

Monitor, troubleshoot and optimize workloads in Azure Databricks

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/1-introduction>

Introduction

Completed



Running data workloads in Azure Databricks at scale requires more than setting up clusters and writing pipelines—it demands continuous attention to resource consumption, job health, and performance optimization. When compute costs climb unexpectedly, jobs fail without clear explanations, or queries that once ran quickly now take hours, you need systematic approaches to identify problems and apply targeted fixes.

Azure Databricks provides comprehensive tools for monitoring and troubleshooting across all compute types. **Cluster metrics** reveal CPU, memory, and network utilization patterns that help you right-size resources. **The Spark UI** exposes job execution details, stage-level metrics, and task distributions that pinpoint bottlenecks. **System tables** enable you to query billing data and usage patterns directly, supporting cost attribution and chargeback processes.

Performance issues often trace back to common patterns: **data skew** that overloads a few tasks while others sit idle, **memory spill** that forces expensive disk operations, **excessive shuffle** that moves data unnecessarily across the network, or **caching problems** that cause repeated reads from remote storage. Understanding how to investigate these issues using the DAG visualization, Spark UI metrics, and query profiles helps you restore performance efficiently.

For enterprise-scale environments, **log streaming to Azure Log Analytics** centralizes diagnostic and audit data from across your Azure Databricks workspaces. This integration enables powerful KQL queries for troubleshooting, proactive alerting for critical events, and comprehensive visibility into platform operations.

Throughout this module, you explore cluster consumption monitoring, Lakeflow Jobs troubleshooting, Spark job diagnostics, performance optimization techniques, and log streaming configuration to build the skills needed to keep your Azure Databricks workloads running reliably and cost-effectively.

2. Monitor and manage cluster consumption

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/2-monitor-manage-cluster-consumption>

Monitor and manage cluster consumption

Completed

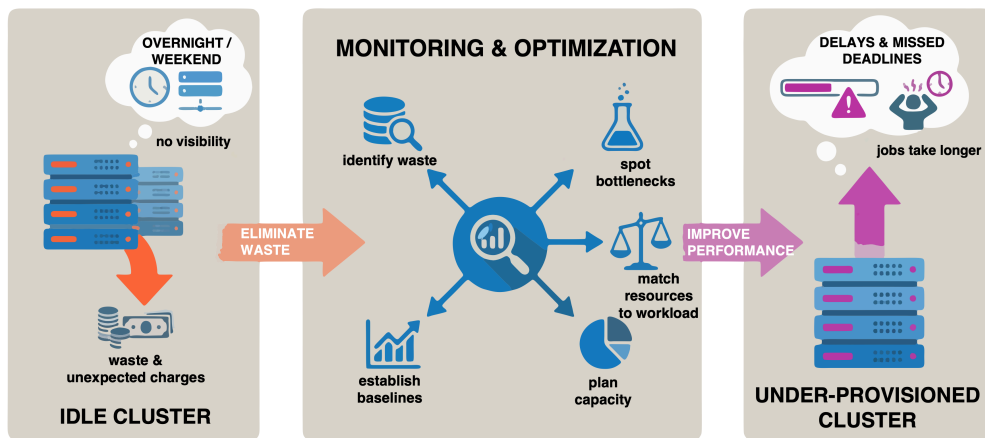


When clusters run longer than needed or consume more resources than necessary, costs can escalate quickly. As a data engineer, you're responsible for **monitoring cluster usage** and taking action to balance **performance requirements** with **cost efficiency**.

Understand the impact of cluster consumption

Unmonitored clusters present real financial risks. A single **idle cluster** running overnight or over a weekend can generate significant unexpected charges. These costs compound when multiple teams work across shared workspaces without visibility into resource usage patterns.

At the same time, **under-provisioned clusters** create their own problems. Jobs take longer to complete, users experience delays, and critical workflows miss their deadlines. The goal isn't simply to minimize costs—it's to **match resources precisely to workload requirements**.



Monitoring cluster consumption helps you **identify waste** before it impacts budgets and **spot performance bottlenecks** before they affect business operations. Regular monitoring also establishes **baselines** that help you plan capacity and justify infrastructure decisions.

Monitor compute metrics

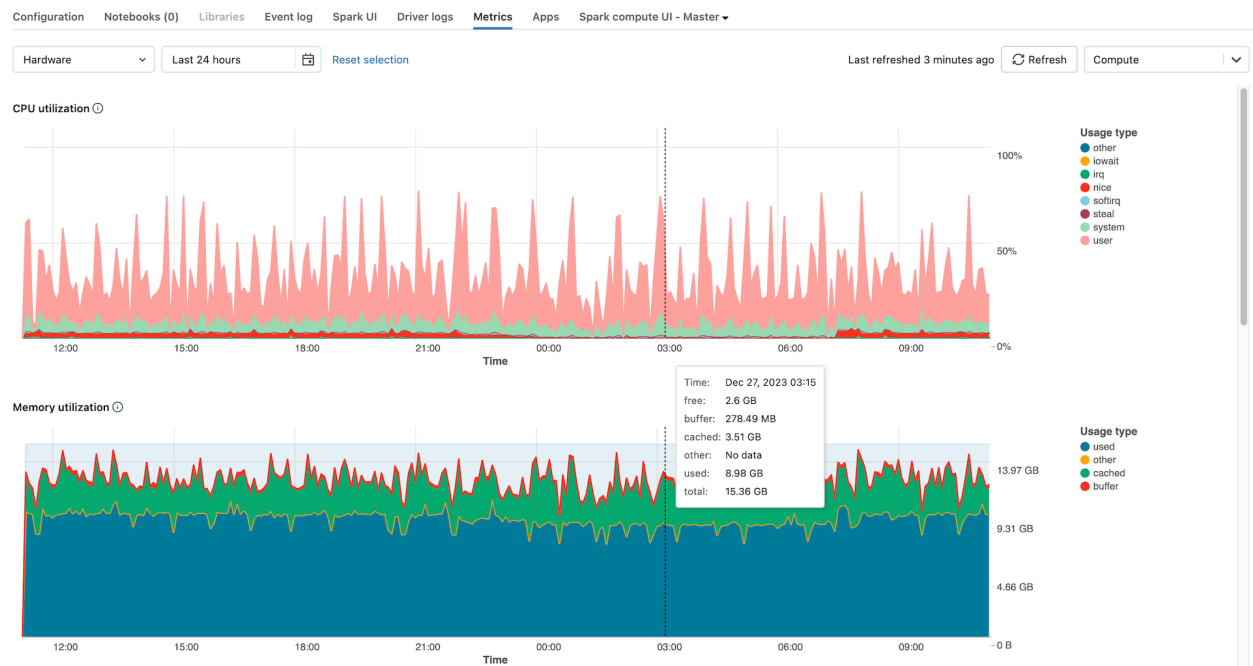
Azure Databricks provides a **native metrics interface** for all-purpose and job compute resources. You access these metrics by selecting **Compute** in the sidebar, choosing your compute resource, and selecting the **Metrics** tab.

The metrics interface displays three categories of data:

Hardware metrics show how physical resources are used. **CPU utilization** reveals whether your cluster is working hard or sitting idle. **Memory utilization** indicates whether you need to scale up or can reduce instance sizes. **Network throughput** helps identify data transfer bottlenecks.

