Domain 4: Deploy and maintain data pipelines and workloads with Azure Databricks

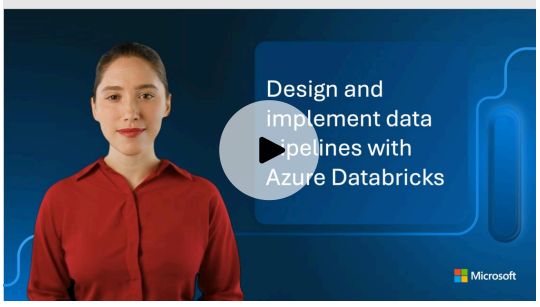
Design and implement data pipelines with Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/1-introduction

Introduction

Completed



Building reliable data pipelines requires more than connecting data sources to destinations. You need to design workflows that handle failures gracefully, scale with growing data volumes, and remain maintainable as business requirements evolve. Azure Databricks provides multiple approaches for creating data pipelines—from flexible **notebooks** with procedural code to **Lakeflow Spark Declarative Pipelines** that automate orchestration and data quality enforcement.

When you design data pipelines, you make decisions that affect every downstream consumer of your data. The order of operations determines whether transformations build on validated, well-structured data. Your choice between notebooks and declarative pipelines influences how much orchestration code you write versus how much the platform manages for you. **Task dependencies** in **Lakeflow Jobs** control execution flow and enable parallel processing that reduces pipeline runtime.

Error handling separates production-ready pipelines from fragile prototypes. Without proper error handling, invalid records corrupt downstream analytics, unnoticed failures accumulate technical debt, and problems surface hours or days after they occur. Azure Databricks provides built-in mechanisms for **data quality expectations**, **retry policies**, and **conditional task flows** that help you build resilient data workflows.

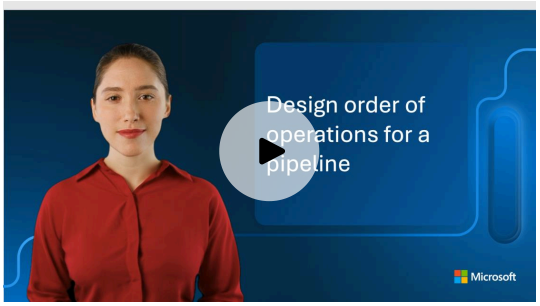
This module guides you through designing and implementing data pipelines in Azure Databricks. You learn how to structure pipeline operations, choose the right approach for your use case, configure task logic in Lakeflow Jobs, and implement error handling strategies. You also create pipelines using both notebook-based and declarative approaches, gaining hands-on experience with the tools that power production data platforms.

2. Design order of operations for a pipeline

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/2-design-order-operations-pipeline

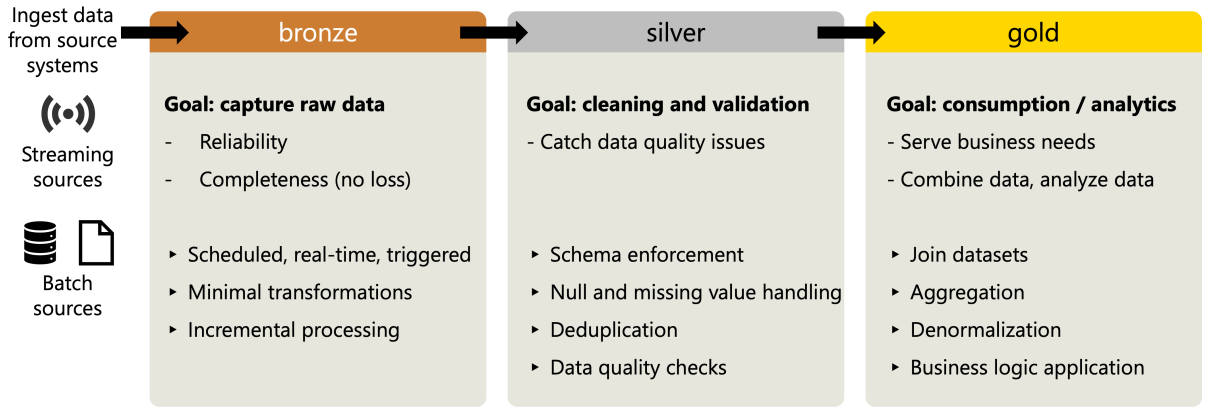
Design order of operations for a pipeline

Completed



Data pipelines move data from source systems through multiple processing stages before delivering it to consumers. Understanding the **order of operations** helps you design pipelines that are reliable, maintainable, and aligned with business requirements. Each stage in the pipeline serves a specific purpose, and the sequence you choose affects **data quality**, **processing efficiency**, and how quickly insights reach your users.

When you design a data pipeline in Azure Databricks, you typically follow a pattern that mirrors the **medallion architecture**: raw data flows through progressively refined layers until it's ready for consumption. This approach ensures that each stage builds on validated, well-structured data from the previous stage.



Ingest data from source systems

Data ingestion is the entry point of your pipeline. At this stage, you focus on capturing data from source systems and preserving it in its original form. The goal is **reliability** and **completeness**—you want to ensure that no data is lost during the transfer.

During ingestion, consider the following factors:

- **Source characteristics**: Different sources require different approaches. **Streaming sources** like message queues deliver data continuously, while **batch sources** like databases or file systems provide data at scheduled intervals.
- **Data preservation**: Store ingested data with minimal transformation. Keeping the **raw state** enables you to reprocess data if downstream requirements change.
- **Incremental processing**: Configure ingestion to process only new or changed data, reducing processing time and compute costs.

This stage corresponds to the **bronze layer** in the medallion architecture. Data engineers typically implement ingestion logic that runs on a schedule or triggers when new data arrives.

Clean and validate data

After ingestion, data enters the **cleaning and validation** stage. You transform raw data into a validated, consistent format that downstream processes can rely on. This stage is critical because **data quality issues** caught early prevent problems from propagating through the rest of your pipeline.

Cleaning operations typically include:

- **Schema enforcement**: Apply consistent data types and structures to ensure data conforms to expected formats.

- **Null and missing value handling**: Define rules for how your pipeline treats incomplete records—whether to fill defaults, drop records, or flag for review.
- **Deduplication**: Identify and remove duplicate records that might have entered through multiple ingestion paths or retry mechanisms.
- **Data quality checks**: Apply validation rules that verify business constraints, such as ensuring dates fall within valid ranges or required fields contain appropriate values.

This stage maps to the **silver layer**. By separating cleaning from ingestion, you maintain the raw data for **auditing** while providing a reliable foundation for transformation.

Transform data for consumption

Transformation shapes your validated data into structures that serve specific business needs. At this stage, you apply **business logic**, combine data from multiple sources, and create the datasets that analysts, data scientists, and applications use.

Common transformation patterns include:

- **Joining datasets**: Combine data from different sources to create unified views. For example, joining customer records with transaction data to create a complete customer activity dataset.
- **Aggregation**: Summarize data at different levels of granularity, such as calculating daily totals, monthly averages, or regional summaries.
- **Denormalization**: Flatten hierarchical or normalized data structures to optimize **query performance** for analytics workloads.
- **Business logic application**: Implement calculations, categorizations, and derived metrics that align with how your organization measures and reports on data.

Transformation logic should be **clear and testable**. When business requirements change, well-structured transformations are easier to modify without disrupting other parts of the pipeline.

Load data to target destinations

Loading represents the handoff from data engineering to data consumption. At this stage, you write processed data to tables, views, or external systems where consumers access it.

Key considerations for the load stage include:

- **Table design**: Structure tables to support common query patterns. Consider **partitioning strategies** that align with how users filter data.
- **Write patterns**: Choose between **full refresh**, **incremental append**, or **merge operations** based on data volumes and freshness requirements.

- **Schema evolution**: Plan for how your tables handle changes to data structure over time without breaking dependent workloads.

Data loaded at this stage often corresponds to the **gold layer**—highly curated datasets optimized for analytics, reporting, and machine learning.

Serve data to consumers

Once data reaches its target destination, the **serving layer** makes it accessible to different consumer groups. This stage focuses on providing appropriate **access patterns** and **performance characteristics** for each use case.

Serving considerations include:

- **Views and access layers**: Create views that present data in formats optimized for specific consumer needs, such as dashboards, reports, or analytical queries.
- **Performance optimization**: Configure **compute resources** and **caching strategies** that meet latency requirements for interactive queries.
- **Access control**: Implement **security policies** that ensure consumers see only the data they're authorized to access.

The serving layer bridges data engineering and data consumption. While data engineers build and maintain the underlying infrastructure, analysts and applications interact with the served data without needing to understand pipeline internals.

Monitor and optimize pipeline performance

Pipeline monitoring ensures your data flows reliably and efficiently over time. Without ongoing observation, issues like **data delays**, **quality degradation**, or **resource inefficiency** can go unnoticed until they affect business decisions.

Establish monitoring for:

- **Processing metrics**: Track **execution times**, **record counts**, and **resource utilization** to identify bottlenecks or unexpected changes in data volumes.
- **Data quality trends**: Monitor validation results over time to detect gradual **drift** in data quality or sudden spikes in invalid records.
- **Cost and performance**: Review compute usage patterns to identify opportunities for **optimization**, such as adjusting cluster sizes or scheduling non-urgent workloads during off-peak hours.

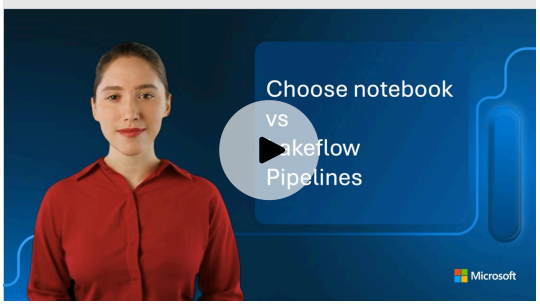
Optimization is an ongoing activity. As data volumes grow and business requirements evolve, revisit your pipeline design to ensure it continues to meet performance and cost objectives.

3. Choose notebook vs Lakeflow Pipelines

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/3-choose-notebook-lakeflow-pipelines

Choose notebook vs Lakeflow Pipelines

Completed

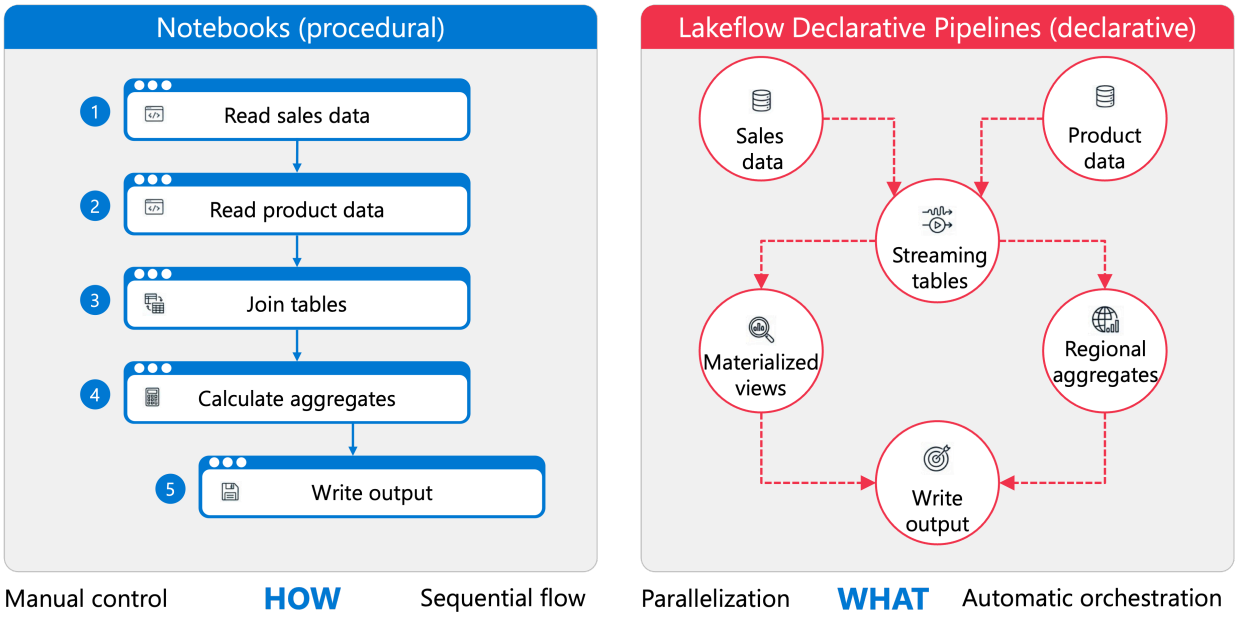


When you build data pipelines in Azure Databricks, you have two primary approaches: **notebooks** with procedural code and **Lakeflow Spark Declarative Pipelines**. Each approach serves different needs, and understanding when to use each helps you deliver maintainable, efficient data solutions.

Understand the two approaches

Notebooks execute code **step by step**. You control every aspect of data processing—from reading sources to writing outputs. This **procedural approach** gives you full control over execution flow, error handling, and optimization decisions.

Lakeflow Spark Declarative Pipelines work differently. Instead of specifying **how** to process data, you define **what** you want as the end result. You declare your **streaming tables** and **materialized views**, and the pipeline engine handles **orchestration**, **parallelization**, and **error recovery** automatically.



Consider a scenario where you need to ingest sales data, join it with product information, and calculate regional aggregates. With a notebook, you write explicit read, join, and aggregation commands in sequence. With Lakeflow Spark Declarative Pipelines, you define the final tables and their relationships—the system determines the most efficient execution plan.

When notebooks fit best

Notebooks excel in scenarios requiring **flexibility** and **detailed control**. Choose notebooks when your pipeline involves:

Complex business logic that's difficult to express declaratively. Some transformations require **conditional branching**, **loops over external APIs**, or **custom retry logic** that doesn't fit a declarative model.

Rapid prototyping and exploration. When you're still discovering the data structure or testing transformation approaches, notebooks let you **iterate cell by cell**, inspect intermediate results, and adjust your logic interactively.

External library dependencies. If you need specialized machine learning libraries, custom connectors, or legacy code integration, notebooks provide the flexibility to install and use any **Python** or **Scala** package.

Fine-grained performance tuning. When you need to manually control **partitioning**, **caching strategies**, or specific **Spark configurations**, notebooks give you direct access to these optimizations.

When Lakeflow Spark Declarative Pipelines fit best

Lakeflow Spark Declarative Pipelines simplify **production data pipelines** by handling operational complexity automatically. Choose this approach when your pipeline needs:

Standardized ETL patterns. For common ingestion and transformation workflows—reading from cloud storage, applying **schema evolution**, maintaining **slowly changing dimensions**—the declarative approach reduces thousands of lines of code to a few statements.

Built-in data quality enforcement. Declarative pipelines include **expectations** that validate data as it flows through. You define **quality rules** directly in your pipeline definition, and the system tracks violations and can halt processing when data quality degrades.

Automatic dependency management. The pipeline engine analyzes relationships between your tables and determines the correct **execution order**. When source data updates, the engine refreshes only the **affected downstream tables**.

Operational visibility. Lakeflow Spark Declarative Pipelines provide **lineage tracking**, **execution graphs**, and **monitoring dashboards** without additional configuration. Operations teams can trace data from source to target and troubleshoot issues faster.

Compare the approaches

Understanding the key differences helps you match the right tool to your requirements:

Criteria	Notebook	Lakeflow Declarative Pipeline
Control level	Full control over execution	System manages execution
Code complexity	Can be verbose for standard patterns	Concise for common ETL
Error handling	Custom implementation required	Built-in retry logic
Data quality	Manual validation code	Declarative expectations
Lineage tracking	Requires external tooling	Automatic
Team skillset	Coding required	SQL-friendly, lower barrier
Best for	Custom logic, prototyping	Production, repeatable pipelines

Make your decision

Start by evaluating your specific requirements. Ask these questions:

- Does the transformation require **procedural logic** that's hard to express declaratively?
- Is this a **one-time analysis** or a **recurring production workload**?
- What level of **operational monitoring** does your team need?
- Who maintains this pipeline—**seasoned developers** or a broader team with varied skills?

For production pipelines with standard ingestion and transformation patterns, Lakeflow Spark Declarative Pipelines **reduce operational burden** and **improve maintainability**. You spend less time

writing orchestration code and more time defining business logic.

For exploratory work, complex integrations, or pipelines requiring extensive customization, notebooks provide the **flexibility** you need. You can always refactor successful notebook prototypes into declarative pipelines once the logic stabilizes.

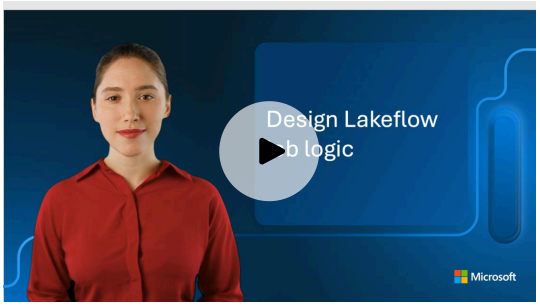
Many teams use **both approaches together**. Notebooks handle custom preprocessing or machine learning model training, while Lakeflow Spark Declarative Pipelines manage the core ETL workflow. This **hybrid approach** lets you use each tool where it performs best.

4. Design Lakeflow job logic

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/4-design-task-logic-lakeflow-job

Design Lakeflow job logic

Completed

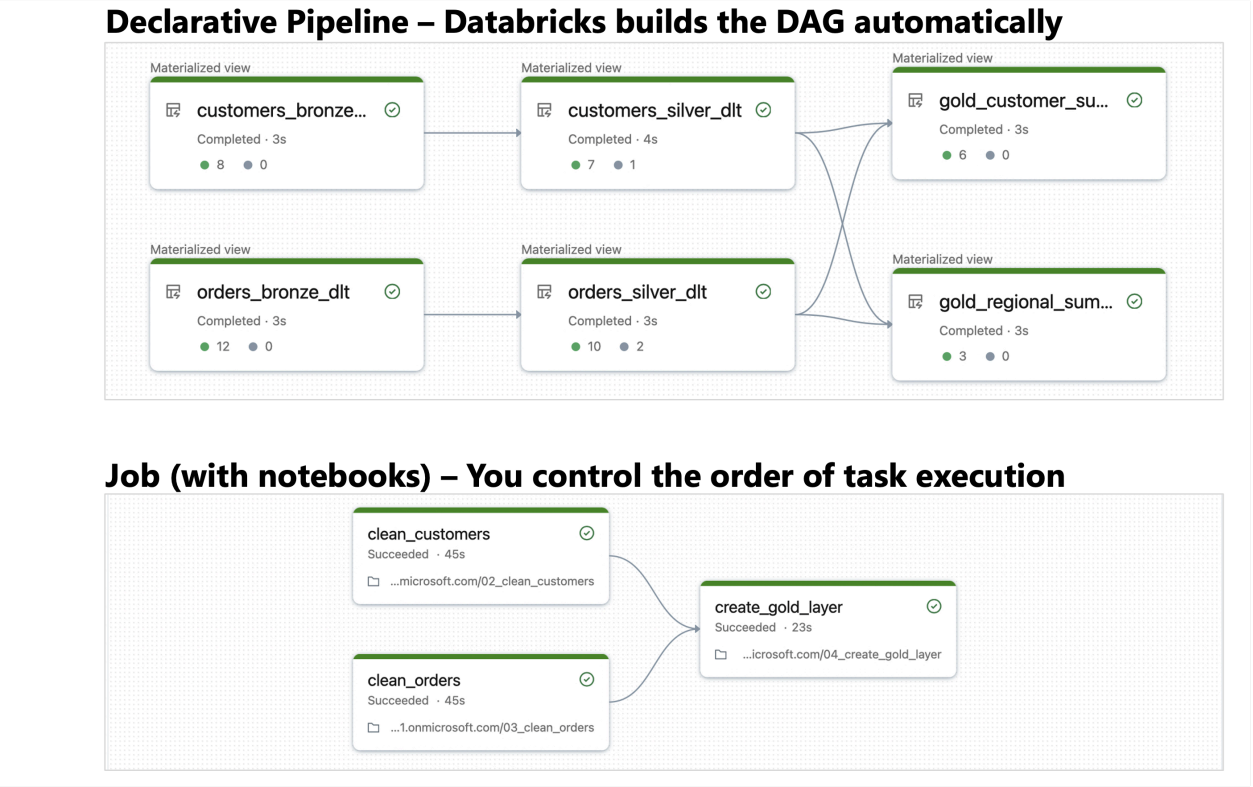


When you design task logic for a **Lakeflow Job**, you make decisions that affect how your workflow executes, scales, and responds to changing conditions. A well-designed job balances performance, cost, and maintainability while meeting business requirements.

