

Cloud Storage for Data Engineers

S3, GCS, and ADLS — patterns, conventions, and tools every DE needs.

Last reviewed 29 Sep 2026 ·

Prerequisites: [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/)

Related: [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) · [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) · [Terraform](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/terraform-for-de/) · [Azure and Fabric](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

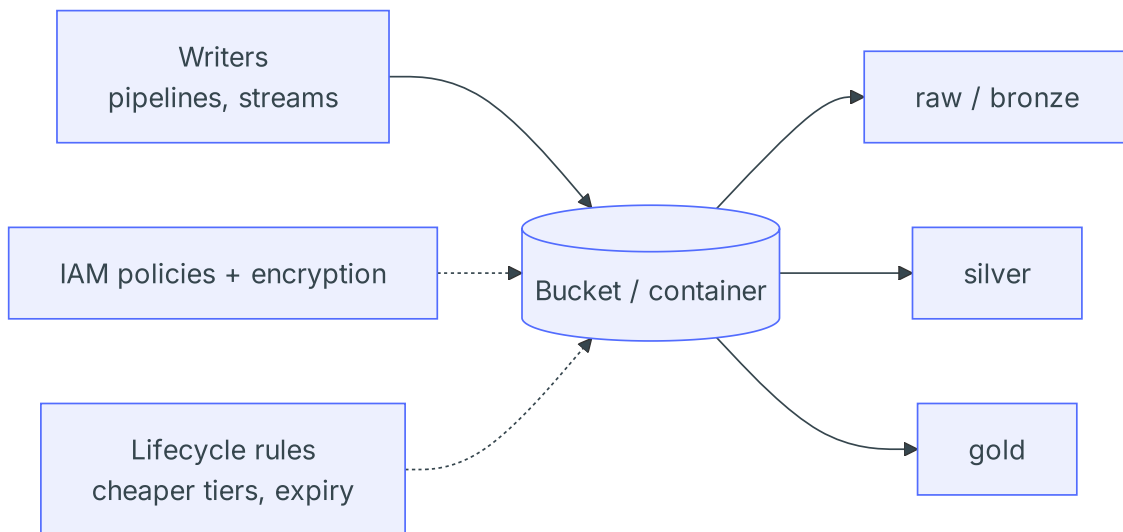
Challenge: Databases are expensive per gigabyte and are not designed to hold petabytes of raw logs, events, and exports. Local disks fill up, fail, and cannot be shared by many compute nodes at once.

Solution: Object storage services — Amazon S3, Google Cloud Storage, and Azure Data Lake Storage — store files ("objects") in buckets with effectively unlimited capacity, very high durability, and low cost. Virtually every processing engine, warehouse, and query service can read from them.



storage is separate from compute:
scale (and pay for) each independently

Trade-offs: Object storage is not a filesystem. Folders are only key prefixes, renames are copy-and-delete operations, and every request has a cost. File layout — partitioning, file sizes, and formats — therefore determines whether queries take seconds or hours, and is the focus of most of this guide.



On this page

Basics - [Object Storage Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#object-storage-concepts) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#object-storage-concepts) - [Amazon S3](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#amazon-s3) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#amazon-s3) - [Google Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#google-cloud-storage) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#google-cloud-storage) - [Azure Data Lake Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#azure-data-lake-storage) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#azure-data-lake-storage)

Intermediate - [Storage Layout Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#storage-layout-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#storage-layout-patterns) - [Partitioning Conventions](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#partitioning-conventions) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#partitioning-conventions) - [Access & IAM](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#access-iam) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#access-iam) - [Python SDK Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#python-sdk-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#python-sdk-patterns)

Advanced - [Performance & Cost Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#performance-cost-optimization) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#performance-cost-optimization) - [Lifecycle Policies](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#lifecycle-policies) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#lifecycle-policies) - [Storage in Spark & Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#storage-in-spark-databricks) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#storage-in-spark-databricks) - [Cross-Cloud Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#cross-cloud-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#cross-cloud-patterns)

Reference - Common Pitfalls (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#common-pitfalls>) - **Cheat Sheet** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/#further-reading>)

Object Storage Concepts

Object storage is fundamentally different from a filesystem. There are no real directories — only **keys** (strings with `/` as a convention to mimic folders) and **objects** (blobs of bytes).