Spark metrics focus on job execution patterns. **Active tasks** show current parallelism levels. Failed and completed task counts reveal job health. **Shuffle read and write metrics** expose data movement between stages that can indicate performance issues.

GPU metrics (available on Databricks Runtime ML 13.3 and later) track specialized compute utilization when running machine learning workloads.



You can filter metrics by time range using the date picker, viewing data from the past 30 days. Select individual nodes from the Compute dropdown to investigate specific worker performance, or view aggregated metrics across all nodes to understand overall cluster behavior.

Note

Serverless compute for notebooks and jobs uses query insights instead of the metrics UI. The monitoring approach differs, but the goal remains the same: understanding resource consumption.

Monitor SQL warehouse performance

SQL warehouses have their own monitoring interface optimized for query analytics. Select a SQL warehouse and then select the **Monitoring** tab to view performance data.



Live statistics at the top of the page show warehouse status, running queries, queued queries, and current cluster count. These metrics update in real-time and help you quickly assess whether the warehouse is keeping up with demand.

The **peak query count** chart displays the maximum number of concurrent queries—both running and queued—during your selected time frame. Spikes in this chart often indicate periods where the warehouse struggled to keep up with demand.

The **running clusters** chart shows how many clusters were allocated over time. If you've enabled **autoscaling**, you'll see this number fluctuate based on query load. Clusters that remain at maximum capacity for extended periods suggest you might need to increase the maximum cluster count.

The **query history** table lists individual queries with their start time, duration, and the user who ran them. Use filters to find **long-running queries** or identify users whose queries consume disproportionate resources.

Configure auto-termination and autoscaling

Monitoring reveals patterns; configuration changes act on those patterns. Two key settings help you manage cluster consumption automatically.

Auto-termination shuts down idle clusters after a specified period of inactivity. For development environments, **30-60 minutes** is typically appropriate. The cluster terminates when no commands have run for the specified duration, preventing costs from accumulating overnight or over weekends.

☒ Use Photon Acceleration ⓘ

Worker type ⓘ		Min workers	Max workers	Current	
Standard_DC4as_v5	16 GB Memory, 4 Cores	2	8	2	☒ Spot instances ⓘ

Driver type

Standard_DC4as_v5 16 GB Memory, 4 Cores

☒ Enable autoscaling ⓘ

☒ Terminate after 120 minutes of inactivity ⓘ

To configure auto-termination, enable the setting during cluster creation or edit an existing cluster. Enter the number of minutes of inactivity before termination. Keep in mind that a cluster is considered inactive only when all commands—including Spark jobs, Structured Streaming, and JDBC calls—have finished executing.

Autoscaling dynamically adjusts the number of worker nodes based on workload demand. Configure **minimum and maximum node counts** based on your workload analysis. The cluster adds workers during intensive processing and removes them during lighter periods, reducing costs by **20-40%** compared to fixed-size clusters.

Important

Auto-termination doesn't monitor DStreams activity. If you run DStream workloads, either disable auto-termination or migrate to Structured Streaming.

Track costs with budgets and system tables

Beyond real-time monitoring, you need visibility into actual spending. Azure Databricks provides several tools for cost tracking.

Budgets let you set financial targets and track spending across your account. Configure **email notifications** when spending approaches or exceeds your budget limits. You can apply filters to track spending by team, project, or workspace.

System tables, specifically `system.billing.usage`, provide detailed usage data you can query directly. Join this table with `compute.clusters` to identify which cluster owners consume the most **Databricks Units (DBUs)**. Use **custom tags** to attribute costs to specific business units or projects.

Tags propagate from clusters and workspaces to billing records, enabling accurate **chargeback**. Apply tags consistently from the start—you can't add tags retroactively to historical usage. Common tags include business unit, project, and environment (development, staging, production).

Note

You need **account admin permissions** to create and manage budgets. Workspace admins can view budgets but typically cannot create them.

The following example queries help you analyze usage patterns:

Daily DBU consumption trend - See how your compute usage changes day by day. This helps you spot unusual spikes or identify patterns like higher usage on certain weekdays.

```
SELECT
  usage_date,
  sum(usage_quantity) as DBUs_Consumed
FROM system.billing.usage
WHERE sku_name = "STANDARD_ALL_PURPOSE_COMPUTE"
GROUP BY usage_date
ORDER BY usage_date ASC
```

Which jobs consumed the most DBUs - Find out which scheduled jobs are your biggest cost drivers. Use this to prioritize optimization efforts on the most expensive workloads.

```
SELECT
    usage_metadata.job_id as job_id,
    sum(usage_quantity) as total_dbus
FROM system.billing.usage
WHERE usage_metadata.job_id IS NOT NULL
GROUP BY job_id
ORDER BY total_dbus DESC
```

Attribute costs to cluster owners - Identify which team members or service accounts are responsible for the most resource consumption. This supports chargeback discussions and helps teams understand their footprint.

```
SELECT
    c.owned_by,
    sum(u.usage_quantity) as total_dbus
FROM system.billing.usage u
JOIN system.compute.clusters c
    ON u.usage_metadata.cluster_id = c.cluster_id
WHERE u.usage_metadata.cluster_id IS NOT NULL
GROUP BY c.owned_by
ORDER BY total_dbus DESC
```

Usage by custom tag - Break down costs by project, department, or any other tag you've applied to your resources. This is essential for accurate cost allocation across business units.

```
SELECT
    sku_name,
    usage_unit,
    sum(usage_quantity) as total_usage
FROM system.billing.usage
WHERE custom_tags['project'] = 'sales-analytics'
GROUP BY sku_name, usage_unit
```

Reviewing monthly cost reports helps you catch anomalies early. Investigate unexpected increases before they become significant budget problems.

3. Troubleshoot and repair Lakeflow Jobs

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/3-troubleshoot-repair-lakeflow-jobs>

Troubleshoot and repair Lakeflow Jobs

Completed



When a Lakeflow Job fails, you must quickly identify the problem, understand the root cause, and take corrective action. Navigating failures efficiently keeps your pipelines running and your downstream consumers satisfied.

Understand job failure states

Before you can fix a problem, you need to recognize the different ways a job run can end unsuccessfully. Azure Databricks determines job run status based on the outcome of *leaf tasks*—tasks with no downstream dependencies.

A job run can result in several states:

- **Failed:** One or more leaf tasks didn't complete successfully.
- **Succeeded with failures:** Some tasks failed, but all leaf tasks completed successfully.
- **Skipped:** The job didn't run because it exceeded maximum concurrent runs for the job or workspace.
- **Timed Out:** The job exceeded its configured maximum duration.
- **Canceled:** A user or automated process stopped the run.

Understanding these states helps you prioritize your response. A failed leaf task requires immediate attention, while a skipped run might indicate a scheduling conflict that resolves itself.

Identify the cause of failure

The Lakeflow Jobs UI provides tools to pinpoint exactly where and why a job failed. The matrix view in the **Runs** tab displays a visual history of runs, with each cell representing a task's status across multiple runs.

To investigate a failed job:

1. Select **Jobs & Pipelines** in the sidebar.
2. Select the job name to open its details.
3. In the **Runs** tab, hover over a failed task (shown in red) to see metadata including start time, end time, status, duration, and error messages.
4. Select the failed task to open the **Task run details** page with complete output and logs.

