Monthly Monitoring of Databricks Jobs: Tracking Time, Memory Utilization, and Costs for Better Efficiency

Anurag Sharma 5 min read · Jan 12, 2026

In the fast-paced world of data engineering and analytics, scaling operations often comes with a hidden challenge: skyrocketing cloud costs. As our data volumes grew exponentially, so did our concerns about the associated expenses in our Databricks environment. We noticed that unchecked job runs were contributing to unnecessary resource consumption, leading to higher bills without proportional value. To address this, we implemented a proactive strategy: monthly checks on key performance metrics for our Databricks jobs, focusing on execution time, memory, CPU utilization, and overall costs. This not only helped us optimize resources but also fostered a culture of cost-awareness across the team.

In this blog post, I will dive into why we started this initiative, how we built a simple reporting job to automate these checks, and the insights we have gained along the way. Whether you are managing a small data pipeline or a large-scale analytics workload, these practices can help you rein in costs while maintaining performance.

The Problem: Rising Data Volumes and Cloud Costs

Our journey began with a familiar pain point for many data teams. As our datasets expanded, driven by more sources, higher ingestion rates, and complex transformations, our Databricks usage followed suit. Jobs that once ran efficiently started taking longer, consuming more memory and CPU, and racking up Databricks Units (DBUs), the core metric for billing in Databricks.

DBUs represent a blend of compute resources (like VM hours) and Databricks' managed services, priced based on your cloud provider (AWS, Azure, or GCP) and workspace type. Without regular oversight, small inefficiencies compound: a job with underutilized clusters might waste DBUs on idle time, or one with memory leaks could balloon costs through spills to disk or failed runs.

We realized that reactive cost management, waiting for the end-of-month bill, wasn't sustainable. Instead, we needed a "tight check" system: automated, monthly reports on job metrics to spot trends early, optimize configurations, and prevent bill shocks.

Our Solution: Building a Reporting Job with Databricks Metadata

To tackle this, we created a dedicated Databricks job that runs monthly (scheduled via workflows) to extract, analyze, and report on key metrics. The beauty of Databricks is its rich metadata ecosystem, accessible through system tables in Unity Catalog. These tables store operational data about jobs, clusters, and billing, making it straightforward to query without external tools.

Here is how we structured it:

Extracting Data from System Tables

Databricks provides a schema-like system.billing and system.compute for this purpose. We query these tables using SQL in a notebook or job task. Key tables we rely on include:

- **system.billing.usage**: This captures DBU consumption per job run, workspace, and user. It contains details like usage type (e.g., Jobs Compute, All-Purpose Compute), start/end times, and DBUs consumed. We convert DBUs to approximate dollars using our pricing tier (e.g., on AWS, Jobs Compute might cost around $0.07–$0.20 per DBU, depending on the plan — always check your specific rates).
- **system.compute.clusters** and **system.compute.cluster_events**: These provide cluster-level metrics, including CPU and memory utilization over

time. For each job run, we pull averages for CPU usage (e.g., percentage utilized vs. idle), memory allocation (e.g., peak usage, spills), and runtime durations.

- **Job Run History (via Jobs API or system tables)**: For execution time, we query job metadata to get start/end timestamps, calculating total runtime in minutes or hours. This helps identify jobs that are trending longer due to data growth or inefficiencies.

