement client-side rate limiting or token-bucket in the Lambda when calling external APIs; back off on 429/5xx.

In short: bigger batches, a short batch window, more CPU, capped concurrency, partial-batch acks, and correct visibility timeout. That combination drains bursts quickly without overload.

16. A fraud detection Lambda needs very low latency, but cold starts are slowing it. How would you solve this?

I would remove cold starts for the critical path and shrink initialization so each invoke is fast.

What I would change

- Turn on Provisioned Concurrency for the function (and schedule it higher during peak hours). This keeps a pool of warm execution environments ready, eliminating cold-start latency.
- Trim initialization. Keep dependencies small, lazy-load heavy modules, and move one-time setup to the initialization phase so request handling is minimal. Use Lambda layers sparingly and keep package size lean.
- Pick a fast runtime and architecture. Python or Node often start faster than heavy JVM stacks; if using Java, enable SnapStart to cut cold-start time. Run on Graviton/ARM for better price/perf and often faster init.
- Avoid or optimize VPC cold-start overhead. If you don't need VPC, run outside. If you must, ensure VPC endpoints are set up and keep ENIs warm with provisioned concurrency.
- Keep connections warm. Reuse HTTP clients and DB connections across invocations (declare them outside the handler). This removes connect latency on the hot path.
- Right-size memory. More memory increases CPU, which lowers per-request time. Measure and pick the cheapest setting that meets latency SLOs.
- If in-function ML inference is heavy, consider an always-on endpoint (Amazon SageMaker real-time or an ECS/Fargate microservice) and call it from Lambda; keep Lambda's work minimal (validation/routing).
- Add autoscaling guardrails. Use reserved concurrency to guarantee capacity for this function and prevent noisy neighbors from starving it. Monitor p50/p95/p99 latencies and cold start counts.

In simple words: use provisioned concurrency to remove cold starts, make the function tiny and fast to init, reuse connections, pick a quick runtime, and keep enough reserved capacity to hit your fraud-detection latency target.

17. Your Lambda → Kinesis → S3 pipeline outputs inconsistent data. How would you trace/debug with logs and X-Ray?

I make the pipeline traceable end-to-end with one correlation id that flows from Lambda to Kinesis to S3, and I turn on structured logs and X-Ray so I can see where data diverges.

Step by step

1. Add a correlation id and carry it through
   I generate or reuse an event_id (for example, a UUID or upstream id). I put it in every Kinesis record (as a field) and in log lines. If a consumer writes to S3, it includes event_id in the object name/metadata or a manifest row. This gives me a single key to search across components.

2. Turn on structured JSON logging everywhere
   In the producer Lambda, consumer Lambda (or Firehose transformer), and any loaders, I log JSON with fields: event_id, kinesis_stream, shard_id, sequence_number, partition_key, s3_bucket, s3_key, version_id, rows_out, and a phase label like "produced", "consumed", "written". With this I can run CloudWatch Logs Insights queries to follow a record.

3. Enable X-Ray active tracing on Lambdas and AWS SDK calls
   Producer Lambda: creates an X-Ray trace segment and logs trace_id alongside event_id.
   Consumer Lambda (from Kinesis): also enables X-Ray; when it handles a record, it logs both trace_id and event_id. I can't trace inside Kinesis itself, but I can correlate via event_id and sequence_number.

4. Add annotations/metadata to traces
   In Lambda code I add X-Ray annotations: event_id, table_name, shard_id. Annotations are indexed, so I can filter traces quickly for a bad event.

5. Capture Kinesis details to find hot shards or reorders
   I log partition_key and sequence_number, and I watch CloudWatch metrics (IncomingBytes/Records, GetRecords.IteratorAgeMilliseconds, ReadProvisionedThroughputExceeded). If IteratorAge spikes on a shard that also shows data issues, I know where to look.

6. Trace S3 writes and verify atomicity
   The consumer logs the exact s3://bucket/key (and versionId) it wrote for that event_id and adds a _SUCCESS marker when finishing a batch. If I find inconsistent Athena results, I query logs for the event_ids in that partition and check which ones lack a matching "written" log or _SUCCESS. I also check that we write to a temp prefix first and only "promote" after success (avoids partial reads).

7. Common root causes I confirm with traces/logs
   • Duplicate consumers or retries writing twice → fix with idempotency keys (event_id) and upserts/overwrites by key.
   • Out-of-order processing across shards → enforce ordering per key or route through SQS FIFO with MessageGroupId.
   • Partial batches written to S3 → ensure writers are transactional (temp path + _SUCCESS).
   • Bad partitioning causing hot shards → redesign partition keys with salted/hash keys and reshard.

In short: one correlation id everywhere, JSON logs, X-Ray on all Lambdas with annotations, and atomic S3 writes with success markers. Then I can search by event_id and see exactly where the pipeline diverged.

18. A Lambda ETL job fails with only "timeout" errors. How would you set up detailed monitoring and alarms?

I add visibility for duration, memory, external calls, and per-step timings, and then alert before timeouts happen.