Jobs & Pipelines >

Job name ☆ Send feedback

Run now

Runs Tasks

Runs

Run total duration

51s

26s

clean_customers

Tasks

Job name

Job ID 674307446072248

Job run ID 117523655278048

Launched Manually

Started Dec 17, 2025 at 09:56 AM

Ended Dec 17, 2025 at 09:57 AM

Duration 52s

Execution time 48s

Queue duration -

Status Failed

Performance Enabled

optimization

Message Task clean_customers failed with message: Workload failed, see run output for details

Next

Dec 17

runs

Start time	Run ID	Launched	Duration	Spark	Status	Error code	Run parameters
Dec 17, 2025 at 09:56 ...	11752365527...	Manually	52s	Logs	Failed	RunExecutionError	

The matrix view helps you identify patterns. If the same task fails repeatedly, the issue likely relates to that task's code or configuration. If failures appear random across different tasks, you might have a cluster or resource problem.

Tip

Use the **Diagnose Error** button in the Jobs UI to get help from Databricks Assistant. This feature analyzes error messages and provides suggestions for common issues.

Fix common failure causes

Once you identify the failing task, you can address the underlying problem. Common causes and their solutions include:

Configuration issues: If the failure relates to task settings, select **Edit task** to modify the configuration. You can update notebook paths, change parameters, or adjust timeout values.

Insufficient compute resources: Resource-related failures might require cluster changes. You can modify the cluster configuration to add workers, change instance types, or switch to a shared all-purpose cluster. If your workspace has resource quotas, you might need an administrator to increase limits.

Maximum concurrent runs exceeded: When jobs queue up behind other runs, you have two options. Wait for existing runs to complete, or increase the **Maximum concurrent runs** setting in the job configuration.

External dependencies: Sometimes failures originate outside your job—an unavailable data source, a network issue, or an upstream process that hasn't completed. After the external issue resolves, you

can use the repair feature to re-run the job without starting from scratch.

Repair failed job runs

The repair feature allows you to re-run only the unsuccessful tasks and their dependents, saving time and compute resources compared to restarting the entire job.

To repair a failed run:

1. Select the failed run in the **Start time** column or from the matrix view.
2. Select **Repair run** to open the repair dialog.
3. Review the list of tasks that will re-run—this includes all failed tasks plus any dependent tasks.
4. Optionally, modify task parameters in the dialog. These values override the original settings for this repair run only.
5. Select **Repair run** to start.

The screenshot displays the Databricks Jobs & Pipelines interface. At the top, the breadcrumb navigation shows 'Jobs & Pipelines > Job name > Run 117523655278048 >'. Below this, the job name 'clean_customers run' is shown, along with a dropdown menu set to 'Retry: 1st (latest) · Failed'. To the right of the job name, there are two buttons: 'Edit task' and 'Repair run', with the 'Repair run' button highlighted by a red box. The main content area is divided into two panels. The left panel, titled 'Output', shows a detailed error message: '[SCHEMA_NOT_FOUND] The schema `ms\_learn.pipeline\_lab` cannot be found. Verify the spelling and correctness of the schema and catalog. If you did not qualify the name with a catalog, verify the current_schema() output, or qualify the name with the correct catalog. To tolerate the error on drop use DROP SCHEMA IF EXISTS. SQLSTATE: 42704'. Below the error message, there is a 'Diagnose error' button, also highlighted by a red box, and a 'Highlight error' button. The right panel, titled 'Details', provides metadata about the job run, including Job ID, Job run ID, Task run ID, Run as (System Administrator), Launched (By retry scheduler), Started (Dec 17, 2025 at 09:57 AM), Ended (Dec 17, 2025 at 09:57 AM), Duration (14s), Status (Failed), Lineage (No lineage information for this job), and Performance optimization (Enabled). At the bottom of the 'Details' panel, there are buttons for 'View run events' and 'View run libraries'. Below the 'Details' panel, there is a 'Notebook' section showing the path to the notebook: '/Workspace/Users/admin@mngenvmcap801981.onmicrosoft.com/databricks-labs/instructions/design-and-implement-data-pipelines/02_clean_customers'. At the bottom of the interface, there is a 'Compute' section.

After the repair completes, the matrix view adds a new column showing the repaired run results. Tasks that were red (failed) should now appear green (successful).

Note

Repair is available only for jobs with two or more tasks. If tasks share a job cluster, the repair creates a new cluster instance (for example, `my_job_cluster_v1`) so you can compare settings between the original and repaired runs.

Stop and restart job runs

Sometimes you need to halt a running job or restart one that's stuck. The Jobs UI provides controls for both actions.

To stop an active run: From the **Runs** tab, active runs display a stop button. Select this button to cancel the specific run. You can also select **Cancel runs** or **Cancel all queued runs** from the dropdown menu to stop multiple runs at once.

To restart continuous jobs: Continuous jobs that fail repeatedly enter an exponential backoff state, where Azure Databricks waits progressively longer between retry attempts. The **Job details** panel shows the number of consecutive failures and the time until the next retry. Select **Restart run** to cancel the active run, reset the retry period, and immediately start a new run.

Jobs & Pipelines

▼ Create new

 **Ingestion pipeline**
Ingest data from apps, databases and files

 **ETL pipeline**
Build ETL pipelines using SQL and Python

 **Job**
Orchestrate notebooks, pipelines, queries and more

Jobs & pipelines **Job runs** [Send feedback](#)

Job ▼ Run as ▼ Start: 12/15/2025, 11:00 AM ● End: 12/17/2025, 11:00 AM ● Run status ▼ Error code ▼ [Create job](#)

1 ———— Top 5 error codes (1 error)
1 ———— RunExecutionError 1

0 ————

15 Dec, 12 PM 16 Dec, 12 AM 16 Dec, 12 PM 17 Dec, 12 AM

Failed Skipped Succeeded

Start time	Job	Run as	Launched	Duration	Status	Error code	Run parameters	Cancel
Dec 17, 2025 at 10:20 AM	Job name	System Admini...	Manually	33s	Running			
Dec 17, 2025 at 09:56 AM	Job name	System Admini...	Manually	52s	Failed	RunExecutionError		

< Previous Next >

Use the stop function when a job consumes excessive resources, processes incorrect data, or needs immediate intervention. Use restart when you've fixed an underlying issue and want to bypass the exponential backoff waiting period.

Monitor job health proactively

Reactive troubleshooting works, but proactive monitoring prevents many failures from impacting your workflows. Consider these practices:

- **Configure notifications:** Set up email, Slack, or webhook alerts for job failures so you learn about issues immediately.
- **Review run history regularly:** The matrix view reveals patterns over time. Frequent failures in specific tasks indicate areas needing attention.
- **Use system tables:** Query the `system.lakeflow` schema to analyze job performance across your account and identify trends before they become critical issues.

Combining proactive monitoring with efficient troubleshooting skills helps you maintain reliable data pipelines that your organization can depend on.