In this unit, you learn how to structure task dependencies, choose execution patterns, and configure parameters that make your jobs flexible and reusable.

Understand task relationships in a job

Lakeflow Jobs represent tasks and their relationships as a **Directed Acyclic Graph (DAG)**. Each task in the DAG can depend on one or more upstream tasks, creating a visual map of your workflow's execution order.



Consider a typical data processing workflow. You might have an ingestion task that loads raw data, followed by transformation tasks that clean and aggregate that data, and finally a task that publishes results. The dependencies between these tasks determine not just the order of execution, but also how failures propagate through your workflow.

When you configure task dependencies, you choose from several dependency conditions:

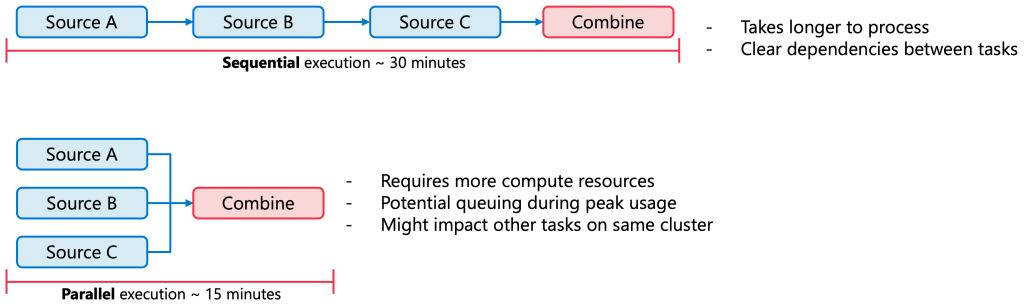
Dependency condition	Behavior
All succeeded	Runs only when all upstream tasks complete successfully
At least one succeeded	Runs when any upstream task succeeds
None failed	Runs when no upstream tasks failed, even if some are skipped
All done	Runs after all upstream tasks finish, regardless of outcome
At least one failed	Runs only when at least one upstream task fails
All failed	Runs only when all upstream tasks fail

These conditions give you control over task execution beyond simple sequential ordering. For example, you might configure a cleanup task with **All done** to ensure temporary resources are released regardless of whether earlier tasks succeeded.

Design task execution patterns

Your choice between **serial** and **parallel execution** affects both runtime performance and resource utilization. Tasks without dependencies on each other can run simultaneously, reducing total job duration.

Consider a scenario where you need to process data from three independent sources. If you design these as sequential tasks, your job waits for each source to complete before starting the next. With parallel execution, all three tasks run simultaneously.



However, parallel execution requires more compute resources. When designing your task graph, balance parallelism against the compute capacity available in your workspace. Azure Databricks limits concurrent task runs per workspace, so highly parallel jobs might experience queuing during peak usage.

Tasks that share a compute resource run on the same cluster, which reduces startup latency but means they share available memory and CPU. Consider grouping related tasks on **shared compute** while keeping resource-intensive tasks on **dedicated clusters**.

Add conditional logic to task flows

Real-world workflows often require decisions based on data conditions or processing outcomes. Lakeflow Jobs provides two primary mechanisms for conditional execution.

If/else branching

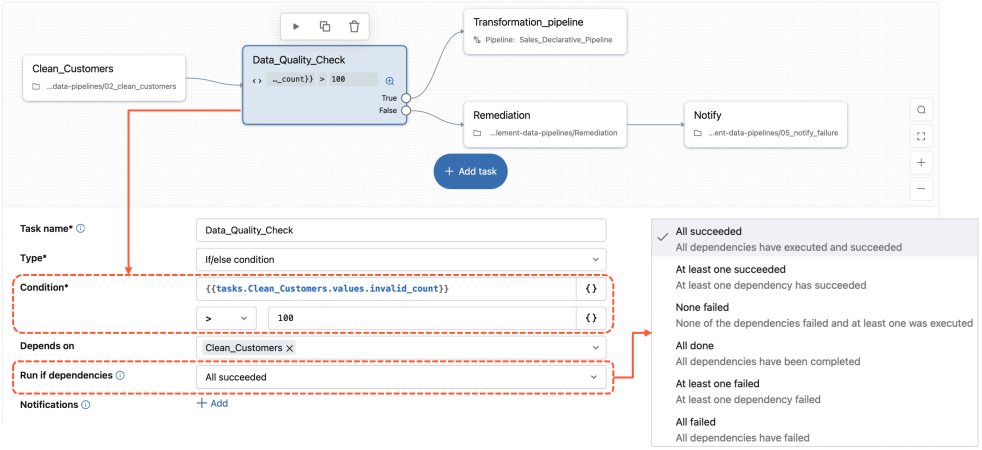
Use **If/else condition** tasks when you need to choose between different processing paths based on values or state. The condition evaluates an expression using task values, job parameters, or dynamic references.

For example, suppose your data quality check produces a count of invalid records. You can configure an If/else task to route execution:

- If invalid records exceed a threshold, run a remediation workflow
- Otherwise, proceed to the standard transformation pipeline

The condition expression might look like: `{{tasks.quality_check.values.invalid_count}} > 100`

When the condition evaluates to true, tasks configured to depend on the true outcome execute. When false, the alternative path runs instead.



Run if conditions

For tasks that should execute based on upstream task outcomes rather than data values, use **Run if** dependencies. This approach handles scenarios like:

- Running a notification task only when upstream tasks fail
- Executing cleanup logic regardless of success or failure
- Proceeding when at least one of several alternative data sources succeeds

Run if conditions evaluate automatically based on task state, without requiring explicit value comparisons.

Implement iterative processing with For each

When you need to apply the same processing logic across multiple items, the **For each** task eliminates repetitive task definitions. You define a single nested task that executes once for each element in an input array.

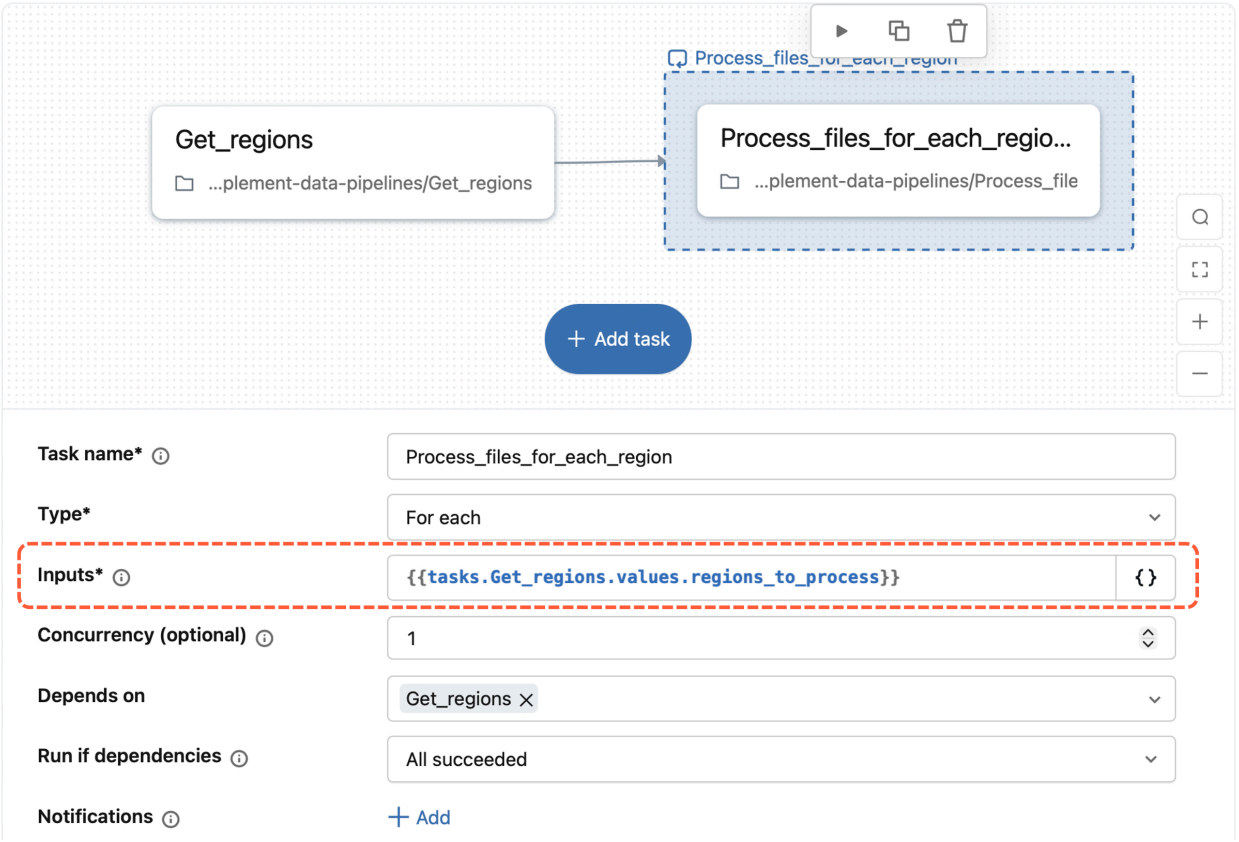
Common scenarios for For each tasks include:

- Processing files from a list of paths
- Transforming data for multiple regions or date ranges
- Running the same analysis across different product categories

The input array can come from several sources. You might define it directly as JSON, reference a task value from a preceding task, or use a job parameter. Each iteration receives the corresponding array element as its input.

```
# A preceding task might set values like this:
regions = ["us-east", "us-west", "eu-central"]
dbutils.jobs.taskValues.set(key="regions_to_process", value=regions)
```

The **For each** task then references `{{tasks.get_regions.values.regions_to_process}}` and runs its nested task three times—once for each region.



You control concurrency by setting how many iterations can run in parallel. Higher concurrency processes items faster but requires more compute resources.

Configure parameters for flexibility

Well-designed jobs accept **parameters** that allow the same workflow to handle different scenarios without modification. Parameters can be defined at the job level and accessed by any task, or passed specifically to individual tasks.

Consider designing your jobs with these parameter patterns:

Date and time parameters: Enable the same job to process different time periods. This supports both scheduled runs with current dates and backfill operations for historical data.

Environment parameters: Allow switching between development, staging, and production configurations. You might parameterize catalog names, storage paths, or connection settings.

Processing parameters: Control job behavior like batch sizes, filtering criteria, or output formats.

To reference a job parameter within a task, use the syntax `{{job.parameters.<parameter_name>}}` in the task configuration. For example, if you define a job parameter called `target_catalog`,

configure the notebook task to pass it as a base parameter with value `{{job.parameters.target_catalog}}`. Then access it in your notebook:

```
# Access the parameter passed to the notebook
target_catalog = dbutils.widgets.get("target_catalog")
spark.sql(f"USE CATALOG {target_catalog}")
```

Azure Databricks provides **dynamic value references** that inject runtime context into parameters. For scheduled jobs, `{{job.trigger.time.iso_date}}` provides the trigger date. For backfill runs, `{{backfill.iso_datetime}}` supplies the backfill timestamp.

Select compute resources for tasks

Each task type has recommended compute options that affect both capability and cost. Your task logic design should account for compute requirements:

Task type	Recommended compute
Notebooks, Python scripts	Serverless jobs compute
SQL queries and files	Serverless SQL warehouse
Lakeflow Spark Declarative Pipelines	Serverless pipeline compute
JAR and Spark Submit	Classic jobs compute

Tasks within the same job can use different compute resources. A common pattern assigns SQL tasks to a SQL warehouse while notebook-based transformations run on jobs compute.

When multiple tasks share a cluster, the cluster remains active until all tasks complete. This reduces startup time but incurs cost during idle periods between task executions. Balance shared compute benefits against the resource isolation of dedicated clusters.

For production jobs, **serverless compute** eliminates cluster management overhead and scales automatically. **Classic compute** provides more configuration control when you need specific Spark versions, custom libraries, or particular instance types.

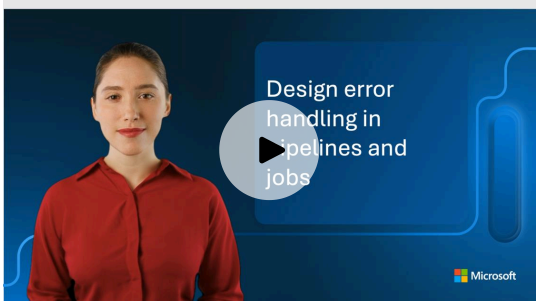
Your task logic design establishes the foundation for reliable, efficient data workflows. As you implement and test your jobs, you can refine dependencies, adjust parallelism, and tune parameters to match actual workload characteristics.

5. Design error handling in pipelines and jobs

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/5-design-error-handling-pipelines

Design error handling in pipelines and jobs

Completed

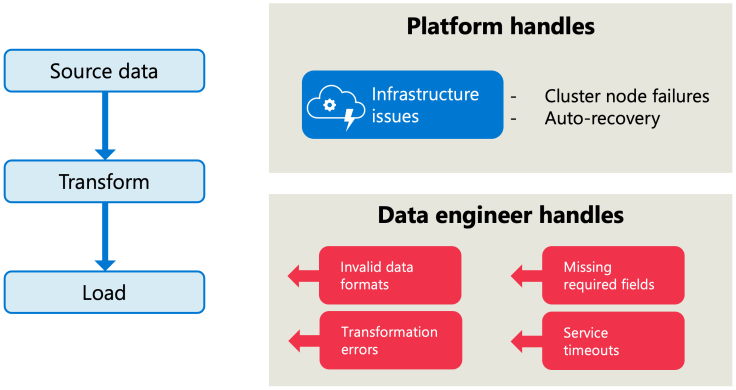


When data pipelines fail, the consequences extend beyond immediate job errors. Invalid records can corrupt downstream analytics, unnoticed failures can accumulate technical debt, and without proper alerting, teams might discover problems hours or days after they occur. Designing robust **error handling** transforms your pipelines from fragile processes into resilient systems that protect data quality and enable rapid recovery.

In this unit, you learn how to design error handling strategies for data pipelines, notebooks, and jobs in Azure Databricks.

Understand error handling responsibilities

Error handling in Azure Databricks operates at multiple levels, each addressing different failure scenarios. The platform handles **infrastructure-level issues** like cluster node failures automatically. Your responsibility as a data engineer focuses on **data-level errors** and **application logic failures**.



Consider a typical ETL pipeline processing customer transactions. Errors can occur when:

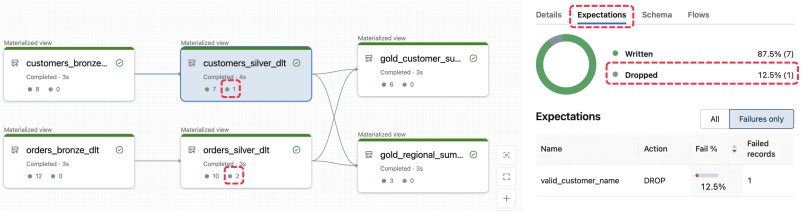
- Source data contains invalid formats or missing required fields

- Transformation logic encounters unexpected values
- External service connections time out
- Compute resources become unavailable

Each scenario requires a different response. Some errors warrant immediate pipeline termination, while others should log the issue and continue processing valid records.

Define data quality expectations in declarative pipelines

Lakeflow Spark Declarative Pipelines provides built-in data quality constraints called **expectations**. These constraints validate records as data flows through your pipeline, giving you control over how to handle invalid data.



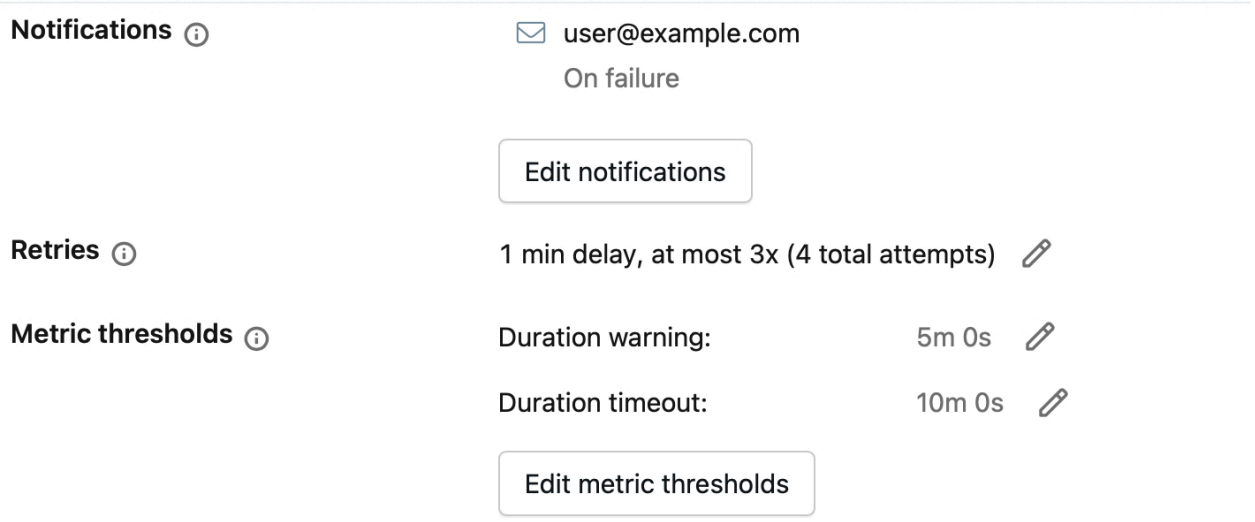
When a record fails validation, three actions determine what happens:

Action	Behavior	Use case
Warn	Invalid records written to target; metrics logged	Monitoring data quality trends
Drop	Invalid records excluded from target	Preventing bad data from propagating
Fail	Pipeline stops immediately	Critical constraints that must never be violated

> **Tip**
>
> When a fail expectation triggers, only that specific flow stops. Other parallel flows in the same pipeline continue processing. This isolation prevents a single data quality issue from halting your entire data platform.

Configure job-level error handling

Lakeflow Jobs provides several mechanisms to handle task failures and ensure pipeline reliability. These settings work together to create resilient workflows.



Set retry policies

Transient failures like network timeouts, temporary resource constraints, or brief service interruptions often resolve themselves. Configure **retry policies** to automatically attempt failed tasks again:

- **Retry count**: Number of attempts before marking the task as failed (typically 1-3 retries)
- **Retry interval**: Time between attempts, allowing systems to recover

For continuous jobs, the platform automatically applies **exponential backoff**. Each consecutive failure increases the wait time before the next retry, preventing rapid repeated failures from overwhelming resources.

Configure timeouts

A task that hangs indefinitely consumes resources and blocks downstream processing. Set **timeout thresholds** to terminate unresponsive tasks:

- **Expected duration**: Triggers a warning notification if exceeded
- **Maximum duration**: Terminates the task if exceeded

Add notifications

Proactive alerting enables rapid response to failures. Configure **notifications** to reach your team through:

- Email addresses
- Slack channels
- Microsoft Teams
- PagerDuty
- Custom webhooks

You can trigger notifications on job start, success, failure, or when duration thresholds are exceeded.

> **Important**
>
> Task-level notifications fire on each retry attempt. To avoid notification fatigue, select **Mute notifications until the last retry** when configuring task notifications.

Design conditional task flows

Beyond simple retries, you can design jobs that respond intelligently to failures using **conditional task dependencies**:

- **All succeeded**: Continue only when every upstream task completes successfully
- **At least one failed**: Trigger error handling or cleanup tasks
- **All done**: Execute regardless of upstream status (useful for cleanup operations)

This approach enables patterns like running a notification task when processing fails, or executing cleanup logic before a retry.

