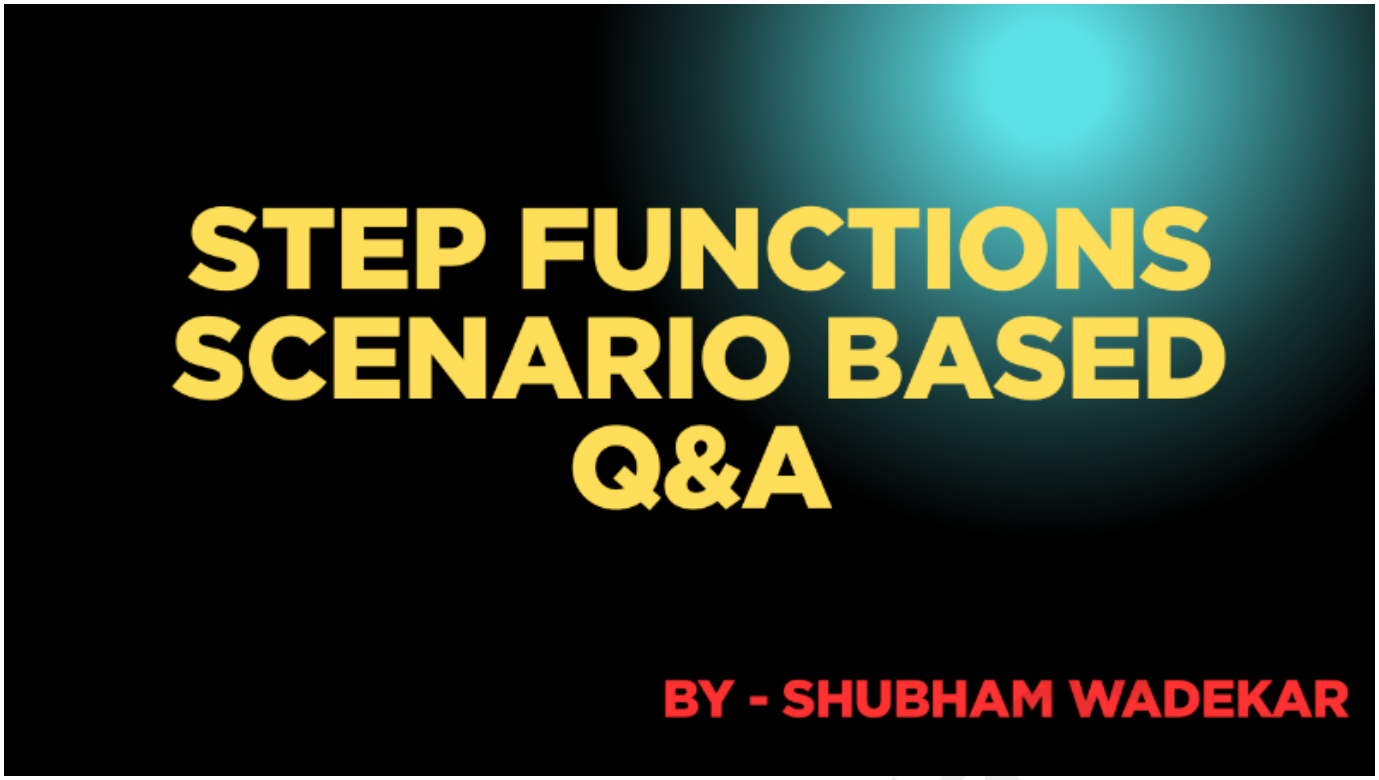

**1. Your pipeline ingests S3 files every 5 minutes for near real-time processing. Would you choose Standard or Express workflow, and why?**

I would choose Express workflow because each run is short and frequent. Express is designed for high-throughput, event-driven tasks that finish quickly. It starts fast, costs less at high volumes, and is perfect when a new file lands every few minutes.

How I would design it

- S3 event → SQS → Step Functions (Express) → small Lambda/Glue job that finishes in a few minutes.
- Make the processing idempotent using an id like bucket + key + version (or ETag). If the same message arrives twice, the job recognizes it and skips duplicate work.
- Use a "temp → validate → publish" pattern: write outputs to a temporary path, validate counts, then move to the final S3 prefix and drop a _SUCCESS marker. This keeps downstream reads consistent.
- Add a daily reconciliation: list yesterday's S3 objects (or use S3 Inventory), compare with a processed-manifest table, and requeue any missing keys.

When I would not use Express here

- If an occasional file can take longer than a few minutes to process, I route those rare cases to a Standard workflow or a dedicated long-running job. Everything else stays on Express for speed and cost.

**2. A workflow orchestrates multi-hour Glue/EMR jobs. What risks arise if you use Express workflow, and how would you redesign it?**

Express is not a good fit for multi-hour jobs. The main risks are time limits and weaker long-run observability for complex, lengthy workflows. For jobs that run for hours, I need durable execution history, reliable retries, and easy troubleshooting.

What could go wrong with Express

- Long jobs may exceed the short-run nature of Express.
- You get less detailed, long-term execution history, which makes audits and debugging harder for nightly jobs.
- High-frequency retries and at-least-once behavior can accidentally trigger duplicate long jobs if you don't build strict idempotency.

How I would redesign it

- Move orchestration to a Standard workflow. Standard supports long durations, detailed state history, robust retries/backoff, human approval steps, and better auditability.
- Pattern: Start Glue/EMR → Wait/Check status → Handle success/failure → Notify. If the job fails, the workflow retries with exponential backoff and then sends an alert.
- Keep small "fast checks" as Express if needed. For example, a quick schema/quality check can run in a nested Express workflow that feeds into the Standard workflow for the heavy lifting.

- Store run metadata (input manifest, job IDs, row counts, checksums) so you can resume or replay safely.