4. Troubleshoot Spark jobs and notebooks

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/4-troubleshoot-repair-spark-jobs-notebooks>

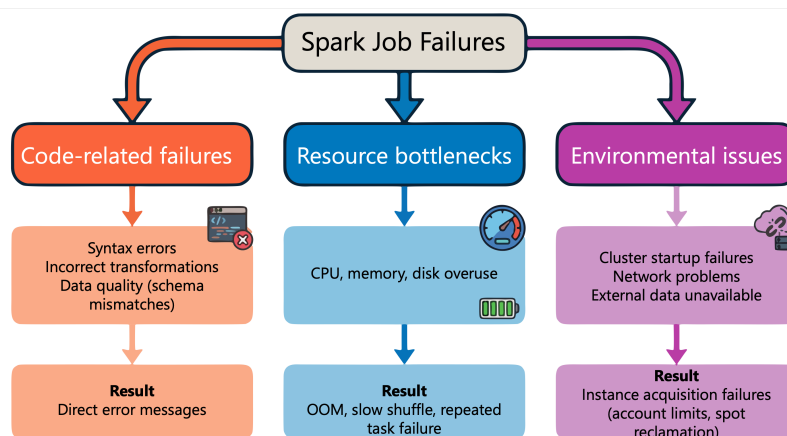
Troubleshoot Spark jobs and notebooks

Completed

When a Spark job fails or runs slower than expected, your ability to quickly diagnose and resolve the issue directly impacts business operations. As a data engineer, you need a systematic approach to investigate problems, identify root causes, and take corrective action.

Understand common causes of Spark job failures

Spark jobs can fail for various reasons, and understanding these patterns helps you focus your investigation. The most frequent causes fall into three categories: code errors, resource constraints, and environmental issues.



Code-related failures include syntax errors in notebooks, incorrect transformations, or data quality issues like schema mismatches. These failures typically produce error messages that point directly to the problematic code.

Resource bottlenecks occur when jobs consume more CPU, memory, or disk than available. You might see out-of-memory (OOM) errors, slow shuffle operations, or tasks that fail repeatedly. These issues often require adjusting cluster configuration or optimizing your code.

Environmental issues involve cluster startup failures, network problems, or external data source unavailability. A cluster might fail to acquire cloud instances due to account limits or spot instance reclamation.

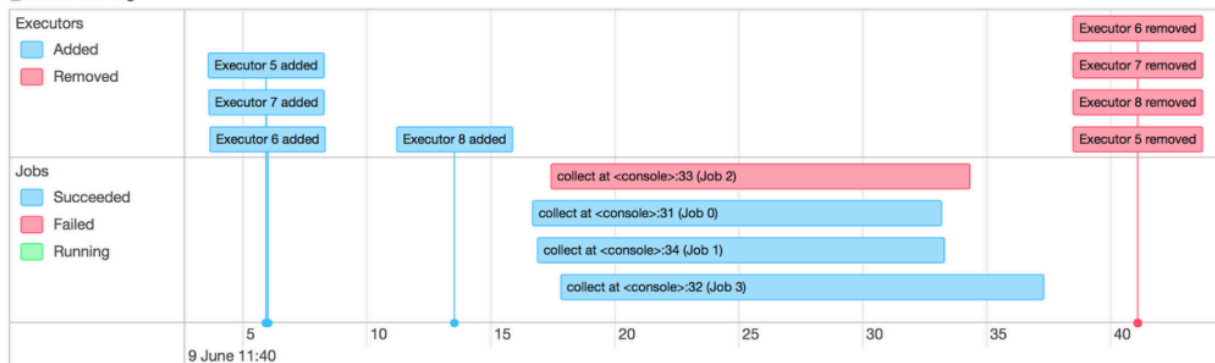
Use the Spark UI to diagnose issues

The Spark UI provides detailed visibility into job execution and is your primary diagnostic tool. To access it, navigate to your cluster's page and select the **Spark UI** tab.

Spark Jobs (?)

Total Uptime: 2.2 min
Scheduling Mode: FIFO
Completed Jobs: 3
Failed Jobs: 1

▼ Event Timeline
✓ Enable zooming



Start your investigation with the **Jobs Timeline**, which shows the execution sequence of all Spark jobs. Look for three key patterns:

Failing jobs appear with a red status indicator. Select any failed job to view the failed stage and specific failure reason. The error description often contains links to more detailed information about task-level failures.

Long-running jobs dominate the timeline with extended execution times. A single job taking significantly longer than others is a prime target for investigation. Select the job to examine its stages and identify bottlenecks.

Gaps in execution represent periods where no Spark work occurred. Short gaps are normal as the driver coordinates tasks, but gaps longer than a minute in the middle of a pipeline suggest the driver might be overloaded or a cluster malfunction occurred.

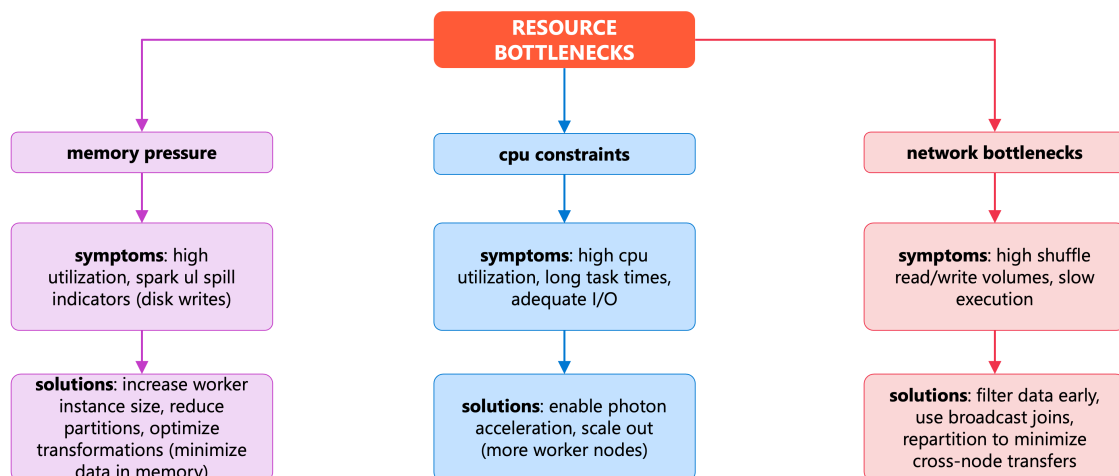
After identifying a problematic job, drill into its longest stage to examine task-level metrics. Look at the **Input**, **Output**, **Shuffle Read**, and **Shuffle Write** columns to understand data movement patterns.

Tip

The number of tasks in a stage provides important diagnostic information. A stage with only one task often indicates insufficient parallelism, while uneven task durations suggest data skew.

Identify and resolve resource bottlenecks

Resource bottlenecks manifest differently depending on which resource is constrained. The compute metrics interface helps you identify these patterns by showing CPU, memory, and network utilization across nodes.

