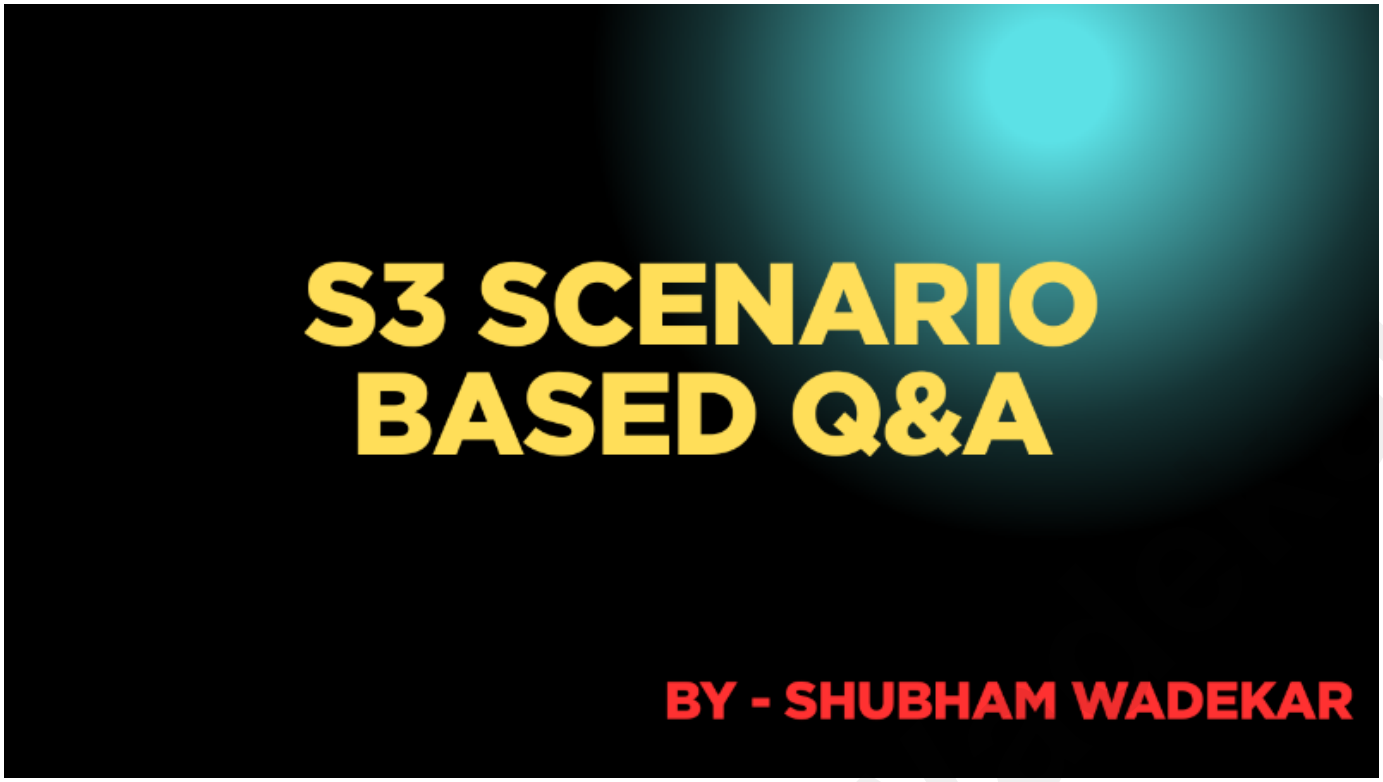

1. Your data lake stores raw data in CSV, and Athena queries are slow/expensive. How would you redesign storage format and structure to improve performance and reduce cost?

I would change two things: how the data is stored (columnar format, compression, right-sized files) and how it is laid out in S3 (clean partitions and atomic writes).

First, convert CSV to a columnar format like Parquet. Columnar files let Athena read only the columns used in a query, which cuts scanned bytes a lot. I would also compress with snappy or zstd. This typically reduces both cost and runtime dramatically compared to plain CSV.

Second, partition by the fields most commonly used in filters, usually a date column like dt=YYYY-MM-DD, and maybe one low-cardinality dimension such as region or source. I avoid over-partitioning; date, and optionally one more dimension, is usually enough.

Third, avoid thousands of tiny files. I aim for 128–512 MB per Parquet file. I batch writes (via Glue/EMR/Firehose) and schedule a compaction job that merges small files inside each partition. Fewer, larger files make planning and reading faster.

Fourth, make writes atomic. Each job writes to a run-scoped temp path, then promotes to the final partition and drops a _SUCCESS marker or manifest. Readers only scan partitions with the marker, so they never see half-written data.

Fifth, register clear table metadata in Glue Data Catalog with proper data types. For large date ranges, I enable partition projection so Athena computes partitions instead of listing millions of folders. I keep table and column names lowercase and consistent.

Sixth, create analyst-friendly presentation tables or views that expose only needed columns and hide raw noise. That reduces scanned data and avoids select *. If late data is common, I materialize a small, recent summary table to speed daily dashboards.

In simple terms: convert CSV to Parquet with compression, use sensible partitions, keep files big, write atomically, and register clean schemas. This combination cuts scan cost and speeds up Athena immediately.

2. A new JSON log source needs onboarding into your S3 data lake. How would you organize raw, processed, and curated layers for efficient downstream use?

I would land JSON safely in a raw area, standardize it in a processed area, and publish business-ready Parquet tables in a curated area. Each layer has a clear purpose.

Raw layer
I store exactly what arrives, unmodified and immutable, for audit and replay. I use newline-delimited JSON, optionally gzipped, under a simple layout like s3://lake/raw/app_logs/dt=YYYY-MM-DD/hr=HH/. Every ingestion writes to a temp path, then promotes and adds a _SUCCESS marker. I also write bad records to a separate rejects prefix with the error reason.

Processed layer
I enforce schema, types, and basic quality rules. I parse the JSON into structured columns, flatten only the fields we actually use, and keep nested objects in a single column if they are rarely queried. I add standard columns like event_time, dt, source, and an event_id for dedup. I mask or tokenize PII here so sensitive values never reach downstream in plain text. Output is Parquet, partitioned by dt (and maybe hour or region), with 128–512 MB files and compaction. I register the table in Glue, and allow only additive schema changes.

Curated layer
I publish business-oriented tables that analysts and BI use directly. These are clean, well-named Parquet tables, often arranged as a star schema (facts and dimensions) or purpose-built aggregates. I keep only the columns needed for analytics, set clear data types, and add surrogate keys if useful. If slowly changing dimensions are required, I maintain them here. I expose stable views that hide internal technical columns.

Cross-cutting practices
I manage metadata in Glue Data Catalog for all layers and use Lake Formation for permissions. I use partition projection for very large partition counts. I keep idempotency by writing to run-scoped paths and promoting only on success. I schedule validation checks (row counts, null rates, duplicate event_ids) and capture simple lineage: raw path → processed table → curated table. If the source adds fields, the pipeline accepts new nullable columns in processed, and I update curated on my schedule so downstream tools are not surprised.

In simple terms: land JSON as-is in raw, standardize and secure it as Parquet in processed, and publish tidy, business-ready Parquet tables in curated. Clear partitions, big files, atomic writes, and a single Glue catalog make the data easy and cheap to query.

3. Daily ETL jobs are creating too many small files in S3, slowing Spark and Athena queries. How would you fix this in the data lake design?

I would attack the root causes (over-partitioning and uncoordinated writers) and then add compaction so we always end up with a small number of large Parquet files per partition. My goal is to land 128–512 MB Parquet files, partitioned sensibly, written atomically, and compacted on a schedule.

Diagnose the issue quickly

- Check a few partitions and count files. If I see hundreds of files that are 1–10 MB, that explains the slow scans in Athena and the small-file overhead in Spark.
- Confirm the partition scheme. If we partition by too many keys (like dt=YYYY-MM-DD/hr=HH/min=MM/app=user_id), each partition becomes tiny and creates small files.
- Look at writer concurrency. Many small Spark tasks writing independently will each produce separate files.

Design changes to reduce small files at the source

- Right-size partitions. I keep dt at day, add hour only if volume is high. I avoid minute-level partitions and avoid putting high-cardinality fields (like user_id) in the partition path.
- Batch the write window. Instead of writing every 5 minutes, I buffer and write hourly where feasible. If upstream must be more frequent, I still compact within the hour.
- Control the number of output files. In Spark I set the number of shuffle partitions and use repartition or coalesce before the final write so each partition ends up with only a few large files.
- Target file size. I tune Spark to aim for ~256 MB files using options like spark.sql.files.maxRecordsPerFile (indirectly controls size) and by adjusting the number of output partitions to match the data volume.

