

Cost Optimization for Data Platforms

Making data platform spend visible, attributable, and efficient — without sacrificing reliability or freshness.

Last reviewed 29 Sep 2026 ·

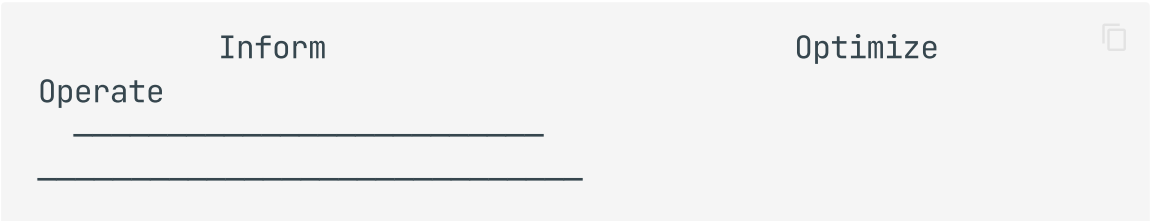
Prerequisites: [System Design](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/system-design/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/)

Related: [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [BigQuery](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/bigquery-reference/) · [Amazon Redshift](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/redshift-reference/) · [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) · [DuckDB & Polars](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/duckdb-polars/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

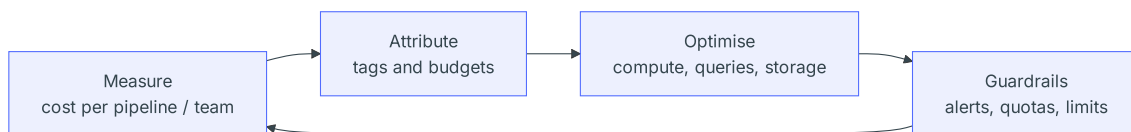
Challenge: Cloud data platforms bill for every query, every compute-second, every stored and transferred gigabyte. Costs grow quietly — an unfiltered dashboard refreshing every five minutes, a cluster that never shuts down, raw data kept forever — until a monthly bill triggers an emergency review. Without attribution, nobody knows which team, pipeline, or query is responsible.

Solution: FinOps applies engineering discipline to cloud spend in three repeating phases: **inform** (make costs visible and attributable), **optimize** (remove waste and right-size), and **operate** (budgets, guardrails, and cost-aware design as part of normal work). For data platforms, most savings come from reading less data, running compute only when needed, and matching freshness to real requirements.



tag and attribute spend	read less data (pruning,
budgets and alerts	
cost per pipeline / team	columns, incremental)
quotas and guardrails	
unit costs (per query,	run compute only when
needed	cost reviews in design and PRs
per table, per customer)	right-size, tier storage
commitments for steady load	

Relevance to data engineering: data engineers control the biggest cost levers — table layout, processing strategy, scheduling, and retention. Cost is a design requirement, like latency or correctness.



On this page

Basic - [Where the Money Goes](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#where-the-money-goes) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#where-the-money-goes) - [Unit Economics](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#unit-economics) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#unit-economics) - [Visibility and Attribution](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#visibility-and-attribution) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#visibility-and-attribution)

Intermediate - [Monitoring Spend by Platform](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#monitoring-spend-by-platform) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#monitoring-spend-by-platform) - [Compute Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#compute-optimization) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#compute-optimization) - [Query and Pipeline Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#query-and-pipeline-optimization) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#query-and-pipeline-optimization) - [Storage Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#storage-optimization) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#storage-optimization)

Advanced - [Data Transfer Costs](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#data-transfer-costs) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#data-transfer-costs) - [Commitments and Pricing Models](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#commitments-and-pricing-models) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#commitments-and-pricing-models) - [Guardrails and Budgets](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#guardrails-and-budgets) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#guardrails-and-budgets) - [Cost-Aware Architecture](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#cost-aware-architecture) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#cost-aware-architecture) - [LLM and AI Workload Costs](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#llm-and-ai-workload-costs) (https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#llm-and-ai-workload-costs)

Reference - Common Pitfalls (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#common-pitfalls>) - **Cheat Sheet** (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/cost-optimization/#further-reading>)

Where the Money Goes

Category	Examples	Typical drivers
Warehouse / query compute	Credits, slots, RPUs, DBUs	Bytes scanned, runtime, idle time, concurrency
Cluster compute	Spark, Flink, orchestrator workers	Instance size and count, uptime, retries
Storage	Object storage, warehouse storage, time travel, backups	Retention, duplication, file inefficiency
Data transfer	Cross-region, cross-cloud, internet egress	Architecture and location choices
Managed services and SaaS	Ingestion connectors (per row), observability, BI licences	Volume, seats
AI workloads	LLM API tokens, embeddings, GPUs	Token volume, model choice, caching

In most data platforms, **compute dominates**; storage is usually cheaper than expected and transfer is often a surprise.