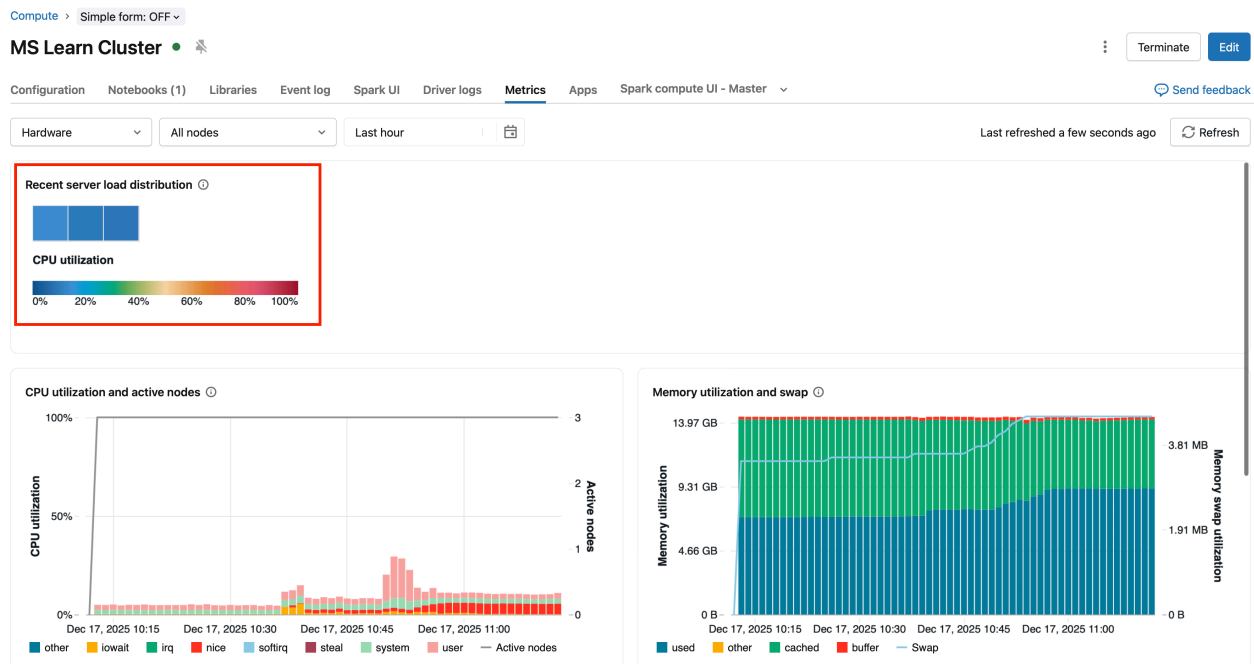


Memory pressure appears as high memory utilization across workers or the driver. In the Spark UI, look for spill indicators showing data being written to disk because memory is insufficient. You can address memory issues by increasing worker instance sizes, reducing partition counts, or optimizing transformations to minimize data held in memory.

CPU constraints show as high CPU utilization with long task execution times despite adequate I/O throughput. Consider enabling Photon acceleration for compatible workloads or scaling out with additional worker nodes.

Network bottlenecks appear as high shuffle read/write volumes with slow execution. Reduce network overhead by filtering data earlier in your pipeline, using broadcast joins for small tables, or repartitioning data to minimize cross-node transfers.

To access compute metrics, select your cluster from the **Compute** page and select the **Metrics** tab. The **Server load distribution** visualization uses color coding—red indicates heavily loaded nodes, while blue shows idle resources. If the driver node appears red while workers are blue, the driver is overloaded and may need a larger instance type.

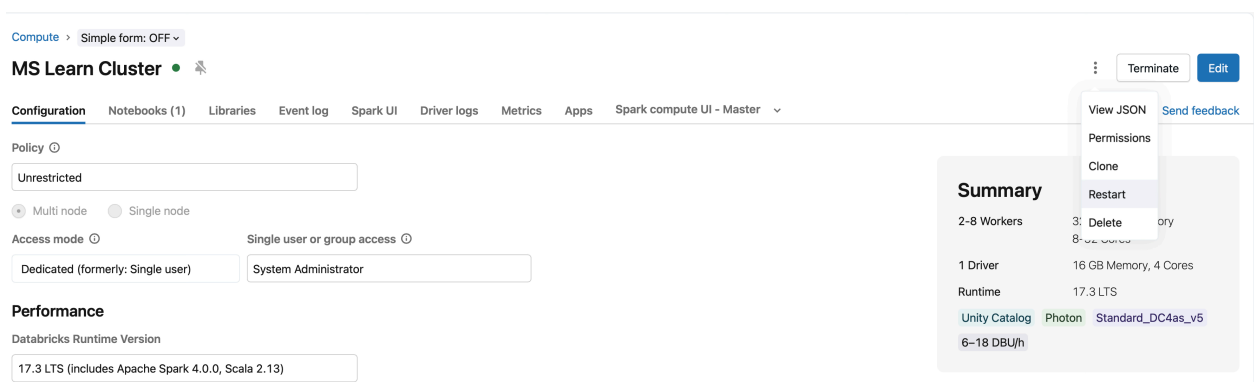


Restart clusters to resolve environmental issues

Sometimes a cluster encounters problems that require a restart to resolve. Resource exhaustion, malfunctioning executors, or stale container images can all necessitate a fresh cluster start.

Before restarting, determine whether a restart is appropriate. Check the **Event log** tab on the cluster details page for lifecycle events that might explain the problem. Look for messages about instance acquisition failures, spot instance reclamation, or executor terminations.

To restart a cluster using the UI, select your cluster from the **Compute** page and select **Restart**.



You can also restart programmatically using the Databricks CLI:

```
databricks clusters restart CLUSTER_ID
```

Replace `CLUSTER_ID` with your cluster's identifier, which you can find on the cluster configuration page or by running `databricks clusters list`.

Important

Restarting a cluster terminates any running jobs and resets the Spark UI history. Save any diagnostic information you need before restarting. For long-running clusters processing streaming data, consider scheduling regular restarts during maintenance windows to ensure the cluster runs on current images.

After making changes—whether adjusting cluster configuration, fixing code, or resolving external dependencies—validate your fix by monitoring the next job run. Check that execution times return to expected levels and that no new errors appear in the Spark UI or job output.

5. Investigate caching, skewing, spilling, shuffle

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/5-resolve-cache-skew-spill-shuffle>

Investigate caching, skewing, spilling, shuffle

Completed

A query that ran fine yesterday now takes three times longer. A job that processed millions of records suddenly fails with memory errors. These situations often trace back to four common performance issues: **caching problems**, **data skew**, **memory spill**, and **excessive shuffle**. Knowing how to investigate these issues using Azure Databricks diagnostic tools helps you restore performance quickly.

Use diagnostic tools to identify issues

Azure Databricks provides three primary tools for investigating performance problems. Each offers a different perspective on what's happening during query execution.

The **Directed Acyclic Graph (DAG)** visualizes your query's execution plan. It shows how data flows through operators and where transformations occur. By examining the DAG, you identify where

bottlenecks happen and understand the sequence of operations Spark performs.

The **Spark UI** provides detailed metrics about job execution, including stages, tasks, and resource consumption. You access it by navigating to your cluster's page and selecting **Spark UI**. The Jobs page shows timeline views, stage durations, and task distributions that reveal problems with skew, spill, and shuffle.

The **query profile** offers a breakdown of query execution for SQL warehouse workloads. It highlights slow operations and shows metrics like time spent, rows processed, and memory consumption for each operator. Access it through **Query History** in the sidebar, then select a query and choose **See query profile**.

Tip

Start your investigation with the DAG to understand the overall execution flow. Then use the Spark UI or query profile to drill into specific stages where time is spent.