Concrete Spark/Glue techniques

- Before writing: compute the target number of files per partition as roughly data_size_in_partition / 256MB, then use df.repartition(n, "dt") or df.repartitionByRange("dt") so Spark writes that many files.
- After write: if I still get small files, run a quick compaction job that reads a partition and writes it back with a lower number of output partitions (coalesce) to merge files.
- Use columnar format. Always write Snappy Parquet (or ORC). It reduces file count and speeds Athena.
- Avoid too many small tasks. Set spark.sql.shuffle.partitions to a sane number based on cluster size and daily input volume.
- Atomic, idempotent writes. Write to a temporary run path, then promote to the final prefix only on success. That prevents partial partitions with a mix of small files from failed runs.

Automated compaction and housekeeping

- Add a daily compaction job per table/partition. It reads yesterday's partitions and rewrites them into 128–512 MB files, then replaces the old files in a single commit.
- For streaming or micro-batch pipelines, I compact hourly and then run a final daily compaction to hit ideal sizes.
- Optional table formats. If the lake uses Apache Hudi, Delta Lake, or Apache Iceberg, I enable their built-in compaction/clustering so the table stays healthy automatically over time.

Improvements for Athena and downstream

- Keep partitions predictable and not overly granular so partition pruning actually helps.
- Use Glue Data Catalog with correct parquet statistics and enable partition projection for very high partition counts to avoid expensive MSCK REPAIR calls.
- Educate producers. If a producer pushes thousands of tiny JSON files, I either batch them upstream (for example, larger flush intervals) or stage them and merge during the processed step.

In simple terms: I reduce partition granularity, control how many files Spark writes per partition, always write Parquet, and schedule compaction so each partition ends up with a handful of large files. That fixes query speed and lowers compute cost.

4. You have 200 TB of logs in S3 Standard, but only the last 3 months are queried. How would you cut costs using storage classes?

I would keep the most recent logs in a fast class for analytics and shift older data to colder, cheaper classes automatically using lifecycle policies. I will also compress and organize the data so we pay less for both storage and retrieval.

Classify access patterns first

- Hot data: last 0–90 days, queried by analysts and jobs. Keep this quickly accessible.
- Warm/cold data: 3–12 months, rarely queried. Keep it cheaper but still retrievable without long waits.
- Archive: older than a year, kept mainly for compliance or audits.

Choose storage classes by age

- 0–30 days: S3 Standard for best performance (or S3 Intelligent-Tiering if access is unpredictable; it automatically moves objects between frequent and infrequent tiers).
- 31–90 days: S3 Standard-IA to drop storage cost while still keeping milliseconds access. Note the 30-day minimum storage charge, which fits this window.
- 91–365 days: S3 Glacier Instant Retrieval if we occasionally need to query these logs with low latency; if access is extremely rare, S3 Glacier Flexible Retrieval is cheaper but has minutes-to-hours retrieval.
- 365 days: S3 Glacier Deep Archive for the lowest cost, accepting hours-level retrieval time for rare audits.

Implement with lifecycle rules

- Create prefix-based rules per dataset, or use object tags like data_class=logs to target policies precisely.
- Example rule: transition to Standard-IA at 30 days, Glacier Instant Retrieval at 90 days, Deep Archive at 365 days, and optionally delete after your retention period (for example, 730 or 1825 days).
- Apply rules only after confirming minimum storage duration for each class to avoid early-deletion fees.

Keep retrieval costs predictable

- For teams that occasionally need older data quickly, I prefer Glacier Instant Retrieval for the 3–12 month window to avoid surprise restore delays.
- For bulk historical pulls, plan ahead: use bulk retrieval options and schedule restores during off-hours.

Reduce the footprint before moving

- Compress the logs. If they are text/JSON, store as gzip or, better, convert to Parquet in the processed layer to massively reduce size and speed up any future analytics.
- Avoid billions of tiny objects. If the raw layer has many small files, compact them in the processed layer so Intelligent-Tiering monitoring overhead and lifecycle actions remain efficient.

Operational guardrails

- Bucket encryption with KMS and bucket-level policies remain unchanged across storage classes.
- Enable S3 Storage Lens or Cost Explorer to verify savings and adjust thresholds based on real access patterns.
- Test restores from each cold class once, document runbooks, and set expectations for retrieval times and costs.

In simple terms: keep the last three months hot, move months 3–12 to a cheaper retrieval-friendly class, and push anything older into deep archive. Do this automatically with lifecycle policies, and shrink the data with compression/Parquet so you pay less from day one.

5. ML datasets are used heavily for the first few weeks, then rarely. Which S3 storage class would you pick, and how would you automate transitions?

I would start the dataset in a fast, general-purpose class while it is actively used, and then let S3 automatically move it to cheaper tiers as access drops. If usage is unpredictable across teams, I prefer S3 Intelligent-Tiering so S3 handles the movement without me guessing thresholds. If the pattern is always the same, I use clear lifecycle steps to the cheapest safe classes.

What I store and how I lay it out

- I keep each dataset version under a clean prefix like s3://ml/datasets/<project>/<version>/, and I tag objects with project, version, data_class=ml_dataset, created_utc, and retention_days.
- If the data is tabular, I write Parquet with Snappy. For image or text corpora, I compress archives or keep reasonably large objects to avoid many tiny files.

Option 1: unpredictable access, use S3 Intelligent-Tiering

- I upload to the dataset prefix with Intelligent-Tiering enabled. Objects start in the frequent access tier during the first weeks while training and experimentation are heavy.
- When objects are not accessed, S3 shifts them to infrequent access automatically, reducing cost with no changes to my code. If needed, I enable the archive tiers within Intelligent-Tiering so long-unused data moves even cheaper, with restore time similar to Glacier classes.
- There are no retrieval fees between frequent and infrequent tiers, so surprise reads are safe. I account for the small per-object monitoring cost; I keep objects reasonably large to keep that overhead tiny.

Option 2: predictable access, use lifecycle transitions

- Hot phase: 0–30 days in S3 Standard for best performance during training.
- Warm phase: 31–90 days in S3 Standard-IA for lower storage cost but instant access if we need to retrain quickly. I avoid moving earlier than 30 days to respect the minimum storage duration.
- Cold phase: 91–365 days in S3 Glacier Instant Retrieval or Glacier Flexible Retrieval, depending on how quickly we might need it back. Instant Retrieval keeps millisecond access with much lower storage price; Flexible Retrieval is cheaper but has minutes-to-hours restore.
- Archive: older than a year in S3 Glacier Deep Archive when it is kept only for audit or reproducibility. I document restore times and costs so no one is surprised.

How I automate it safely

- I attach lifecycle rules to the dataset prefix or to objects with data_class=ml_dataset, so I don’t affect other data in the bucket.
- I keep versioning on for safety, then add noncurrent version expiration (for example, delete noncurrent versions after 30–60 days) to avoid cost creep.
- I set abort-incomplete-multipart-uploads after 7 days to stop paying for abandoned parts.

- I use S3 Storage Lens or Cost Explorer to verify that transitions actually happen and that cost drops as expected. If I see repeated re-access right after a transition, I shift the thresholds or switch to Intelligent-Tiering.
- I store a manifest file per dataset version (list of keys, sizes, checksums) so restores or moves are easy.

In simple terms: I keep new ML datasets hot for a few weeks, then either let Intelligent-Tiering auto-optimize cost, or I step them down with lifecycle rules from Standard to Standard-IA and finally to Glacier tiers. I tag, version, and monitor so transitions are automatic and safe.

6. Old intermediate ETL files in S3 are driving up costs. How would you apply lifecycle rules and storage classes to clean this up safely?

I treat intermediate files as disposable once the final outputs are validated. I separate them clearly, tag them on write, move them quickly to cheaper classes, and then expire them on a short timer. I also clean up failed uploads and old versions so the bucket does not accumulate hidden costs.

Structure and tagging so cleanup is easy

- I keep clear prefixes for each stage, for example:
    - s3://lake/raw/ for immutable landings
    - s3://lake/processed/ for parquet outputs used by downstream
    - s3://lake/stage/tmp/ and s3://lake/stage/work/ for intermediate and scratch files
- I tag intermediate objects at write time with stage=intermediate, pipeline=name, run_id, and created_utc. Final tables get stage=curated or stage=processed.

Make deletion safe and idempotent

- Every job writes to a run-scoped temp prefix, then promotes committed outputs to the final processed prefix only after all checks pass. That way, if a run fails, its intermediates are already isolated.
- I drop a _SUCCESS marker with the final outputs and record expected row counts or checksums. The lifecycle policy for intermediates starts only after the corresponding final partition shows _SUCCESS.

Lifecycle rules I apply

- Transition quickly: move stage=intermediate objects to a cheaper class after a short delay. Common pattern is transition to S3 Standard-IA at 7 days because we rarely need intermediates beyond the first week. If the data is almost never re-read, I go straight to a Glacier tier at 7–14 days.
- Expire aggressively: delete stage=intermediate after 14–30 days. For massive pipelines I often choose 14 days; for mission-critical jobs I keep 30–60 days to allow reprocessing investigations.
- Version cleanup: enable versioning for safety but add a noncurrent version expiration, for example keep only the last 1 noncurrent version and expire older noncurrent versions after 7–14 days on intermediate prefixes.