**3. You have a nightly batch ETL pipeline and a real-time clickstream pipeline. Which workflow type would you use for each, and why?**

Nightly batch ETL → Standard workflow

- Runs for hours, may include multiple long steps (extract, transform, load, quality checks).
- Needs detailed execution history, strong error handling, retries, and alerts.
- Easier to pause/resume, add manual approval, or reprocess a specific partition.

Real-time clickstream → Express workflow

- Very high event rate, each run is short (enrich, validate, write).
- Low-latency starts and lower cost at scale.
- Easy to fan out per event or per small micro-batch (for example, every 1–5 minutes).

A simple end-to-end design

- Clickstream (real-time): Events land in S3/Kinesis → Express workflow validates/enriches → writes to a processed S3 path in Parquet, partitioned by minute/hour. A small compaction job keeps file sizes healthy.
- Nightly batch: Standard workflow takes the day's processed data → runs heavy aggregations/dimensional modeling → publishes curated tables → runs data quality checks → notifies stakeholders and tools.
- Both pipelines are idempotent and publish with a "temp → validate → publish + _SUCCESS" pattern so downstream queries are always consistent.

**4. A streaming pipeline using Standard workflows has high Step Functions costs. What changes would you recommend?**

I would redesign it so frequent, short, event-driven work uses Express (or SQS + Lambda) and I reduce the total number of state transitions. Standard is great for long, complex runs, but it gets expensive when you start thousands of executions with many states per event.

What I'd change first

- Move hot, short paths to Express: If each event does quick validation/enrichment and finishes in a few seconds, run that part as an Express workflow. Keep Standard only for the occasional long backfill or daily aggregation.
- Micro-batch events: Buffer events in SQS or Kinesis and trigger one execution per small batch (for example, 1–5 minutes or N records). That cuts executions and state transitions dramatically.
- Collapse tiny steps: Combine multiple small Lambda steps into a single Lambda where it makes sense (validate + enrich + route). Fewer states = fewer transitions = lower cost.
- Use service integrations directly: Call SQS, SNS, DynamoDB, or Glue directly from one Task state instead of multiple Lambdas doing one-liners.
- Remove unnecessary Choice/Pass states: Push simple routing into application code or a single Choice.
- Throttle concurrency: Cap Express/consumer concurrency so you don't create excessive parallel branches.
- Consider EventBridge Pipes for simple "source → filter → enrich → target" flows: It can replace many tiny Step Functions states for light transformations.
- Separate long work from short work: If something can take minutes (or hours), don't hold a fast pipeline open. Publish a message to start that long job (Glue/EMR), then track it asynchronously (standard workflow) or via a callback pattern.

Safety and operations

- Make each execution idempotent (use object key + version or event ID) so switching to micro-batches and Express doesn't create duplicates.
- Keep the "temp → validate → publish + _SUCCESS" pattern so downstream reads stay consistent.

In simple words: move the per-event orchestration to Express or micro-batches, merge tiny steps, call services directly, and keep Standard only for the slow, complex parts. That lowers state transitions and your Step Functions bill.

**5. You need sequential Glue jobs, but one often fails on S3 read errors. How would you design retries for only the failed job?**

I would orchestrate each Glue job as its own Step Functions Task state and put a targeted Retry on the flaky step only. The other jobs won't rerun if they already succeeded.

Simple sequence

- Job A → Job B → Job C (each a separate Task state).
- Add Retry on Job B with exponential backoff and a small MaxAttempts.
- Catch on Job B routes to a notify/stop path only if it still fails after retries.
- Job A and Job C have no special retry (or a minimal one), so they don't run again when B retries.

What I include in the retry for Job B

- Retry only transient errors: S3 read timeouts, throttling, 5xx, networking issues.
- Backoff plan: for example, 30s → 60s → 120s (jitter helps).
- Preflight check: a tiny Lambda before Job B that HEADs the expected S3 prefixes and waits if files aren't fully published (avoids reading half-written data).
- Idempotent outputs: Job B writes to a run-specific temp path and only promotes on success. If a retry happens, it overwrites the temp path safely and publishes once.

Extra guardrails

- Use Glue's own retry (maxRetries) as a second layer, but keep the main control in Step Functions so only B repeats.
- S3 consistency pattern: upstream jobs write to temp → validate → final + _SUCCESS marker. Job B reads only partitions with _SUCCESS, reducing read errors.
- After Job B completes, Step Functions moves to Job C. If B ultimately fails, send an alert with the exact partition and error, and stop the chain without re-running A.

In simple words: break the chain into A, B, C; add a targeted Retry only on B for transient S3 issues, use temp/manifest markers so B sees complete data, and keep outputs idempotent so retries are safe. Only B retries; A and C don't.

**6. Your S3 data lake has partitions that need parallel Glue ETL processing. How would you design Step Functions for parallel runs while ensuring all jobs complete before the next stage?**

I would use a Step Functions Map state to fan out one Glue job per partition, limit how many run at once, wait for each to finish, and then move to the next stage only after the Map has completed. The Map state acts like a "parallel for-loop" with a built-in barrier.

How I set it up

- Prepare a partition list. A small Lambda builds a list like [{dt:"2025-08-24", hr:"00"}, ...] by listing S3 or reading a manifest. This becomes the Map input.
- Map state for parallel runs. The Map iterates over that list and starts a Glue job for each partition. I pass the partition values as job arguments.
- Wait for completion inside the Map. I use the Glue "sync" pattern (the task waits for the job to finish). If the job succeeds, that item in the Map is done; if it fails, the item fails.
- Limit concurrency. I set MaxConcurrency to a safe number (for example, 10 or 20) so I don't overwhelm the Glue capacity or hit API limits.
- Retries and idempotency. The Map's task has a Retry block for transient errors (S3 throttling, network). Each Glue run writes to a run-specific temp path and promotes only on success, so a retry won't duplicate data.
- Failure handling. If any partition keeps failing, the Map can Catch and send that partition to a DLQ path (for example, store the partition key in DynamoDB/SNS) and continue, or it can fail the whole workflow depending on business needs.