Implement error handling in notebooks

When building pipelines with notebooks, you implement error handling directly in your code. Python's **exception handling** provides granular control over failure responses.

Catch and handle exceptions

Wrap operations that might fail in **try-except blocks**:

```
from pyspark.errors import PySparkException

try:
    df = spark.read.table("source_data")
    transformed = df.filter("amount > 0")
    transformed.write.mode("append").saveAsTable("target_data")
except PySparkException as e:
    if e.getErrorClass() == "TABLE_OR_VIEW_NOT_FOUND":
        # Handle missing source table
        print(f"Source table not found: {e.getMessageParameters()['relationName']}"
    else:
        raise  # Re-raise unexpected errors
```

This example catches `PySparkException`, the base class for PySpark errors. The `getErrorClass()` method returns a standardized error code like `TABLE_OR_VIEW_NOT_FOUND`, allowing you to handle specific errors differently. The `getMessageParameters()` method provides context about the error, such as the name of the missing table. Errors you don't explicitly handle are re-raised to avoid silently ignoring unexpected issues.

Implement retry logic

For operations prone to transient failures, implement retry with exponential backoff:

```
import time

def run_with_retry(operation, max_retries=3, base_delay=10):
    for attempt in range(max_retries + 1):
        try:
            return operation()
        except Exception as e:
            if attempt == max_retries:
                raise
            delay = base_delay * (2 ** attempt)
            print(f"Attempt {attempt + 1} failed, retrying in {delay}s: {e}")
            time.sleep(delay)
```

Signal success or failure to jobs

When a notebook runs as a job task, use `dbutils.notebook.exit()` to communicate results:

```
try:
    # Processing logic
    records_processed = process_data()
    dbutils.notebook.exit(f"SUCCESS: Processed {records_processed} records")
except Exception as e:
    dbutils.notebook.exit(f"FAILED: {str(e)}")
```

Downstream tasks can then use these values in conditional logic or for reporting.

Apply error handling best practices

Effective error handling balances thoroughness with practicality. Consider these guidelines:

Fail fast for critical errors: When data integrity is at stake, stop processing immediately. Continuing with corrupt data creates larger problems downstream.

Log meaningful context: Include enough information to diagnose issues without exposing sensitive data. Record timestamps, record counts, and error classifications.

Plan for recovery: Design pipelines so failed runs can be repaired. Use idempotent operations where possible, and structure jobs so only failed tasks need re-running.

Monitor trends: Track data quality metrics over time. A gradual increase in dropped records might indicate upstream system changes that need attention.

Test failure scenarios: Validate that your error handling works as expected by deliberately introducing failures during development.

6. Create pipeline with notebook

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/6-create-pipeline-notebook-precedence

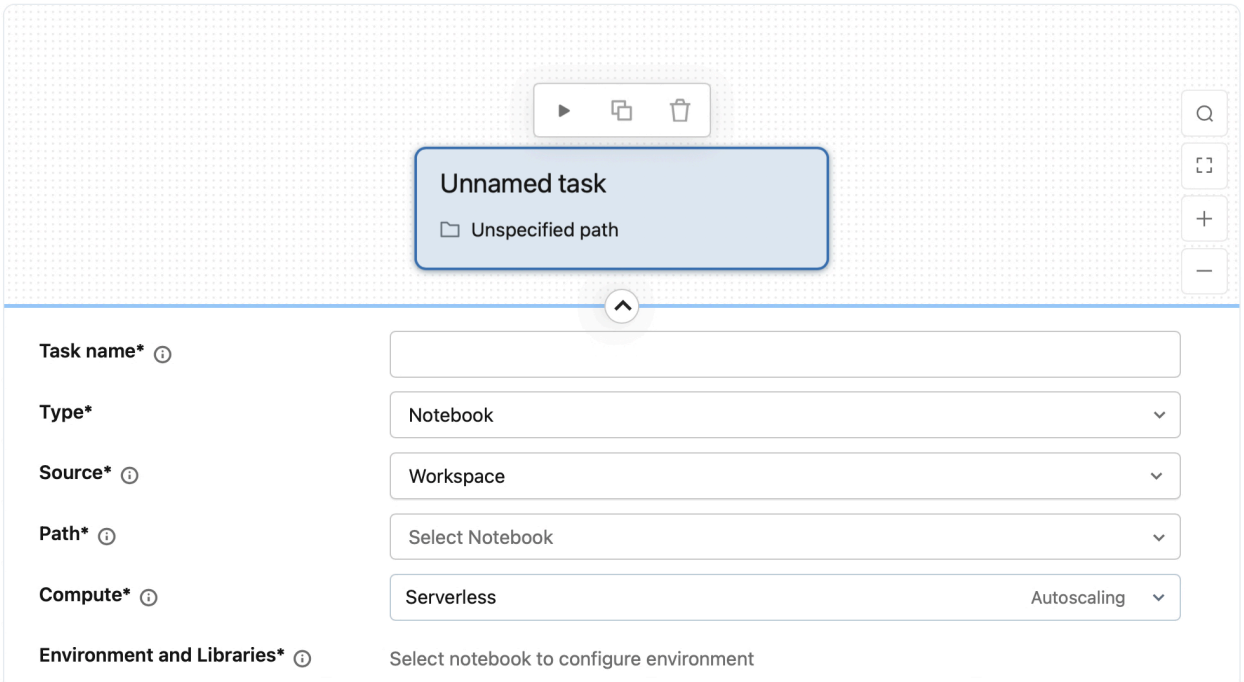
Create pipeline with notebook

Completed

Data pipelines often require multiple notebooks to run in a specific sequence. When you build a pipeline using notebooks in Azure Databricks, you configure **task dependencies** that control the execution order. These dependencies ensure that each step completes before dependent steps begin.

Add a notebook task to a Lakeflow Job

A **notebook task** runs a Databricks notebook as part of a job workflow. You specify the notebook path, configure compute resources, and define any parameters the notebook requires. Multiple notebook tasks can be chained together to form a complete data pipeline.



When you add notebook tasks to a job, each task appears as a node in a **Directed Acyclic Graph (DAG)**. The DAG provides a visual representation of your pipeline's execution flow, showing which tasks run first and which tasks depend on others.

Consider a typical data pipeline that processes sales data. The pipeline might include three notebooks: one for data ingestion, one for transformation, and one for loading data into a final table. Without dependencies configured, these notebooks could run simultaneously and cause failures when the transformation notebook tries to read data that hasn't been ingested yet.

Configure task dependencies

Task dependencies control when each notebook runs relative to other tasks in your job. When you configure a dependency, you specify that one task must wait for another task to complete before starting.

To add a dependency between tasks:

1. Select a task in the job DAG.
2. In the **Depends on** field, select the upstream task from the drop-down menu.
3. Select **Save task**.

When you select a task in the DAG before creating a new task, the new task automatically has a dependency on the selected task. This default behavior helps you build sequential pipelines quickly.

Dependencies create a clear **execution path** through your pipeline. In a four-task job where Task 2 and Task 3 both depend on Task 1, and Task 4 depends on both Task 2 and Task 3, Azure Databricks runs Task 1 first. After Task 1 completes successfully, Task 2 and Task 3 run **in parallel**. Task 4 starts only after both Task 2 and Task 3 finish.

```
Task 1 (ingest_data)
    ├── Task 2 (transform_customers)
    └── Task 3 (transform_orders)
            └── Task 4 (generate_report)
```

Create a notebook task from scratch

Building a notebook-based pipeline starts with creating the job and adding your first notebook task. The following steps walk through the complete configuration process.

1. In your Azure Databricks workspace, select **Jobs & Pipelines** in the sidebar.
2. Select **Create**, then select **Job**.
3. Enter a task name that describes the notebook's purpose, such as `ingest_sales_data`.
4. In the **Type** drop-down menu, select **Notebook**.
5. In the **Source** drop-down menu, select **Workspace** to use a notebook stored in your workspace.
6. Select the **Path** field and browse to your notebook location.
7. Configure compute resources using the **Compute** field.
8. Select **Create task**.

After creating the first task, add additional notebook tasks and configure their dependencies to complete your pipeline structure. Each new task you add while another task is selected automatically inherits a dependency on that task.

Pass parameters between notebooks

Notebook tasks can receive **parameters** at runtime through the **Parameters** configuration. These parameters are accessible in the notebook using `dbutils.widgets`. Parameter passing enables you to reuse notebooks across different jobs with varying inputs.

To configure parameters for a notebook task:

1. Select the task in your job configuration.
2. Under **Parameters**, select **Add**.
3. Enter the parameter name in the **Key** field.
4. Enter the parameter value in the **Value** field.
5. Select **Save task**.

In your notebook, retrieve parameters using `dbutils.widgets.get()`:

```
# Retrieve a parameter named 'process_date' with a default value
dbutils.widgets.text("process_date", "2024-01-01", "Process Date")
process_date = dbutils.widgets.get("process_date")

# Use the parameter in your processing logic
df = spark.read.table("sales_raw").filter(f"date = '{process_date}'")
```

This approach keeps your notebooks flexible and allows the same notebook to process different data based on job parameters.

Best practices for notebook pipelines

When building production notebook pipelines, follow these practices to maintain reliability and clarity.

- **Use descriptive task names** that indicate the notebook's function. Names like `bronze_ingest_orders` or `silver_transform_customers` make the pipeline's purpose immediately clear in the DAG view.
- **Organize notebooks in folders** that mirror your pipeline structure. A folder hierarchy such as `/pipelines/sales/bronze/` and `/pipelines/sales/silver/` helps team members locate notebooks quickly.
- **Minimize task dependencies** where possible. While dependencies ensure correct execution order, excessive dependencies can reduce parallelism and slow down your pipeline. Only add dependencies where a true data relationship exists.

- **Use workspace notebooks for production jobs** rather than notebooks in Git folders when you need to track MLflow experiments. Notebooks run from remote Git repositories are ephemeral and can't reliably track MLflow runs or models.
- **Consider output limits** when designing your notebooks. Notebook cell output is limited to 20 MB total, with individual cells limited to 8 MB. If your notebooks generate large outputs, write results to tables instead of displaying them in cell output.

7. Create pipeline with Lakeflow Spark Declarative Pipelines

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/7-create-pipeline-lakeflow-declarative

Create pipeline with Lakeflow Spark Declarative Pipelines

Completed

Production data pipelines require reliability, maintainability, and clear data quality enforcement. As a data engineer, you likely spend significant time writing code to handle incremental processing, orchestrate dependencies, and validate data quality. **Lakeflow Spark Declarative Pipelines** in Azure Databricks addresses these challenges by letting you define *what* your data should look like rather than *how* to process it step by step.

In this unit, you learn how to create data pipelines using the declarative approach, define streaming tables and materialized views, and apply data quality expectations to enforce constraints on your data.

Understand the declarative approach

Traditional data pipelines require you to write imperative code that specifies every processing step. You handle incremental processing logic, manage checkpoint recovery, and orchestrate dependencies between tables. With Lakeflow Spark Declarative Pipelines, you instead declare the **desired end state**, and the framework handles the execution details.

The declarative approach provides three key benefits for production pipelines:

- **Automatic orchestration**: The framework analyzes dependencies between your tables and determines the correct execution order. If you define a silver table that reads from a bronze table, the framework automatically processes the bronze table first.

- **Incremental processing**: Streaming tables process each record exactly once. Materialized views automatically identify and process only changed data when possible, avoiding expensive full recomputations.
- **Built-in retry logic**: When failures occur, the framework retries at the most granular level possible—first at the Spark task level, then at the flow level, and finally at the pipeline level.

Consider a scenario where you build an analytics pipeline for customer transactions. Instead of writing hundreds of lines of Structured Streaming code with checkpoint management and state handling, you declare your tables and let the framework manage the complexity.

Define streaming tables for data ingestion

A **streaming table** is a Delta table optimized for append-only data processing. Each input record is processed **exactly once**, making streaming tables ideal for ingesting data from sources like cloud storage or message queues.

The following SQL example creates a streaming table that ingests JSON files from cloud storage using Auto Loader:

```
CREATE OR REFRESH STREAMING TABLE customers_bronze
AS SELECT * FROM STREAM read_files(
  "/Volumes/raw_data/customers",
  format => "json"
);
```

The `STREAM` keyword tells the pipeline to treat the source as a **streaming dataset**. On each pipeline update, only new files are processed and appended to the table.

You can achieve the same result with Python:

```
from pyspark import pipelines as dp

@dp.table
def customers_bronze():
  return (
    spark.readStream.format("cloudFiles")
     .option("cloudFiles.format", "json")
     .option("cloudFiles.inferColumnTypes", "true")
     .load("/Volumes/raw_data/customers")
  )
```

The `@dp.table` decorator marks this function as a streaming table definition. The function returns a streaming DataFrame, which the framework processes incrementally with each update.

Create materialized views for transformations

While streaming tables handle append-only ingestion, **materialized views** are better suited for transformations where you need to reason about aggregations or join dimensions that might change. A materialized view caches query results and automatically keeps them **synchronized with upstream changes**.

The following example creates a materialized view that joins customer and transaction data:

```
CREATE OR REPLACE MATERIALIZED VIEW customer_transactions
AS SELECT
  c.customer_id,
  c.customer_name,
  c.region,
  t.transaction_date,
  t.amount
FROM customers c
INNER JOIN transactions t ON c.customer_id = t.customer_id;
```

Unlike streaming tables, materialized views handle complex queries including aggregations and joins. When upstream tables change, the framework determines the most efficient way to update the materialized view—often processing only the affected data rather than recomputing everything.

In Python, you use the `@dp.materialized_view` decorator:

```
from pyspark import pipelines as dp

@dp.materialized_view
def regional_sales():
  customers_df = spark.read.table("customers")
  transactions_df = spark.read.table("transactions")

  return (
    customers_df.join(transactions_df, on="customer_id", how="inner")
    .groupBy("region")
    .agg({"amount": "sum"})
  )
```

> **Tip**
>
> Use streaming tables when your source data is append-only and you need low-latency processing. Use materialized views when you need aggregations, complex joins, or must handle updates and deletes in source data.

Apply data quality expectations

Production pipelines require data quality enforcement. **Expectations** are declarative constraints that validate records as they flow through your pipeline. You define what valid data looks like, and the framework tracks metrics and takes action when violations occur.

When a record fails validation, the framework can respond in different ways depending on the action you specify:

Action	Behavior
`EXPECT` (default)	Retains invalid records, tracks violation counts in metrics
`ON VIOLATION DROP ROW`	Removes invalid records from the output
`ON VIOLATION FAIL UPDATE`	Stops the pipeline and rolls back the transaction

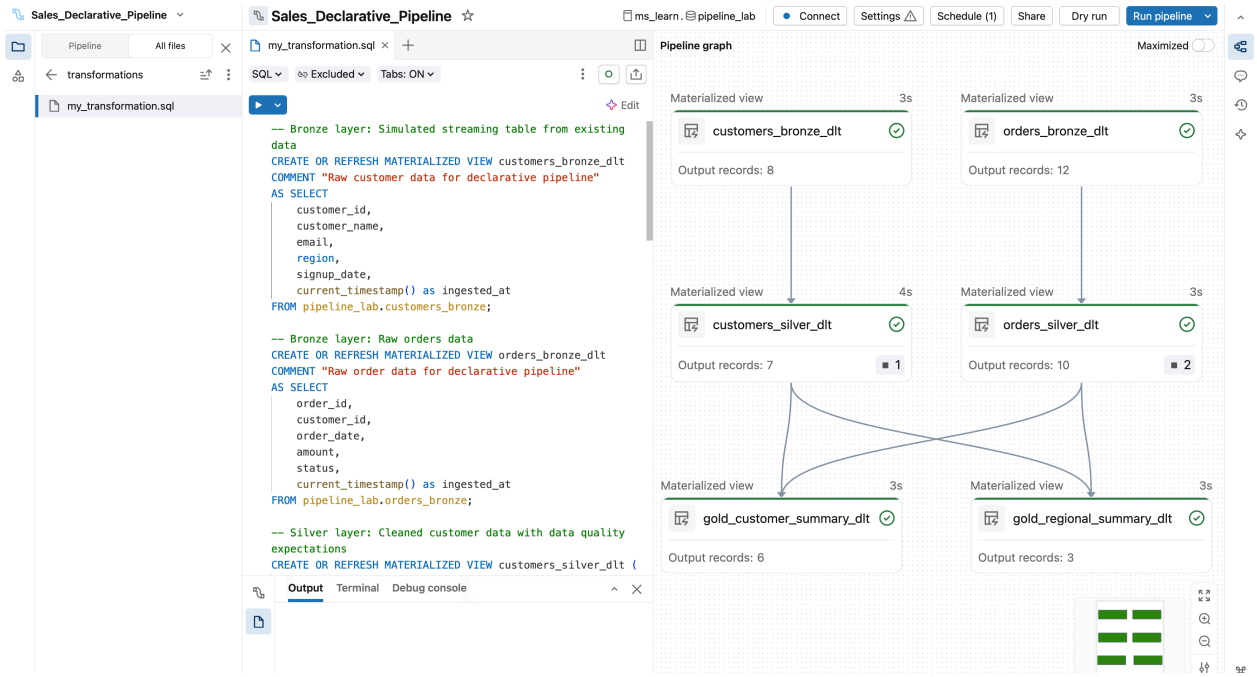
Use `ON VIOLATION FAIL UPDATE` for critical constraints where any violation indicates a serious data problem that requires investigation before processing can continue.

> **Important**
>
> The Lakeflow Spark Declarative Pipelines **Advanced** product edition is required to use expectations. If your pipeline includes expectations with the Core or Pro editions, you receive an error.

Develop pipelines with the Lakeflow Pipelines Editor

The Lakeflow Pipelines Editor provides an integrated development environment for creating and testing pipelines. When you create a new ETL pipeline in Azure Databricks, the editor provides a default folder structure with separate directories for transformation source code, exploratory notebooks, and utility modules.

> **Note**
>
> The code examples in this unit are **pipeline definitions**, not interactive notebook code. You cannot run these statements directly in a regular notebook. The `pyspark.pipelines` module and streaming table DDL statements are only available within the Lakeflow pipeline runtime. To execute your pipeline code, use the **Dry run** feature to validate, then **run the pipeline** through the Jobs & Pipelines interface.



The editor supports iterative development through several features:

- **Dry run**: Validates your pipeline code without processing data, allowing you to catch syntax errors and missing dependencies before execution.
- **Selective execution**: Run individual files or single table definitions rather than the entire pipeline, enabling faster iteration during development.
- **Interactive DAG**: Visualize the dependency graph between your tables, select multiple tables for targeted refreshes, and inspect execution metrics.
- **Data preview**: Sample data from streaming tables and materialized views directly in the editor to verify transformation logic.

With the declarative approach and integrated tooling, you build production-grade data pipelines that are maintainable, observable, and enforce consistent data quality. The framework handles the complexity of incremental processing and orchestration, so you can focus on defining the business logic that matters to your organization.

8. Exercise - Design and Implement Data Pipelines with Azure Databricks

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/8-exercise-design-implement-data-pipelines

Exercise - Design and Implement Data Pipelines with Azure Databricks

Completed

Now it's your chance to design and implement data pipelines with Azure Databricks. In this lab, you build a medallion architecture pipeline (Bronze → Silver → Gold) for GlobStay hotel booking data, applying cleaning rules such as deduplication, null filtering, and date validation before producing Gold-layer aggregations for property and channel performance. You implement error handling and parameterize notebooks for job orchestration. You then configure a Lakeflow Job in the Azure Databricks UI with sequential task dependencies, retry policies, failure notifications, and an If/else condition task for data quality routing.

> **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

9. Module assessment

https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/9-knowledge-check

Module assessment

Completed

10. Summary

Summary

Completed

Designing and implementing data pipelines in Azure Databricks requires understanding both the conceptual framework and the practical tools available. Throughout this module, you explored the **order of operations** that guide reliable pipeline design—from data ingestion through cleaning, transformation, loading, and serving. You learned how the **medallion architecture** provides a structured approach where each stage builds on validated data from the previous stage, ensuring data quality and maintainability.