- Abort incomplete uploads: set abort-incomplete-multipart-uploads after 7 days to stop paying for abandoned parts from failed runs.
- Scope rules precisely: target the tmp/work prefixes or tag filters stage=intermediate so I never touch raw or curated data by accident.

Extra cost and performance hygiene

- Compact intermediates during the job so there are fewer, larger files even before deletion. This reduces listing costs and lifecycle evaluation overhead.
- If intermediates must be kept briefly for debugging, I still compress them (gzip or Parquet) to reduce storage immediately.
- For datasets with a legal hold or audit needs, I exclude those prefixes via tags like legal_hold=true so lifecycle never deletes them.
- I monitor with S3 Storage Lens to confirm the object count and total bytes in stage=intermediate are trending down. If not, I adjust rules or fix pipelines that forgot to tag outputs.

Runbook for exceptions

- If a downstream job fails and we need to re-run, we read from raw or processed, not from intermediates. This lets us keep intermediate retention short.
- If a team occasionally needs intermediates for root-cause analysis, I extend the transition window slightly or place them in Intelligent-Tiering for the first week, then let deletion kick in.

In simple terms: I keep intermediates in their own prefix with tags, move them to a cheaper class within a few days, and delete them soon after final outputs are confirmed. I also clean up failed uploads and old versions. This keeps storage bills low without risking important data.

7. IoT raw data must be kept for 1 year, but analysts only query 30 days. How would you design lifecycle rules for compliance and cost efficiency?

I would separate what we query from what we only keep for compliance, use clear prefixes, and let lifecycle rules move and delete data automatically.

What I store where

- I keep the last 30 days in a "hot" prefix that Athena reads, for example: s3://iot/hot/device_events/dt=YYYY-MM-DD/
- I keep day 31 to day 365 in an "archive" prefix for compliance, for example: s3://iot/archive/device_events/dt=YYYY-MM-DD/

Storage classes I use

- Hot data (0–30 days): S3 Standard or S3 Intelligent-Tiering so queries stay fast
- Archive data (31–365 days): if we might need quick access sometimes, S3 Glacier Instant Retrieval; if access is almost never, S3 Glacier Flexible Retrieval or Deep Archive to save more

Lifecycle rules I set

- Rule for hot prefix: after 30 days, either delete the hot copy or move it to the archive prefix via a small job so we do not keep two copies
- Rule for archive prefix: at 365 days, expire the object to meet the one-year retention
- I add "abort incomplete multipart uploads after 7 days" to avoid paying for failed uploads
- If compliance requires write-once, I enable Object Lock (governance or compliance mode) on the archive bucket so no one can delete early

How I keep Athena safe

- Athena tables only point to the hot prefix
- I publish a default view that always filters to the last 30 days, so dashboards never scan old data by mistake
- If someone needs older data, we restore specific partitions from the archive into a temporary "restore" prefix and query from there

Basic security and housekeeping

- I encrypt with SSE-KMS and restrict archive access to a small group
- I tag objects with data_class=iot_raw and retention_until=YYYY-MM-DD so rules target the right data
- I watch S3 Storage Lens and Cost Explorer to check that hot data stays small and archive grows as expected

In simple words: keep 30 days in a fast area for Athena, push older days to a cheap archive for the rest of the year, and delete on day 366. Queries never touch the archive, so they do not break or get slow.

8. Athena queries fail because some files moved to Glacier. How would you redesign lifecycle policies to prevent this issue?

I would stop moving anything that Athena needs into classes it cannot read, and keep archived copies separate. The key is to split queryable data and archived data, and aim lifecycle rules only at the archive.

Why it broke

- Athena can read Standard, Standard-IA, One Zone-IA, Intelligent-Tiering, and Glacier Instant Retrieval
- Athena cannot read objects that are in Glacier Flexible Retrieval or Deep Archive unless they are restored first
- Lifecycle rules pushed table data into an archive class that Athena cannot read, so queries failed

How I redesign the layout

- I create two clear locations:

  s3://lake/analytics/<table>/dt=YYYY-MM-DD/ for data that Athena queries

  s3://lake/archive/<table>/dt=YYYY-MM-DD/ for long-term storage that Athena does not touch

- Athena tables and views only point to the analytics location

Storage classes I allow for analytics

- Keep analytics data in Standard or Intelligent-Tiering
- If we need to cut cost more but still be queryable, I can use Glacier Instant Retrieval because Athena can read it without a restore
- I never let analytics data transition to Glacier Flexible Retrieval or Deep Archive

Lifecycle policies I apply

- On analytics prefixes:

  Either do nothing, or at most move to Intelligent-Tiering (no retrieval surprises)

  Never transition to Glacier Flexible Retrieval or Deep Archive

- On archive prefixes:

  Move older data to Glacier Flexible Retrieval or Deep Archive on a schedule

  Expire after the retention period

- I scope rules by prefix or by object tags like stage=analytics or stage=archive so the right data follows the right policy

Safe migration from the current broken state

- For partitions already in Deep Archive or Flexible Retrieval that we need, I restore only the required dates to a temporary restore prefix, then copy them back into the analytics prefix in a supported class
- I update the Athena table or MSCK REPAIR to include the fixed partitions
- I add a guardrail check in the pipeline: before publishing a partition, we assert the storage class is allowed for analytics

Operational guardrails

- I tag analytics objects with stage=analytics and enforce a policy that blocks transitions to forbidden classes for that tag
- I enable an S3 EventBridge rule or a daily check that lists any analytics objects not in an allowed class and alerts the team
- I document a short runbook: where to restore from, which classes Athena supports, and how to re-point partitions if needed

In simple words: keep Athena's data in classes it can read, put true archives in a different place with their own lifecycle, and add small checks so nothing from analytics ever gets pushed into a non-readable class again.

9. ETL jobs generate temporary files in S3 that aren't needed later. How would you auto-clean them without deleting raw or curated data?

I would separate temporary files from important data and use lifecycle rules to auto-delete them after a short time. The key is to clearly tag or store temporary files in their own prefix so cleanup rules never touch raw or curated layers.

How I organize

- I create clear prefixes:
    - s3://lake/raw/ → permanent source of truth
    - s3://lake/processed/ → business-ready data
    - s3://lake/tmp/ or s3://lake/stage/ → temporary ETL files
- I also tag temporary files with stage=tmp or stage=intermediate at the time they are written.

How I auto-clean

- I set a lifecycle rule only for the tmp prefix (or objects with stage=tmp tag).
- The rule deletes files after a short time, usually 1–7 days depending on how long jobs might need them for retries.
- I also configure "abort incomplete multipart uploads after 7 days" so half-written files don't accumulate.

How I make it safe

- Rules are scoped only to the tmp prefix or tmp-tagged objects, so raw and curated are untouched.
- Jobs always write temp files to a run-specific subfolder like s3://lake/tmp/run_id=UUID/, then promote only the final good outputs into processed. This makes cleanup predictable.
- If debugging is needed, I can extend retention to 7–14 days, but it's still auto-cleaned.

In simple words: I keep temp files in their own folder, tag them, and set lifecycle rules to delete them after a few days. This keeps costs low and makes sure raw and curated data are never touched.

10. A developer accidentally deleted raw data, but versioning is enabled. How would you recover and prevent future issues?

Since versioning is on, the deleted objects aren't gone they just have delete markers. I can restore by removing those markers or copying older versions back. After recovery, I would add guardrails so this doesn't happen again.

How I recover

- In S3 console, I enable "show versions" and see that the deleted objects have previous versions still available.
- I either:
    - Delete the delete marker to bring the last version back, or
    - Copy the previous version to a new prefix like s3://lake/raw_restored/ and point Glue/Athena to it.
- For large datasets, I would use S3 Batch Operations or AWS CLI with --version-id to restore in bulk.

How I prevent this in future

- Enable MFA Delete on the raw bucket so no one can permanently delete or overwrite without MFA approval.
- Use S3 Object Lock (governance mode) to enforce write-once for a set period if compliance allows. That way raw data cannot be deleted until the retention expires.
- Restrict IAM permissions so developers only have write access to raw, not delete. Only admins can delete.
- Set up CloudTrail or S3 EventBridge to alert if anyone tries to delete raw data.

In simple words: I recover by rolling back to the previous versions since versioning kept them. Then I put guardrails like MFA Delete, Object Lock, and stricter IAM permissions so raw data can't be deleted by mistake again.

11. You need data to remain immutable for 7 years due to compliance. How would you use S3 Object Lock to meet this need while still supporting analytics?

I would keep an immutable "archive copy" of the raw data under Object Lock for 7 years, and do all analytics from a separate processed copy. This gives me legal-grade immutability without blocking day-to-day work.