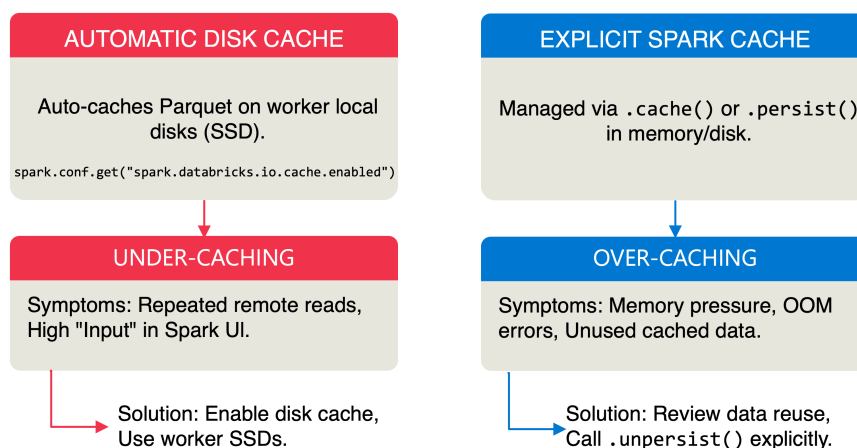
Investigate caching issues

Caching keeps frequently accessed data in memory or on local disks, avoiding repeated reads from remote storage. Azure Databricks uses **disk cache** (formerly called Delta cache) to automatically cache Parquet data files on worker nodes. This caching happens transparently when you read data.

To check if disk cache is enabled, run:

```
spark.conf.get("spark.databricks.io.cache.enabled")
```

When investigating caching problems, consider these scenarios:



Under-caching means data is read repeatedly from remote storage when it could be served from cache. The Spark UI shows high **Input** values for stages that read the same data multiple times. Enable disk cache and use worker nodes with SSD storage for better performance.

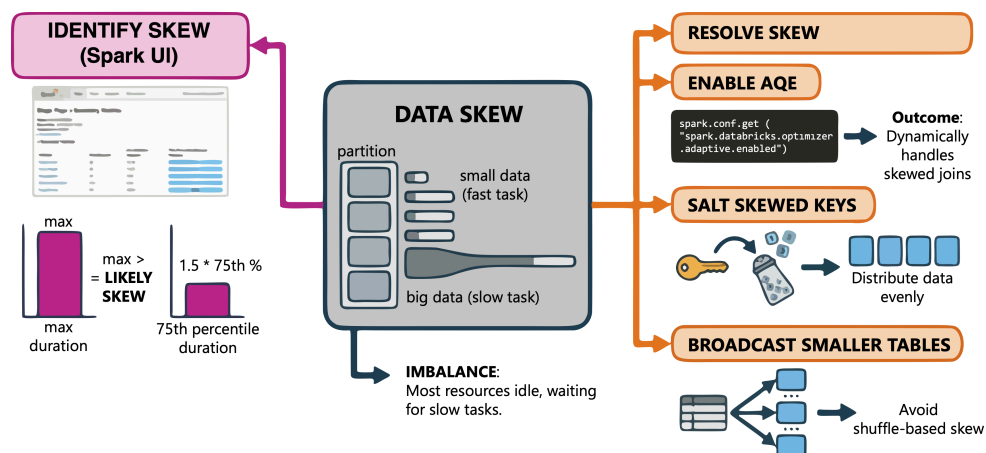
Over-caching consumes memory that Spark needs for processing. If you see memory pressure or out-of-memory errors, review whether cached data is actually being reused. Spark cache (using `.cache()` or `.persist()`) requires explicit management, unlike automatic disk caching.

Unlike automatic disk cache, Spark cache requires you to call `.unpersist()` when you no longer need the cached data:

```
df.unpersist()
```

Investigate data skew

Data skew occurs when some partitions contain significantly more data than others. This imbalance causes a few tasks to run much longer than the rest, leaving most cluster resources idle while waiting for slow tasks to complete.



To identify skew in the Spark UI, navigate to the stage's page and scroll to **Summary Metrics**. Compare the **Max** duration to the **75th percentile**. If the Max is more than 50% higher than the 75th percentile, you likely have skew.

Common causes of skew include:

- **Uneven key distribution** in joins or aggregations (for example, a single customer ID appearing in 90% of records)
- **Null values** that all route to the same partition
- **Time-based partitioning** where recent dates have far more data

To resolve skew:

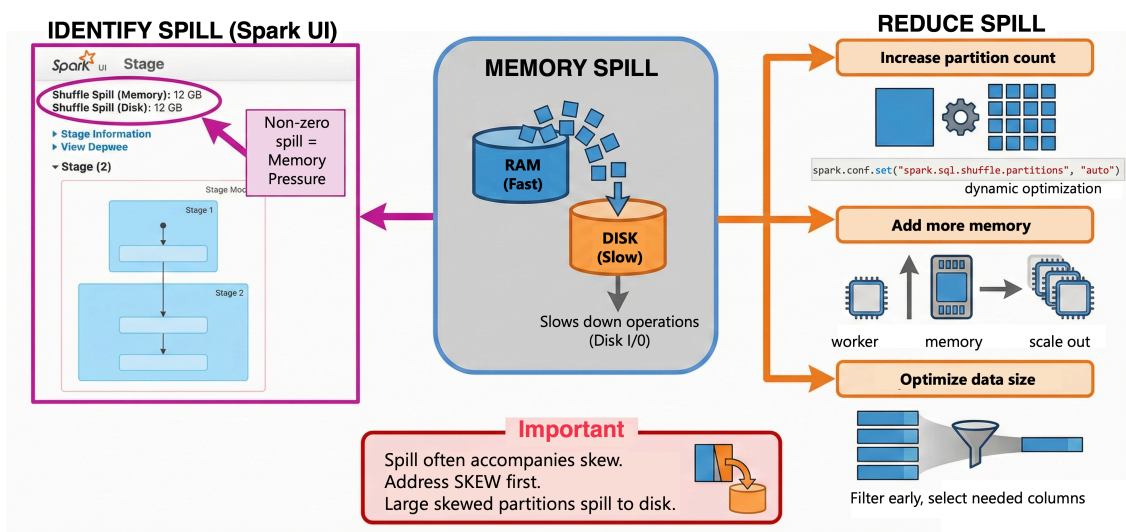
- **Enable Adaptive Query Execution (AQE)**, which dynamically handles skewed joins. Verify it's enabled:

```
spark.conf.get("spark.databricks.optimizer.adaptive.enabled")
```

- **Salt skewed keys** by adding a random component to distribute data more evenly across partitions.
- **Broadcast smaller tables** for joins where one table fits in memory to avoid shuffle-based skew.

Investigate memory spill

Spill happens when Spark runs out of memory during processing and writes intermediate data to disk. This disk I/O significantly slows down operations. Spill commonly occurs during shuffle operations, aggregations, or when partitions are too large.



The Spark UI shows spill metrics at the top of each stage's page. Look for **Shuffle Spill (Memory)** and **Shuffle Spill (Disk)** values. Any non-zero spill indicates memory pressure.