Filesystem:	Object Storage:
/data/	bucket: my-data-lake
raw/	key:
raw/orders/2024/03/15/orders.parquet	
orders/	key:
raw/orders/2024/03/16/orders.parquet	
2024/	key: silver/orders/2024-03-15/part-
00000.parquet	
03/	
15/	
orders.parquet	

Why it matters for DE

Property	Implication
Infinitely scalable	No capacity planning — store petabytes
Cheap	~\$0.02/GB/month for object storage vs ~\$0.10/GB for SSD block storage (illustrative — check provider pricing pages for current rates)
No rename	Renaming = copy + delete (expensive for large files)
No atomic directory ops	"Moving" a folder = copy all objects + delete originals
Eventual consistency → Strong consistency (S3 since 2020)	Reads after writes are now consistent on S3, GCS, ADLS
Durability	11 nines (99.999999999%) — multiple copies across AZs

Amazon S3

CLI (aws s3)

```
# List
aws s3 ls s3://my-bucket/
aws s3 ls s3://my-bucket/raw/orders/ --recursive
aws s3 ls s3://my-bucket/ --recursive --human-readable |
tail -1 # total size

# Copy / Upload / Download
aws s3 cp local_file.parquet s3://my-
bucket/raw/file.parquet
aws s3 cp s3://my-bucket/raw/file.parquet ./
aws s3 cp s3://source-bucket/ s3://dest-bucket/ --recursive
# bucket-to-bucket

# Sync (like rsync – only transfers changed/new files)
aws s3 sync ./local/ s3://my-bucket/raw/
aws s3 sync s3://my-bucket/raw/ ./local/
aws s3 sync s3://source/ s3://dest/ --exclude "*.tmp"

# Delete
aws s3 rm s3://my-bucket/raw/file.parquet
aws s3 rm s3://my-bucket/raw/old/ --recursive

# Move (copy + delete – no atomic rename)
aws s3 mv s3://bucket/source.parquet
s3://bucket/dest.parquet

# Presigned URL – time-limited URL for sharing
aws s3 presign s3://my-bucket/file.parquet --expires-in
3600 # 1 hour

# Check object metadata
aws s3api head-object --bucket my-bucket --key
raw/orders/file.parquet
```

Storage classes

Class	Use case	Cost (relative)	Retrieval
Standard	Frequently accessed data	\$\$\$	Instant
Standard-IA	Infrequent access (monthly)	\$\$	Instant
One Zone-IA	Infrequent, non-critical	\$	Instant
Glacier Instant	Archive, accessed quarterly	\$	Instant
Glacier Flexible	Archive, accessed rarely	¢	1-12 hours
Glacier Deep Archive	Long-term archive	¢¢	12-48 hours
Intelligent-Tiering	Unknown access patterns	Auto	Instant

Google Cloud Storage

```
# gcloud storage CLI (replaced the legacy gsutil)
gcloud storage ls gs://my-bucket/
gcloud storage ls -l gs://my-bucket/raw/          # with
sizes
gcloud storage ls --recursive gs://my-bucket/raw/

gcloud storage cp local.parquet gs://my-bucket/raw/
gcloud storage cp gs://my-bucket/file.parquet ./
gcloud storage cp --recursive gs://bucket/dir/ ./local/ #
copy directory

gcloud storage rsync --recursive ./local/ gs://my-
bucket/raw/    # sync
gcloud storage mv gs://bucket/src.parquet
gs://bucket/dst.parquet

gcloud storage rm gs://my-bucket/file.parquet
gcloud storage rm --recursive gs://my-bucket/old-dir/

# Signed URL
gcloud storage sign-url --duration=1h --private-key-
```

```
file=service_account.json gs://my-bucket/file.parquet

# Storage classes: STANDARD, NEARLINE (monthly access),
COLDLINE (quarterly), ARCHIVE
gcloud storage objects update --storage-class=NEARLINE
"gs://bucket/archive/**"
```