How I set up the immutable archive

- I create a dedicated archive bucket just for compliance data, for example: s3://company-archive-raw/.
- When creating the bucket, I enable bucket versioning and turn on S3 Object Lock.
- I choose Compliance mode (cannot be shortened or removed by any user, even admins) with a default retention of 7 years. If the business needs a little flexibility, I use Governance mode but still restrict who can bypass it.
- I optionally use legal holds for specific investigations. A legal hold freezes objects until the hold is removed, independent of the 7-year timer.
- I lock only the archive bucket. This way, day-to-day analytics isn't slowed down.

How I write data into the archive

- Ingest pipelines write raw files once into the archive bucket using write-once keys (no overwrites).
- I tag objects with source, dt, and retention metadata so auditing is simple.
- If we need a second region for resilience, I set up S3 Replication with Object Lock enabled on the destination bucket so retention carries over.

How I support analytics safely

- I keep a separate analytics bucket/prefix, for example: s3://lake/processed/ and s3://lake/curated/. These are not object-locked.
- ETL jobs read from the immutable archive and produce clean Parquet tables in the processed/curated area. Analysts and Athena/Glue point only to processed/curated, not to the locked archive.
- If a transformation bug is found, I can always reprocess from the immutable archive because it's untouched and guaranteed complete.
- I use lifecycle policies on processed/curated to control cost (transition older partitions to cheaper classes that Athena can still read, or expire if fully replaceable).

Operational guardrails

- I restrict delete/overwrite permissions on the archive bucket to prevent mistakes. Even if someone tries, Object Lock will block it.
- I monitor with S3 Inventory and Storage Lens to prove retention and show auditors the evidence.
- I encrypt both buckets with KMS and log access with CloudTrail for audit trails.

In simple words: I lock the raw data for 7 years in a special bucket so nobody can change or delete it, and I do all reporting from a separate processed copy. If something goes wrong, I can always rebuild from the locked raw data.

12. Versioning caused multiple large file versions, increasing costs. How would you balance versioning with lifecycle rules to control cost?

I keep versioning where I truly need safety (raw and critical data), and I clean up noncurrent versions everywhere else with clear lifecycle rules. I also change how we write data to avoid creating too many versions in the first place.

Where I keep versioning and where I don't

- Raw and archive buckets: keep versioning on for safety and audit.
- Processed/curated buckets: keep versioning on, but aggressively clean up old versions because we can always rebuild from raw.
- Scratch/tmp buckets: versioning usually not needed; if enabled, clean them very quickly.

Lifecycle rules I apply to control cost

- Noncurrent version transitions: move noncurrent versions to cheaper classes after 30 days (for example, Standard-IA or Glacier Instant Retrieval if we may need a quick rollback; Glacier Flexible Retrieval for rarely needed rollbacks).
- Noncurrent version expiration: delete noncurrent versions after 60–90 days, or keep only the last N noncurrent versions (for example, keep the last 1–2) and expire the rest. This keeps a safety net without paying for dozens of copies.
- Abort incomplete multipart uploads after 7 days so abandoned parts don't waste money.
- Scope rules by prefix or tags. For example, stage=processed gets aggressive cleanup; stage=raw keeps longer retention.

Change write patterns to reduce versions

- Write new data to new keys instead of overwriting the same key. For example, write by partition path dt=YYYY-MM-DD/hr=HH/ so a new run doesn't create versions of the same object.
- Use atomic "publish": write to a run_id path, validate, then copy/rename into the final partition. This avoids repeated overwrites of the same object and therefore avoids many noncurrent versions.
- Compact outputs so there are fewer, larger files. Fewer files means fewer versions overall if something is updated.

Monitor and tune

- Use S3 Storage Lens and Cost Explorer to see where noncurrent bytes are growing.
- If I see frequent rollbacks needed, I extend the noncurrent retention a bit (for example, from 30 to 45 days). If no rollbacks happen, I shorten it.

In simple words: keep versioning where it protects you, but set lifecycle rules to move old versions to cheaper storage and delete them after a short, safe window. Also, prefer writing new files instead of overwriting the same ones so you don't create lots of versions in the first place.

13. Customer data in India must also be stored in the EU for DR. How would you configure Cross-Region Replication (CRR), and what factors must you consider?

I would set up one S3 bucket in India as the source and one S3 bucket in the EU as the destination, enable versioning on both, and create a replication rule that copies new and changed objects automatically. I also make sure encryption and permissions are correct so the replication never gets stuck.

How I set it up

- Create the source bucket in India (for example, ap-south-1) and the destination bucket in the EU (for example, eu-west-1 or eu-central-1).
- Turn on bucket versioning on both buckets. Replication requires versioning.
- Create a replication rule in the source bucket:
    - Scope it by prefix or object tags if we only want to replicate specific data (for example, only customer data).
    - Choose replicate delete markers if DR requires keeping deletes in sync, or skip it if you want the EU copy to be "backup style".
    - Turn on replication metrics and optional Replication Time Control (RTC) if we need a time-bound SLA for replication.
- Configure object ownership in the rule to make the destination bucket owner the owner of replicated objects. This avoids access issues later.
- Encryption:
    - If we use SSE-S3, no extra work.
    - If we use SSE-KMS, create or pick KMS keys in both regions. Update the key policies so the S3 replication role can decrypt with the source key and encrypt with the destination key. Multi-Region KMS keys make this simpler.
- IAM and roles:
    - Let S3 create the replication role automatically or provide a role with s3:ReplicateObject, s3:ReplicateDelete, and KMS permissions.
- Test with a small file and check the object's replication status in its metadata.

Factors I consider before choosing regions and settings

- Compliance and data transfer rules. Confirm cross-border transfer is allowed and documented. Sometimes legal asks for specific EU region choices.
- Recovery point and time objectives. If we need tighter guarantees, I enable RTC; otherwise standard CRR is usually enough.
- Cost. Cross-region data transfer and extra PUT requests add cost. I set lifecycle rules on the EU bucket to use cheaper storage classes after a few weeks if the DR copy is rarely read.
- Storage classes and query behavior. If analytics will also run in the EU during DR, I keep the EU copy in a class Athena can read (Standard, Intelligent-Tiering, or Glacier Instant Retrieval). If it is true cold DR, I may move older objects to Glacier classes in the EU.

- Security. Encrypt both sides, restrict who can read the EU copy, and log all access. Consider S3 Object Lock in the EU if the DR copy must also be immutable.
- Existing data. CRR only replicates new objects by default. If we must copy historical data, I run S3 Batch Replication once.

In simple terms: I enable versioning, add a replication rule from India to the EU, give S3 the right KMS and IAM permissions, and test. Then I tune lifecycle, cost, and security so the DR copy is safe, compliant, and affordable.

14. Some objects are missing in your Same-Region Replication (SRR) bucket. How would you troubleshoot and fix this?

I would check the basics first (versioning, rule scope, encryption, and permissions), look at replication metrics to see which objects failed, and then re-replicate only the missing ones.

What I check first

- Versioning. Make sure versioning is enabled on both source and destination. Without it, replication does not work.
- Rule scope. Confirm the replication rule actually covers the objects' prefixes or tags. If the rule filters by a tag that the objects do not have, they will be skipped.
- Encryption. If the source objects use SSE-KMS, make sure the replication role can decrypt with the source KMS key and encrypt with the destination KMS key. Replication does not work with customer-provided SSE-C keys.
- Ownership and ACLs. Use bucket owner enforced and set the replication rule to change object ownership to the destination bucket owner. This avoids "replicated but not readable" cases.
- Replication status. On a missing object in the source bucket, check the x-amz-replication-status. If it shows FAILED or PENDING, we know replication attempted or is stuck.

Where I look for errors

- S3 replication metrics and dashboard in the S3 console. I look for failed operations, bytes pending replication, and any recent spikes.
- CloudWatch logs and EventBridge notifications if we enabled them for replication failures.
- KMS key policy. Very often the missing objects are the ones encrypted with a key that the replication role cannot use.

Common root causes and fixes

- Rule did not apply to those objects. Fix the rule filter (prefix or tags) and re-run replication for those objects using S3 Batch Replication.
- No permission to use KMS. Update the key policies to allow the S3 replication role to decrypt source and encrypt destination. After fixing, trigger Batch Replication for affected keys.
- Objects existed before SRR was turned on. Set up S3 Batch Replication once to backfill historical objects.

- Delete markers or metadata-only changes. If we need deletes to replicate, enable “replicate delete markers” in the rule. Otherwise the destination will keep old copies.
- Application overwrote keys too often, creating many versions rapidly. Replication may lag or fail if the role or limits were misconfigured. Reduce overwrites by writing to run-specific keys and publishing atomically.

How I catch up and prevent repeats

- Use S3 Batch Replication to copy only the missing objects. I scope it by date or prefix so it is fast and cheap.
- Keep replication metrics and alarms on so we know quickly if failures return.
- Add a small daily check that lists source keys for the last day and confirms they exist in the destination. If not, it alerts and optionally triggers Batch Replication automatically.