Unit Economics

Absolute spend is hard to judge; **unit costs** tie spend to value and reveal trends.

Unit metric	Example	Useful for
Cost per pipeline run	\$4.10 per daily <code>orders</code> run	Spotting regressions after code changes

Unit metric	Example	Useful for
Cost per table per month	\$620/month to maintain <code>fact_sessions</code>	Deciding whether a table earns its cost
Cost per query / dashboard	\$0.35 per dashboard load	Prioritizing pre-aggregation
Cost per customer or tenant	\$0.12 per active customer per month	Pricing and margin analysis
Cost per TB processed	\$3.20 per TB transformed	Comparing engines and approaches

Track unit costs over time: if volume doubles and cost per unit stays flat, the platform scales well; if cost per unit rises, something is degrading.

Visibility and Attribution

- **Tag everything:** cloud resources, clusters, warehouses, and jobs carry `team`, `pipeline`, `environment`, and `cost_center` tags (enforce with IaC policies)
- **Label queries:** set query tags or labels from pipelines so warehouse spend maps to jobs (e.g. Snowflake `QUERY_TAG`, BigQuery job labels, Databricks custom tags)
- **Separate compute per workload:** distinct warehouses/workgroups/clusters for ingestion, transformation, BI, and data science make attribution trivial
- **Centralize billing data:** export cloud billing (e.g. AWS CUR, GCP billing export, Azure cost exports) and platform usage tables into the warehouse, and build a cost dashboard like any other data product
- **Show back before you charge back:** share per-team costs first; charging budgets comes later

Monitoring Spend by Platform

```
-- Snowflake: credits per warehouse, last 30 days
SELECT warehouse_name, SUM(credits_used) AS credits
FROM snowflake.account_usage.warehouse_metering_history
WHERE start_time >= DATEADD(day, -30, CURRENT_TIMESTAMP())
GROUP BY warehouse_name
ORDER BY credits DESC;
```

```

-- BigQuery: TiB billed per user, last 7 days
SELECT user_email, ROUND(SUM(total_bytes_billed) /
POW(1024, 4), 2) AS tib_billed
FROM `region-us`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE creation_time >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(),
INTERVAL 7 DAY)
GROUP BY user_email
ORDER BY tib_billed DESC;

-- Databricks: DBUs per SKU per day (system tables)
SELECT usage_date, sku_name, SUM(usage_quantity) AS dbus
FROM system.billing.usage
WHERE usage_date >= current_date() - INTERVAL 30 DAYS
GROUP BY usage_date, sku_name
ORDER BY usage_date, dbus DESC;

-- Redshift Serverless: compute usage per day
SELECT TRUNC(start_time) AS day, SUM(charged_seconds) /
3600.0 AS rpu_hours_charged
FROM sys_serverless_usage
WHERE start_time >= DATEADD(day, -30, GETDATE())
GROUP BY 1
ORDER BY 1;

```

What to look for: the top 10 most expensive queries and jobs (usually most of the spend) · compute running with no queries · costs that jumped after a deployment · dashboards refreshing far more often than anyone views them

Compute Optimization

Lever	Practice
Run only when needed	Auto-suspend/auto-terminate (e.g. 60 s for warehouses, 10–30 min for interactive clusters); schedule dev environments off outside working hours
Right-size	Match size to workload; bigger compute that finishes faster can cost the same or less — measure

Lever	Practice
Scale elastically	Autoscaling clusters, multi-cluster warehouses for concurrency, serverless for spiky workloads
Use cheaper capacity	Spot/preemptible instances for fault-tolerant batch workers; ARM-based instances where supported
Isolate workloads	Separate compute so one team's heavy job doesn't force everything onto a larger size
Right engine for the size	Small and medium data on a single-node engine (DuckDB, Polars) instead of a cluster
Job compute, not interactive	Scheduled jobs on job clusters or serverless jobs instead of always-on interactive clusters