Barrier to next stage

- After the Map finishes, I know all parallel jobs are done (or skipped to DLQ per design). Only then do I run the "next stage" task, such as compaction, quality checks, or a downstream aggregation.
- I drop a _SUCCESS marker per partition during the Glue job and a table-level _SUCCESS after the Map, so downstream steps can trust completeness.

In simple words: build a list of partitions, run a Map state that starts one Glue job per partition with limited concurrency, wait for each to finish, and only then continue. The Map itself gives me the "wait until all are done" behavior.

**7. A Glue Crawler must finish before ETL jobs start. How would you design the workflow to enforce this dependency?**

I would run the crawler first, wait until it is finished and in a ready state, and only then start the ETL jobs. I do this in Step Functions with a simple "start → wait → check → loop" pattern.

Simple workflow

- StartCrawler task. Call Glue to start the crawler for the target prefix or table.
- Wait state. Sleep for 30–60 seconds to avoid hammering the API.
- GetCrawler status. Call Glue to read the crawler's state.
- Choice state.
    - If state is "READY" (and last crawl status is "SUCCEEDED"), proceed to ETL.
    - If state is "RUNNING", go back to Wait and check again.
    - If state is "FAILED", go to a failure path: notify, log the error, and stop.
- ETL stage. Once the crawler is ready, start the Glue job(s). If there are many partitions, I use a Map state as described above.

Reliability and safety

- Timeout guard. Add a maximum total wait time (for example, 1–2 hours). If exceeded, alert and stop to avoid stuck workflows.
- Idempotency. The ETL reads only partitions with a _SUCCESS marker or a manifest and writes to temp → validate → publish so reruns are safe.
- Optional backoff. Increase the wait interval on each loop (30s, 60s, 120s) to reduce API calls during long crawls.
- Optional EventBridge. If I want to avoid polling, I can have the crawler emit an event and let Step Functions wait on that event, but the polling loop is simple and reliable.

In simple words: start the crawler, keep checking its status in a small wait–check loop until it's ready, and only then kick off the ETL. If the crawler fails or takes too long, stop and alert.

**8. A daily pipeline has both short and long Glue jobs. How would you balance retries, error handling, and parallel execution in Step Functions?**

I split the flow into small, fast steps and long, heavy steps, then give each the right retry strategy. I run independent work in parallel, but I keep a barrier before publishing final outputs so partial success never leaks to downstream.

How I structure the state machine

- Ingest/validate (short) → Transform/aggregate (long) → Publish/quality checks (short).
- I use a Parallel or Map state to fan out independent partitions or domains, then a single "barrier" state where I check that all branches completed.

Retries and error handling

- Short steps (Lambdas, quick Glue jobs under ~10–15 min): small, fast retries with exponential backoff (for example, 10s → 30s → 60s). These usually fail for transient reasons (throttling, metadata race, S3 HEAD timeouts).
- Long Glue jobs (hour+): fewer retries with longer backoff (for example, 2–5 attempts with 5–15 minutes between). I only retry on transient errors (network, S3 5xx, capacity), not on data-quality errors.
- I put the Retry block only on the failing Task state. Upstream and downstream states don't re-run if they already succeeded.
- Catch blocks send details to an alert path (SNS/Slack) with the partition, job name, and error. I also write a failure record into a DynamoDB "runs" table.

Parallel execution safely

- I use Map to process a list of partitions (for example, all dt from yesterday). MaxConcurrency limits how many Glue jobs run at once so I don't overload DPUs.
- Each branch writes to a run-specific temp path and, on success, promotes to the final partition and drops a _SUCCESS marker.
- If some branches fail after all retries, I record those partitions and decide: either fail the whole workflow (strict mode) or continue but list the missing partitions for a follow-up rerun (tolerant mode).

Data quality and idempotency

- Before publish, I run a quick DQ step (row counts, null checks, duplicate keys). If DQ fails, I stop publish and alert.
- I keep an idempotency key per partition (bucket + key prefix + run_date). If a step is retried or re-executed, it overwrites the temp path and publishes once, so duplicates do not occur.

Operational hygiene

- I set Glue job maxRetries as a secondary safety net, but Step Functions controls the main retry logic so only the failing step repeats.
- I add timeouts per state so hangs don't block the pipeline.
- I write execution and DQ stats to a small summary file so downstream tools and dashboards know which partitions are complete.

In simple words: short steps retry quickly, long steps retry cautiously; independent partitions run in parallel with a limit; every branch publishes atomically; and a final barrier ensures all required parts are done before moving on.

**9. How would you orchestrate multi-step Spark jobs on EMR with Step Functions while respecting dependencies?**

I model each Spark step as its own Task in Step Functions, wire them in the exact order they must run, and only move ahead when the previous step has finished successfully. I also make the cluster lifecycle explicit so I do not pay when idle.

Cluster strategy

- For one big daily pipeline: create an EMR cluster at the start, run all Spark steps, then terminate it at the end.
- For teams that share compute or need autoscaling: use a persistent cluster with managed scaling or EMR Serverless, and submit steps that wait for completion.

Wiring the steps with dependencies

- State 1: “Create/Start Cluster” (or skip if using EMR Serverless).
- State 2: “Step A – Ingest” (AddStep with wait-for-completion).
- State 3: “Step B – Transform” (runs only if A succeeded).
- State 4: “Step C – Aggregate/Publish”.
- Optional Map: if I process multiple partitions with the same Spark job, I use a Map state to submit steps per partition (with MaxConcurrency).
- Final state: “Terminate Cluster” (always in a Finally/Catch path so the cluster is shut down even on failure).

Retries and failure handling

- Add targeted Retry to transient errors for each AddStep task (for example, YARN capacity, S3 5xx, throttling).
- For deterministic data errors (schema mismatch, bad records), I Catch and stop the workflow with a clear message and pointers to the Spark logs.
- I upload Spark event logs and driver/executor logs to S3/CloudWatch so the failure is easy to inspect.