In simple terms: I verify versioning, rule filters, KMS permissions, and ownership. I use the S3 replication metrics to find what failed, fix the cause, and backfill the missing objects with Batch Replication. Then I keep simple alarms so it does not happen again.

15. You need to replicate only logs/objects to another account's bucket in a different region. How would you design this policy and IAM setup?

I would use Cross-Region Replication (CRR) with cross-account permissions, scope the rule only to the log prefix, and set IAM and bucket policies so replication is automatic and secure.

How I design it

- Source bucket: in my account, where raw logs land (for example, ap-south-1).
- Destination bucket: in the other account, in a different region (for example, eu-west-1). This bucket is owned by the partner account but grants permissions to my replication role.
- Both buckets must have versioning enabled.

Replication rule setup

- On the source bucket, I create a replication rule scoped only to the logs prefix, for example s3://my-source-bucket/logs/.
- The rule includes "replicate delete markers" if we want deletes mirrored; if it's DR, we usually skip delete replication so the destination keeps a backup copy.
- I enable replication metrics and possibly Replication Time Control (RTC) if SLA guarantees are needed.

IAM and bucket policies

- Replication role (in my account):
    - Trust policy: trusted by S3 service.
    - Permissions: s3:GetObjectVersionForReplication, s3:GetObjectVersionAcl, s3:GetObjectVersionTagging on the source bucket.
    - Permissions: s3:ReplicateObject, s3:ReplicateDelete, s3:ReplicateTags on the destination bucket.
    - If encrypted with KMS, the role also needs kms:Decrypt on the source key and kms:Encrypt on the destination key.
- Destination bucket policy (in the other account):
    - Grants my replication role permission to put objects.
    - Example statement: Allow principal = arn:aws:iam::<source-account-id>:role/S3ReplicationRole, action = s3:ReplicateObject, s3:ReplicateDelete, resource = arn:aws:s3:::destination-bucket/*

Encryption

- If using SSE-S3, nothing extra.
- If using SSE-KMS, I set up a KMS key in the destination account and give the replication role encrypt permissions.

Testing

- I upload a small test object into s3://my-source-bucket/logs/.
- In the destination account's bucket, I confirm it appears and its metadata shows Replication status = COMPLETED.

In simple words: I scope replication to just the logs prefix, create a replication role in my account, grant it permission in the other account's bucket, and test with a small object. This way only logs are replicated, and nothing else.

16. A raw data bucket was found to be public. How would you secure it immediately and prevent future risks?

First, I would lock it down right away, then I would put permanent guardrails so it can't accidentally go public again.

Immediate actions

- Disable public access using the S3 Block Public Access settings on the bucket and at the account level. This blocks any ACL or policy that tries to make it public.
- Review and delete any bucket policies or object ACLs that grant public access (for example, Principal = *).
- Confirm using the S3 console "Access Analyzer" that the bucket is no longer publicly accessible.

Prevent future risks

- Keep Block Public Access enabled by default at the account level.
- Enforce IAM policies so developers cannot change public access settings. For example, deny s3:PutBucketAcl with public grants.
- Use Service Control Policies (if in an organization) to block making buckets public.
- Enable CloudTrail and GuardDuty to alert if any changes happen to bucket policies or ACLs.
- Use AWS Config rules such as s3-bucket-public-read-prohibited to continuously check and auto-remediate if any bucket turns public.
- Keep data encrypted (SSE-KMS) and give access only via IAM roles with least privilege.

Extra governance

- Enable versioning and Object Lock (governance mode) if the data must remain immutable.
- If logs or sensitive data are inside, review access logs to confirm whether anyone actually accessed the bucket while it was public.

In simple words: I immediately block public access and remove open policies, then set account-wide Block Public Access, monitoring, and IAM restrictions so no one can accidentally expose raw data again.

17. Your S3 data lake is accessed by multiple teams with different permissions. How would you design IAM roles and bucket policies for least-privilege access?

I would split access by "who does what" (producer, consumer, curator, admin), scope permissions to only the prefixes each team needs, and enforce guardrails at the bucket level so mistakes don't open up data. I keep encryption and network controls consistent for everyone.

How I organize the data and access

- I organize data by domains and layers so it's easy to grant narrow access, for example:
    - s3://lake/raw/<domain>/...
    - s3://lake/processed/<domain>/...
    - s3://lake/curated/<domain>/...
- I create IAM roles per team persona:
    - producer_role_<domain>: can write only to raw/<domain> and list the bucket paths it needs
    - curator_role_<domain>: can read raw/<domain>, write processed/<domain>, and manage schema/metadata if required
    - consumer_role_<team>: read-only to curated/<team_or_domain> paths
    - lake_admin_role: limited set of admins who can manage policies, not used for day-to-day work
- Users and jobs assume these roles (no long-lived IAM users).

Least-privilege IAM policies (key ideas)

- For read-only consumers:
    - s3:ListBucket with a condition to allow listing only keys that start with allowed prefixes (s3:prefix, s3:delimiter)
    - s3:GetObject only on arn:aws:s3:::lake/curated/<team_or_domain>/*
- For producers:
    - s3:PutObject and s3:AbortMultipartUpload only on arn:aws:s3:::lake/raw/<domain>/*
    - s3:ListBucket limited to raw/<domain> prefixes
    - Optional s3:GetObject if their job needs to read its own uploads for validation
- For curators:
    - s3:GetObject on raw/<domain>/*
    - s3:PutObject on processed/<domain>/* (and maybe curated/<domain>/* if they publish)
    - s3:ListBucket scoped to those prefixes

Bucket-level guardrails (one-time controls)

- Block public access at account and bucket level.
- Enforce TLS only: bucket policy denies any request where aws:SecureTransport is false.
- Enforce encryption on write: bucket policy denies s3:PutObject if x-amz-server-side-encryption is missing or not aws:kms.
- Enforce bucket-owner-enforced object ownership so uploads default to bucket owner and ACLs are not needed.
- Optional network control: require S3 access via specific VPC endpoints using aws:SourceVpce in the bucket policy.

KMS key access

- Use a dedicated KMS key for the lake.
- KMS key policy allows only the lake roles to encrypt/decrypt.
- For consumer roles, allow kms:Decrypt; for producer roles, allow kms:Encrypt and kms:Decrypt if they need to verify; scope with conditions on the bucket and prefixes.

Scaling and simplicity with S3 Access Points (optional but helpful)

- Create one access point per domain or team, each with its own policy limited to that prefix.
- Optionally restrict access points to specific VPCs (VPC-only access).
- Point jobs and Athena workgroups to the right access point alias to avoid long bucket policies.

Extra safety and hygiene

- Permission boundaries for developer roles so they can't grant themselves more S3 rights.
- AWS Config rules to alert on any bucket that becomes public or any policy that allows wild access.
- CloudTrail and S3 server access logs enabled for auditing.
- For extremely large partition sets, partition projection and read-only views to further reduce accidental scans.

In simple words: I give each team a role that can only see and touch its own folders, I block public and unencrypted access at the bucket, and I control KMS keys so only approved roles can read or write. This keeps access tight and easy to manage.

18. A partner needs temporary read-only access to certain S3 objects. How would you grant this securely (e.g., pre-signed URLs, access points)?

I choose the simplest secure method based on the use case duration and scale. For one-off or small batches, I use short-lived pre-signed URLs. For ongoing access, I set up a cross-account role or an access point with a tight policy. In all cases, I make sure KMS permissions match.

Option 1: pre-signed URLs (fastest for small, short-term sharing)

- When to use: a few files, access for hours or days, minimal setup.
- How I do it:
    - Generate pre-signed GET URLs with a short expiry (for example, 1–24 hours).
    - If using SSE-KMS, the signing role must have kms:Decrypt permission; the URL will work without exposing KMS keys to the partner.
    - Send the links over a secure channel. Rotate them if needed.
- Pros: no partner IAM setup, time-limited, easy to revoke by rotating objects or expiring URLs.
- Cons: not ideal for thousands of objects or long-term access.

Option 2: cross-account IAM role the partner can assume (best for ongoing integrations)

- When to use: repeated access, automation, many objects.
- How I do it:
    - In my account, create role PartnerReadOnlyRole with a trust policy allowing the partner's AWS account to assume the role (use an external ID to prevent confused-deputy issues).
    - Attach an IAM policy that allows s3:ListBucket (scoped by prefix) and s3:GetObject only on the allowed prefixes, for example arn:aws:s3:::lake/exports/partnerX/*.
    - Update the bucket policy to allow that role to GetObject and List with the same prefix scope.
    - If encrypted with KMS, allow that role kms:Decrypt on the lake's KMS key with a condition limiting it to the bucket and prefix.
    - Partner configures an IAM role or user in their account to assume PartnerReadOnlyRole and uses temporary credentials.
- Pros: least-privilege, auditable, easy to turn off by removing trust or policy.
- Cons: initial setup coordination.

Option 3: S3 Access Point for the partner (clean scoping, can be VPC-restricted)

- When to use: you want a dedicated endpoint and policy per partner.