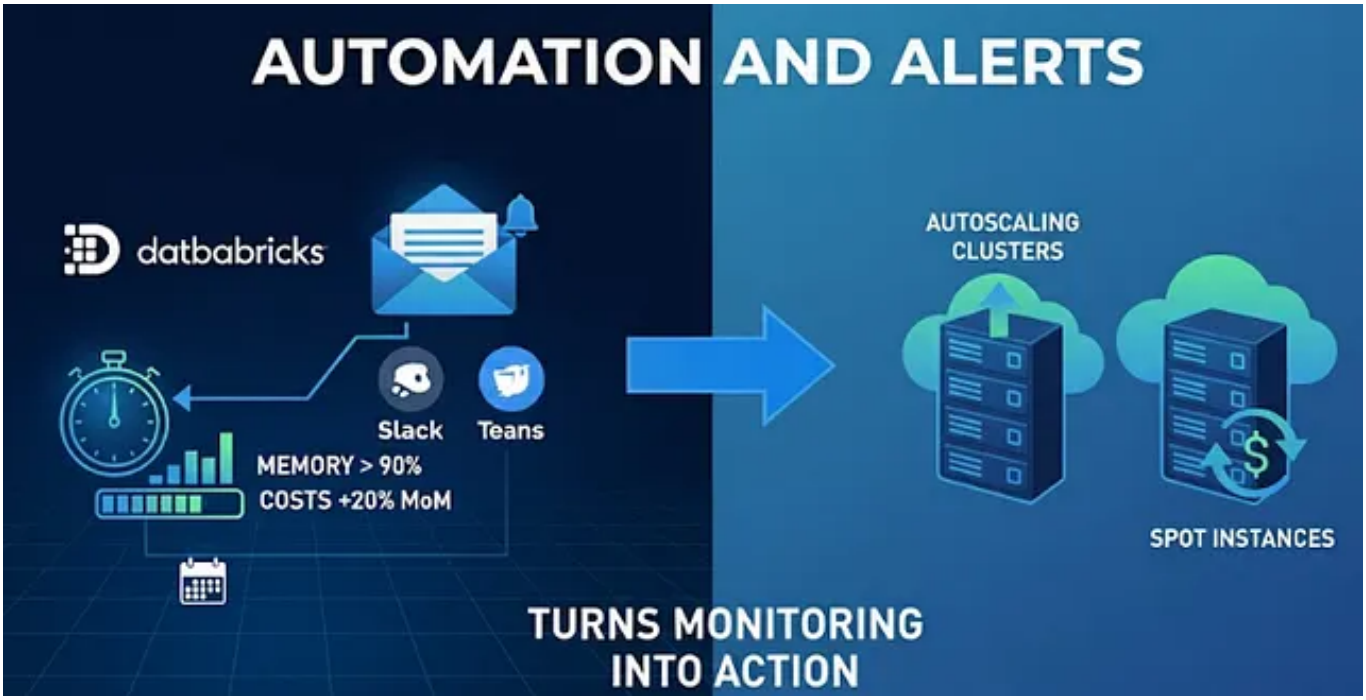
```
-- Query for monthly DBU usage and costs
SELECT
    workspace_id,
    sku,  -- JOBS_COMPUTE
    usage_type,
    SUM(dbus) AS total_dbus,
    -- Approx cost: Multiply by your DBU rate (0.10 USD/DBU)
    SUM(dbus) * 0.10 AS estimated_cost_usd,
    date_trunc('month', usage_date) AS month
FROM system.billing.usage
WHERE usage_type = 'JOBS_COMPUTE'
AND usage_date >= date_sub(current_date(), 30)  -- Last month
GROUP BY workspace_id, sku, usage_type, month;

-- Query for cluster metrics (CPU/Memory utilization)
SELECT
    cluster_id,
    AVG(cpu_utilization_percent) AS avg_cpu_util,
    AVG(memory_utilization_percent) AS avg_memory_util,
    AVG(runtime_seconds / 3600) AS avg_runtime_hours
FROM system.compute.cluster_events
WHERE event_time >= date_sub(current_date(), 30)
GROUP BY cluster_id;
```

Key Metrics We Monitor

- **Execution Time:** We track average and maximum run times per job. If a job's time increases month-over-month, it signals issues like data skew, inefficient Spark configurations, or the need for larger clusters. Goal: Keep times stable or reduce them through optimizations like partitioning or caching.
- **Memory and CPU Utilization:** High memory usage (e.g., >80%) can lead to out-of-memory errors or expensive spills, while low utilization (<50%) means overprovisioned clusters wasting DBUs. CPU metrics help spot bottlenecks — e.g., if CPU is pegged at 100% but memory is low, we might need more cores. We aim for balanced utilization: 60–80% for efficiency.
- **Costs (DBU to Dollars):** By aggregating DBUs and applying our rate, we get a tentative monthly cost breakdown per job or cluster. This isn't the final bill (which includes cloud VM costs), but it's a close proxy. We flag jobs consuming >10% of total DBUs for review.