Azure Data Lake Storage

ADLS Gen2 (Azure Data Lake Storage Gen2) combines Blob Storage with a hierarchical namespace for better performance on large-scale analytics.

```
# Azure CLI
az storage blob list --container-name raw --account-name
mystorageaccount

az storage blob upload \
  --file local.parquet \
  --container-name raw \
  --name orders/2024/03/15/orders.parquet \
  --account-name mystorageaccount

az storage blob download \
  --container-name raw \
  --name orders/file.parquet \
  --file ./local.parquet \
  --account-name mystorageaccount

# azcopy – high-performance copy tool
azcopy copy ./local/
"https://account.blob.core.windows.net/container/" --
recursive
azcopy sync ./local/
"https://account.blob.core.windows.net/container/"

# URL format
#
abfss://container@account.dfs.core.windows.net/path/to/file
#
```

```
wasbs://container@account.blob.core.windows.net/path/to/file (legacy)
```

Storage Layout Patterns

A consistent layout is critical — it determines partition pruning, access patterns, and cost.

Medallion (Bronze / Silver / Gold)

```
s3://my-data-lake/
bronze/                ← raw, as-is, immutable
  orders/
    ingested_date=2024-03-15/
      part-00000.parquet
silver/                ← cleaned, validated, typed
  orders/
    order_date=2024-03-15/
      part-00000.parquet
gold/                  ← aggregated, business-ready
  daily_revenue/
    snapshot_date=2024-03-15/
      part-00000.parquet
checkpoints/          ← streaming checkpoints
logs/                 ← pipeline run logs
schemas/              ← schema registry files
```

Raw landing zone

```
s3://my-landing-zone/
source_name/
  table_name/
    YYYY/MM/DD/HH/
      filename.csv      ← time-partitioned by arrival
```

Archive zone

```
s3://my-archive/
source_name/
```

```
table_name/  
  YYYY/  
    filename.parquet  
landing
```

← year-level partitions
← consolidated from raw

Partitioning Conventions

Object storage partitions are directory-level splits. The engine (Spark, Athena, BigQuery) reads only the directories matching your filter — **partition pruning**.

```
# Hive-style partitioning (most common – Spark/Athena read these natively)
orders/order_date=2024-03-15/part-00000.parquet
orders/order_date=2024-03-16/part-00000.parquet

# Explicit path partitioning (when Hive style isn't required)
orders/2024/03/15/part-00000.parquet
```

Good partitioning keys

Strategy	Key	When
Date	<code>order_date=YYYY-MM-DD</code>	Event/transaction data — almost always use this
Hour	<code>event_hour=YYYY-MM-DD-HH</code>	High-volume streaming data
Region	<code>region=us-east</code>	Geographic fan-out, compliance requirements
Source	<code>source=crm</code>	Multi-source ingestion to one table

```
# Write Hive-partitioned Parquet in PySpark
df.write \
    .mode("overwrite") \
    .partitionBy("order_date") \
    .parquet("s3://my-data-lake/silver/orders/")

# Dynamic partition overwrite – overwrite only partitions in the data
```

```
spark.conf.set("spark.sql.sources.partitionOverwriteMode",
"dynamic")
df.write.mode("overwrite").partitionBy("order_date").parquet(
path)

# Read with partition filter (Spark prunes automatically)
df = spark.read.parquet("s3://my-data-lake/silver/orders/")
df.filter("order_date = '2024-03-15'") # reads only that
directory
```