Choosing between **notebooks** and **Lakeflow Spark Declarative Pipelines** depends on your specific requirements. Notebooks offer flexibility for complex business logic, rapid prototyping, and custom integrations. Lakeflow Spark Declarative Pipelines reduce operational complexity by automatically managing orchestration, incremental processing, and dependency analysis. Many production environments benefit from combining both approaches—using notebooks for specialized processing while declarative pipelines handle core ETL workflows.

Task logic design in **Lakeflow Jobs** enables sophisticated workflow patterns. You configured **task dependencies** to control execution order, implemented **conditional branching** with If/else tasks, and used **For each** tasks for iterative processing. **Error handling** proved essential for production reliability—you applied data quality expectations in declarative pipelines, configured retry policies and notifications at the job level, and implemented exception handling in notebook code.

Apply these concepts by starting with clear pipeline architecture that matches the medallion model. Choose the pipeline approach that fits your team's skills and the complexity of your transformations. Design task dependencies that maximize parallelism while respecting data relationships. Implement error handling strategies that protect data quality and enable rapid recovery when failures occur. These practices form the foundation for building data platforms that scale with your organization's needs.

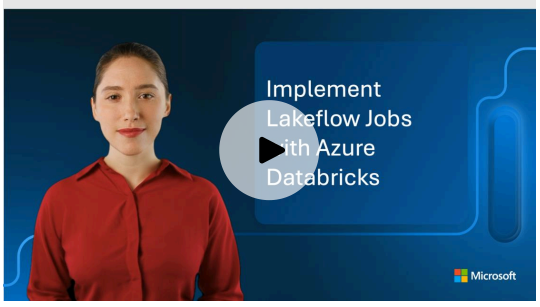
Implement Lakeflow Jobs with Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/1-introduction

Introduction

Completed



Data pipelines rarely succeed when run manually at arbitrary times. **Production workloads** demand automation—jobs that execute reliably, respond to data changes, and recover from failures without human intervention. **Lakeflow Jobs** in Azure Databricks provides the **orchestration layer** that transforms ad hoc data processing into robust, automated workflows.

Building effective data pipelines requires more than writing transformation logic. You need to configure how tasks execute, when they run, and what happens when something goes wrong. A well-designed job coordinates multiple tasks with proper **dependencies**, allocates appropriate **compute resources**, and maintains visibility through alerts and notifications. Without these capabilities, even the best data transformations become operational liabilities that require constant monitoring and manual restarts.

Lakeflow Jobs addresses these challenges through a comprehensive set of features. You create jobs that organize tasks as **directed acyclic graphs (DAGs)**, defining execution order and dependencies. You configure **triggers** that respond to table updates, file arrivals, or continuous processing needs—eliminating rigid schedules that miss data changes or waste resources. You set up **schedules** using simple intervals or cron expressions when time-based execution fits your requirements. **Alerts and notifications** keep your team informed about job status without constant dashboard monitoring.

Automatic restart policies handle transient failures gracefully, maintaining pipeline reliability during inevitable infrastructure hiccups.

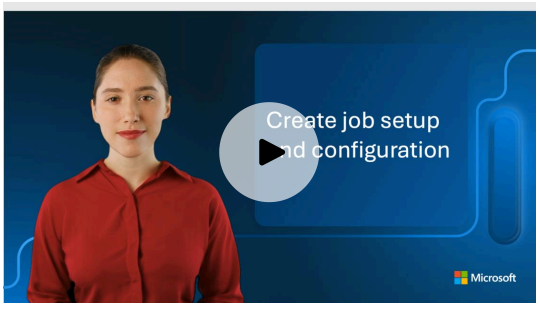
This module guides you through implementing these Lakeflow Jobs capabilities. You explore **job creation** and **task configuration**, event-based and scheduled triggers, alerting strategies, and **retry policies** that keep your data pipelines running smoothly. By mastering these concepts, you build automation that your organization can depend on for **critical data processing workloads**.

2. Create job setup and configuration

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/2-create-lakeflow-job

Create job setup and configuration

Completed



When you need to automate data processing or orchestrate multiple operations in Azure Databricks, you create a **Lakeflow Job**. A job coordinates tasks, manages their execution order, and allocates compute resources to run your workloads reliably.

In this unit, you learn how to create and configure a Lakeflow Job, including setting up tasks, selecting compute resources, and organizing task dependencies.

Understand job structure

A Lakeflow Job consists of one or more **tasks** organized as a **Directed Acyclic Graph (DAG)**. The DAG defines execution order and dependencies between tasks, allowing you to build workflows that range from a single notebook execution to complex multi-step data pipelines.

Every job requires at minimum:

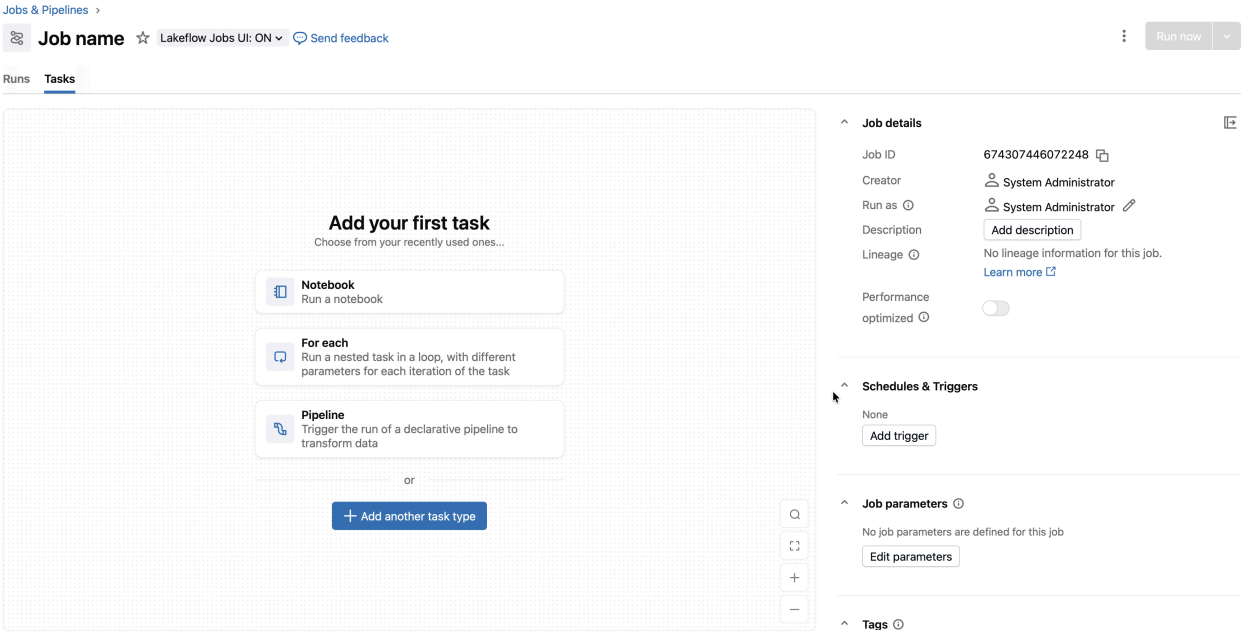
- A **task** containing the logic to run
- A **compute resource** to execute the task
- A **unique name** to identify the job

Tasks within a job can execute notebooks, Python scripts, SQL queries, or Lakeflow Spark Declarative Pipelines. You configure each task type differently, but all tasks follow the same pattern: define what to run, specify where to run it, and configure any required parameters.

Create a job and add tasks

To create a new job in Azure Databricks:

1. In your workspace sidebar, select **Jobs & Pipelines**.
2. Select **Create**, then **Job**.
3. Enter a descriptive name for your job.
4. Configure your first task by specifying the **Task name** and selecting the **Type** (such as Notebook, Python script, or SQL).



The task type determines which configuration options appear. For a notebook task, you specify the notebook path and any parameters. For a SQL task, you select a query and SQL warehouse. The following table summarizes common task types and their configuration requirements:

Task type	Key configuration	Compute options
Notebook	Notebook path, parameters	Serverless, classic jobs, all-purpose
Python script	Script path, CLI arguments	Serverless, classic jobs, all-purpose
SQL	Query or file, SQL warehouse	Serverless SQL warehouse, pro SQL warehouse
Pipeline	Existing pipeline selection	Serverless or classic pipeline compute

After configuring a task, select **Create task** to add it to your job.

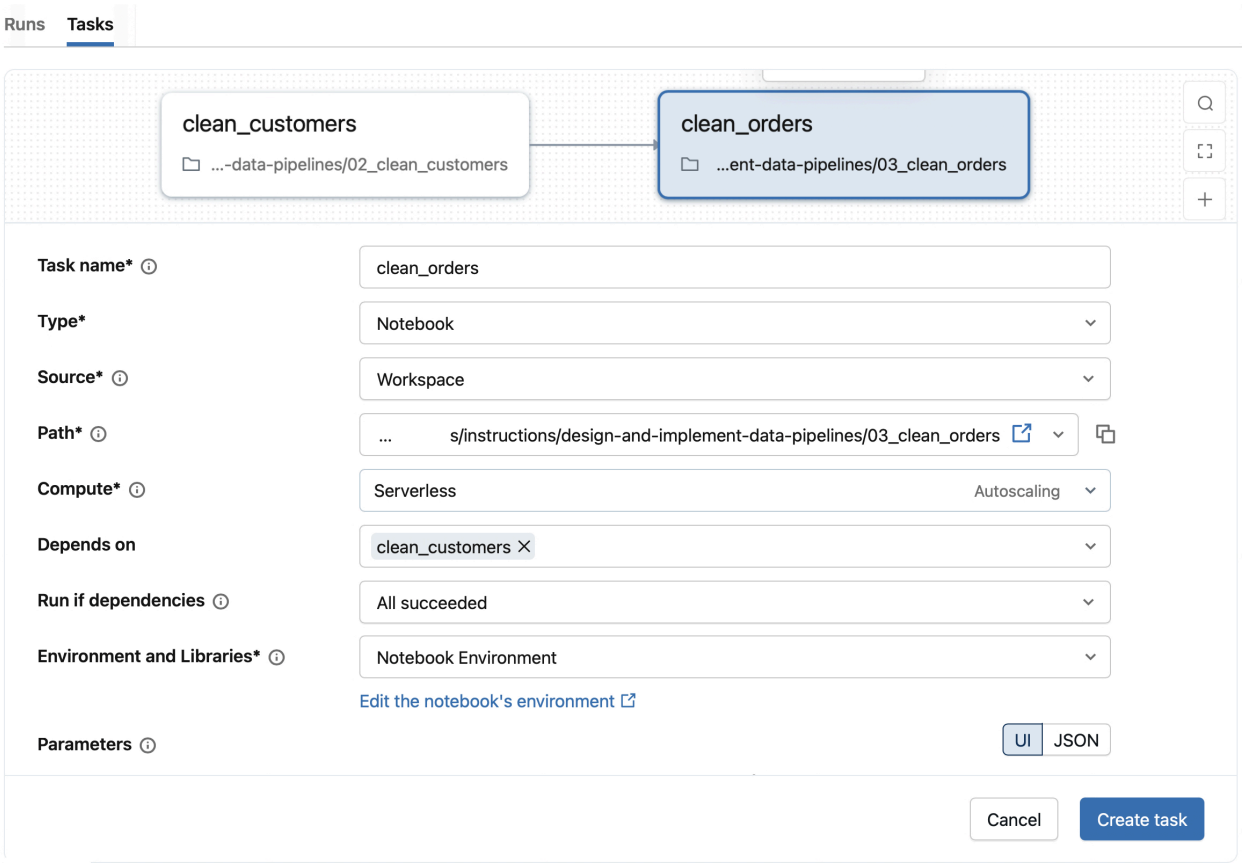
Configure task sources

Tasks that run code (notebooks, Python scripts, SQL files) need a source location. You have three options for specifying where your code lives:

Workspace stores code directly in your Azure Databricks workspace. Use the file browser to navigate to your notebook or script, then confirm your selection. This option works well for development and simple workflows.

Git provider connects to a remote repository. You specify the repository URL, branch or tag, and the relative path to your file. All tasks in a job share the same Git reference, ensuring consistent code versions across the workflow. When you use Git, Azure Databricks captures a snapshot of the code at run time, so your job executes against a specific commit.

DBFS/ADLS (for Python scripts) allows you to reference files stored in volumes or cloud storage. Provide the full URI, such as `abfss://container@storage.dfs.core.windows.net/path/script.py`.



Configure compute resources

Each task needs compute resources to execute. Azure Databricks offers several compute options optimized for different workloads.

Serverless compute is the default for supported task types. Azure Databricks manages the infrastructure, so you don't configure cluster settings. Serverless compute reduces operational overhead and scales automatically.

Classic jobs compute gives you control over cluster configuration. You specify the Spark version, instance types, and autoscaling policies. Use classic compute when you need specific configurations or libraries not supported by serverless.

SQL warehouses run SQL tasks. Select an existing serverless or pro SQL warehouse from your workspace.

When multiple tasks share the same compute resource, the cluster remains active until all tasks complete. Sharing compute reduces startup time between tasks but incurs cost during idle periods. You can balance this by grouping related tasks on shared compute while isolating resource-intensive operations.

To view and modify compute configuration:

1. Open your job and select the **Job details** panel.
2. Under **Compute**, review the resources assigned to each task.
3. Use **Configure** to modify classic jobs compute or **Swap** to change compute for all tasks using a resource.

Set up task dependencies

Jobs with multiple tasks use dependencies to control execution order. Dependencies create the DAG structure that determines which tasks run in sequence and which can run in parallel.

To add a dependency:

1. Select a task in the task graph.
2. In the **Depends on** field, select the upstream tasks that must complete first.
3. Choose a **Run if** condition to specify when the downstream task should execute.

The available run-if conditions let you handle various scenarios:

Condition	When the task runs
All succeeded	All upstream tasks completed successfully
At least one succeeded	Any upstream task succeeded
None failed	No upstream tasks failed (some may be skipped)
All done	All upstream tasks finished, regardless of outcome
At least one failed	At least one upstream task failed
All failed	All upstream tasks failed

Use **All done** for cleanup tasks that should run regardless of earlier results. Use **At least one failed** to trigger error-handling logic when problems occur.

Add job parameters

Parameters make your jobs reusable by letting you pass different values to each run. You define parameters at the job level, and they're automatically available to all tasks that accept key-value inputs.

To add job parameters:

1. In the **Job details** panel, locate the **Parameters** section.
2. Select **Add** and enter a key-value pair.

Tasks access parameters differently based on their type. In notebooks, use `dbutils.widgets.get("parameter_name")` to retrieve parameter values. Python scripts receive parameters as command-line arguments.

You can also reference dynamic values in parameters. For example, `{{job.trigger.time.iso_date}}` inserts the trigger date, useful for processing data based on when the job runs.

Organize with tags

Tags help you categorize and filter jobs in your workspace. Add tags as labels or key-value pairs to group related jobs by team, project, or environment.

To add tags:

1. In the **Job details** panel, select **+ Tag**.
2. Enter a key and optionally a value.

Tags also propagate to job clusters, enabling consistent monitoring and cost attribution across your organization.

Configure job access permissions

Job permissions control who can view, run, and manage your Lakeflow Jobs independently from compute permissions. Azure Databricks provides four permission levels:

Permission level	Capabilities
CAN VIEW	View job configuration, task definitions, and run history
CAN RUN	Everything in CAN VIEW plus trigger job runs manually

Permission level	Capabilities
CAN MANAGE RUN	Everything in CAN RUN plus cancel runs, view output, and restart failed runs
CAN MANAGE	Everything in CAN MANAGE RUN plus edit configuration, modify tasks, change schedules, and set permissions

To configure permissions, navigate to **Jobs & Pipelines**, select your job, open the **Permissions** tab, and add users or groups with the appropriate level. Job creators and workspace admins automatically receive `CAN MANAGE` permissions.

When a job runs, it executes with the job owner's permissions or the configured service principal's permissions—not the triggering user's permissions. For production jobs, grant `CAN MANAGE` to the pipeline team, `CAN RUN` to users who need manual execution, and `CAN VIEW` to stakeholders requiring visibility.

Configure run identity and Unity Catalog access

When your job accesses Unity Catalog objects—such as tables, views, or volumes—the job's **run identity** must have the required Unity Catalog privileges. This is a critical prerequisite before configuring any job that reads from or writes to Unity Catalog-managed data.

The run identity is the principal whose permissions Unity Catalog evaluates during job execution. By default, jobs run as the **job owner** (the user who created the job). For production workloads, you can configure a **service principal** as the run identity to avoid dependency on individual user accounts.

Before creating your job, verify that the run identity has the necessary privileges:

Operation	Required Unity Catalog privilege
Read from a table	`SELECT` on the table
Write to a table	`MODIFY` on the table
Create tables in a schema	`CREATE TABLE` and `USE SCHEMA` on the schema
Access a volume	`READ VOLUME` or `WRITE VOLUME` on the volume

To grant privileges to a service principal or user, use SQL commands like:

```
GRANT SELECT, MODIFY ON TABLE catalog.schema.table TO `service-principal-id`;
GRANT USE SCHEMA ON SCHEMA catalog.schema TO `service-principal-id`;
```

If the run identity lacks the required privileges, the job fails at runtime with an authorization error—even if the job configuration itself is valid. Always verify Unity Catalog access before scheduling production jobs.

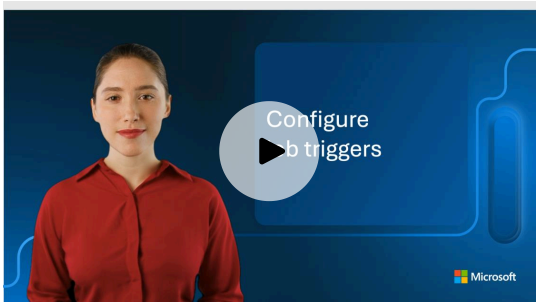
With your job created, tasks configured, dependencies set, and permissions assigned, you're ready to run your workflow. The next step is understanding how to monitor job execution and handle run outcomes.

3. Configure job triggers

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/3-configure-job-triggers

Configure job triggers

Completed

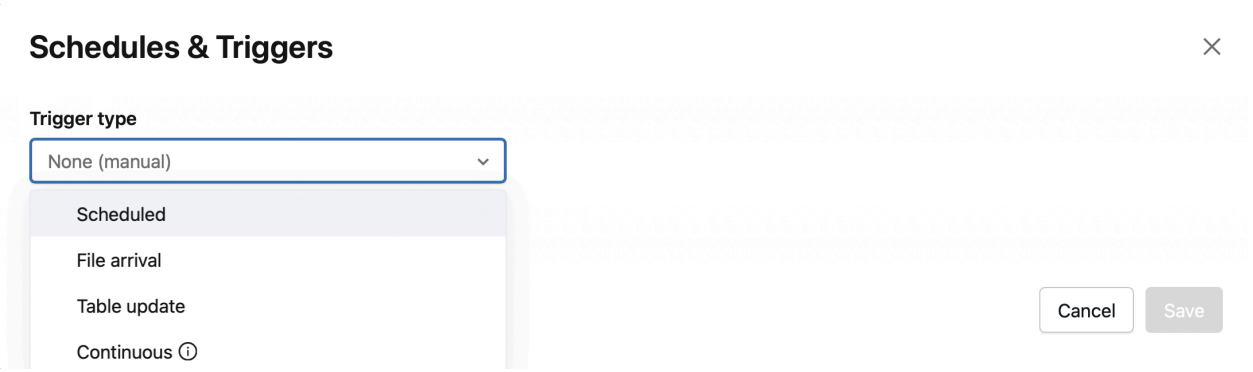


When you configure job triggers in Lakeflow Jobs, you enable automated pipeline execution based on data events rather than fixed schedules. Your jobs can respond to changes in source tables, new file arrivals, or run continuously to process data as it becomes available.