- How I do it:
    - Create an access point pointing to the bucket, set its policy to allow s3:GetObject on the specific prefixes for the partner's account or role.
    - Optionally restrict the access point to a specific partner VPC (if they connect via PrivateLink or have a peering setup).
    - Ensure KMS permissions include the partner principal that uses the access point.
- Pros: keeps bucket policy simple, easy to see and manage partner-specific access.
- Cons: still requires KMS coordination and careful policy writing.

Revocation and monitoring

- To revoke access, remove the trust relationship (cross-account role), delete or change the access point policy, or let pre-signed URLs expire.
- Enable S3 server access logs or CloudTrail data events to audit what the partner accessed.
- Use object tags (for example, partner=partnerX) and scope policies to those tags for flexible control.

Extra safeguards

- Deny list in bucket policy for any action outside the approved prefixes or without TLS.
- Set narrow expirations for pre-signed URLs and avoid sharing entire listings unless required.
- If the data is sensitive, package it under an "exports" prefix so nothing else is ever exposed by mistake.

In simple words: for quick, short-term sharing I use pre-signed URLs. For ongoing read-only access, I create a cross-account role or an access point that can only read a specific folder, and I make sure KMS allows just that role. It's easy to audit and easy to turn off.

19. Athena queries large CSV files but only need a few columns. How would you use S3 Select to cut cost and improve performance?

I would use S3 Select to read only the columns and rows I need from each CSV object instead of downloading the whole file. This reduces data scanned and speeds up downstream processing. I typically use it in a small pre-filter step that writes a lighter dataset for Athena to query.

How I apply it in practice

- I add a lightweight Lambda or Glue job that reads each CSV object with S3 Select and runs a simple SQL like "SELECT col_a, col_b FROM S3Object WHERE event_date BETWEEN ...".
- The job writes the filtered result as Parquet into a processed prefix. Athena then queries the Parquet table, which is much faster and cheaper.
- If a tool or notebook needs to read directly from S3, I call S3 Select from the SDK and pull only the required columns, not the whole file.

Key setup choices

- Keep CSV newline-delimited and clean (consistent headers, types).
- Push as many filters into S3 Select as possible (date range, status flags).
- Convert the S3 Select output to Parquet with Snappy so future queries are columnar and small.
- Use parallelism by running one S3 Select call per object in parallel, then merge outputs.

In simple words: I let S3 Select do the first cut on each CSV object (pick columns and filter rows), then I store that reduced data in Parquet for Athena. This lowers scanned bytes and speeds everything up.

20. A data science team queries JSON logs but only needs some fields. How would you use S3 Select with Parquet/ORC to optimize efficiency?

For raw JSON, I use S3 Select to project only needed fields and to filter rows before I write Parquet. For Parquet, S3 Select can also read specific columns inside each object, which is useful for lightweight extraction tools. For ORC, I would not rely on S3 Select; instead I would use normal engines (Athena/Spark) and let them prune columns.

How I design the flow

- Raw JSON → use S3 Select to pull only the fields needed (for example, event_time, user_id, country) and only the rows I care about (date range, event_type). Then I write Parquet.
- Parquet → if a small script or Lambda needs data, I can use S3 Select on Parquet to fetch just a few columns from each object. For big analytics, Athena or Spark already prune Parquet columns efficiently, so I just query the Parquet table directly.
- ORC → I keep using Athena/Spark for column pruning. If the team wants S3 Select style reads, I prefer converting to Parquet.

Quality and performance tips

- Normalize timestamps and types before writing Parquet so downstream tools don't spend time fixing schemas.
- Partition by date so both S3 Select and analytics engines read fewer objects.
- Keep Parquet files in the 128–512 MB range for good scan efficiency.
- Mask or drop PII during the JSON→Parquet step so sensitive fields never travel further than needed.

In simple words: use S3 Select to trim raw JSON to only the fields and rows you need, store the result as Parquet, and then let Athena or Spark query that Parquet efficiently. If you must read Parquet from a small script, S3 Select can return just the columns you need.

21. Spark ETL jobs on S3 are slow due to small files and poor partitions. How would you redesign file format, partitioning, and compression for performance and cost?

I would switch to a columnar format, fix the partition strategy, and control how many files Spark writes so each partition ends up with a few large Parquet files. I also schedule compaction so the table stays healthy over time.

File format and compression

- Write Snappy-compressed Parquet (or ZSTD if your stack supports it well) for all processed/curated data. Parquet is columnar, so queries scan only needed columns.
- Avoid plain CSV/JSON for analytics tables; keep them only in the raw layer for audit.
- Disable schema merge at write time unless you truly need it, to avoid extra overhead.

Partitions that actually help

- Partition by date (dt=YYYY-MM-DD). Add hour (hr=HH) only if the daily volume is very high.
- Do not partition by high-cardinality fields like user_id or session_id; that creates too many tiny partitions and many small files.
- Keep partition columns stable and few in number so engines can prune effectively.

Write fewer, larger files

- Before the final write, repartition the DataFrame by the partition column to control the number of output files per partition.
- Aim for 128–512 MB per Parquet file. As a rule of thumb, target file_count_per_partition ≈ partition_size / 256 MB.
- Tune Spark settings so you don't create thousands of small tasks: set a sane spark.sql.shuffle.partitions and coalesce before the write when needed.
- Use an atomic publish pattern: write to a temporary run path, compact if needed, then move to the final table path.

Ongoing compaction and housekeeping

- Add a daily compaction job that reads yesterday's partitions and rewrites them into the target file size if they're fragmented.
- If you use Delta Lake, Apache Hudi, or Apache Iceberg, turn on their OPTIMIZE/compaction/clustering features so the table maintains good file sizes automatically.
- Remove leftover _temporary and failed-run folders, and set "abort incomplete multipart uploads" on the bucket.

Extra read speed wins

- Keep a concise schema: drop unused columns, use correct types (timestamp, bigint, decimal).
- Enable predicate and column pushdown (Parquet filter pushdown) and avoid functions that break it (for example, apply date filters on partition columns, not on transformed expressions).

- Register tables in Glue Data Catalog with clear partitions; use partition projection if you have very many partitions to avoid heavy metastore operations.
- Cache small dimension tables in Spark if you join frequently.

In simple words: store data as Snappy Parquet, partition mainly by date, and make sure each partition has only a handful of large files. Use compaction regularly and avoid high-cardinality partitions. This makes Spark and Athena much faster and cheaper.

22. You want Glue ETL to run automatically when new files arrive in S3. How would you design S3 event notifications to avoid duplicates/misses?

I design the flow knowing S3 events are "at-least-once" (duplicates can happen) and deliveries can be delayed. So I never start Glue directly from S3. I put a reliable queue in the middle, make the job idempotent, and add a daily reconciliation.

How I wire it

- S3 sends only the events I care about (ObjectCreated:Put and ObjectCreated:CompleteMultipartUpload) into SQS. I use prefix/suffix filters so only the right folder and file types trigger it, for example raw/appX/dt=/ and *.json.gz.
- A small Lambda reads from SQS in batches and starts or bookmarks the Glue job with the object key, ETag, and versionId.
- If I expect many files per day, I do micro-batches: Lambda groups keys by partition (for example, same dt) and triggers Glue once per partition, passing a manifest list to Glue to process together.

How I avoid duplicates

- I make Glue idempotent: before writing processed data, it checks a DynamoDB table (or a _SUCCESS marker) to see if this object/version was already processed. The idempotency key is bucket + key + versionId (or ETag).
- Glue writes to a run-scoped temp path and promotes to the final partition only once. If the same file comes again, the job does nothing.
- In Lambda, I also use a short-lived de-dup cache (for example, DynamoDB TTL of a few hours) to skip obvious repeats before starting Glue.

How I avoid misses

- I enable an SQS dead-letter queue (DLQ). If the Lambda can't process a message after retries, it lands in DLQ for manual review.
- I set SQS visibility timeout > Lambda's max runtime so messages don't reappear mid-processing.
- I add a daily "reconciliation" Glue job that reads the S3 listing (or S3 Inventory) for yesterday's prefix and compares against the processed manifest. Any missing keys are queued again.

Operational details

- Increase Lambda memory to speed up S3 HEAD/manifest work, and set reserved concurrency so it scales but doesn't start too many Glue runs at once.
- Use EventBridge rules for additional routing if later I need to send a copy of the event to monitoring or a second pipeline.

In simple terms: send S3 events to SQS, trigger Glue through Lambda, make the job idempotent, and run a daily catch-up so duplicates don't hurt and misses are re-queued automatically.

23. S3 → Lambda integration misses some large files. How would you troubleshoot and redesign it for both small and large files?

I check whether the event fired, whether Lambda received it, and whether processing failed due to time or memory. Then I redesign to use SQS between S3 and Lambda, tune timeouts, and handle multi-part uploads properly.