Partition sizing

```
Target: 128 MB - 1 GB per partition file
Too small (< 10 MB): "small file problem" - metadata
overhead dominates
Too large (> 2 GB): single task takes too long, memory
pressure

# If partitions are too small - coalesce before writing
df.coalesce(10).write.parquet(path)

# If files are too large - repartition
df.repartition(100).write.parquet(path)

# Rule of thumb for daily Hive partitions:
# < 1M rows/day → 1 file per partition
# 1M-50M rows/day → 4-8 files per partition
# > 50M rows/day → scale files with data volume
```

Access & IAM

S3 IAM policy patterns

Pipeline role — read raw, write silver:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```

        "Effect": "Allow",
        "Action": ["s3:GetObject", "s3:ListBucket"],
        "Resource": [
            "arn:aws:s3::my-data-lake/bronze/*",
            "arn:aws:s3::my-data-lake"
        ]
    },
    {
        "Effect": "Allow",
        "Action": ["s3:GetObject", "s3:PutObject",
"s3:DeleteObject"],
        "Resource": "arn:aws:s3::my-data-
lake/silver/*"
    }
]
}

```

BI tool role — read gold only:

```

{
    "Statement": [{
        "Effect": "Allow",
        "Action": ["s3:GetObject", "s3:ListBucket"],
        "Resource": [
            "arn:aws:s3::my-data-lake/gold/*",
            "arn:aws:s3::my-data-lake"
        ]
    }]
}

```

Best practices

- Use **IAM roles** (for EC2/ECS/Lambda), not access keys
- Use **instance profiles** for machines, not user credentials
- Apply **least privilege** — read-only for BI, write only to the layer each pipeline owns
- Enable **S3 Block Public Access** on all buckets — no exceptions
- Enable **S3 server-side encryption** (SSE-S3 or SSE-KMS)
- Enable **versioning** on landing zones to protect against accidental overwrites
- Use **bucket policies** to restrict access to specific VPCs or IAM roles

Python SDK Patterns

boto3 (AWS)

```
import boto3
from botocore.exceptions import ClientError

s3 = boto3.client("s3", region_name="us-east-1")
resource = boto3.resource("s3")

# List objects
response = s3.list_objects_v2(Bucket="my-bucket",
Prefix="raw/orders/2024/")
for obj in response.get("Contents", []):
    print(obj["Key"], obj["Size"])

# Paginator – for buckets with > 1000 objects
paginator = s3.get_paginator("list_objects_v2")
for page in paginator.paginate(Bucket="my-bucket",
Prefix="raw/orders/"):
    for obj in page.get("Contents", []):
        print(obj["Key"])

# Upload / Download
s3.upload_file("local.parquet", "my-bucket",
"raw/orders/file.parquet")
s3.download_file("my-bucket", "raw/orders/file.parquet",
"local.parquet")

# Upload from memory (BytesIO)
import io
buffer = io.BytesIO()
df.to_parquet(buffer, index=False)
buffer.seek(0)
s3.upload_fileobj(buffer, "my-bucket",
"raw/orders/file.parquet")

# Read directly into pandas
import pandas as pd
df = pd.read_parquet("s3://my-
bucket/raw/orders/file.parquet") # via s3fs
df = pd.read_csv("s3://my-bucket/raw/orders/file.csv")
```

```

# Check if object exists
def object_exists(bucket, key):
    try:
        s3.head_object(Bucket=bucket, Key=key)
        return True
    except ClientError as e:
        if e.response["Error"]["Code"] == "404":
            return False
        raise

# Delete
s3.delete_object(Bucket="my-bucket",
Key="raw/file.parquet")

# Delete all objects with a prefix
objects = s3.list_objects_v2(Bucket="my-bucket",
Prefix="raw/old/")
delete_keys = [{"Key": o["Key"]} for o in
objects.get("Contents", [])]
if delete_keys:
    s3.delete_objects(Bucket="my-bucket", Delete=
{"Objects": delete_keys})

```

google-cloud-storage (GCS)

```

from google.cloud import storage

client = storage.Client()
bucket = client.bucket("my-data-lake")