Query and Pipeline Optimization

- **Read less:** partition and cluster by common filters; select only needed columns; filter early
- **Process incrementally:** transform only new or changed partitions instead of rebuilding full tables
- **Materialize deliberately:** views for cheap, rarely used logic; tables or incremental models for expensive, frequently read logic; pre-aggregates for dashboards
- **Match refresh frequency to use:** a dashboard viewed once a morning doesn't need 5-minute refreshes
- **Avoid rework:** reuse intermediate results; cache results where the platform supports it
- **Fix the top offenders first:** the most expensive few queries typically account for a large share of spend

```
-- Before: full rebuild every hour, scanning two years of events
CREATE OR REPLACE TABLE gold.daily_active_users AS
SELECT event_date, COUNT(DISTINCT user_id) AS dau FROM
silver.events GROUP BY event_date;

-- After: recompute only the last 3 days (covers late
events) and merge
MERGE INTO gold.daily_active_users t
```

```

USING (
  SELECT event_date, COUNT(DISTINCT user_id) AS dau
  FROM silver.events
  WHERE event_date >= CURRENT_DATE - 3
  GROUP BY event_date
) s
ON t.event_date = s.event_date
WHEN MATCHED THEN UPDATE SET dau = s.dau
WHEN NOT MATCHED THEN INSERT (event_date, dau) VALUES
(s.event_date, s.dau);

```

Storage Optimization

Lever	Practice
Columnar formats and compression	Parquet/ORC with ZSTD or Snappy instead of CSV/JSON after the raw layer
Lifecycle tiers	Move raw data to infrequent-access and archive tiers after 30–90 days; expire temporary data
Retention by classification	Keep raw data only as long as reprocessing or compliance requires
Table-format housekeeping	Compact small files; expire snapshots; remove orphan files; limit time travel on large staging tables
Remove duplicates	Delete redundant copies and abandoned tables — use lineage and access logs to find tables no one reads
Transient/staging tables	No fail-safe or long time travel for data that can be rebuilt

Data Transfer Costs

- Keep compute in the **same region** as the data
- Avoid repeatedly reading data across clouds or regions — replicate once and read locally, or process where the data lives
- Use **private endpoints** (e.g. VPC gateway endpoints for object storage) to avoid NAT gateway processing charges
- Compress data before transferring it; send changes, not full copies

- Watch BI tools and notebooks that pull large result sets out of the cloud
-

Commitments and Pricing Models

Workload shape	Pricing approach
Steady, predictable baseline	Commitments: reserved capacity, savings plans, capacity/slot commitments, pre-purchased credits
Spiky or unpredictable	On-demand or serverless, billed only while running
Mixed	Commit to the steady baseline; burst on demand

Commit only after usage has stabilized and been optimized — committing to waste locks it in.

Guardrails and Budgets

- **Budgets and alerts** per team and environment, alerting on forecasted overspend, not only actual spend
 - **Platform limits:** resource monitors and credit quotas (Snowflake), `maximum_bytes_billed` and custom quotas (BigQuery), usage limits and maximum capacity (Redshift Serverless), cluster policies with maximum size and auto-termination (Databricks)
 - **Policies as code:** enforce tags, instance types, and auto-termination through IaC and cluster policies
 - **Anomaly detection:** alert when daily spend deviates significantly from its trailing average
 - **Cost in code review:** estimate the cost impact of new pipelines and schedules before they ship
-

Cost-Aware Architecture

Decision	Cheaper option	When the expensive option is justified
Freshness	Daily or hourly batch	Decisions genuinely need minutes or seconds
Processing	Incremental	The logic can't be made incremental, or data is small
Engine	Single-node or warehouse SQL for small/medium data	Data volume or concurrency requires a cluster
Storage	Open formats on object storage with tiering	Low-latency serving requirements
Serving	Pre-aggregated tables	Truly ad hoc exploration over raw detail
Retention	Keep what reprocessing and compliance need	Contractual or regulatory retention

State the cost of each major component in design documents, and revisit it after launch against actual spend.

LLM and AI Workload Costs

- **Choose the smallest model that meets the quality bar** on your evaluation set
- **Cache** repeated prompt prefixes (prompt caching) and results for identical inputs
- **Batch APIs** for offline workloads — typically about half the price of synchronous calls
- **Trim tokens:** shorter prompts, fewer retrieved chunks, capped output length
- **Embed incrementally:** hash content and embed only new or changed text
- **Track cost per feature and per request** with token usage logged on every call

See [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) and [AI Observability](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-observability/).