To reduce spill:

- **Increase partition count** to reduce data per partition. Set shuffle partitions to `auto` for dynamic optimization:

```
spark.conf.set("spark.sql.shuffle.partitions", "auto")
```

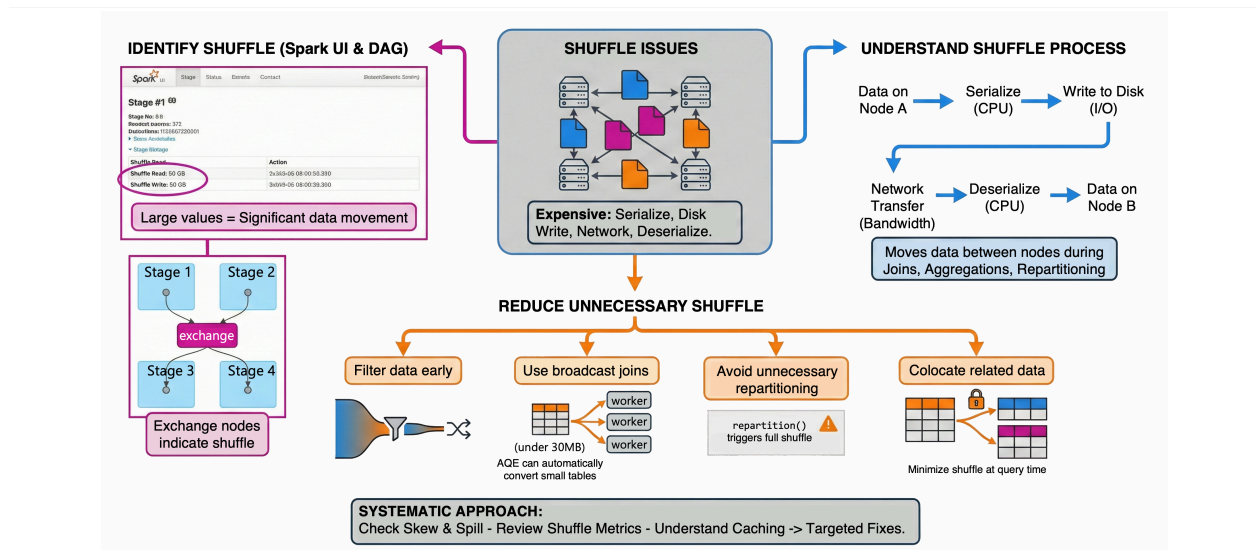
- **Add more memory** by choosing worker nodes with higher memory-to-core ratios or increasing the number of workers.
- **Optimize data size** by selecting only needed columns and filtering early in your pipeline. Reducing the data processed at each stage lowers memory requirements.

Important

Spill often accompanies skew. When a single partition grows too large for memory, it spills to disk. Address skew first, and spill problems often resolve as a side effect.

Investigate shuffle issues

Shuffle moves data between nodes during operations like joins, aggregations, and repartitioning. While sometimes necessary, excessive shuffle is expensive because it involves serializing data, writing to disk, transferring across the network, and deserializing.



In the Spark UI, check the **Shuffle Read** and **Shuffle Write** columns for each stage. Large shuffle values indicate significant data movement. The DAG shows where shuffle operations occur as exchange nodes.

Reduce unnecessary shuffle with these approaches:

- **Filter data early** before joins or aggregations to reduce the data that must be shuffled.
- **Use broadcast joins** when one table is small (under 30 MB by default). AQE can automatically convert sort-merge joins to broadcast joins when it detects a small table at runtime.
- **Avoid unnecessary repartitioning.** Each call to `repartition()` triggers a full shuffle.
- **Colocate related data.** If you frequently join two tables on the same key, consider bucketing or partitioning them similarly to minimize shuffle at query time.

The query profile in SQL warehouses visualizes shuffle as **Exchange** nodes in the DAG. Select these nodes to see metrics like data size and number of partitions exchanged.

By systematically checking for skew and spill in the Spark UI, reviewing shuffle metrics, and understanding your caching behavior, you can diagnose most performance issues and apply targeted fixes.

6. Implement log streaming with Azure Log Analytics

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/6-implement-log-streaming-azure-analytics>

Implement log streaming with Azure Log Analytics

Completed

When your Azure Databricks environment grows beyond a single workspace, tracking what happens across clusters, jobs, and users becomes a challenge. **Log streaming to Azure Log Analytics** solves this by centralizing all your diagnostic and audit logs in one place, enabling powerful queries, alerts, and dashboards.

In this unit, you learn how log streaming works, how to query Databricks logs in Azure Log Analytics, and how to create alerts for proactive monitoring.

Understand the log streaming architecture

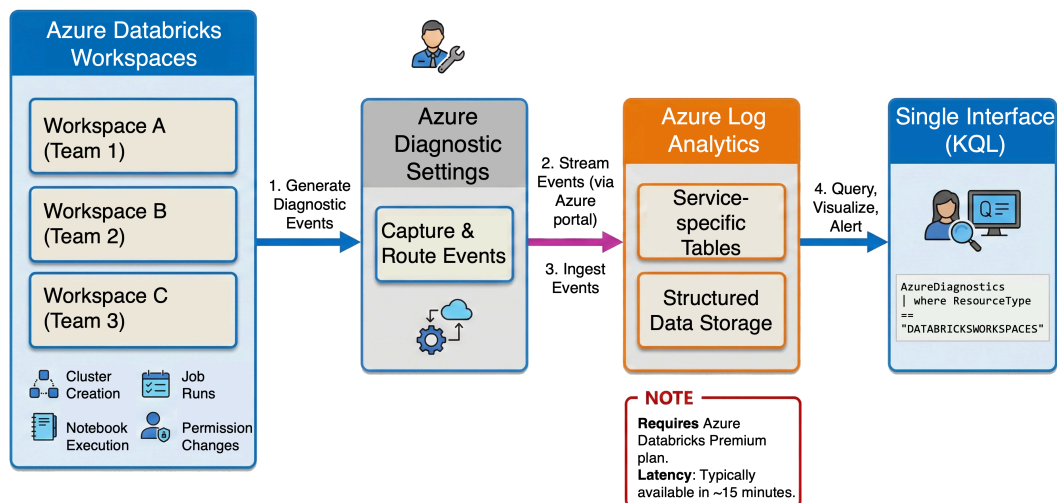
Log streaming sends **diagnostic data** from Azure Databricks to Azure Log Analytics in **near real-time**. This integration uses **Azure diagnostic settings**, which capture events from your Azure Databricks workspace and route them to a Log Analytics workspace.

Consider a typical scenario: Your organization runs multiple Azure Databricks workspaces across different teams. Without **centralized logging**, investigating a failed job requires connecting to each workspace individually and searching through scattered logs. With log streaming configured, you query all events from a **single interface**.

The data flow works as follows:

1. Azure Databricks generates audit and diagnostic events for actions like cluster creation, job runs, notebook execution, and permission changes.

2. Azure diagnostic settings capture these events and stream them to your configured destination.
3. Log Analytics ingests the events into **service-specific tables**.
4. You query, visualize, and alert on this data using **Kusto Query Language (KQL)**.



Platform administrators typically configure the diagnostic settings through the Azure portal. As a data engineer, you focus on using the logs for monitoring and troubleshooting.

Note

Diagnostic logs require the **Azure Databricks Premium plan**. Logs typically become available in Log Analytics within **15 minutes** of the event occurring.