Data consistency between steps

- Each Spark step writes to a temp location, validates, then promotes to the final S3 path and creates a _SUCCESS marker.
- The next step only reads partitions where _SUCCESS exists. This prevents a downstream step from reading half-written data.

Passing data between steps

- I pass S3 paths, partition lists, and config flags in the Step Functions input/output.
- If a step generates a dynamic list (for example, which partitions were built), I store it in the state context and use it to drive the next Map.

Cost and performance tips

- Reuse the same cluster across sequential steps to avoid spin-up overhead.
- Enable EMR managed scaling so executors grow/shrink with load.
- Use consistent Parquet + partitioning so later steps and Athena queries are fast.
- Compact small files at the end to keep the lake healthy.

In simple words: I make each Spark step a separate state, run them in order with AddStep and "wait for completion," retry only transient failures, use _SUCCESS markers between steps, and shut the cluster down at the end. This respects dependencies and keeps cost under control.

**10. A team keeps an EMR cluster always running. How would you redesign with Step Functions to create/run/terminate clusters on demand?**

I would make the cluster ephemeral: create it only when a workflow starts, run the Spark steps, and terminate it automatically at the end. Step Functions becomes the orchestrator that guarantees the cluster lifecycle and keeps costs down.

How I design it

- State 1: validate inputs and build a run configuration (S3 paths, partitions, Spark args).
- State 2: create cluster task that calls EMR to create a right-sized cluster (instance fleets or EMR Serverless). I attach tags for cost tracking and set log URI to S3 so all logs are saved.
- State 3..N: submit Spark steps (AddStep) and wait for each to finish. Each step writes to a temp S3 path, validates, then promotes to the final path with a _SUCCESS marker.
- Final state: terminate cluster always. I put this in a finally path so the cluster shuts down even if a step fails.
- Scaling and price: enable managed scaling so executors grow and shrink with load. Use a mix of On-Demand and Spot with safe allocation strategies. Choose the smallest cluster that meets SLAs.
- Security and config: bootstrap actions or configurations for consistent JDK, Spark settings, and connector jars; KMS encryption; S3 VPC endpoints; fine-grained IAM roles for EMR and steps.

Why this helps

- You stop paying when there is no work.
- Every run is clean and reproducible because you create a fresh cluster with known config.
- Logs and outputs are tied to the run for easy auditing and rollback.

In simple words: Step Functions creates the EMR cluster at the start, runs your Spark jobs, and then always terminates the cluster. You only pay while work is running, and every run is clean and consistent.

**11. An EMR cluster sometimes fails to start due to misconfigurations. How would you add retries and cleanup logic in Step Functions?**

I would add a guarded create step with targeted retries, detect whether the failure is transient or a bad config, and always clean up any partial clusters.

Pattern I use

- Preflight check: a small Lambda validates configs before creation (subnet, security groups, release label, bootstrap URIs, KMS, instance types and quotas). If this fails, stop early with a clear message.
- Create cluster task with Retry: retry only transient errors (capacity, throttling, Spot interruptions) using exponential backoff and jitter. Limit attempts so it does not loop forever.
- Wait-and-poll: after creation, poll cluster status until it is WAITING or RUNNING, with a total timeout in case it hangs.
- Catch blocks:
    - Config error path: if the error indicates an invalid configuration, fail fast, notify with the exact field to fix.
    - Transient failure path: after retries are exhausted, notify and stop.
- Always-terminate step: a cleanup state that terminates the cluster if it exists. This runs in a finally path so orphaned clusters do not linger.
- Safeguards:
    - Auto-termination timer on the cluster as a last resort.
    - Tags like app=xyz, run_id=uuid so a periodic janitor job can find and terminate any orphaned clusters.
    - Store create parameters and error messages in S3/DynamoDB so the next attempt is easy to debug.

In simple words: validate before creating, retry only transient problems, fail fast on bad configs, and always run a cleanup step that terminates any half-created cluster.

**12. You need to orchestrate multiple Spark jobs on EMR where some run in parallel and others sequentially. How would you design this?**

I build a small DAG with Step Functions: independent jobs run in parallel, dependent jobs run in sequence, and I add a barrier so the next stage starts only after all required jobs complete.

Design example

- Create cluster.
- Parallel state with two branches:
    - Branch A (sequential): Step A1 → Step A2 → Step A3. Each AddStep waits for completion and only proceeds on success.
    - Branch B (sequential): Step B1 → Step B2.
- Barrier: when both branches finish, continue.
- Step C (final aggregation) runs after the barrier, then compaction and data quality checks.
- Terminate cluster always at the end.

Operational details

- MaxConcurrency inside any Map that fans out partitions so the cluster is not overloaded.
- Targeted retries per step: transient EMR or S3 errors retry; deterministic data errors do not.
- Data handoff: each step writes to a temp S3 path, validates counts, promotes to final, and writes a _SUCCESS marker. A downstream step reads only partitions with _SUCCESS to avoid half-written data.
- Passing outputs: the output S3 locations or generated partition lists from earlier steps are placed in the state context and consumed by later steps.
- Logs and metrics: Spark event logs to S3, CloudWatch for driver logs, and a small summary object with row counts and durations per step.

In simple words: I use Step Functions to run independent Spark jobs in parallel and dependent ones in order, wait for all to finish, then run the final aggregation and shut down the cluster. Each step is retried safely on transient issues and publishes its results atomically so downstream steps always read complete data.

**13. How would you build a workflow to run Athena transformations on S3 data before making it available for reporting?**

I would keep the flow very clear: land raw data, transform it with Athena into clean Parquet, validate it, and publish it to a curated area that BI tools read. I make publishing atomic so reports never see half-written data.