Troubleshooting steps

- Verify event configuration: are we listening to ObjectCreated:CompleteMultipartUpload as well as Put? Large files often use multi-part, so only "Put" may miss them.
- Check filters: wrong prefix/suffix filters can exclude large files if they land in a different folder or extension.
- Look at S3 object metadata: confirm the upload actually completed and isn't a partial or aborted upload.
- Check CloudWatch logs and Lambda metrics: look for timeouts, out-of-memory, or throttling. Also check concurrent executions; throttling can drop events if not using a queue.
- Confirm permissions: ensure Lambda's role can GetObject on the source, and KMS decrypt if objects are SSE-KMS.

Redesign for reliability and size

- Put SQS in the middle: S3 → SQS (notifications) → Lambda (polls SQS). This gives retries, buffering, DLQ, and removes the risk of direct-event throttling.
- Include both ObjectCreated:Put and ObjectCreated:CompleteMultipartUpload in the S3 notification so large multi-part uploads trigger only after they fully complete.
- Set SQS visibility timeout longer than Lambda's max runtime. Add a DLQ for poison messages.
- Increase Lambda timeout and memory so it can handle larger headers/manifests. If Lambda copies data, consider moving that work to Glue or Step Functions instead of doing heavy work inside Lambda.
- If objects are extremely large, avoid reading them in Lambda. Have Lambda write a manifest entry, and let a Glue/Spark job process in parallel from S3.

Performance and cost tips

- Batch SQS messages (for example, 5–10 keys per batch) so Lambda does fewer cold starts.
- For very hot buckets, use Standard SQS (not FIFO) for higher throughput, and add idempotency in processing.
- Add a periodic reconciliation using S3 Inventory to catch any stragglers.

In simple terms: include the right event types for big files, put SQS between S3 and Lambda, tune timeouts/memory, and move heavy processing to Glue so both small and large files are handled reliably.

24. You need to push S3 file metadata into SQS for real-time processing. How would you configure S3 event notifications for scale and reliability?

I configure S3 to send only the necessary events into SQS, design message shape clearly, and add buffering, retries, and DLQ so it scales without losing messages.

Event configuration

- Turn on notifications for ObjectCreated:Put and ObjectCreated:CompleteMultipartUpload. I usually skip Copy/Post unless needed.
- Use prefix/suffix filters so only relevant folders/extensions generate messages, for example raw/appY/ and *.parquet.
- If many teams need the events, I fan out via SNS (S3 → SNS → multiple SQS queues), or use EventBridge rules to route to multiple targets.

SQS queue setup

- Choose Standard SQS for very high throughput. If strict ordering and de-dup are required, use FIFO with content-based deduplication (throughput will be lower).
- Set a visibility timeout comfortably larger than the consumer's processing time.
- Configure a DLQ with an appropriate maxReceiveCount (for example, 5). Alert on DLQ messages.

Message content and idempotency

- The S3 event already includes bucket, key, size, ETag, and (if versioning) versionId. I keep these as the idempotency key in my consumer so reprocessing the same object does nothing.
- If I need extra metadata (tags, custom headers), the consumer can call HeadObject to enrich the message before downstream processing.

Consumer design

- Use an autoscaling consumer (Lambda with reserved concurrency, or a container service polling SQS) so it keeps up with spikes.
- Process messages in small batches (for example, 5–10). For each message, validate the object exists and is in a "ready" state.
- On success, delete the message from SQS. On transient failure, let SQS retry; on repeated failure, the message lands in DLQ for manual action.

Reliability and observability

- Enable CloudWatch metrics on SQS (ApproximateNumberOfMessagesVisible, NotVisible, DLQ depth) and add alarms.
- Add a daily reconciliation job using S3 Inventory: list objects created yesterday under the prefixes and compare with what was processed. Re-queue any gaps.
- If encrypted with SSE-KMS, ensure S3 can publish to SQS (SQS policy) and the consumer has kms:Decrypt for the data it needs to read.

Cost and noise control

- Keep filters tight so you don't flood SQS with unneeded events.
- For extremely chatty workloads, consider aggregating events: have S3 send to EventBridge, then a small rule batches keys into a manifest and pushes fewer, larger messages to SQS.

In simple terms: S3 sends only the right "object created" events to SQS, the consumer scales and is idempotent, there's a DLQ for failures, and a daily reconciliation catches anything that slips through. This gives real-time, reliable metadata flow at scale.

25. IoT data in S3 makes Athena queries slow because of wide scans. How would you redesign the partitioning strategy to improve performance?

I would make partitions match how people actually filter the data, keep them simple and predictable, and avoid very high-cardinality columns. My main goal is that every Athena query touches only a small set of folders, not the whole table.

What I change in the layout

- I organize data by date first, because almost all queries filter by time. For example: s3://lake/iot/device_events/dt=YYYY-MM-DD/hr=HH/.
- If we also filter by geography or site, I add a second low-cardinality partition such as region or site. For example: dt=YYYY-MM-DD/region=APAC/ or region first if most queries filter by region.
- I never partition by device_id because it creates millions of tiny folders. Instead I keep device_id as a normal column and let the engine filter it inside the files.

How this speeds Athena

- When queries include predicates like dt between and region in, Athena prunes all other partitions and scans only the needed folders.
- Fewer partitions per query means fewer files to open, lower scanned bytes, and much faster results at lower cost.

Practical settings and guardrails

- I store data as Parquet with Snappy so Athena can read just the needed columns.
- I size files to about 128–512 MB per file so each partition has a handful of files, not thousands.
- I register the table in Glue with the same partition columns (order matters). I enable partition projection for large date ranges so Athena does not need to load millions of partitions into the catalog.
- I use clear naming and keep partition columns stable. Changing partition columns later is painful and slows queries.

In simple words: partition mainly by time (and maybe region/site), not by device_id. Keep files columnar and reasonably large. Then Athena only reads the few folders that match the filter and runs much faster.

26. You're worried about read-after-write consistency for new files in S3. How would you design the ingestion pipeline to ensure consistent downstream reads?

I design the pipeline so readers only see a partition after it is fully written and validated. I use a temporary path, a success marker or manifest, and an atomic "promote" step. This makes readers consistent even if infrastructure is slow.

Write pattern

- The writer lands data in a run-scoped temporary folder like s3://lake/tmp/run_id=UUID/... .
- After writing all files, it does validation checks (row counts, schema, basic quality).
- Only then it moves or copies the files into the final partition path like s3://lake/processed/table/dt=YYYY-MM-DD/hr=HH/.
- Finally it writes a small _SUCCESS file or a manifest file listing the exact objects for that partition.

Read pattern

- Downstream jobs and Athena read only partitions that have the _SUCCESS file or a manifest present.
- If a downstream job runs while a partition is still being written, it simply skips it because the marker is not there yet.

Idempotency and retries

- The writer includes a run_id and an idempotency key (bucket + key + version or ETag) in a small tracking table. If a retry happens, it does not publish the same data twice.
- If a run fails, the tmp folder is left behind and can be cleaned by lifecycle rules. No half-written data appears in the final path.

Operational tips

- For event-driven loads, I queue S3 events into SQS and trigger processing from there to avoid race conditions.
- I compact small files before the promote step to keep final partitions healthy.
- I keep schema and partition columns consistent so readers can rely on predictable locations.

In simple words: write to a temp area, validate, then publish and drop a _SUCCESS or manifest. Readers only look at published partitions, so they always see complete, consistent data.

27. Spark ETL produces thousands of tiny Parquet files per partition in S3, slowing Athena. How would you optimize partitioning and file sizes?

I would reduce partition granularity, control the number of output files, and add compaction so each partition ends up with only a few large Parquet files. This makes both Spark and Athena much faster.

Fix partitioning

- Keep partitions simple, usually by dt=YYYY-MM-DD, and add hr=HH only if the daily volume is very high.
- Do not partition by high-cardinality columns like device_id or user_id. That is the main cause of tiny files.

Control file counts at write time

- Before the final write, I repartition by the partition column and set the number of output partitions to target 128–512 MB per file. A simple rule is target_files ≈ partition_size / 256 MB.
- I coalesce output partitions if the data is small to avoid creating dozens of tiny files.
- I set reasonable Spark shuffle partitions so I do not create thousands of tiny tasks that each write a file.

Use the right format and compression

- I write Snappy Parquet for processed and curated data. Parquet is columnar and compresses well, which reduces both storage and scan cost.
- I avoid writing CSV/JSON in analytics layers; I keep those only in raw.

Add compaction

- I schedule a small daily job that reads yesterday's partitions and rewrites them into large files if they are fragmented. After compaction, I replace the old files in one atomic step.
- If I use Delta Lake, Apache Hudi, or Apache Iceberg, I enable their optimize/compaction features so tables stay healthy automatically.

Safe publish

- I write to a run-scoped temp path, compact there, and then promote to the final partition with a _SUCCESS marker. This prevents half-written partitions.