What I put in place

1. Right metrics and alarms
   • CloudWatch alarms on Duration approaching Timeout (for example, p95 > 80% of timeout), Errors > 0, and Throttles > 0.
   • If pulling from SQS/Kinesis, also alarm on AgeOfOldestMessage / IteratorAge.
   • Lambda max memory used alarm when MaxMemoryUsed > 85% of allocated (from the REPORT line or Lambda Insights). High memory → slow GC → timeouts.
2. Lambda Insights and X-Ray
   • Enable Lambda Insights to see CPU, memory, init time, and cold starts.
   • Turn on X-Ray active tracing; wrap external calls (S3, DynamoDB, Redshift, HTTP) so each shows up as a subsegment with timing. Timeouts often come from a single slow dependencyX-Ray makes that obvious.
3. Structured, per-phase timing logs
   • Emit JSON logs at start/end of each ETL phase: read_input_ms, transform_ms, write_output_ms, rows_in, rows_out, bytes_read, bytes_written, and event_id.
   • Use CloudWatch Embedded Metric Format (EMF) so these become custom metrics without a separate agent.
   • Always log a final "handler_complete" with total_ms in a finally blockif missing, I know the code exited early.
4. Idempotent retries and DLQ
   • Configure a DLQ or on-failure destination. Alarms on DLQ depth and age catch silent failures.
   • Use idempotency keys so retries after timeouts don't double-write.
5. Dependency timeouts and backoff
   • Set explicit, short client timeouts on AWS SDK/HTTP calls (connect/read timeouts) and exponential backoff with jitter. Many "Lambda timeouts" are actually stuck I/O waits.
   • If calling Redshift/third-party APIs, add circuit breakers and per-call metrics; alarm on high 4xx/5xx.
6. Dashboards for fast triage
   • One dashboard showing: Duration p50/p95, init duration, MaxMemoryUsed, Invocations, Errors, Throttles, and for sources, backlog metrics.
   • A Logs Insights saved query tying event_id to per-phase times, so I can filter slow runs immediately.

7. Early warnings
   • Add a “pre-timeout” watchdog: if elapsed time in the function exceeds a threshold (say 70% of timeout), log a WARN with phase and pending work. This helps pinpoint which phase consistently runs long.

8. After observability, fix or re-shape
   • If p95 is close to Timeout, either optimize the slow phase, raise memory (more CPU), increase Timeout, or split the ETL into smaller steps via Step Functions/Glue for long work.

In simple words: turn on Lambda Insights and X-Ray, add per-phase timing logs/metrics, alarm before you hit the timeout, and set strict timeouts on downstream calls. That makes timeouts diagnosable and preventable, not mysterious.

19. Millions of daily Lambda → S3 invocations are costly. How would you cut costs with batching or alternate services?

I would stop writing one file per event and switch to a design that batches many events into one S3 write. That reduces Lambda invocations, S3 PUT requests, and tiny-file overhead, which also makes Athena faster later.

What I would change, in simple steps:

1. Add a buffer and batch before S3
   Instead of every producer Lambda writing directly to S3, I put events onto Kinesis or SQS. A single "writer" component then drains in batches and writes bigger Parquet files. This slashes invocations and S3 PUTs.
2. Prefer Kinesis Data Firehose when possible
   Firehose can do the batching for me. It buffers by size/time, converts to Parquet, compresses, and writes to S3 at target sizes like 128–512 MB. I go from millions of small writes to hundreds of large ones with almost no code.
3. If I must use Lambda, micro-batch it
   Have a dedicated writer Lambda read up to N messages (for example 10–10,000) and write one file per batch. Use a short batch window (1–5 seconds) so latency stays low. Keep file sizes around 128–512 MB.
4. Partition and compact
   Write into Hive-style partitions (for example dt=YYYY-MM-DD/hr=HH). Run a periodic compaction job (Glue or EMR Serverless) to merge any stragglers so there are no tiny files.
5. Reduce invocations inside Lambda
   Increase batch size on the event source (SQS/Kinesis), use a small batching window, and raise memory (more CPU) so each invocation processes more data. Fewer, fuller invocations cost less than many tiny ones.
6. Consider alternate services for heavy or constant flow
   If transforms are simple, Firehose with record format conversion is ideal. For heavier transforms, use Glue streaming or EMR Serverless to handle high throughput with fewer moving parts than thousands of Lambdas.
7. Control concurrency and schedules
   Throttle the writer to a safe, fixed concurrency and run bursts on a cadence (for example every minute) instead of per-record.

Quick win summary: buffer → batch → Parquet to S3 via Firehose or a writer Lambda → partition properly → compact. This cuts Lambda cost, S3 request cost, and speeds up downstream queries.

20. A team wants Lambda for heavy ETL. How would you explain when to use Lambda vs Glue vs EMR?

I pick the tool based on job length, data size, shuffle needs, and how much control I need over Spark.

When I use Lambda

- Short, simple functions under 15 minutes with modest CPU/RAM.
- Stateless transforms, small fan-in/fan-out, API calls, light parsing.
- I can batch with SQS/Kinesis and write a few bigger files, not one per event.
- No big joins, no wide shuffles, no complex Python/native deps.
  Simple example: validate events, mask PII, batch to S3 via a writer.

When I use AWS Glue (Jobs/Studio/Streaming)

- I want managed Spark ETL with minimal infra work: crawlers, bookmarks, job scheduling, retries, and connectors built-in.
- Medium to large batch or streaming transforms, incremental loads from S3 to Redshift, CDC merges.
- Need Spark but don't want to manage clusters; autoscaling DPUs is fine.
  Simple example: daily S3 → transform to Parquet → COPY to Redshift with bookmarks.

When I use EMR or EMR Serverless

- Large-scale Spark/Hive jobs with heavy shuffles, joins, or ML that need more tuning or specialized instances (memory/NVMe).
- Need full Spark control, libraries, and sometimes HDFS or very fast local disks.
- EMR Serverless if I want "no cluster ops" but still EMR's Spark runtime; EMR on EC2 if I need deep OS tweaks or multi-step pipelines with custom services.
  Simple example: multi-terabyte joins, feature engineering for ML, or big compaction jobs.

Simple rule I give teams:

- If it's small, event-driven, and finishes fast → Lambda.
- If it's Spark ETL and you want managed simplicity → Glue.
- If it's big/complex Spark with heavy shuffle or special hardware needs → EMR (or EMR Serverless to avoid cluster management).

This keeps costs down and matches each workload to the right engine without fighting service limits.