What I create in S3 and Glue

- Buckets and folders
    - s3://lake/raw/<source>/dt=YYYY-MM-DD/ (raw files exactly as received, immutable)
    - s3://lake/processed_staging/<table>/dt=YYYY-MM-DD/ (temporary output from Athena CTAS/INSERT)
    - s3://lake/curated/<table>/dt=YYYY-MM-DD/ (final, published data for reporting)
- Glue Data Catalog tables
    - raw_<source> pointing to raw (CSV/JSON with the right SerDe)
    - processed_<table> pointing to processed_staging (Parquet, Snappy)
    - curated_<table> pointing to curated (Parquet, Snappy)
- A results bucket for Athena query outputs with encryption turned on

How the daily job runs end to end

1. validate inputs
    - check that yesterday's raw partition exists and is not tiny (size and file count checks)
    - if expected files are missing, stop and alert
2. transform with Athena (CTAS or INSERT INTO)
    - use a CTAS query that reads raw, parses fields, fixes types, drops duplicates, applies simple business rules, and writes Parquet to processed_staging/dt=<date>
    - only select the columns needed for analytics to keep files small
    - choose partitioning (usually dt, sometimes dt and hr for high volume)
3. optional enrichment and joins
    - join with small dimensions (country, product, user attributes) that are already in Parquet
    - keep joins selective (filter fact table by date first, then join)
4. data quality checks before publish
    - row count compared to raw (within a threshold)
    - key columns not null (for example, event_time, id)
    - duplicate checks on natural keys or event_id
    - basic distribution checks (for example, no future timestamps)

5. publish atomically
    - move or copy the processed_staging dt folder into curated
    - write a tiny _SUCCESS marker or a manifest JSON file describing the exact objects published
    - if something fails, do not touch the curated area
6. refresh metadata for consumers
    - if using Glue static partitions: ALTER TABLE ADD PARTITION for the new dt
    - if using partition projection: nothing to update; still keep the _SUCCESS marker so consumers know the partition is ready
    - update a stable view if you version curated paths (for example, swap v1 to v2)
7. notify and log
    - send a message with run_id, row counts, durations, and the published dt
    - store a small audit record (run status, counts, S3 paths) in DynamoDB or a log folder

How I operate it safely and cheaply

- file format and size: always Parquet + Snappy with 128–512 MB files per partition to keep Athena scans fast and cheap
- idempotency: write to processed_staging/<run_id>/ first, then promote; if the job retries, it overwrites the staging location and publishes once
- security: encrypt all buckets with KMS, require TLS, and block public access; limit read access to curated for analysts
- scheduling: EventBridge daily at a fixed time or Step Functions orchestrating the whole flow with clear retries and alerts
- error handling: if CTAS fails, capture the SQL and error, leave curated unchanged, and send an alert with links to the failed query results
- cleanup: lifecycle rules to delete old query results, expire old staging outputs after a few days, and transition raw to cheaper storage after 30–90 days if allowed

In simple terms: I turn raw files into clean Parquet with an Athena CTAS, check the data, then publish to a curated folder with a success marker. Reports read only curated, so they always see complete, clean data.

**14. Athena queries sometimes start before all S3 files are written. How would you ensure data consistency before queries run?**

I make writers use a two-step publish pattern and make readers check a “ready” signal before they query. This guarantees that queries never read partial data.

Writer pattern (two-phase publish)

- step 1: write to a temporary run path, for example s3://lake/tmp/run_id=UUID/table=xyz/dt=YYYY-MM-DD/
- step 2: validate the temp output (row counts, schema, duplicates)
- step 3: atomically publish by moving or copying the finished files into s3://lake/curated/xyz/dt=YYYY-MM-DD/
- step 4: create a tiny _SUCCESS file or a manifest.json that lists the final object keys and sizes

Reader pattern (only query published partitions)

- BI and Athena read only from curated
- the SQL or the view includes a check that the partition is published, for example: join to a tiny “manifests” table that lists dt values with status = ‘READY’, or simply rely on Glue to add the partition only after publish
- if using partition projection (no Glue adds), keep a small table manifests(dt, status, published_at) and filter queries with “where dt in (select dt from manifests where status = 'READY')”

Orchestration to avoid races

- Step Functions order: write → validate → publish → add partition (or update manifests) → trigger reports
- if you use a crawler, run it only after publish and wait until it finishes before starting queries
- use S3 events on the _SUCCESS object to trigger downstream tasks, not on the first data file

Extra safety tips

- compaction before publish so each partition has a handful of large Parquet files; this avoids readers hitting partially written small files
- retries and idempotency: if publish fails halfway, re-run safely because temp paths are isolated
- time-based holds: if data arrives late, the workflow waits for a completeness signal (for example, an upstream manifest that lists all expected files) before starting the CTAS

Monitoring and quick checks

- a tiny “readiness check” query that counts rows in curated for the new dt and compares with expected counts before scheduling dashboards
- CloudWatch alarms if any partition lacks _SUCCESS after a cutoff time
- daily reconciliation using S3 Inventory to spot files that were written but never published

In simple terms: writers don't publish until everything is complete; they drop a success/manifest file when ready. Readers and schedulers look for that signal (or a registered partition) before running queries. This prevents Athena from reading half-written data.

**15. A reporting pipeline must run multiple Athena queries in sequence and pass outputs downstream. How would you design this?**

I would orchestrate the queries with Step Functions so each query runs in order, writes its result to S3 as Parquet, and then passes the produced S3 path to the next step. I keep the flow "temp → validate → publish" so downstream reports only see complete data.

How I lay it out

- Athena workgroup with result encryption and its own results bucket.
- Three S3 areas per table: raw, processed_staging, curated.
- SQL stored as templates in S3 (parameterized by date), so I can update logic without redeploying code.

State machine flow (simple and reliable)

- Prepare step: compute report_date (usually yesterday) and build parameters (S3 paths, workgroup, output prefixes).
- Query 1 task (CTAS or INSERT INTO): produce a cleaned dataset to processed_staging/table1/dt=report_date as Parquet. The task waits until Athena says the query succeeded.
- Validate 1: quick count checks and basic quality rules. If bad, stop and alert.
- Publish 1: move processed_staging/table1/dt=... to curated/table1/dt=... and write a _SUCCESS marker. Pass the curated path to the next step.
- Query 2 task: use the curated path from step 1 as input (for example, join table1 with another table). Write to processed_staging/table2/dt=... .
- Validate 2 → Publish 2 → pass output path forward.
- Continue for all queries.
- Final step: refresh partitions (or rely on partition projection), update a stable view, and notify consumers.