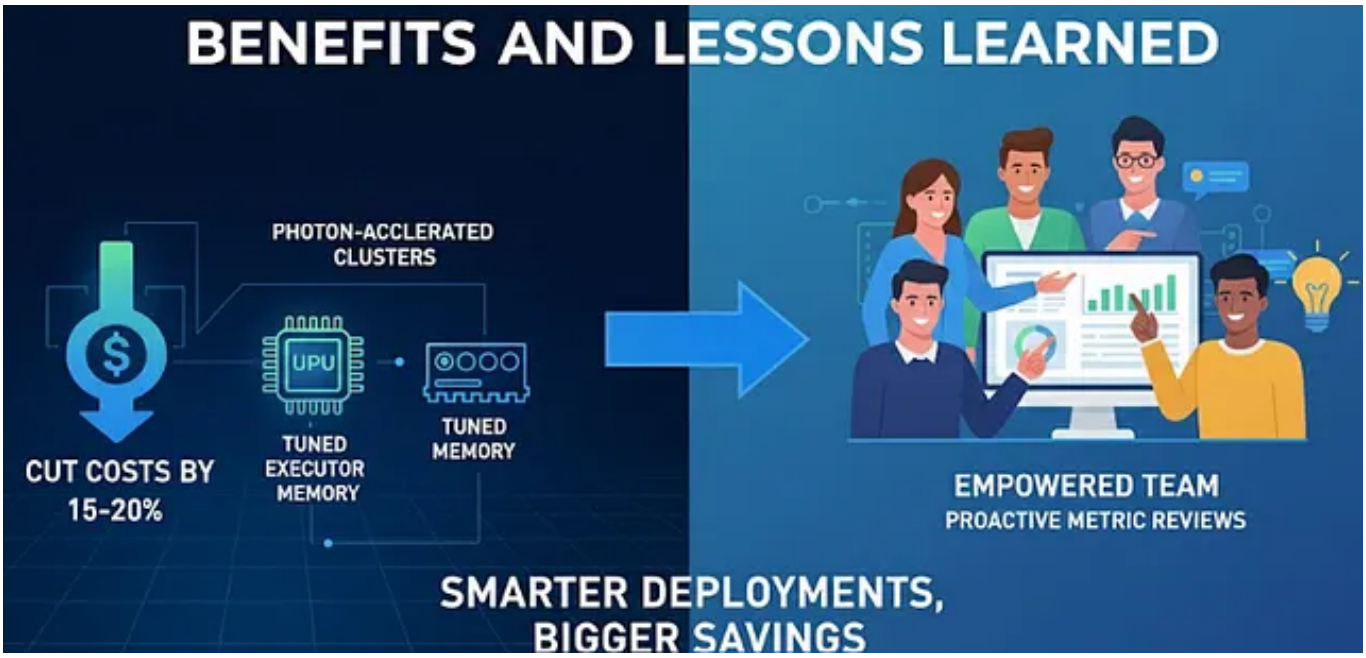
Automation and Alerts

The reporting job emails summaries (using Databricks' notification features) or posts to Slack/Teams. We set thresholds: e.g., alert if memory utilization exceeds 90% or costs rise 20% MoM. This turns monitoring into action, e.g., rightsizing clusters with autoscaling or switching to spot instances.



Benefits and Lessons Learned

Since implementing this, we've cut our Databricks costs by 15–20% through targeted optimizations, like tuning executor memory or using photon-accelerated clusters for CPU-intensive jobs. More importantly, it empowers our team: engineers now proactively review metrics before deploying changes.



Key takeaways:

- Start small: Begin with one or two jobs, then scale.
- Use system tables wisely; they are free to query and update in near real-time.
- Combine with external monitoring: Tools like Datadog can integrate for deeper insights, but system tables cover the basics.
- Review pricing regularly: DBU rates vary, so factor in your contract.

If you're facing similar cost pressures in Databricks, setting up monthly checks like this is a game-changer. It's not just about saving money — it's about building scalable, efficient data operations.

Databricks Spark Data Engineering Automation Optimization