Athena-friendly practices

- I keep file counts per partition low (for example, 5–20 files). That reduces open/close overhead.
- I register partitions in Glue and consider partition projection for very large date ranges.
- I keep schemas clean and typed correctly so predicate pushdown works well.

In simple words: stop over-partitioning, write Parquet, control the number and size of output files, and compact regularly. Each partition should have a handful of big files, not thousands of tiny ones. This makes Athena scan much less data and run much faster.

28. Your team is running multiple queries on a large dataset stored in an S3 bucket, and the queries are taking too long to execute. How would you optimize Athena queries to improve performance?

I would reduce the amount of data Athena scans and cut the number of files it has to open. I also make queries more selective and keep the table design simple.

What I fix first

- Store data in columnar format (Parquet or ORC) with Snappy compression. This lets Athena read only the columns it needs instead of whole rows.
- Right-size files. Aim for about 128–512 MB per file so each partition has a small number of large files, not thousands of tiny ones.
- Partition by how we filter. Usually this is by date (dt=YYYY-MM-DD) and maybe region or site. I avoid high-cardinality partitions like user_id.

Cut scanned bytes in queries

- Always filter on partition columns (for example, WHERE dt BETWEEN … AND … AND region IN …). This prunes folders before reading files.
- Select only the columns needed instead of using SELECT *.
- Avoid functions that break pushdown (for example, don't wrap partition columns in functions; filter them directly).

Table and catalog hygiene

- Keep a clean schema with correct data types (timestamp, bigint, decimal). Drop unused columns.
- Use Glue Data Catalog with the same partition columns and order as the S3 layout. For very large tables, turn on partition projection so Athena doesn't need to load millions of partitions.
- If data is messy, create CTAS (CREATE TABLE AS SELECT) tables that are already clean and columnar, then point analysts to those.

File layout and compaction

- Add a daily compaction job that merges small files into large Parquet files per partition.
- For streaming feeds, compact hourly and then daily.

Query patterns that help

- Use approximate functions or aggregates where exact detail is not needed.
- Pre-aggregate heavy reports (daily or hourly summaries) into separate curated tables so dashboards hit small tables.

In simple words: convert to Parquet, partition by time (and maybe region), keep a few big files per partition, filter on partitions, and only select the columns you need. This makes Athena scan a lot less and run much faster.

29. You receive JSON data with deeply nested structures in an S3 bucket. How would you flatten and extract specific attributes using Athena?

I would first land the raw JSON as-is, then create a processed table that either flattens important fields or converts JSON to Parquet with those fields extracted. The goal is to make queries simple and cheap.

How I set it up

- Create a raw external table over the JSON location using a JSON SerDe. This lets me inspect the structure quickly.
- Identify the fields analysts need most (for example, event_time, user.id, device.os, items array).

How I query nested JSON

- Use json_extract_scalar for single values (strings, numbers) and json_extract for objects.
- For arrays, use CROSS JOIN UNNEST to explode them into rows, then pick only what is needed.
- Keep the filter on top-level or partition columns (for example, dt) so Athena prunes data before parsing JSON.

Make it easy for the team

- Create a CTAS query that reads raw JSON, extracts the important fields into proper columns, and writes Snappy Parquet into a processed prefix. Partition by dt.
- Register a clean processed table with real types (timestamp, bigint, map/struct only if truly needed). Most users should query this table, not the raw JSON.

Lightweight examples (short and readable)

- Extract scalar fields:
    - SELECT json_extract_scalar(payload, '$.user.id') AS user_id
- Explode an array:
    - SELECT t.event_id, i.item_id FROM raw t CROSS JOIN UNNEST(cast(json_extract(t.payload, '$.items') AS array(json))) AS u(i)

Operational tips

- Keep the raw layer immutable for audit. Do masking for PII when creating the processed table.
- If new fields arrive, add them as new nullable columns in the processed table to avoid breaking downstream tools.

In simple words: read the raw JSON to understand it, use json_extract and UNNEST to pull just the fields you need, then write a clean Parquet table with those fields so future queries are fast and simple.

30. Your company handles sensitive financial data stored in Amazon S3, and you need to enforce strict access controls for Athena queries. What measures would you take?

I would lock down S3, restrict who can query what in Athena, encrypt everything with KMS, and log all access. I also prevent network and public access issues.

Secure S3 first

- Block Public Access at the account and bucket level. Use bucket owner enforced (no ACLs).
- Bucket policy denies any request without TLS (deny if aws:SecureTransport is false).
- Bucket policy enforces encryption on write (deny PutObject if server-side encryption header is missing or wrong).
- Store data in clearly separated prefixes (for example, raw/processed/curated) so it's easy to grant narrow access.

Encryption and keys

- Encrypt all objects with SSE-KMS using a dedicated KMS key for finance data.
- KMS key policy allows only finance roles and specific Athena workgroups to decrypt.
- Encrypt Athena query results and the query history bucket with the same or another controlled KMS key.

Control who can query what

- Use AWS Lake Formation or fine-grained Glue/Athena permissions to grant database/table/column access per role. For sensitive columns (like card_number), use column-level or row-level filters.
- Expose views that mask or tokenize sensitive columns for most users. Only a small break-glass role can query the raw columns.
- Create separate Athena workgroups per team, and enforce:
    - Allowed data locations (only curated/finance paths)
    - Result encryption required
    - Per-workgroup limits and CloudWatch metrics

Network and endpoint controls

- Require access via S3 VPC endpoints and limit bucket policy to specific VPC endpoints (aws:SourceVpce).
- For partner or cross-account access, use cross-account roles with external IDs rather than opening bucket policies.

Operations and monitoring

- Enable CloudTrail data events for S3 and Athena to capture who read what and when.
- Turn on S3 server access logs or CloudTrail Lake for deeper analysis.
- Add AWS Config rules to alert if any bucket becomes public or if encryption is disabled.
- Use lifecycle rules to remove old query results from the results bucket and to control noncurrent versions.

Least privilege by design

- Give analysts read-only to curated tables they need, not raw.
- Give ETL roles write-only to their output paths and read-only to inputs.
- Use permission boundaries so developers cannot grant themselves extra access.

In simple words: keep data private by default, encrypt with KMS, let only specific roles and workgroups read certain tables or columns, force TLS and VPC endpoints, and log everything. Most users see masked views; only a few audited roles can see raw sensitive fields.

31. Your team wants to schedule daily reports from AWS Athena and store the results in S3 for downstream analysis. How would you automate this process?

I would run a daily SQL that writes results to S3 as Parquet, and automate it with a scheduler. I also keep results partitioned by date and encrypted.

Design

- Write the report as a CTAS or INSERT INTO query that produces Parquet in s3://reports/<report_name>/dt=YYYY-MM-DD/.
- Use a dedicated Athena workgroup with an S3 query-results bucket and KMS encryption.

Automation options

- Use Athena Scheduled Queries or an EventBridge rule that triggers a Lambda. The Lambda calls StartQueryExecution, polls GetQueryExecution until it succeeds, and handles retries.
- On success, the job writes Parquet files under dt=today. On failure, it sends an SNS or Slack alert.

Good practices

- Parameterize the date (for example, run for dt = yesterday).
- Keep files 128–512 MB and partition by dt so downstream tools can filter easily.
- Store a manifest row in DynamoDB or a small CSV alongside each run with row counts and a checksum to prove completeness.
- Expire old report partitions or move them to cheaper storage classes after some days.

In simple words: schedule a daily CTAS/INSERT query in Athena, write Parquet to a dated folder in S3, encrypt the results, and alert on failures. Downstream jobs read the dt partition they need.

32. A new data source is added to your existing dataset in S3, introducing additional fields not part of the current schema, and queries start failing. How would you handle this schema evolution?

I would make schema changes additive, keep raw and processed separate, and shield downstream queries with views. The idea is to accept new columns safely without breaking existing queries.

Steps I take

- Keep the raw JSON/CSV as-is in a raw layer. Do not change it.
- In the processed Parquet table, add new columns as nullable with sensible defaults. Use ALTER TABLE ADD COLUMNS or recreate via CTAS if needed.
- If types changed (for example, amount string → decimal), fix the types during the processed step using safe casts or TRY_CAST, and write the corrected types into Parquet.

Protect current users

- Avoid SELECT * in production queries. Publish a stable view that lists the known columns. When new fields arrive, the view still works.
- When ready, extend the view to expose the new columns.

Fix existing bad partitions

- If older partitions have the old schema and newer ones have the new schema, rewrite recent partitions with CTAS so the Parquet schema is consistent.
- If there are corrupt values causing failures, use TRY_CAST or CASE to convert or null out only the bad rows, and send them to a rejects prefix for review.

Governance

- Document the data dictionary (column name, type, description) and version it.
- For high-change sources, land to a flexible raw table, then normalize to a stable curated schema that changes less often.

In simple words: accept new fields as nullable in processed Parquet, avoid breaking queries by using views that list known columns, and standardize types during the ETL. If needed, rewrite recent partitions so the schema stays consistent.