# Upload
bucket.blob("raw/orders/file.parquet").upload_from_filename(
"local.parquet")

# Download
bucket.blob("raw/orders/file.parquet").download_to_filename(
"local.parquet")

# List blobs
for blob in client.list_blobs("my-data-lake",

```

```

prefix="raw/orders/2024/"):
    print(blob.name, blob.size)

# Check existence
blob = bucket.blob("raw/orders/file.parquet")
blob.exists()

# Read into bytes
data =
bucket.blob("raw/orders/file.parquet").download_as_bytes()

# Read directly into pandas
import pandas as pd
df = pd.read_parquet("gs://my-data-
lake/raw/orders/file.parquet")

```

azure-storage-blob (ADLS)

```

from azure.storage.blob import BlobServiceClient

client = BlobServiceClient.from_connection_string(conn_str)
container = client.get_container_client("raw")

# Upload
container.upload_blob("orders/file.parquet",
open("local.parquet", "rb"))

# Download
blob = container.get_blob_client("orders/file.parquet")
with open("local.parquet", "wb") as f:
    f.write(blob.download_blob().readall())

# List
for b in
container.list_blobs(name_starts_with="orders/2024/"):
    print(b.name, b.size)

# Read directly into pandas (via adlfs)
import pandas as pd
df =

```

```
pd.read_parquet("abfs://raw@account.dfs.core.windows.net/orders/file.parquet")
```

Performance & Cost Optimization

Avoid the small file problem

```
# Bad – thousands of tiny 1 MB files per partition
df.write.partitionBy("order_date", "hour").parquet(path)
# → 24 files * 365 days = 8,760 files in a year partition

# Better – one partition per day
df.write.partitionBy("order_date").parquet(path)

# Coalesce before writing to control file count
df.coalesce(4).write.partitionBy("order_date").parquet(path)

# Compact small files periodically (Delta OPTIMIZE or Spark job)
spark.sql("OPTIMIZE delta.`s3://my-data-lake/silver/orders`")
```

Request costs

```
S3 pricing (US East, approximate – check
aws.amazon.com/s3/pricing for current rates):
  GET requests:    $0.0004 per 1,000
  PUT requests:    $0.005 per 1,000
  Storage:         $0.023 per GB/month

Cost trap: listing 1 million objects = 1,000 LIST requests
= $0.005
          But Glue/Athena crawlers listing daily = costs
add up

Cost saver: use partition discovery over manual listing
           batch small writes into fewer larger files
```

Multipart upload (large files)

```
# boto3 handles this automatically for files > 8MB
# Force multipart for large transfers:
from boto3.s3.transfer import TransferConfig

config = TransferConfig(
    multipart_threshold=25 * 1024 * 1024,    # 25 MB
    max_concurrency=10,
    multipart_chunksize=25 * 1024 * 1024,
    use_threads=True,
)

s3.upload_file("large_file.parquet", "my-bucket",
               "path/file.parquet",
               Config=config)
```

Lifecycle Policies

Automatically transition or delete objects based on age.

S3 lifecycle policy — transition and expire raw data (objects under `bronze/` expire after 2,555 days, about 7 years):

```
{
  "Rules": [
    {
      "ID": "archive-old-raw-data",
      "Status": "Enabled",
      "Filter": {"Prefix": "bronze/"},
      "Transitions": [
        {"Days": 30, "StorageClass": "STANDARD_IA"},
        {"Days": 90, "StorageClass": "GLACIER_IR"},
        {"Days": 365, "StorageClass": "GLACIER"}
      ],
      "Expiration": {"Days": 2555}
    },
    {
```

```

        "ID": "delete-temp-files",
        "Status": "Enabled",
        "Filter": {"Prefix": "tmp/"},
        "Expiration": {"Days": 7}
    }
]
}

```

Storage in Spark & Databricks