In this unit, you learn how to configure event-based triggers that automate your data workflows.

Understand trigger types for automated job execution

Lakeflow Jobs supports several trigger types that determine when your jobs run. While scheduled triggers run at fixed times, event-based triggers respond to data changes in your environment.



Trigger type	Behavior
Table update	Runs when monitored tables receive new data or modifications
File arrival	Runs when new files appear in a Unity Catalog storage location
Continuous	Runs immediately after the previous run completes or fails

Each trigger type addresses different automation scenarios. Table update triggers work well when your job depends on upstream data processing. File arrival triggers handle scenarios where external systems drop files into storage locations. Continuous triggers keep your jobs running around the clock for near real-time processing.

To add a trigger to an existing job, open the job in your workspace, navigate to the **Job details** pane, and select **Add trigger** under **Schedules & Triggers**. From there, you select the trigger type and configure its settings.

Configure table update triggers

Table update triggers run your job when source tables receive updates. This approach eliminates the need for continuously running clusters while ensuring your pipeline processes new data promptly.

A table update trigger monitors one or more Unity Catalog tables for data changes such as inserts, updates, merges, and deletes. The supported table types include:

- Unity Catalog Delta and Iceberg managed tables
- Unity Catalog external tables backed by Delta Lake
- Materialized views and streaming tables
- Unity Catalog views that depend on supported tables

When you configure a table update trigger with multiple tables, you choose between two behaviors. Select **Any table is updated** to run the job when any monitored table changes. Select **All tables are updated** to wait until every monitored table receives an update before triggering a run.

Consider a scenario where your transformation job depends on data from three source tables. If your business logic requires all three sources to be current before processing, configure the trigger with **All tables are updated**. If processing partial updates is acceptable, use **Any table is updated** for faster response times.

The following advanced options give you finer control over trigger timing:

- **Minimum time between triggers**: Sets a waiting period after a job completes before the next run can start, even if tables are updated during this window
- **Wait after last change**: Delays the trigger for a specified time after a table update, resetting if additional updates occur

These settings help you batch updates efficiently. For example, if your upstream processes update tables in rapid succession, a 60-second wait after last change ensures your job processes the complete batch rather than triggering multiple runs.

When you add a table update trigger, your job gains access to dynamic parameter values that provide context about what triggered the run:

```
{{job.trigger.table_update.updated_tables}}
{{job.trigger.table_update.<catalog.schema.table>.commit_timestamp.iso_datetime}}
{{job.trigger.table_update.<catalog.schema.table>.version}}
```

You can pass these values to your tasks to customize processing based on which tables changed or to track data lineage.

> **Tip**
>
> For optimal performance with table update triggers, enable file events on the external location where your tables are stored. This one-time setup step uses cloud provider change notifications to track updates more efficiently.

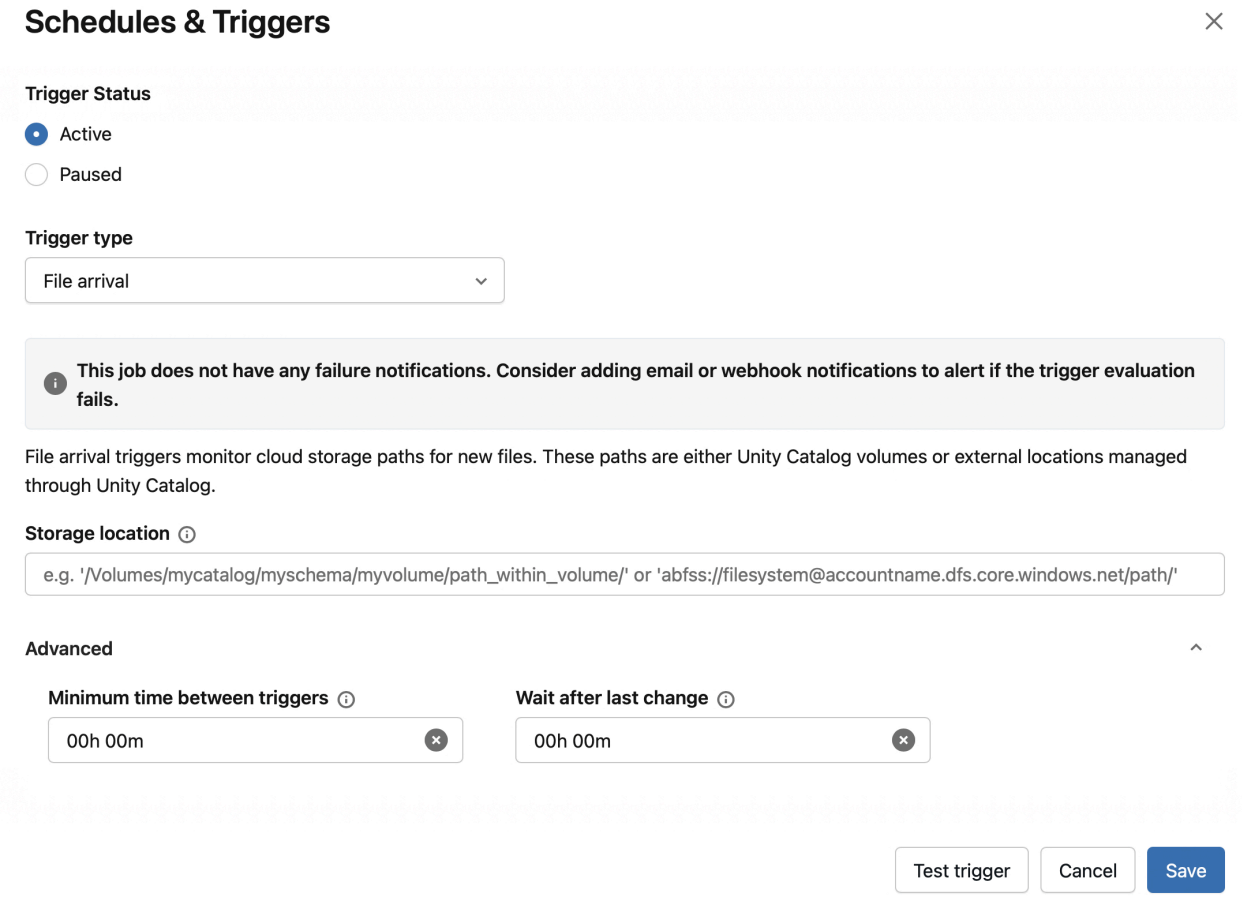
Configure file arrival triggers

File arrival triggers run your job when new files appear in a Unity Catalog external location or volume. This trigger type integrates well with scenarios where external systems deliver data files on irregular schedules.

A file arrival trigger monitors a specified path and recursively checks all subdirectories for new files. The trigger makes a best-effort check approximately every minute, though cloud storage performance can affect timing.

To configure a file arrival trigger, specify the storage location using the full path to an external location or volume:

```
/Volumes/mycatalog/myschema/myvolume/
/Volumes/mycatalog/myschema/myvolume/incoming/
```



Before creating a file arrival trigger, verify you have:

- A workspace with Unity Catalog enabled
- READ permission on the storage location
- CAN MANAGE permission on the job

Like table update triggers, file arrival triggers support advanced timing options. Use **Minimum time between triggers** to control run frequency, and **Wait after last change** to batch multiple file arrivals into a single processing run.

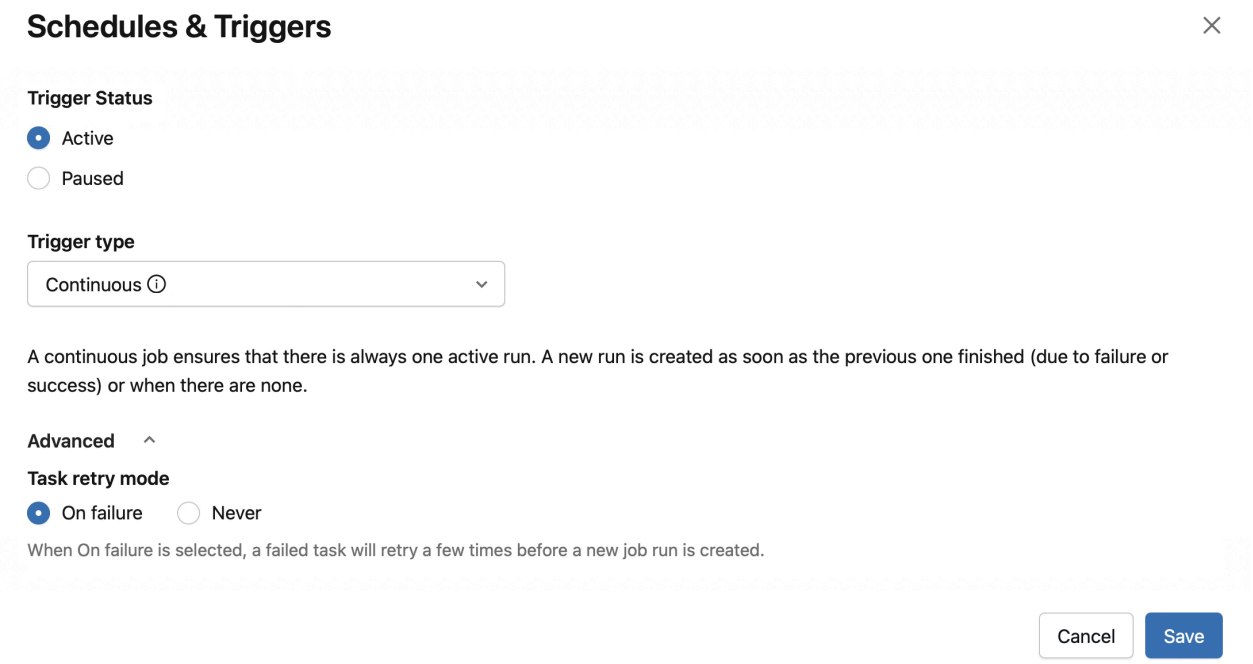
> **Important**
>
> File arrival triggers respond only to new files. Overwriting an existing file with the same name doesn't trigger a run. Design your file delivery processes to use unique file names for each batch.

For production workloads, enable file events on your external location. Without file events enabled, you're limited to 50 jobs with file arrival triggers per workspace, and each monitored location can contain at most 10,000 files. With file events enabled, these limitations don't apply.

Configure continuous triggers

Continuous triggers keep your job running by starting a new run immediately after the previous one completes or fails. This approach suits scenarios requiring near real-time data processing.

When you configure a continuous trigger, your job enters a perpetual cycle: complete a run, start the next run. If a run fails, the trigger still starts a new run, making continuous jobs resilient to transient errors.



Continuous triggers work well for:

- Processing streaming data where low latency matters
- Monitoring systems that need constant operation
- ETL workflows that must keep pace with high-volume data sources

Before enabling a continuous trigger, consider the cost implications. Your job continuously consumes compute resources, which can add up over time. Use serverless compute to optimize costs, as it scales automatically based on workload.

> **Note**
>
> When you resume a paused continuous trigger while a run is still active, the scheduler waits for that run to complete before triggering a new one.

Manage trigger lifecycle

After you configure triggers, you can pause and resume them without deleting the configuration. In the **Job details** pane under **Schedules & Triggers**, the **Pause** and **Resume** buttons control trigger state.

Pausing a trigger stops new runs from starting while allowing any active run to complete. This capability is useful during maintenance windows or when you need to investigate issues without losing your trigger configuration.

To receive notifications when trigger evaluation fails, configure email or system destination notifications on your job. This approach ensures you're informed when triggers encounter problems such as permission changes or table deletions.

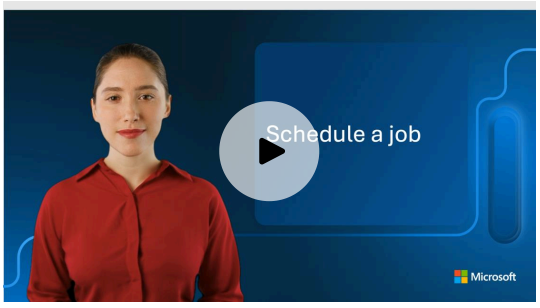
Now that you understand how to configure triggers that automate job execution based on data events, you can build responsive pipelines that process data as it becomes available.

4. Schedule a job

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/4-schedule-job

Schedule a job

Completed



When you schedule a Lakeflow Job, you define exactly when your workflow runs without manual intervention. Scheduling transforms ad hoc data processing into reliable, automated pipelines that keep your data fresh and your business processes running smoothly.

Azure Databricks provides two scheduling approaches. Simple schedules handle recurring intervals like "every 12 hours," while advanced schedules give you precise control through cron expressions that can target specific days, times, and even months.

Create a simple schedule

Simple schedules work well when you need consistent intervals between job runs. You specify a number and a time unit—minutes, hours, days, or weeks—and Azure Databricks handles the rest.

To add a simple schedule to your job:

1. In your Azure Databricks workspace sidebar, select **Jobs & Pipelines**.

2. Select your job's name to open its details.
3. In the **Job details** panel, select **Add trigger**.
4. Set **Trigger type** to **Scheduled**.
5. Set **Schedule type** to **Simple**.
6. Specify your interval (for example, 12 hours).
7. Select **Save**.

With simple schedules, you can't control when the first run happens—the scheduler picks a time when you save the configuration. Each subsequent run then follows your specified interval.

> **Note**
>
> Azure Databricks enforces a minimum interval of 10 seconds between job runs, regardless of your configuration.

Configure an advanced schedule

Advanced schedules use cron expressions to define precisely when your job runs. This approach handles complex requirements like "at 8:00 AM every weekday" or "on the last Friday of each month."

To create an advanced schedule:

1. In the Job details panel, select **Add trigger**.
2. Set **Trigger type** to **Scheduled**.
3. Set **Schedule type** to **Advanced**.
4. Specify the period, starting time, and time zone.
5. Optionally, select **Show Cron Syntax** to view and edit the expression directly.
6. Select **Save**.

Understand cron expression format

A cron expression contains six or seven fields separated by spaces:

Field	Required	Allowed values	Special characters
Seconds	Yes	0-59	`, - * /`
Minutes	Yes	0-59	`, - * /`
Hours	Yes	0-23	`, - * /`
Day of month	Yes	1-31	`, - * ? / L W`
Month	Yes	1-12 or JAN-DEC	`, - * /`
Day of week	Yes	1-7 or SUN-SAT	`, - * ? / L #`

Field	Required	Allowed values	Special characters
Year	No	empty, 1970-2099	`, - * /`

The special characters let you build flexible schedules:

- `*` selects all values in a field ("every minute" or "every hour")
- `?` means "no specific value"—use it in day-of-month or day-of-week when you specify the other
- `-` defines ranges like `10-12` for hours 10, 11, and 12
- `,` lists multiple values like `MON,WED,FRI`
- `/` specifies increments— `0/15` in minutes means every 15 minutes starting at 0
- `L` means "last"—in day-of-month, it's the last day; `6L` in day-of-week means "last Friday"
- `#` specifies "nth occurrence"— `6#3` means "third Friday of the month"

Common cron expression examples

These expressions cover typical scheduling scenarios:

Expression	Schedule
`0 0 12 * * ?`	Every day at noon
`0 15 10 ? * MON-FRI`	Weekdays at 10:15 AM
`0 0 8 1 * ?`	First day of every month at 8:00 AM
`0 30 6 ? * 6L`	Last Friday of every month at 6:30 AM
`0 0 */2 * * ?`	Every 2 hours

Choose the right time zone

Time zone selection affects when your scheduled jobs actually run. Azure Databricks lets you choose any time zone, but your choice has implications for consistency.

Time zones that observe daylight saving time (DST) create complications. When DST begins or ends, an hourly job might skip a run or appear delayed by an hour or two. If your business logic depends on consistent hourly execution—measured in absolute time—select UTC instead.

Consider these factors when choosing a time zone:

- **Business alignment**: If stakeholders expect reports at 8:00 AM local time, schedule in their time zone and accept DST shifts.
- **Data consistency**: If you process data from systems using UTC timestamps, schedule in UTC to maintain alignment.
- **Cross-region coordination**: When jobs must run simultaneously across regions, UTC provides a single reference point.

> **Tip**
>
> For jobs that must run at exact intervals regardless of local time changes, always use UTC.

Control concurrent job runs

When scheduled jobs take longer than expected, a new run might start before the previous one finishes. This overlap can cause data corruption, duplicate processing, or resource contention. Azure Databricks provides concurrency settings to control this behavior.

Configure maximum concurrent runs

The **Maximum concurrent runs** setting limits how many instances of the same job can execute simultaneously. By default, jobs allow multiple concurrent runs. For jobs that must not overlap—such as those writing to the same tables—set this value to **1**.

To configure maximum concurrent runs:

1. Open your job in the **Jobs & Pipelines** workspace UI.
2. In the **Job details** panel, locate the **Maximum concurrent runs** setting.
3. Set the value to control how many runs can execute at once.

When a new run is triggered but the maximum concurrent runs limit is reached, Azure Databricks must decide what to do with the incoming run.

Configure queue behavior for overlapping runs

When concurrent runs exceed your configured limit, you choose how the scheduler handles the new run:

Behavior	Description	Use case
Queue the run	The new run waits until a slot becomes available, then executes	Jobs that must eventually run—no triggers should be missed
Cancel the run	The new run is immediately canceled	Jobs where stale triggers are not valuable
Skip the run	Similar to cancel—the run doesn't execute	Jobs where missing occasional runs is acceptable

For most data pipelines, **queue the run** ensures that all scheduled executions eventually complete. This approach prevents data gaps when a job occasionally runs longer than its schedule interval.

Consider a job scheduled to run every hour. If a run takes 75 minutes to complete, the next scheduled trigger arrives while the job is still running. With concurrency set to 1 and queue enabled:

1. The first run continues processing.

2. The second run enters the queue.
3. When the first run completes, the queued run starts immediately.

This pattern ensures sequential, non-overlapping execution while preserving all scheduled runs.

> **Note**
>
> Queued runs consume no compute resources while waiting. They only start when a concurrent slot becomes available.

Scheduling considerations for production workloads

The Azure Databricks job scheduler handles most scenarios reliably, but it's not designed for low-latency requirements. Network conditions or cloud service issues can occasionally delay job starts by several minutes. When service recovers, scheduled jobs run immediately.

Keep these points in mind:

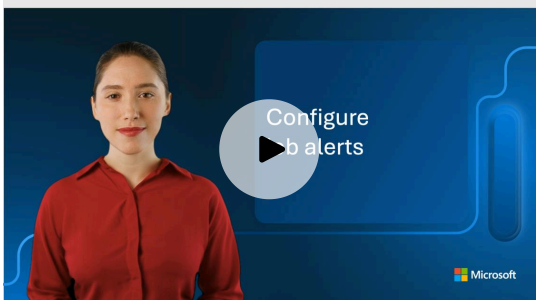
- Schedule jobs with buffer time between dependent pipelines
- Don't rely on sub-minute precision for business-critical timing
- Monitor job run history to identify scheduling patterns or delays
- Use the minimum 10-second interval rule when planning high-frequency jobs

5. Configure job alerts

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/5-configure-job-alerts

Configure job alerts

Completed



When your data pipelines run in production, you need to know when something goes wrong—or even when everything goes right. Configuring **job alerts** in Azure Databricks gives you visibility into

job execution without constantly monitoring dashboards. Alerts enable you to respond quickly to failures, track long-running jobs, and maintain confidence in your data workflows.

In this unit, you learn how to configure notifications for jobs and tasks, choose appropriate alert destinations, and apply best practices that keep your alerts actionable.

Understand notification events

Azure Databricks can notify you when specific events occur during job execution. Each event type serves a different operational purpose.

Event type	When it fires	Use case
Start	Job run begins	Track job initiation, confirm scheduled jobs started
Success	Job completes successfully	Verify pipeline completion, trigger downstream processes
Failure	Job terminates with error	Investigate issues, initiate incident response
Duration warning	Run exceeds configured threshold	Identify performance degradation, catch runaway jobs
Streaming backlog	Backlog exceeds threshold for 10 minutes	Monitor streaming job health

When a job completes in a **Succeeded with failures** state—meaning some tasks failed but the job itself finished—Databricks considers this a success. If you need visibility into partial failures, configure your Success notifications accordingly or add task-level notifications.