Passing outputs between steps

- Each Athena task returns the exact S3 prefix of its output partition.
- The next step reads those prefixes from the Step Functions state input.
- I also store a small manifest file (JSON with table, dt, files, row count) for audit.

Retries and errors

- Retry only transient errors (throttling, network). If SQL is bad or data violates rules, fail fast and do not publish.
- If a step fails, previous published outputs remain intact because they were already validated and marked complete.

Cost and speed

- All CTAS outputs are Parquet with Snappy, files sized around 128–512 MB.
- Queries filter by partition columns to keep scanned bytes low.

In simple words: run Athena queries one after another, each writes Parquet to staging, validate, publish to curated, and pass the published S3 path to the next query. If anything fails, stop before publish so reports never see partial data.

**16. Schema changes in S3 break queries. How would you design Step Functions to run Glue Crawlers, update schema, and then trigger Athena queries safely?**

I would put the crawler at the start of the workflow, wait until it finishes, handle schema changes safely, and only then run the Athena transformations. I protect downstream users with views and default values so additive changes don't break queries.

Workflow design

- Start crawler task: run the Glue Crawler on the raw or processed location.
- Wait and check loop: poll crawler status until it's ready and succeeded. If failed, stop and alert.
- Schema compatibility step:
    - If new columns appeared, add them as nullable in the processed schema (CTAS or ALTER TABLE ADD COLUMNS).
    - If a type changed, normalize during CTAS using safe casts (try_cast) and write the corrected types into Parquet.
    - Keep a stable curated view that lists only the fields consumers rely on; extend the view when ready so old queries still work.
- Transform with Athena: run CTAS to produce Parquet into processed_staging with the updated schema.
- Data quality checks: verify key columns, row counts, and that type casts did not drop important values.
- Publish: move to curated, write _SUCCESS, and refresh partitions or rely on partition projection.
- Notify: send a message that schema version X was published for date Y, with a link to the data dictionary.

Guards that prevent breakage

- Never use SELECT * in production SQL; list columns explicitly so new fields don't change column order unexpectedly.
- Maintain a small schema registry or dictionary (column, type, nullable, pii flag) in Git; the workflow reads it and enforces it during CTAS.
- Keep rollback simple: curated is versioned (v1, v2). If the new schema causes trouble, point the reporting view back to the previous version.

In simple words: run the crawler first, detect and normalize schema changes, write clean Parquet with safe casts, and only then trigger Athena queries. Use views and nullable columns so new fields don't break existing queries.

**17. A multi-step Glue workflow has one job failing on transient S3 errors. How would you retry just that job?**

I would make each Glue job its own Step Functions state and put a targeted Retry on the failing job only. That way other successful jobs don't rerun, and we only repeat the flaky step with backoff.

Structure

- Job A → Job B → Job C, each as a separate Task state that starts a Glue job and waits for completion.
- Only Job B state has a Retry block for transient errors (S3 5xx, timeouts, throttling). Use exponential backoff and a small number of attempts (for example, 30s, 60s, 120s, up to 3–5 tries).
- If Job B still fails, Catch routes to an alert path with the partition/date, error code, and a link to logs.

Make retries safe

- Idempotent writes: Job B writes to a run-specific temp path and publishes to final only on success. A retry overwrites temp safely and publishes once.
- Read consistency: Job B reads only partitions with a _SUCCESS marker from upstream jobs, so it doesn't try to read half-written data.
- Optional preflight: a tiny Lambda before Job B that HEADs the expected S3 prefixes; if files aren't visible yet, wait and retry the preflight, reducing read errors.

Do not rerun other jobs

- Because A and C are separate states without a Retry that restarts the chain, they won't run again when B retries.
- If B ultimately fails, the state machine stops before C and sends an alert. On manual rerun, Step Functions can start from Job B using an input that points to the same partition.

Extra reliability

- Also set Glue's own maxRetries on Job B as a second layer, but keep orchestration retries in Step Functions to control which job is retried.
- Add timeouts per state so a stuck job doesn't hold the pipeline forever.
- Record a small run manifest (partition, attempts, final status) for auditing.

In simple words: break the workflow into separate Glue tasks, attach a retry policy only to the failing job with backoff and idempotent writes, and keep the other jobs untouched. Only the flaky step retries; the rest of the pipeline stays stable.

**18. A Spark job on EMR fails due to cluster capacity issues. How would you retry with a new cluster while terminating the failed one?**

I would make the cluster lifecycle explicit in Step Functions and separate "submit work" from "manage cluster." When capacity-related failures happen, I always clean up the bad cluster first, then create a fresh cluster with safer capacity settings and retry once or twice.

How I design the state machine

- CreateCluster step: creates EMR with instance fleets, multiple subnets/AZs, managed scaling on, and tags (app, run_id).
- SubmitStep step: adds the Spark step and waits for completion.
- Monitor step: polls until the step finishes, collecting logs to S3/CloudWatch.
- Choice + Catch:
    - If step succeeded → proceed.
    - If failed with capacity/Spot errors → go to a "RetryWithNewCluster" branch.
    - Otherwise → alert and stop.
- TerminateCluster (finally): always runs on exit paths to shut down any existing cluster.

RetryWithNewCluster branch

- Terminate the failed cluster first (guard against orphans).
- Backoff wait (for example, 2–5 minutes) to let capacity free up.
- Create a new cluster with safer settings:
    - Prefer instance fleets with capacity-optimized Spot + a baseline of On-Demand.
    - Add more acceptable instance types.
    - Allow multiple subnets and availability zones.
    - Increase core capacity or set a higher max for managed scaling.
- Re-submit the same Spark step (idempotent outputs):
    - The job writes to a run-specific temp path, validates, then promotes to final and drops a _SUCCESS marker.
    - If it already published, the rerun detects _SUCCESS and exits quickly.