```

# Configure S3 access in Spark (using instance profile on EMR/EC2)
# No credentials needed if IAM role is attached

# Direct path reads
df = spark.read.parquet("s3://my-bucket/silver/orders/")
df = spark.read.parquet("gs://my-bucket/silver/orders/")
df =
spark.read.parquet("abfss://container@account.dfs.core.windows.net/orders/")

# Databricks: mount cloud storage (legacy)
dbutils.fs.mount(
    source="s3a://my-bucket/silver",
    mount_point="/mnt/silver",
)
df = spark.read.parquet("/mnt/silver/orders/")

# Databricks: Unity Catalog external location (modern)
# Configure via UI: Catalog > External Locations
# Then use Unity Catalog table references directly

# Read all partitions matching a filter
df = spark.read \
    .parquet("s3://my-bucket/silver/orders/") \
    .filter("order_date >= '2024-01-01'") # partition
pruning applied automatically

# Read specific partitions (faster than filter on large
datasets)

```

```

from datetime import date, timedelta
dates = [(date(2024,3,1) + timedelta(d)).strftime("%Y-%m-%d") for d in range(31)]
paths = [f"s3://bucket/orders/order_date={d}/" for d in dates]
# basePath keeps order_date as a column – without it Spark drops the partition column
df = spark.read.option("basePath", "s3://bucket/orders/").parquet(*paths)

# Write with dynamic partition overwrite
spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")
df.write \
    .mode("overwrite") \
    .partitionBy("order_date") \
    .parquet("s3://my-bucket/silver/orders/")

```

Cross-Cloud Patterns

Many organizations run data on more than one cloud — through acquisitions, customer requirements, or deliberate vendor diversification. The main concerns are cost, identity, consistency of access, and data residency.

Egress is the dominant cost

Storage is cheap; moving data out of a cloud (or across regions) is not. Reading a 10 TB table from another cloud every day can cost more than storing it for a year.

Pattern When to use	Egress per run
Query data where it lives Default – keep compute next to data	Results only
Replicate once, read locally many times Many readers in the other cloud	Once per change
Stream only changes (CDC / incremental) Near-real-time sync between clouds	Deltas only

Copy everything on every run
Avoid

Full dataset

Replication options

Need	AWS	GCP	Azure	Cloud-neutral
Same-cloud, other region	S3 Cross-Region Replication	Dual-/multi-region buckets, Turbo Replication	Object replication, GRS/GZRS	—
Into this cloud from elsewhere	DataSync	Storage Transfer Service	AzCopy, Data Factory	<code>rclone</code> , table-format replication
Table-level sync	—	—	—	Copy Iceberg/Delta snapshots incrementally; share via open catalogs or Delta Sharing

One access layer across clouds