With these event types in mind, you can now configure where these notifications should go.

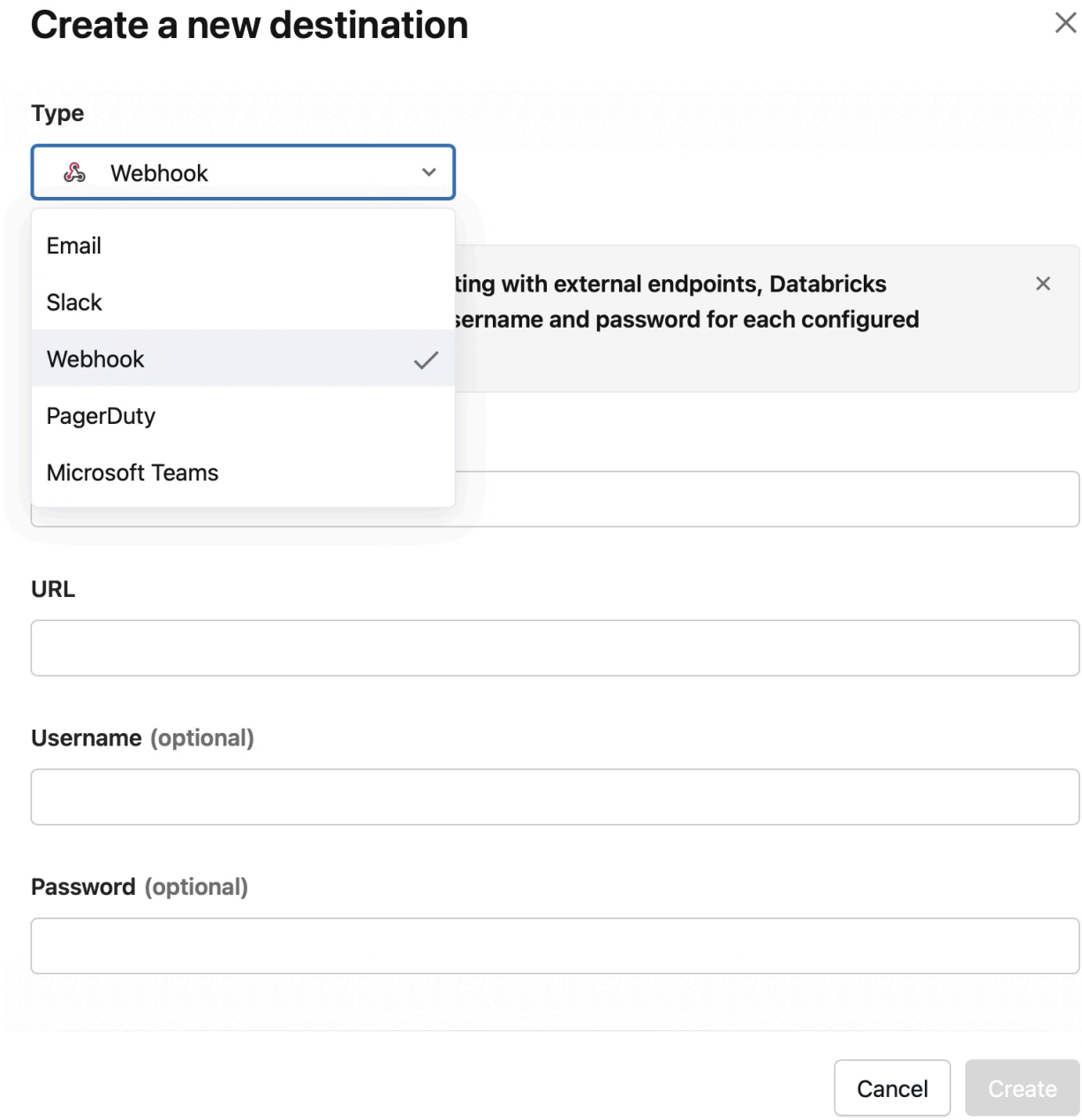
Configure notification destinations

Notifications can go to email addresses or third-party systems. Third-party **system destinations** integrate with tools your team already uses for incident management and collaboration.

Supported system destinations include:

- **Slack**: Channel notifications for team visibility
- **Microsoft Teams**: Integration with your collaboration platform
- **PagerDuty**: Incident management and on-call escalation
- **HTTP webhooks**: Custom integrations with any service that accepts HTTP requests

Before you can use system destinations, a workspace administrator must configure them. Navigate to **Admin Settings** and select **Notifications** to create destinations. Each destination requires appropriate credentials—webhook URLs for Slack and Teams, integration keys for PagerDuty.



Tip

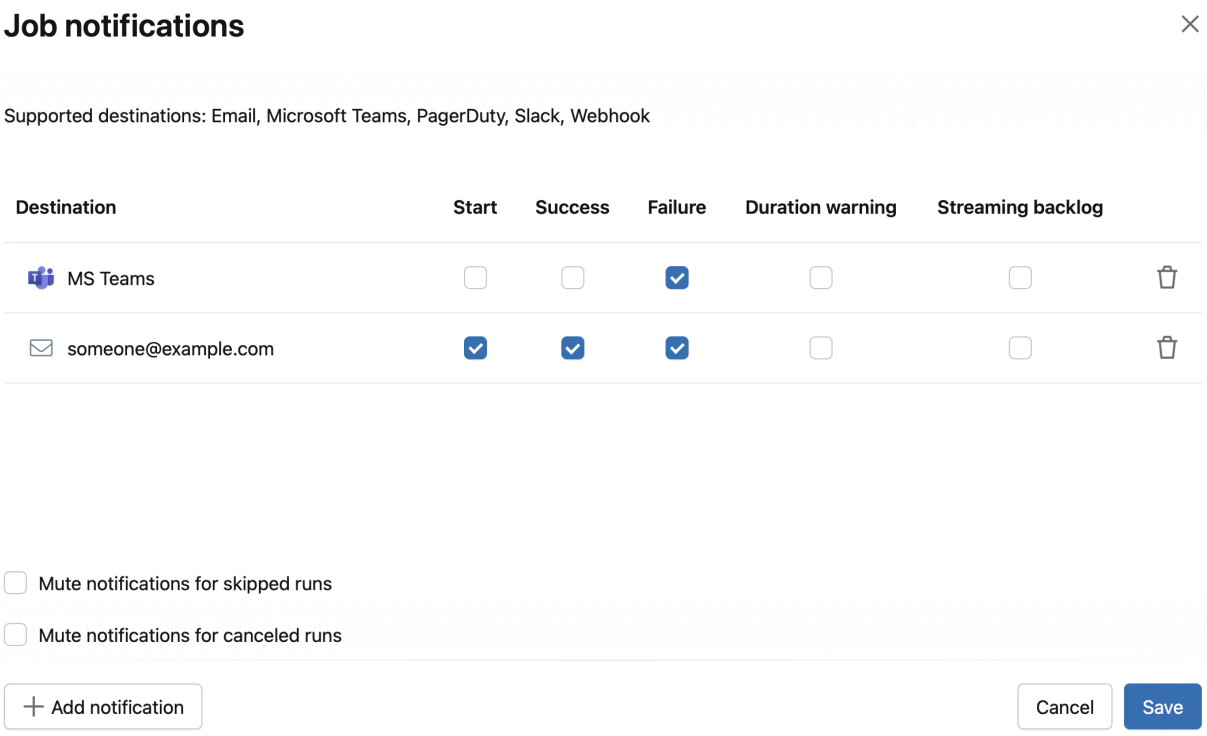
Use different credentials for each configured destination. If one third-party endpoint is compromised, you can revoke its access without affecting other notification destinations.

For each job or task, you can configure up to three system destinations per event type. This limit encourages thoughtful notification design rather than flooding multiple channels.

Add notifications to a job

To configure job-level notifications, follow these steps:

1. Open your job in the Azure Databricks workspace.
2. In the **Job details** panel, locate the **Job notifications** section.
3. Select **Edit notifications**.
4. Select **Add notification**.
5. Choose a destination type—either **Email address** or one of your configured system destinations.
6. Select the event types you want to trigger notifications: **Start**, **Success**, **Failure**, **Duration warning**, or **Streaming backlog**.
7. Select **Save** when you finish adding notifications.



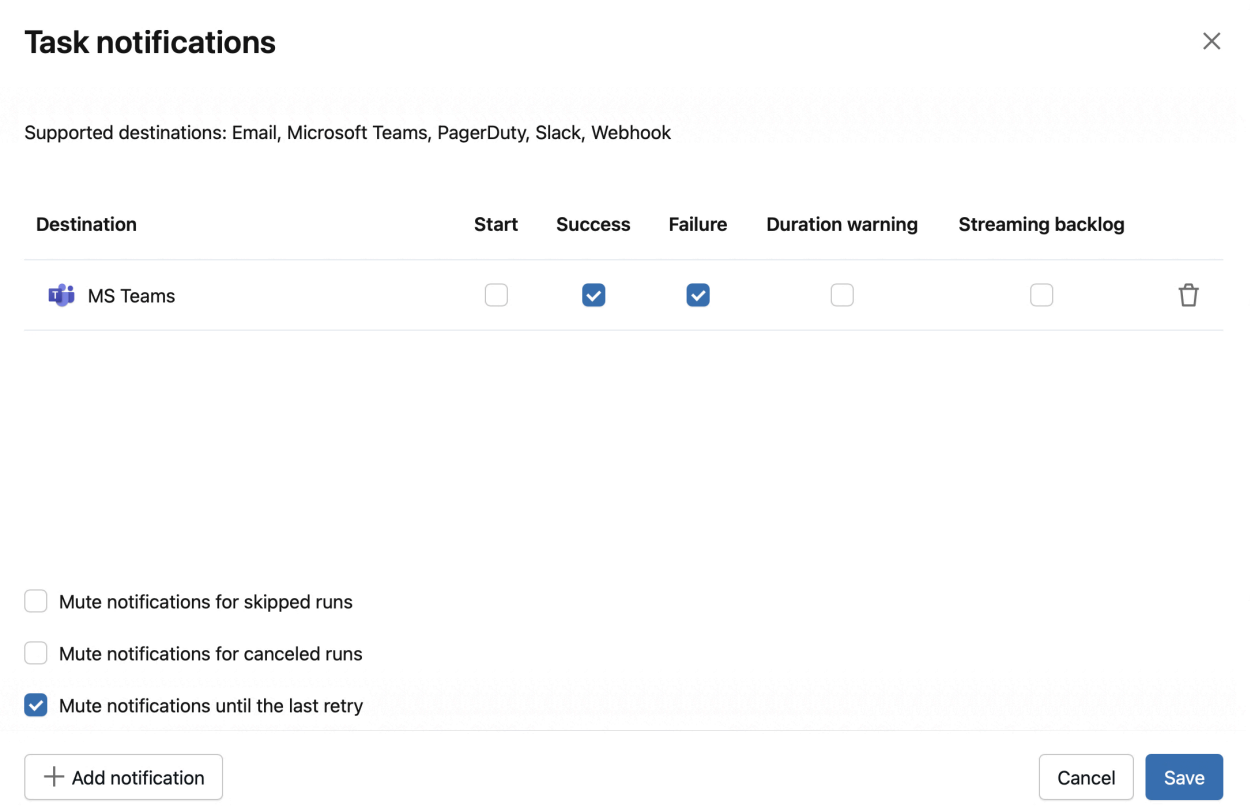
Job-level notifications apply to the overall job run. However, job-level notifications aren't sent when individual tasks fail and retry. Consider this behavior when designing your notification strategy.

Configure task-level notifications

For more granular visibility, configure notifications on individual tasks within your job. Task notifications fire for each task run, giving you insight into which specific step encountered issues.

To add task notifications:

1. Edit your job and select the task you want to monitor.
2. In the task panel, locate **Notifications** and select **Add**.
3. Configure the destination and event types just as you would for job notifications.
4. Save the task configuration.

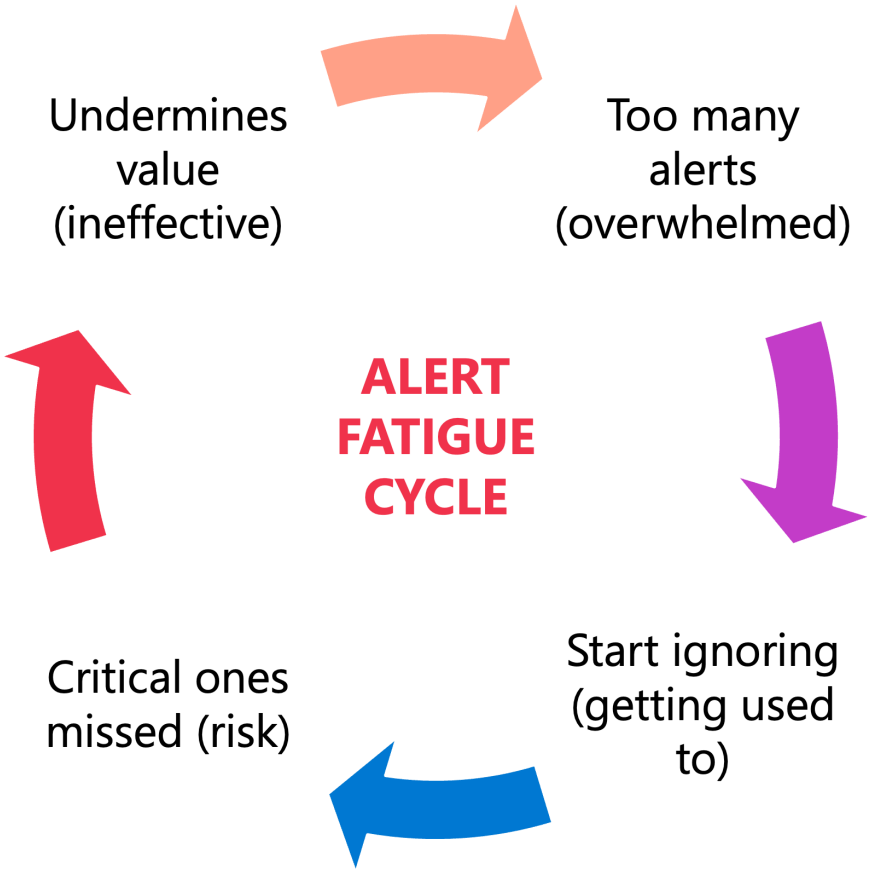


Task notifications are particularly valuable when your job contains multiple independent tasks. If Task A fails, you receive an immediate notification rather than waiting for the entire job to complete or fail.

By default, Azure Databricks retries failed tasks three times. If you don't want notifications for every retry attempt, select **Mute notifications until the last retry**. This reduces noise while still alerting you when a task ultimately fails.

Reduce notification noise

Alert fatigue undermines the value of notifications. When teams receive too many alerts, they start ignoring them—including the critical ones. Apply these strategies to keep your alerts meaningful.



Filter out skipped and canceled runs: When you cancel a job or a run gets skipped due to concurrent run limits, you might not need a notification. Select **Mute notifications for skipped runs** or **Mute notifications for canceled runs** to suppress these.

Use duration warnings strategically: Rather than alerting on every long-running job, set duration thresholds based on historical performance. A job that usually takes 30 minutes might warrant a warning at 45 minutes—not at 35.

Separate task and job notifications: If you enable job failure notifications, you don't necessarily need task failure notifications for every task. Choose based on whether you need to know *that* a job failed or *which task* failed.

Document your alert logic: Record why each alert exists, what conditions trigger it, and what action the recipient should take. This documentation helps when troubleshooting alerts or onboarding new team members.

Monitor with HTTP webhooks

When built-in integrations don't meet your needs, HTTP webhooks provide flexibility. Azure Databricks sends a JSON payload to your endpoint when events occur, enabling custom processing, logging, or integration with internal systems.

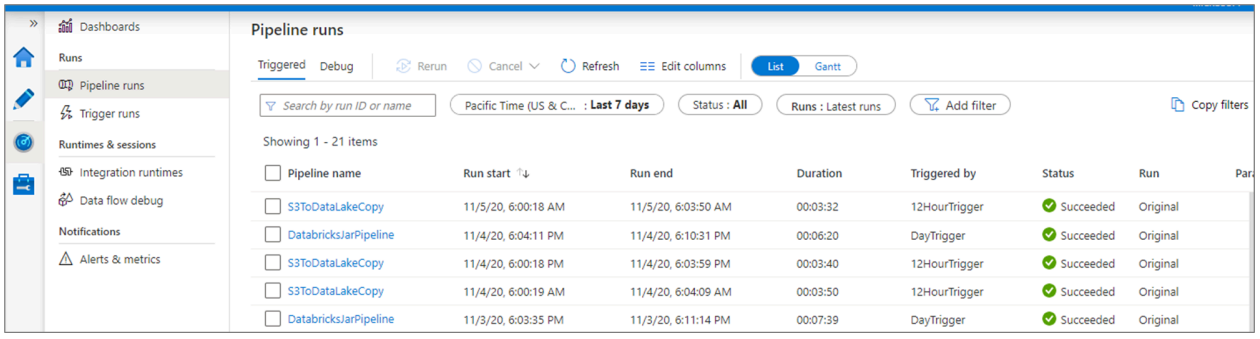
Webhook payloads include the event type, workspace ID, job information, and run details:

```
{
  "event_type": "jobs.on_failure",
  "workspace_id": "your_workspace_id",
  "run": {
    "run_id": "run_id"
  },
  "job": {
    "job_id": "job_id",
    "name": "job_name"
  }
}
```

If you require specific formatting for notifications, webhooks let you control the message structure completely. This approach is useful when integrating with monitoring platforms that have particular schema requirements.

Monitor pipelines in Azure Data Factory

When your Databricks jobs are orchestrated by Azure Data Factory (ADF), you have additional monitoring options. ADF provides visual monitoring in the Azure portal where you can track pipeline runs, activity status, and execution duration.



ADF also supports creating alerts on metrics such as:

- Pipeline run status (failed, succeeded, canceled)
- Activity run counts
- Integration Runtime utilization

To configure ADF alerts, navigate to **Monitor** > **Alerts & metrics** in the Data Factory portal. Create alert rules that specify conditions, severity levels, and action groups for notifications via email, SMS, or other channels.

This layered approach—Databricks notifications for job-level events and ADF alerts for orchestration-level metrics—gives you comprehensive visibility across your data platform.

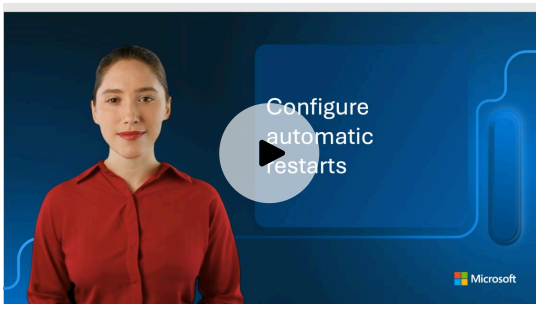
With alerts configured at the right granularity and sent to the right channels, you can respond to issues quickly while avoiding the distraction of unnecessary notifications. As you deploy more jobs to production, revisit your alerting strategy to ensure it scales with your workload.

6. Configure automatic restarts

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/6-configure-job-automatic-restarts

Configure automatic restarts

Completed



When a production job fails at 3 AM, **automatic restarts** can mean the difference between a quick recovery and hours of downtime. Network blips, temporary resource constraints, and **transient service interruptions** happen regularly in distributed systems. Configuring your jobs to handle these failures automatically reduces manual intervention and keeps your data pipelines running reliably.

In this unit, you learn how to configure automatic restarts and retry policies for Lakeflow Jobs in Azure Databricks.

Understand retry behavior in Lakeflow Jobs

Lakeflow Jobs provides multiple retry mechanisms to handle failures at different levels. The behavior depends on your job configuration and trigger type.