Extra guardrails

- Preflight checks for quotas and subnet AZ availability; fail fast if configuration is invalid.
- Cluster-level auto-termination timer as last resort.
- A small "janitor" Lambda that terminates any cluster tagged with an old run_id, just in case.

In simple words: I always terminate the failing cluster, then create a new one with broader capacity options and retry the Spark step once or twice. All outputs are idempotent, and termination runs even on errors so I don't leak clusters.

**19. A Lambda validator sometimes times out on large inputs. How would you use Retry, Catch, or Fallback to handle this?**

I treat the validator as "fast path," but I give it a safe fallback for big inputs. I also split work so each Lambda run stays small.

My approach

- Short Lambda first: try the validator with higher memory (more CPU) and a sensible timeout (for example, 60–120 seconds). Pass only S3 locations, not big payloads.
- Chunk large inputs: if the input is big (many files/records), I use a Map state to process chunks (for example, 1k–10k records per chunk). This prevents one Lambda from doing too much.
- Retry only transient errors:
    - Add a Retry on the Lambda task for throttling or 5xx, with exponential backoff (for example, 5s, 15s, 45s).
    - Do not retry on States.Timeout repeatedly; timeouts usually mean the work is too big for Lambda.

Catch and fallback path

- Catch States.Timeout and States.TaskFailed and route to a fallback:
    - Option A: push the same validation job to a container/Batch/Fargate task with a longer timeout.
    - Option B: enqueue the S3 manifest to SQS and let a Glue job or EMR step do the heavy validation.
- Mark validation idempotent: the validator writes a small result object (valid/invalid, counts, errors) per chunk. Re-runs overwrite that result safely.

Operational tips

- Increase Lambda memory to speed CPU-bound parsing (memory increases CPU share).
- Use streaming/line-by-line parsing and early exits where possible.
- Surface partial results: if a chunk fails, the Map only retries that chunk; other chunks continue.

In simple words: try a fast Lambda with small chunks and quick retries; if it times out, catch that and hand the work to a longer-running engine like Batch/Fargate or Glue. Only the failing chunk reruns, and results are idempotent.

**20. In parallel Glue jobs, one fails while others succeed. How would you capture failure, notify, and still use successful outputs?**

I run the jobs in a Map or Parallel state with per-branch error handling. I collect a success/failed summary at the end, notify the team, and let downstream stages use only the partitions that finished successfully.

Design

- Build a list of partitions or domains to process (for example, all dt partitions for yesterday).
- Map state launches one Glue job per item with MaxConcurrency to control load. Each job writes to a temp path, validates, publishes to final, and creates a _SUCCESS file for that partition.
- Per-item Retry: transient S3/Glue errors retry with backoff. Deterministic data errors do not.
- Per-item Catch: on final failure, write an entry to a “failures” list (partition, error, logs URL).

End-of-map barrier

- After the Map completes, a summarizer step reads the “successes” and “failures” arrays:
    - If strict mode: fail the workflow and alert with the failed list.
    - If tolerant mode: proceed with only successful outputs by reading partitions that have _SUCCESS, and separately queue failed partitions for a rerun later.

Notifications and reruns

- Send a single alert summarizing totals: processed N, succeeded M, failed K, list of failed partitions and a link to logs.
- Persist a manifest (JSON) with the run_id, succeeded partitions, failed partitions, and timings.
- Optional automatic backfill: create a follow-up execution that targets only failed partitions.

Downstream safety

- Aggregation or reporting steps consume only the partitions with _SUCCESS markers or entries in the success manifest.
- No successful partition is reprocessed unless needed.

In simple words: run partitions in parallel with per-item retries and catches, record exactly which ones failed, notify once with a summary, and let downstream steps use only the successful partitions. Failed ones get queued for a targeted rerun without blocking the whole pipeline.

**21. You have date-partitioned S3 data, and each partition needs a Glue job. How would you use Parallel states to process and merge results?**

I would make Step Functions fan out the work per partition, run those Glue jobs in parallel with a safe concurrency limit, and then run one merge/aggregation step after all partitions finish. For a fixed, small number of branches (like two domains or two pipelines), I use a Parallel state. For a variable list of dates or partitions, I use a Map state (it's still "parallel" but scales with the list). The key is: fan out → wait for all to finish → merge.

How I set it up

- Build the partition list. A small Lambda lists S3 or reads a manifest like [{dt:"2025-08-24"},{dt:"2025-08-25"}, ...].
- Fan-out execution.
    - If I have a dynamic list (most common): Map state with MaxConcurrency (for example, 10–20) starts one Glue job per partition and waits for completion.
    - If I have a fixed set of flows (for example, two domains that each process all of yesterday's data): Parallel state with two branches; each branch runs its own sequence of Glue steps.
- Each Glue job writes to s3://lake/processed_staging/table/dt=YYYY-MM-DD/, validates, then promotes to s3://lake/curated/table/dt=YYYY-MM-DD/ and drops a _SUCCESS marker. This makes retries safe and prevents half-written data from leaking.
- Barrier to merge. When the Map/Parallel finishes, I know all partitions are done. I then run a "merge" Glue job that:
    - compacts small files into 128–512 MB Parquet files
    - builds daily rollups (for example, one partitioned summary table)
    - optionally writes a single dated manifest for downstream reads
- Failure handling and retries. Each parallel item has its own Retry for transient S3/Glue errors. On final failure, I capture the (dt, error, logs_url) into a failures list, alert once with the summary, and decide:
    - strict mode: fail the workflow so nothing proceeds
    - tolerant mode: continue the merge using only partitions that have _SUCCESS while scheduling a follow-up run for the failed dt
- Concurrency and cost. MaxConcurrency avoids overloading Glue DPUs and reduces API throttling. File compaction after the barrier reduces Athena scan cost.