```
import fsspec
import pyarrow.dataset as ds

# The same code reads from any provider; only the URI
# scheme changes
for uri in ["s3://lake-aws/orders/", "gs://lake-
gcp/orders/",
"abfs://lake@account.dfs.core.windows.net/orders/"]:
    fs, path = fsspec.core.url_to_fs(uri)
    dataset = ds.dataset(path, filesystem=fs,
format="parquet")
    print(uri, dataset.count_rows())
```

Open table formats (Iceberg, Delta) with a shared catalog extend this to tables: engines in either cloud read the same table metadata, so there is one definition of the data rather than two diverging copies.

Identity without long-lived keys

Use **workload identity federation** instead of copying access keys between clouds: a workload in cloud A presents its native identity token, and cloud B exchanges it for short-lived credentials.

From → To	Mechanism
AWS → GCP	GCP Workload Identity Federation trusting the AWS account/role
GCP → AWS	AWS IAM <u>OIDC</u> identity provider trusting Google-issued tokens
Azure ↔ others	Microsoft Entra ID workload identity federation / federated credentials
CI (GitHub Actions) → any cloud	OIDC federation — no stored cloud secrets

Data residency and governance

- Tag datasets with their permitted regions and enforce it in replication jobs, not just in documentation
- Keep personal data in its home region; replicate aggregated or pseudonymized derivatives instead
- Apply the same classification, encryption, and retention rules in every cloud, ideally from one IaC codebase
- Centralize audit logs so access can be reviewed across clouds in one place

Common Pitfalls

Pitfall	Symptom	Fix
The small file problem (thousands of KB-sized files)	Slow queries, slow listings, high request costs	Target 128 MB–1 GB files; <code>coalesce</code> before writing; compact regularly (<code>OPTIMIZE</code> , Iceberg <code>rewrite_data_files</code>)
Partitioning on a high-cardinality column (<code>user_id</code>)	Millions of directories, each with a tiny file	Partition on date (and maybe one low-cardinality column); cluster/Z-order on the rest

Pitfall	Symptom	Fix
Long-lived access keys in code or on laptops	Leaked keys, unauthorized access	<u>IAM</u> roles / instance profiles / workload identity; short-lived credentials only
A bucket left public by accident	Data breach headlines	Block Public Access at the account level; audit with AWS Config / Security Hub
<code>mode("overwrite")</code> without dynamic partition overwrite	One day's rerun deletes the whole table	<code>partitionOverwriteMode=dynamic</code> , or use a table format (Delta/Iceberg) with <code>replaceWhere</code> / <code>MERGE</code>
Reading explicit partition paths without <code>basePath</code>	Partition column missing from the DataFrame	<code>spark.read.option("basePath", root).parquet(*paths)</code>
Treating a "rename" as atomic (e.g. write to <code>_tmp/</code> then move)	Readers see half-moved data; large moves are slow	Use a table format — commits are atomic metadata swaps
No lifecycle rules	Storage bill grows forever with temp files and old raw data	Lifecycle transitions (IA → Glacier) and expiration for <code>tmp/</code>
Cross-region reads/writes	Surprise data-transfer bill; slower jobs	Keep compute in the same region as the bucket
Listing huge prefixes to find new files	Slow and expensive at millions of objects	Event notifications (S3 → SQS), Auto Loader, or table-format metadata
Still scripting with <code>gsutil</code>	Slower transfers; the tool is legacy	Use <code>gcloud storage</code> (same verbs: <code>ls</code> , <code>cp</code> , <code>rsync</code> , <code>rm</code>)

Cheat Sheet

Cross-cloud equivalents

Concept	AWS	GCP	Azure
Object store	S3	Cloud Storage (GCS)	Blob Storage / ADLS Gen2

Concept	AWS	GCP	Azure
URI (Spark)	s3://bucket/key (s3a:// on OSS Hadoop)	gs://bucket/key	abfss://container@account.dfs.core.windows.net/path
CLI	aws s3	gcloud storage (legacy gsutil)	az storage / azcopy
Python SDK	boto3	google-cloud-storage	azure-storage-blob / azure-storage-file-datalake
pandas/fsspec backend	s3fs	gcsfs	adlfs
Infrequent tier	Standard-IA	Nearline	Cool
Archive tier	Glacier / Deep Archive	Coldline / Archive	Cold / Archive
Machine identity	IAM role / instance profile	Service account / workload identity	Managed identity
Time-limited link	Presigned URL	Signed URL	SAS token
New-file events	S3 Event Notifications → SQS/SNS/EventBridge	Pub/Sub notifications	Event Grid
SQL over files	Athena	BigQuery external tables	Synapse serverless SQL

Everyday commands

Task	Command
List with sizes	aws s3 ls s3://b/prefix/ --recursive --human-readable --summarize
Copy / sync	aws s3 cp f s3://b/k · aws s3 sync ./dir s3://b/prefix/
GCS equivalents	gcloud storage ls -l gs://b/p/ · gcloud storage cp · gcloud storage rsync -r
Object metadata	aws s3api head-object --bucket b --key k

Task	Command