For most job configurations, the default setting doesn't retry tasks on failure. You must explicitly configure retry policies to enable automatic restarts. However, two scenarios behave differently:

- **Serverless jobs** auto-optimize retries by default to ensure critical workloads complete successfully

- **Continuous jobs** use an **exponential backoff** retry policy that automatically retries the entire job on failure

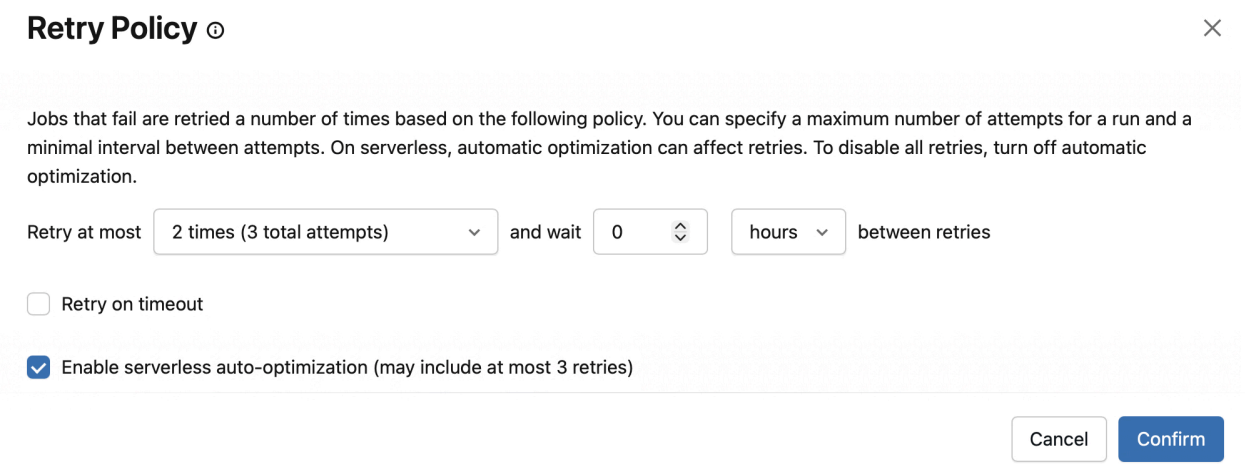
Understanding these defaults helps you choose the right configuration for your workload. Jobs that must execute exactly once, such as **non-idempotent operations**, require disabling auto-optimization. Jobs that can safely retry benefit from automatic restart policies.

Configure task-level retry policies

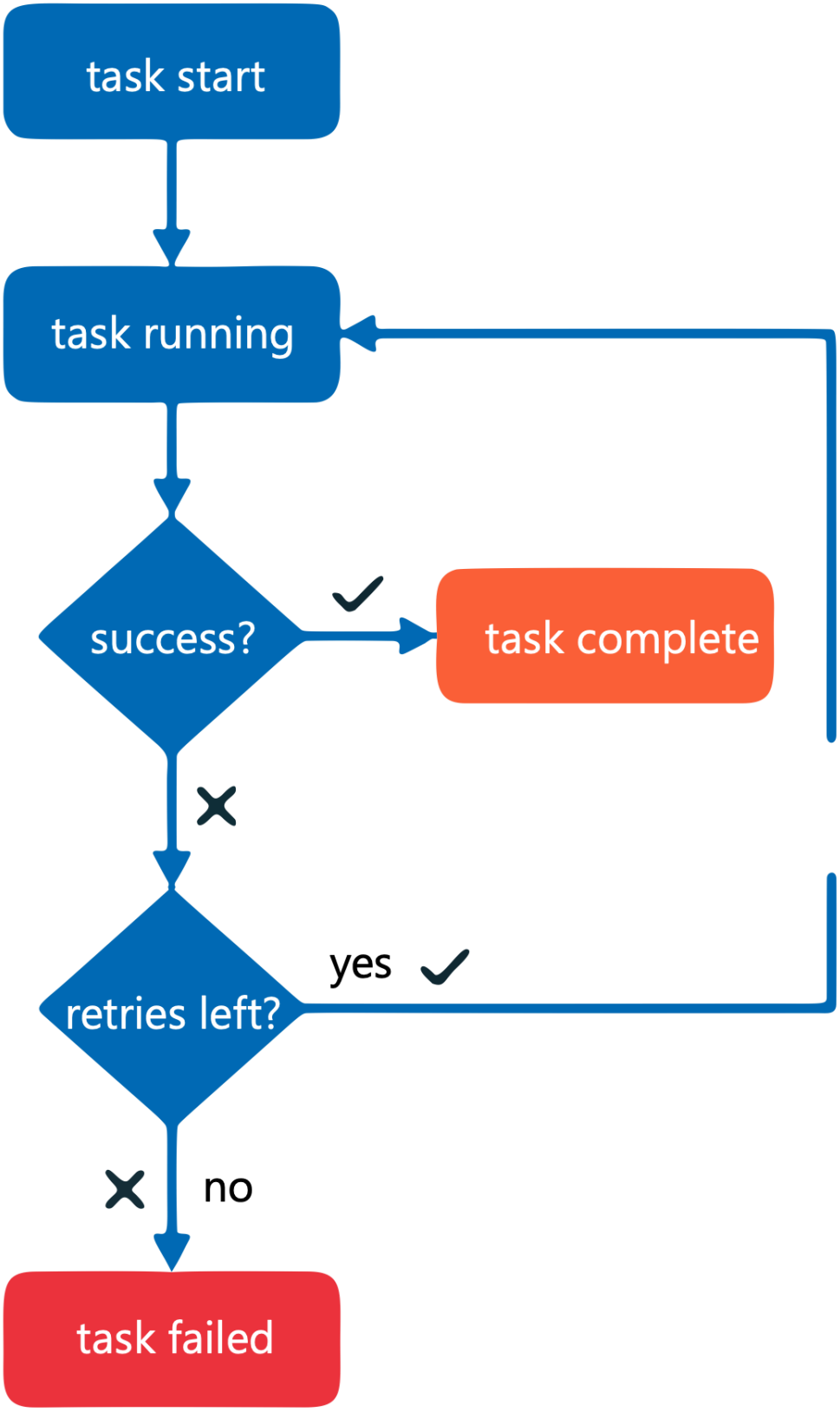
You configure retry policies at the task level in the Lakeflow Jobs UI. This approach gives you granular control over how individual tasks respond to failures.

To set a retry policy for a task:

1. Open your job in the **Jobs & Pipelines** workspace UI
2. Select the **Tasks** tab and select the task you want to configure
3. In the task configuration panel, select **+ Add** next to **Retries**
4. Specify the number of retry attempts and the interval between retries



The **retry interval** is calculated in milliseconds between the start of the failed run and the subsequent retry run. For example, if you set an interval of 60,000 milliseconds (1 minute) and your task took 30 seconds before failing, the next retry starts 30 seconds after the failure.


task start
task running
success?
task complete
retries left?
yes
no
task failed

Tip

> Set reasonable retry limits—typically 1 to 3 retries for most workloads. Avoid configuring unlimited retries, as a persistent failure will waste resources without resolving the underlying issue.

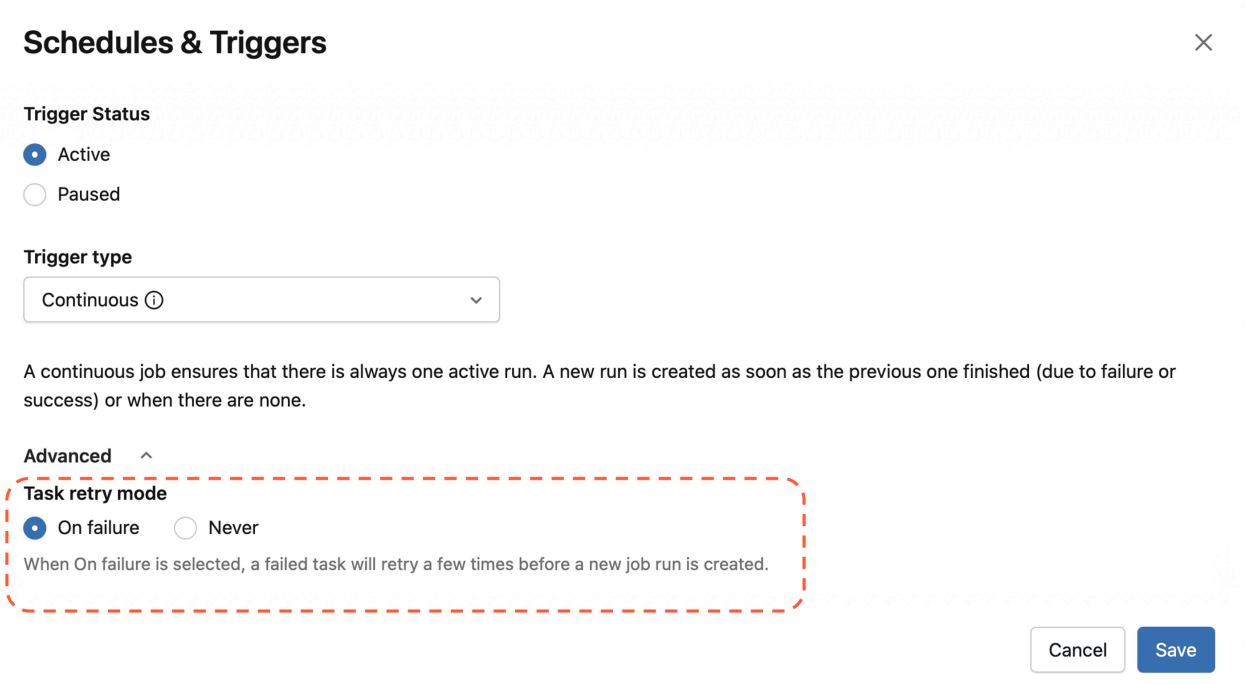
When you configure both timeout and retry settings, the **timeout applies to each retry attempt**. If a task times out, it counts as a failed attempt and triggers a retry if attempts remain.

Configure continuous jobs for always-on workloads

Continuous mode keeps jobs running constantly, making it ideal for **streaming workloads** that process data as it arrives. When a continuous job fails, the platform automatically handles restarts using exponential backoff.

To configure a job to run in continuous mode:

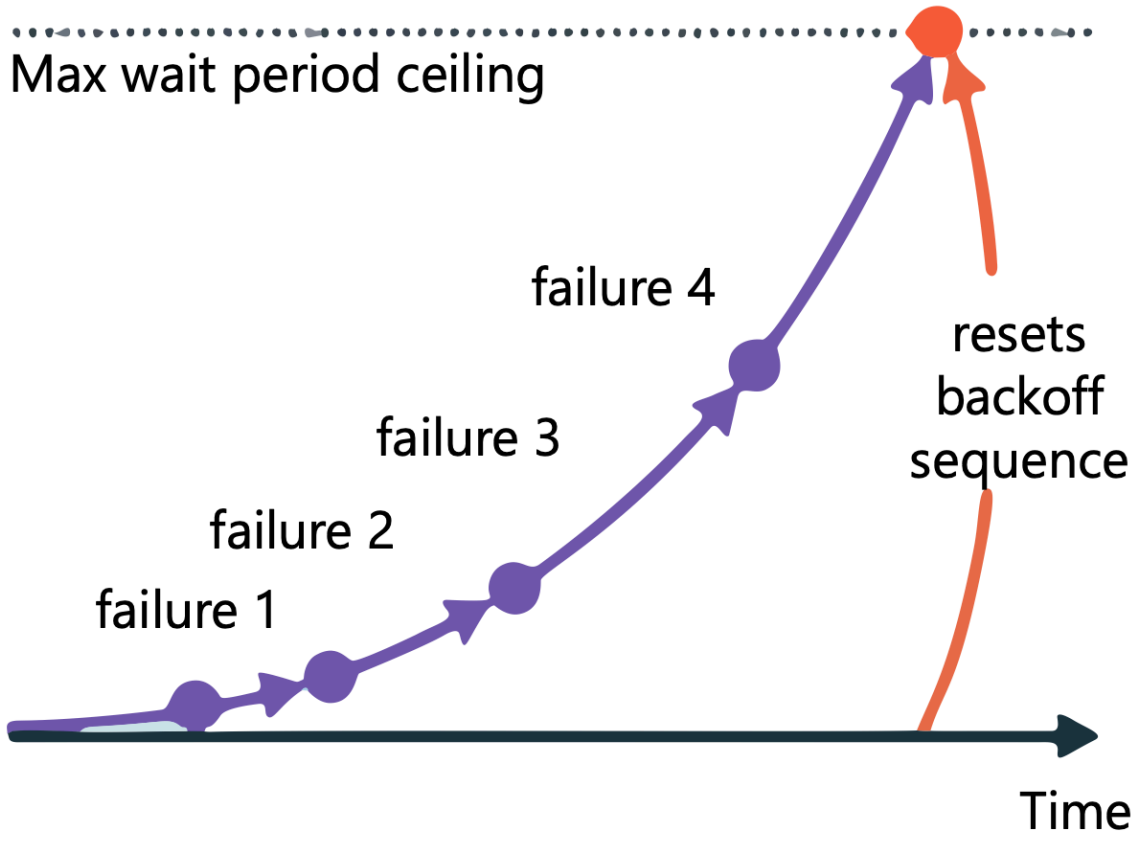
1. In the **Jobs & Pipelines** sidebar, open your job
2. In the **Job details** panel, select **Add trigger**
3. Select **Continuous** in the **Trigger type** dropdown
4. Optionally, select a **Task retry mode**—choose **On failure** to retry failed tasks within a job, or **Never** to only retry at the job level
5. Select **Save**



The **exponential backoff algorithm** works as follows:

1. When consecutive failures exceed a threshold, the job waits before the next retry
2. Each subsequent failure increases the wait period up to a maximum set by the system

3. If a run completes successfully or runs without failure for a threshold period, the **backoff sequence resets**



> **Note**
>
> Continuous jobs don't support task dependencies or task-level retry policies in the traditional sense. Instead, set the **Task retry mode** to control whether failed tasks retry before triggering a new job run.

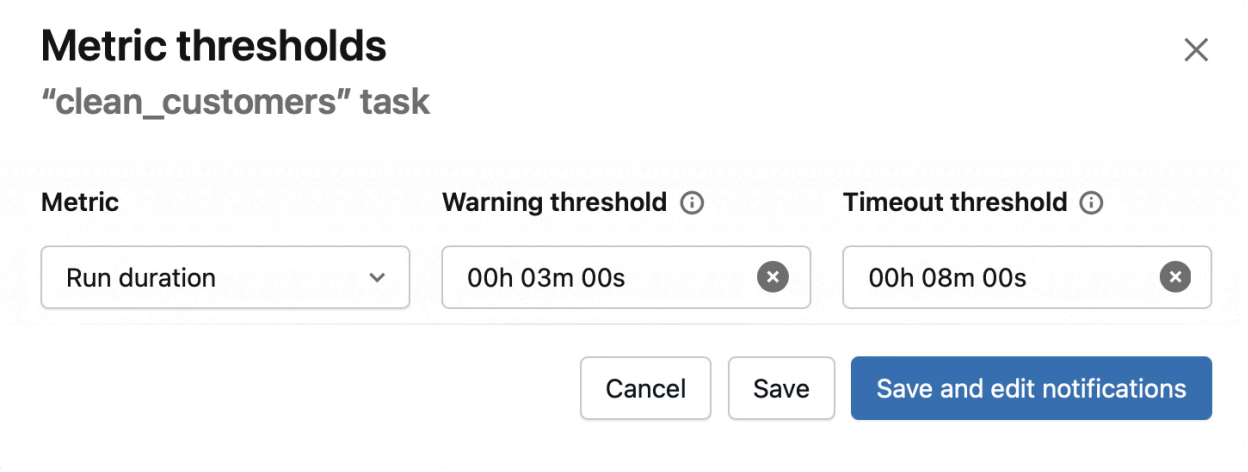
You can monitor continuous jobs in the exponential backoff state through the **Job details** panel, which displays the number of consecutive failures, the success threshold period, and the time before the next retry.

Set timeout thresholds to prevent hung tasks

A task that hangs indefinitely blocks downstream processing and wastes compute resources. **Timeout thresholds** terminate unresponsive tasks so your job can either retry or fail cleanly.

To configure timeout thresholds:

1. In the task configuration panel, select **Metric thresholds**
2. Select **Run duration** in the **Metric** dropdown
3. Enter a duration in the **Warning** field to trigger a notification when the task exceeds expected completion time
4. Enter a duration in the **Timeout** field to terminate the task if it exceeds maximum completion time



When a task times out, Azure Databricks sets its status to "Timed Out" and handles it according to your retry policy. If retries remain, the task restarts. If all retries are exhausted, the task fails and any dependent tasks are affected based on your job's dependency configuration.

Combine automatic restarts with notifications

Automatic restarts work best when combined with alerting. While restarts handle transient failures, persistent issues require human investigation.

Configure notifications to alert your team when:

- A task exceeds its expected duration (warning threshold)
- A task fails after exhausting all retry attempts
- A continuous job enters exponential backoff state

When configuring task-level notifications with retries enabled, select **Mute notifications until the last retry** to prevent **alert fatigue**. This setting ensures you're only notified after all retry attempts fail.

Apply best practices for production workloads

Effective automatic restart configuration balances resilience with efficiency. Consider these guidelines when configuring your jobs:

Match retry strategy to failure type: Transient failures like network timeouts benefit from quick retries with short intervals. Resource-related failures might need longer intervals to allow systems to recover.

Set reasonable limits: A maximum of 3 retries with intervals of 30 seconds to 5 minutes works well for most transient failures. Avoid unlimited retries—if a job fails repeatedly for the same reason, automatic restarts won't help.

Monitor retry patterns: Track how often jobs use retries. A gradual increase in retry usage might indicate underlying infrastructure issues or data quality problems that need attention.

Design for idempotency: Jobs that can safely re-execute produce the same results regardless of how many times they run. **Idempotent operations** make automatic restarts safe and predictable.

Document your configuration: Record why specific retry limits and intervals were chosen. This context helps future maintainers understand the tradeoffs and adjust settings as requirements change.

7. Exercise - Implement Lakeflow Jobs with Azure Databricks

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/7-exercise-implement-lake-flow-jobs

Exercise - Implement Lakeflow Jobs with Azure Databricks

Completed

Now it's your chance to implement Lakeflow Jobs with Azure Databricks. In this lab, you configure and automate a CDR data pipeline for TelConnect using Lakeflow Jobs. You run a prebuilt parameterized notebook that processes Call Detail Records through bronze, silver, and gold layers, then configure a Lakeflow Job in the Azure Databricks UI with task dependencies, a job parameter, scheduled and event-based triggers, failure notifications, and retry policies.

> **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/implement-lakeflow-jobs/8-knowledge-check

Module assessment

Completed

9. Summary

https://learn.microsoft.com/en-us/training/modules/implement-lakeflow-jobs/9-summary

Summary

Completed

Automating data pipelines requires more than scheduling scripts to run at fixed times. Throughout this module, you explored how **Lakeflow Jobs** provides comprehensive orchestration capabilities that transform ad hoc processing into reliable, production-ready workflows. You learned to create jobs that coordinate multiple tasks as **directed acyclic graphs (DAGs)**, configure compute resources appropriately for each workload type, and establish task dependencies that control execution order.

Triggering job execution emerged as a critical design decision. You configured **table update triggers** that respond to data changes in Unity Catalog tables, **file arrival triggers** that activate when new files appear in storage locations, and **continuous triggers** that maintain always-on processing for streaming workloads. For time-based requirements, you created **schedules** using simple intervals or advanced cron expressions, understanding how time zone selection affects execution timing.

Operational visibility proved essential for maintaining pipeline reliability. You configured **job alerts and notifications** that inform your team about starts, completions, failures, and duration warnings

without requiring constant dashboard monitoring. You implemented **automatic restart policies** with task-level retries and exponential backoff for continuous jobs, ensuring transient failures don't derail your data processing.

Apply these Lakeflow Jobs capabilities incrementally as you build your data platform. Start with simple job configurations and add triggers, alerts, and retry policies as your requirements mature. Design your tasks for **idempotency** so automatic restarts produce consistent results. Monitor retry patterns to identify underlying infrastructure or data quality issues. These practices create automation that your organization can depend on for critical data processing workloads.