In simple words: run one Glue job per date in parallel, wait for all of them to finish, then run a single merge/compaction step. Use success markers so merges only include completed partitions.

**22. A daily batch of S3 files must each be transformed before loading into Redshift. How would you design with Map state to process files dynamically?**

I would drive the pipeline from a daily manifest of files, use a Map state to process them in small batches, stage the transformed outputs in S3, and then run one Redshift COPY using a manifest for atomic, consistent load.

End-to-end flow

- Build the file manifest. A Lambda lists s3://landing/dt=YYYY-MM-DD/ and emits a list of file keys, or I use S3 Inventory generated nightly.
- Map state to transform.
    - Use a batch size (for example, 50–200 files per item) so each Glue/Lambda task processes a bundle, not a single file. This reduces Step Functions cost and API calls.
    - Each item transforms raw (CSV/JSON) to Parquet or cleanses CSVs to a consistent schema, writes to s3://stage/redshift/dt=YYYY-MM-DD/batch_id=..., and writes a tiny per-batch _SUCCESS file with row counts.
    - Retries target only transient errors; bad data is written to a rejects prefix with reasons.
- Barrier and validation.
    - After the Map completes, a validator checks that total staged row counts match expectations and that every batch wrote _SUCCESS.
    - Create a Redshift COPY manifest (JSON listing staged objects) in s3://stage/redshift/manifests/dt=.../manifest.json.
- Load to Redshift.
    - Run a single COPY with MANIFEST and proper file format/compression options.
    - Use IAM role auth and KMS keys as needed. Analyze/Vacuum or Auto-Analyze afterwards.
- Idempotency and reruns.
    - Stage paths include run_id, and the manifest for COPY includes only the run's staged files. If a rerun happens, I build a new manifest; the prior data can be truncated or loaded into a temp table and swapped.
- Cost/performance hygiene.
    - Transform to columnar (Parquet) or compressed CSV to reduce S3 → Redshift network and COPY time.
    - Keep files 100–512 MB so COPY uses parallelism well.
    - Limit Map MaxConcurrency to match Glue/Redshift capacity and avoid throttling.

In simple words: create a daily file list, use Map to transform files in manageable batches to a staging area, then run one atomic Redshift COPY using a manifest. Validate before COPY, and make everything idempotent.

**23. Some Glue jobs can run in parallel while others must wait. How would you balance parallelism with dependency management?**

I build a small dependency graph in Step Functions: independent jobs run together; dependent jobs wait behind a barrier. I keep MaxConcurrency to protect capacity and I only retry the steps that fail.

How I structure it

- Group jobs by dependency.
    - Stage 1 (independent): jobs A, B, C can run in parallel (Map or Parallel state).
    - Barrier: wait for A/B/C to finish.
    - Stage 2 (dependent): job D depends on A and B; job E depends on C. I run D and E in parallel now.
    - Barrier.
    - Stage 3 (final): job F aggregates outputs of D and E.
- Control concurrency.
    - Use MaxConcurrency on Map to avoid overloading Glue DPUs.
    - If some jobs are heavy, give them their own branch so they don't starve smaller tasks.
- Safe publishing and reads.
    - Each job writes to temp, validates, then publishes with a _SUCCESS marker.
    - Downstream jobs read only from partitions with _SUCCESS.
- Targeted retries and failure policy.
    - Put Retry only on transient errors for each job; deterministic data errors go to Catch → notify.
    - After Stage 1: if A fails but B and C succeed, I either halt (strict) or proceed with B/C-dependent paths and mark A-dependent paths as pending (tolerant). I persist a run manifest of successes/failures.
- Observability.
    - One summary notification per stage with which jobs succeeded/failed, run IDs, and links to logs.

In simple words: run independent jobs together, wait at a barrier, then run the next dependent set. Limit concurrency to match capacity, publish atomically, and retry only the jobs that fail.

**24. A Map state processes thousands of S3 objects, causing high costs and throttling. How would you redesign for cost efficiency and scaling?**

I would stop doing "one state per object." Instead, I would aggregate work into micro-batches, let Glue/Spark process whole prefixes, and add buffering so Step Functions runs far fewer states with higher work per state.

Changes I make

- Batch the items.
    - Replace per-file items with batches of N files (for example, 100–1,000 files per batch), built from a daily manifest. One Map item = one batch = one Glue/Lambda task.
    - For very large sets, use a two-level approach: outer Map per partition (dt, region), inner job (Glue/Spark) reads all files in that partition directly from S3 (no per-file orchestration at all).
- Use engines that handle many files natively.
    - Prefer a single Glue/Spark task to read many small files from a prefix and compact them into large Parquet outputs. Spark is built to parallelize within one job; you don't need a state per file.
- Control concurrency and backpressure.
    - Set Map MaxConcurrency to a safe number (for example, 10–25) based on Glue DPUs and S3/KMS limits.
    - If events are continuous, buffer with SQS and trigger one Map execution per time slice (for example, every 5–15 minutes), creating fewer, larger runs.
- Reduce Step Functions transitions.
    - Merge tiny validation steps into the same Lambda/Glue script instead of separate states.
    - Use service integrations (call Glue directly) rather than chaining many Lambdas that each do a one-liner.
- Idempotency and manifests.
    - Each batch writes to a unique run/batch path and drops a _SUCCESS with row counts; reruns overwrite batch outputs safely.
    - Keep a run manifest recording which batches succeeded/failed so you can retry only failed batches later.
- Cost hygiene.
    - Compact outputs to 128–512 MB Parquet so Athena scans fewer files later.
    - Use lifecycle rules to clean temporary batch outputs after publish.
    - Tune Spark (shuffle partitions, coalesce) so it produces a handful of large files per partition, not thousands of small ones.

In simple words: stop creating a state per object. Group objects into batches, let a single Glue/Spark job process many files or whole partitions, cap concurrency, and publish atomically with manifests. You'll cut Step Functions cost, avoid throttling, and finish faster