Explore the Azure Databricks log tables

Azure Log Analytics organizes Azure Databricks logs into **purpose-specific tables**. Each table captures events from a particular service or feature, making it easier to focus your queries.

The most commonly used tables include:

Table	Description
DatabricksClusters	Cluster creation, termination, resizing, and configuration changes
DatabricksJobs	Job creation, runs, failures, and schedule modifications
DatabricksNotebook	Notebook execution, creation, and modification events
DatabricksSQL	SQL warehouse operations and query-related events
DatabricksUnityCatalog	Catalog, schema, and table access events
DatabricksWorkspace	Workspace-level operations and administrative actions

Table	Description
DatabricksSecrets	Secret scope and secret access events

Each table contains **standard columns** for event tracking:

- **TimeGenerated** - When the event occurred
- **OperationName** - The action performed
- **Identity** - The user or service principal that initiated the action
- **SourceIPAddress** - Where the request originated
- **RequestParams** - Details about the request
- **Response** - The outcome including status codes

Understanding these tables helps you target your queries efficiently. For job-related issues, start with **DatabricksJobs**. For security investigations, focus on **DatabricksSecrets** and **DatabricksUnityCatalog**.

Query logs with Kusto Query Language

Kusto Query Language (KQL) enables you to search, filter, and analyze your Azure Databricks logs. You access Log Analytics by navigating to your workspace in the Azure portal and selecting **Logs** from the sidebar.

Start with a simple query to explore recent job events:

```
DatabricksJobs
| where TimeGenerated > ago(24h)
| project TimeGenerated, OperationName, Identity, SourceIPAddress
| order by TimeGenerated desc
| take 100
```

This query retrieves the most recent 100 job-related events from the past 24 hours, displaying the timestamp, operation, user, and source IP.

Investigate failed job runs

When jobs fail, you need to **identify patterns**. The following query finds all job failures and groups them by operation:

```
DatabricksJobs
| where TimeGenerated > ago(7d)
| where Response contains "error" or Response contains "failed"
| summarize FailureCount = count() by OperationName, bin(TimeGenerated, 1h)
| order by FailureCount desc
```

This query reveals which operations fail most frequently and when failures cluster together, helping you spot **systemic issues**.

Track cluster usage patterns

Understanding cluster behavior helps optimize costs. This query shows cluster events by type:

```
DatabricksClusters
| where TimeGenerated > ago(7d)
| summarize EventCount = count() by OperationName
| order by EventCount desc
```

Monitor Unity Catalog access

For security and compliance, track who accesses sensitive data:

```
DatabricksUnityCatalog
| where TimeGenerated > ago(24h)
| where OperationName has "getTable" or OperationName has "selectFromTable"
| project TimeGenerated, Identity, OperationName, RequestParams
| order by TimeGenerated desc
```

Tip

Save frequently used queries to the **Query explorer** in Log Analytics. You can organize queries into folders and share them with your team.

Create alerts for proactive monitoring

Queries help you investigate past events. **Alerts** notify you when specific conditions occur, enabling **proactive response**.

To create an alert from a query:

1. Run your query in Log Analytics.
2. Select **New alert rule** from the toolbar.
3. Configure the **Condition** with thresholds that match your requirements.
4. Define an **Action group** to specify who receives notifications and how.

Consider setting up alerts for these common scenarios:

Job failures exceed threshold: Alert when more than five job failures occur within an hour.

```
DatabricksJobs
| where TimeGenerated > ago(1h)
| where Response contains "error" or Response contains "failed"
| summarize FailureCount = count()
| where FailureCount > 5
```

Unusual access patterns: Alert when access to sensitive Unity Catalog objects occurs outside business hours.

```
DatabricksUnityCatalog
| where TimeGenerated > ago(1h)
| where hourofday(TimeGenerated) < 6 or hourofday(TimeGenerated) > 20
| summarize AccessCount = count() by Identity
| where AccessCount > 10
```

Cluster creation spikes: Alert when cluster creation events exceed normal patterns, which might indicate runaway automation or unauthorized activity.

Action groups support multiple notification methods including email, SMS, Azure mobile app, and integration with IT service management tools.

Apply log analysis for troubleshooting

Log streaming transforms troubleshooting from **reactive investigation** to **systematic analysis**. Here are practical approaches for common scenarios:

Diagnosing job failures: Start with the `DatabricksJobs` table to find the failed run. Cross-reference with `DatabricksClusters` to check if the underlying cluster had issues. Review `DatabricksNotebook` logs if the job runs notebook code.

Investigating performance issues: Query cluster events to identify scaling patterns. Look for frequent autoscaling events that might indicate undersized initial configurations.

Security incident response: Use `DatabricksUnityCatalog` and `DatabricksSecrets` tables to trace data access. Correlate with `SourceIPAddress` to identify unusual access locations.

Cost attribution: Combine cluster and job logs with Azure cost data to understand which teams or projects drive consumption.

With logs centralized in Azure Log Analytics, you build comprehensive views of your Azure Databricks environment. This visibility supports both day-to-day operations and long-term planning decisions.

7. Exercise - Monitor, Troubleshoot and Optimize Workloads in Azure Databricks

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/7-exercise-monitor-troubleshoot-optimize-workloads-azure-databricks>

Exercise - Monitor, Troubleshoot and Optimize Workloads in Azure Databricks

Completed

Now it's your chance to monitor, troubleshoot, and optimize workloads in Azure Databricks. In this lab, you generate synthetic workloads with intentional data skew and excessive shuffle, use the Spark UI to diagnose the performance problems, and apply targeted fixes using broadcast joins, Adaptive Query Execution, and shuffle reduction techniques.

Note

To complete this lab, you need an [Azure subscription](#) in which you have administrative access.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Module assessment

<https://learn.microsoft.com/en-us/training/modules/monitor-troubleshoot-optimize-workloads-azure-databricks/8-knowledge-check>

Module assessment

Completed

9. Summary

Summary

Completed

Monitoring, troubleshooting, and optimizing workloads ensures your Azure Databricks environment runs reliably while controlling costs. Throughout this module, you explored **cluster consumption monitoring** with metrics, auto-termination, and budgets, **Lakeflow Jobs troubleshooting** using the matrix view and repair run feature, **Spark job diagnostics** through the Spark UI and compute metrics, and **performance optimization** by investigating caching, skew, spill, and shuffle issues.

Understanding how to read cluster metrics and configure auto-termination prevents idle resources from driving up costs, while autoscaling automatically adjusts worker counts to match workload demands. The **system.billing.usage** table enables you to query consumption data directly, supporting cost attribution and chargeback across teams and projects.

When jobs fail, the Jobs UI provides tools to identify failure causes quickly—from the matrix view showing patterns across runs to the repair run feature that re-executes only failed tasks and their dependents. For Spark performance issues, the **DAG visualization**, **Spark UI metrics**, and **query profile** help you pinpoint bottlenecks caused by data skew, memory spill, or excessive shuffle operations.

Centralizing logs through **Azure Log Analytics** transforms troubleshooting from reactive investigation to systematic analysis. KQL queries across DatabricksClusters, DatabricksJobs, and DatabricksUnityCatalog tables provide visibility into platform operations, while alerts notify you proactively when conditions require attention. Apply these monitoring and optimization practices to maintain performant, cost-effective data engineering solutions in Azure Databricks.