Share temporarily	<code>aws s3 presign s3://b/k --expires-in 3600</code>
pandas read	<code>pd.read_parquet("s3://b/k.parquet", columns=[...])</code>
Spark: overwrite only touched partitions	<code>spark.conf.set("spark.sql.sources.partitionOverwriteMode", "dynamic")</code>

Layout rules of thumb: Hive-style `col=value/` partitions · date first · 128 MB–1 GB files · Parquet (or Iceberg/Delta) everywhere after Bronze · one bucket or prefix per layer, so IAM can differ per layer

Interview Questions

Q: How is object storage different from a filesystem, and why does it matter for data engineering? A: Object storage is a flat key → blob store: "folders" are just key prefixes, there's no atomic rename or directory move, and every operation is an HTTP request with latency and a cost. That means renames are copy + delete, listing millions of keys is slow, and the classic "write to a temp dir then rename" commit pattern isn't safe. Table formats like Iceberg and Delta exist largely to work around this, using atomic metadata commits instead of renames.

Q: What is the small file problem and how do you fix it? A: When a dataset is spread across thousands of tiny files, engines spend more time opening files, listing, and reading footers than reading data, and request costs go up. Causes include over-partitioning, streaming micro-batches, and high-parallelism writes. Fixes: partition by a coarser key, `coalesce/repartition` before writing, and compact regularly (Delta `OPTIMIZE`, Iceberg `rewrite_data_files`, or a scheduled compaction job). Aim for roughly 128 MB–1 GB per file.

Q: How would you design the storage layout for a new data lake? A: Separate zones: a landing/Bronze zone for raw data exactly as received (immutable, partitioned by ingestion date), Silver for cleaned and typed data (Parquet or a table format, partitioned by the business date), and Gold for modeled, business-ready tables. Use separate buckets or prefixes per zone so IAM can grant pipelines write access only to their own layer and BI tools read-only access to Gold. Add lifecycle rules, versioning on landing, encryption, Block Public Access, and a naming convention (`source/table/dt=YYYY-MM-DD/`).

Q: How do you give a Spark job running on AWS access to S3 securely? A: Attach an IAM role to the compute (EMR instance profile, an EKS service account through IRSA, or a Databricks instance profile / Unity Catalog storage credential) with least-privilege

permissions on specific bucket prefixes. No access keys in code or config. Add bucket policies that restrict access to the expected roles or VPC endpoints, and turn on encryption (SSE-KMS if you need key-level audit).

Q: How do you reduce cloud storage costs? A: Lifecycle policies that move old data to infrequent-access or archive tiers and expire temp data; Intelligent-Tiering when access patterns are unknown; compression and columnar formats (Parquet with ZSTD is often 5–10× smaller than CSV); compaction to reduce request counts; keeping compute in the same region to avoid transfer charges; and cleaning up old table-format snapshots and orphan files (`VACUUM` , `expire_snapshots`).

Q: A downstream team says yesterday's partition is missing rows. How do you investigate? A: Check whether the files landed (`aws s3 ls` for that partition: count and sizes), then whether the job that wrote them succeeded and how many rows it reported. Look for an `overwrite` that replaced the partition with partial data, a late-arriving upstream file, or a filter bug. With Delta or Iceberg, the table history and time travel show exactly which commit changed the partition, and let you compare row counts across versions.

Further Reading

- [Amazon S3 user guide](https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html) (https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html) — especially the performance guidelines and storage classes pages
 - [Google Cloud Storage documentation](https://cloud.google.com/storage/docs) (https://cloud.google.com/storage/docs)
 - [Azure Data Lake Storage Gen2 documentation](https://learn.microsoft.com/azure/storage/blobs/data-lake-storage-introduction) (https://learn.microsoft.com/azure/storage/blobs/data-lake-storage-introduction)
 - [boto3 S3 reference](https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html) (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html)
 - [fsspec](https://filesystem-spec.readthedocs.io/) (https://filesystem-spec.readthedocs.io/) — the layer that lets pandas, Dask, and DuckDB read `s3://` , `gs://` , and `abfs://` paths
-

Previous: [Git for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/) • **Next:** [Docker](https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/06-infrastructure/docker-reference/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026