Common Pitfalls

Pitfall	Symptom	Fix
No tagging or labels	Cost reports say "compute" with no owner	Mandatory tags via IaC; query labels from pipelines
Always-on interactive compute	Spend continues nights and weekends	Auto-suspend and auto-termination everywhere
Full refreshes of large tables	Costs grow linearly with history	Incremental processing with a lookback window
Real-time pipelines for daily consumers	Always-on streaming bills	Match freshness to decisions; micro-batch or batch
Dashboards refreshing constantly	Warehouse busy all day	Refresh schedules aligned with usage; pre-aggregation; caching
Keeping everything forever	Storage and time-travel bills creep up	Retention by classification; lifecycle rules; snapshot expiration
Cross-region reads	Large, unexplained data-transfer charges	Co-locate compute and data
Committing before optimizing	Paying for capacity you don't need	Optimize first, then commit to the stable baseline
Optimizing tiny costs	Engineering time exceeds savings	Rank by spend; fix the top offenders first

Cheat Sheet

The big five levers: read less data · run compute only when needed · process incrementally · match freshness to decisions · tier and expire storage

First-week audit checklist

1. List the top 10 queries, jobs, and dashboards by cost
2. Find compute with no auto-suspend or auto-termination
3. Find tables rebuilt in full on every run
4. Find tables with no readers in 90 days
5. Check storage lifecycle rules and time-travel/snapshot retention
6. Check data-transfer charges by region pair
7. Confirm budgets and alerts exist per team and environment

Platform guardrails: Snowflake resource monitors · BigQuery `maximum_bytes_billed` and quotas · Redshift Serverless usage limits · Databricks cluster policies and budgets · cloud budgets with forecast alerts

Interview Questions

Q: A data platform's cloud bill doubled in a month. How do you investigate? A: Break the increase down by service, then by resource and tag, to find where it came from — compute, storage, or transfer. For warehouse compute, rank queries and jobs by cost in the platform's usage views and compare with the previous month. Typical causes are a new or changed pipeline doing full refreshes, a dashboard refreshing too often, compute left running, a backfill, or a new cross-region data flow. Fix the root cause, then add a guardrail (budget alert, quota, auto-suspend) so it can't recur silently.

Q: How do you make data platform costs attributable to teams? A: Tag all cloud resources and jobs with team and pipeline, label warehouse queries from pipelines, and give each major workload its own compute (warehouse, workgroup, or cluster). Load billing exports and platform usage tables into the warehouse, join them on tags, and publish a cost dashboard by team and pipeline — starting with showback before any chargeback.

Q: What are the most effective ways to reduce warehouse costs? A: Read less data (partitioning, clustering, column selection), process incrementally instead of full rebuilds, pre-aggregate for dashboards and align refresh schedules with actual usage, auto-suspend idle compute and right-size it, separate workloads, and set guardrails like quotas and resource monitors. Usually a small number of queries and pipelines account for most of the spend, so start there.

Q: When is it worth paying for streaming or real-time processing? A: When the business decision it supports genuinely needs low latency and the value of acting faster exceeds the cost of always-on compute and operational complexity — fraud prevention, operational alerting, customer-facing features. For reports and analytics read periodically, hourly or daily batch delivers the same value at a fraction of the cost.

Further Reading

- [FinOps Foundation — FinOps framework](https://www.finops.org/framework/) (https://www.finops.org/framework/)
- [AWS Well-Architected: Cost Optimization Pillar](https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html) (https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html)
- [Google Cloud: cost optimization best practices for BigQuery](https://cloud.google.com/bigquery/docs/best-practices-costs) (https://cloud.google.com/bigquery/docs/best-practices-costs)

- [Snowflake: managing cost](https://docs.snowflake.com/en/guides-overview-cost) (<https://docs.snowflake.com/en/guides-overview-cost>)
 - [Databricks: monitor costs using system tables](https://docs.databricks.com/en/admin/system-tables/billing.html) (<https://docs.databricks.com/en/admin/system-tables/billing.html>)
 - *Cloud FinOps, 2nd Edition* — J.R. Storment & Mike Fuller (O'Reilly)
-

Previous: [Choosing a Stack](https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/) (<https://sarangambekar1997.github.io/data-engineering-handbook/08-architecture/choosing-a-stack/>) • **Next:** [Prompt Engineering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) (<https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/>) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026