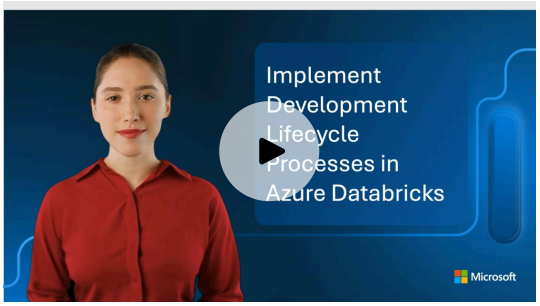
Implement development lifecycle processes in Azure Databricks

1. Introduction

https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/1-introduction

Introduction

Completed



Building reliable data engineering solutions in Azure Databricks requires more than writing code that works—it demands processes that enable collaboration, ensure quality, and support consistent deployments across environments. When data pipelines grow in complexity and team members contribute simultaneously, you need systematic approaches to track changes, validate code, and move work from development to production reliably.

Azure Databricks integrates with established development practices through **Git folders**, which bring version control directly into your workspace. This integration lets you clone repositories, create branches, commit changes, and collaborate with team members without leaving your development environment. Combined with **Databricks Asset Bundles (DABs)**, you gain an infrastructure-as-code approach for defining and deploying jobs, pipelines, and other resources consistently across workspaces.

Effective development lifecycle management also depends on robust testing strategies. The **testing pyramid**—spanning unit tests, integration tests, end-to-end tests, and user acceptance testing—helps you catch issues early, validate component interactions, and confirm business requirements are met before production deployment. By automating these tests and incorporating them into your deployment workflows, you reduce risk and build confidence in your data solutions.

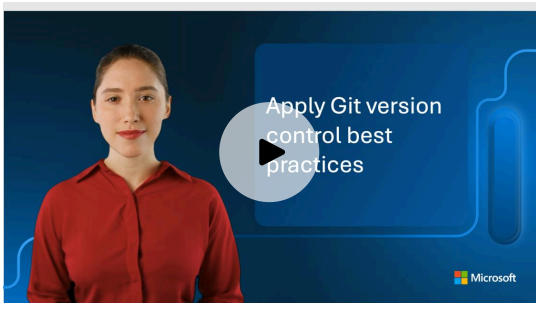
Throughout this module, you work with Git version control practices, branching strategies and pull request workflows, comprehensive testing approaches, and bundle configuration with CLI deployment. These skills enable you to implement professional development lifecycle processes that scale with your team and project complexity.

2. Apply Git version control best practices

https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/2-apply-version-control-git

Apply Git version control best practices

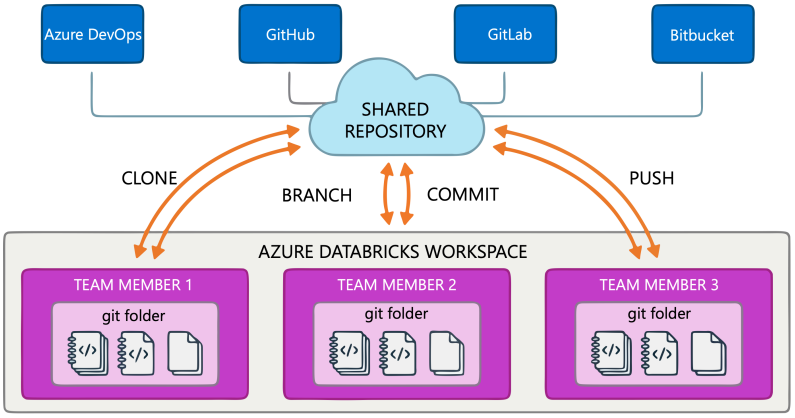
Completed



Data engineering projects require systematic tracking of code changes, collaboration across teams, and the ability to **reproduce work reliably**. When you apply version control best practices using Git within Azure Databricks, you establish a foundation for professional development workflows. **Git folders** integrate directly into your workspace, providing a visual interface for managing repositories without leaving your development environment.

Understand Git folders in Azure Databricks

Git folders bring version control capabilities into your Azure Databricks workspace through a **visual Git client** and API. This integration lets you perform standard Git operations such as **cloning repositories**, **creating branches**, **committing changes**, and **pushing updates** directly from the workspace interface. You don't need to switch between external tools and your development environment.



When you create a Git folder, you connect it to a **remote repository** hosted by providers like Azure DevOps, GitHub, GitLab, or Bitbucket. The folder mirrors your repository structure, allowing you to work with notebooks, Python files, SQL scripts, and other supported assets while maintaining version control. All team members can have their own Git folders mapped to the same remote repository, enabling **collaboration through shared branches** and coordinated commits.

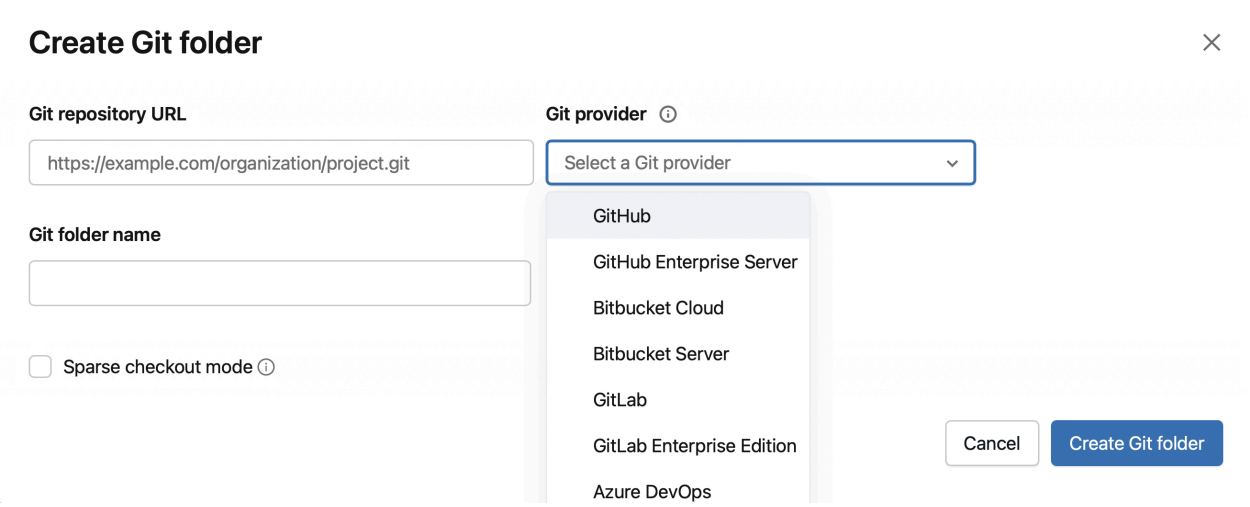
> **Tip**
>
> Each team member should work in their own Git folder connected to the shared repository. This practice prevents accidental branch switches or conflicts that can occur when multiple users perform Git operations on the same workspace folder.

Clone a repository to start working

Before you can work with version-controlled code in Databricks, you need to **clone your team's repository**. Cloning creates a **local copy** of the remote repository in your workspace, giving you access to the latest codebase.

To clone a repository:

1. In the sidebar, select **Workspace** and navigate to where you want to create the Git folder.
2. Select **Create** > **Git folder**.
3. Enter the Git repository URL in the format `https://example.com/organization/project.git`.
4. Select your Git provider from the dropdown menu.
5. Provide a name for the folder in your workspace.
6. Select **Create Git folder**.



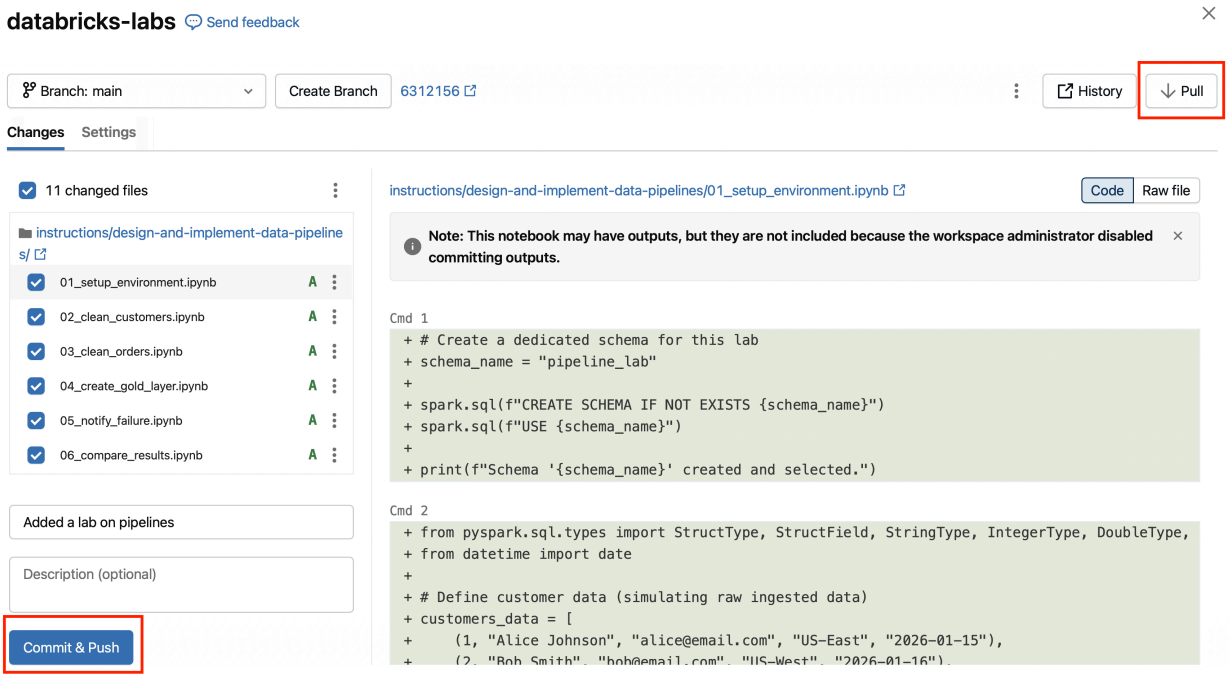
After cloning, the repository contents appear in your workspace. You can open notebooks, edit files, and begin development immediately.

Pull changes from the remote repository

When collaborating with a team, others push changes to the shared repository that you need to incorporate into your local work. **Pulling updates** ensures you're working with the **latest version** of the codebase.

To pull changes:

1. Open the Git dialog.
2. Select **Pull** to fetch and merge changes from the remote repository.
3. If conflicts exist between remote changes and your local modifications, resolve them using the **conflict resolution interface**.



Pull regularly, especially before starting new work, to minimize integration issues. When you pull changes that modify notebook source code, the **notebook state resets**. This means cell outputs, comments, and version history for that notebook clear to reflect the updated content.

Keep your repository organized

A well-organized repository makes collaboration easier and reduces confusion. Apply these practices to maintain a clean, navigable codebase:

- **Use `.gitignore` files** to exclude files that shouldn't be tracked, such as temporary outputs, credentials, or environment-specific configurations. Files already tracked by Git require explicit removal before `.gitignore` applies to them.
- **Structure folders logically** by grouping related notebooks, libraries, and configuration files. Git folders support creating structures at any depth under your user directory.
- **Remove stale branches** from your Git provider after merging to keep the branch list manageable. Local branches persist for up to 30 days after the remote branch is deleted.

> **Note**
>
> Git folders have size constraints. Working branches are limited to 1 GB, and Databricks recommends keeping the total number of assets below 20,000. For large repositories, consider using **sparse checkout** to work with only the directories you need.

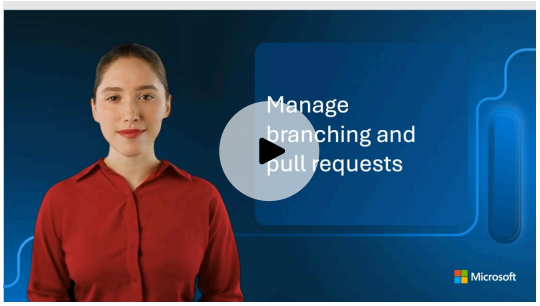
When everyone follows consistent organization practices, team members can find what they need quickly and understand how the codebase fits together.

3. Manage branching and pull requests

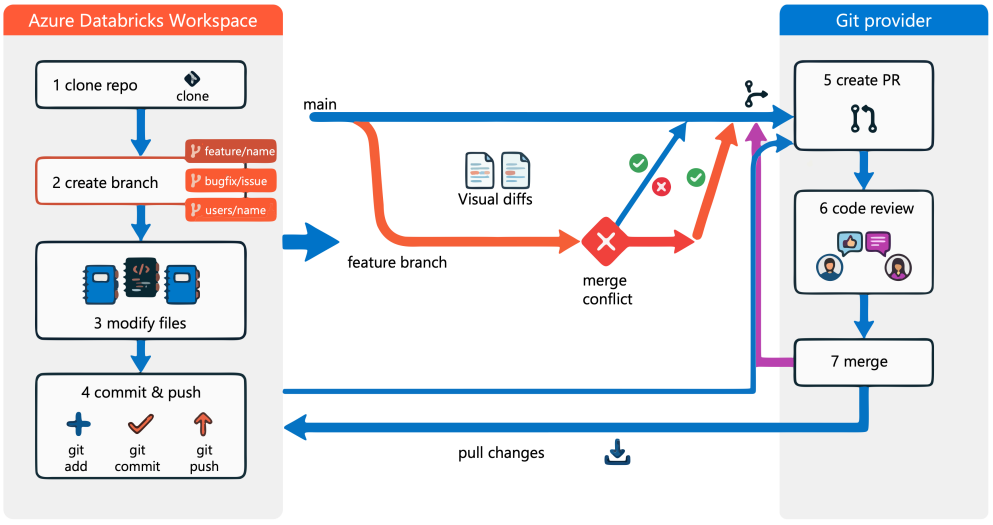
https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/3-manage-branching-pull-requests-conflicts

Manage branching and pull requests

Completed



Collaborative development in Azure Databricks relies on effective **version control practices**. When multiple team members work on notebooks and code files simultaneously, you need a structured approach to **isolate changes**, **review code**, and **integrate work** without disrupting production workflows.



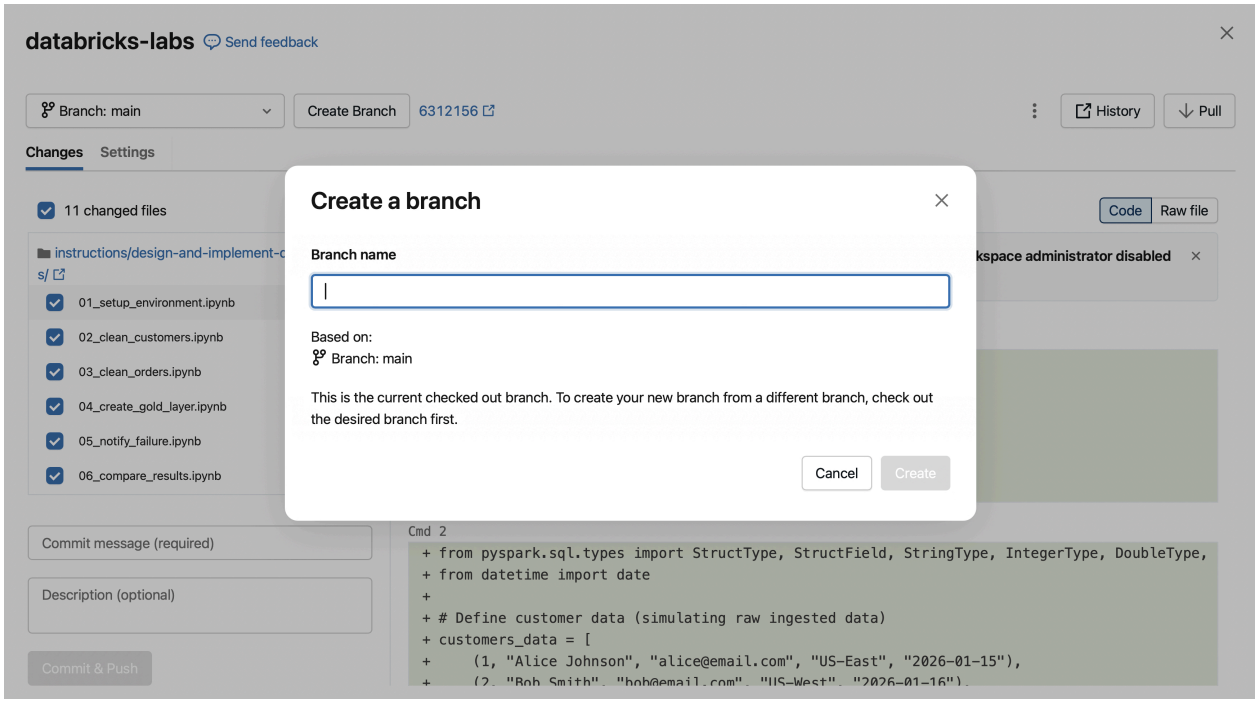
Azure Databricks Git folders provide a visual Git client that integrates directly with your workspace. This integration lets you perform common Git operations—branching, committing, and merging—without leaving the Azure Databricks environment.

Create and manage branches

Feature branches isolate your work from the main codebase. When you create a branch in Azure Databricks, you can develop and test changes independently before merging them into shared branches.

To create a new branch:

1. Open the Git dialog by selecting the branch button next to your notebook name.
2. Select **Create branch** and enter a descriptive name.
3. Base your branch on the appropriate source branch, typically `main`.



Use a **consistent naming convention** for branches to identify work clearly. Common patterns include:

- `feature/feature-name` for new functionality
- `bugfix/issue-description` for fixes
- `users/username/description` for personal development work

When you switch branches, **uncommitted changes carry over** to the new branch if they don't conflict with existing code. Discard changes before switching if you don't intend to carry them to the new branch.

> **Important**
>
> Switching branches may delete workspace assets when the new branch doesn't contain those assets. Verify the asset exists on the target branch before switching, especially if you've shared or bookmarked notebooks.

Commit and push changes

After modifying notebooks or files, **commit your changes** to create a snapshot in your branch history. The Git folders UI highlights modified files and shows **visual diffs** so you can review changes before committing.

To commit and push changes:

1. Open the Git dialog from your notebook or the workspace browser.
2. Review the list of changed files in the dialog.

3. Enter a descriptive commit message that explains what changed and why.
4. Select **Commit & Push** to save changes and sync with the remote repository.

If you lack permission to commit directly to **protected branches** like `main`, create a feature branch first. You can then use your Git provider's interface to create a pull request for merging.

> **Note**
>
> Notebook outputs aren't included in commits by default when notebooks use source file formats like `.py`, `.scala`, `.sql`, or `.r`. Use the IPYNB format if you need to version control outputs.

Work with pull requests

Pull requests let team members **review code changes** before merging them into shared branches. While Azure Databricks provides the Git operations for branching and committing, you create and review pull requests through your Git provider's interface—GitHub, Azure DevOps, GitLab, or Bitbucket.

A typical collaboration workflow follows these steps:

1. Clone the repository to your Azure Databricks workspace.
2. Create a feature branch from `main`.
3. Make modifications to notebooks and files.
4. Commit and push changes to the remote repository.
5. Open your Git provider's website and create a pull request.
6. Review code with your team and address feedback.
7. Merge the pull request into the deployment branch.

Databricks recommends that each developer **work on their own branch** to minimize conflicts. When contributors push to the same Git folder, one user switching branches affects everyone sharing that folder.

After a pull request merges, you can sync the changes to your local workspace by pulling the updated branch.

Resolve merge conflicts

Merge conflicts occur when multiple users modify the same lines of a file, and Git can't automatically determine which changes to keep. Conflicts can arise during **pull**, **merge**, or **rebase** operations.

When a conflict occurs, the Git folders UI displays a list of conflicting files with resolution options:

Keep current or take incoming changes: Use this approach when you know you want all changes from one side. Select the kebab menu next to the file and choose **Keep all current changes** or **Take all incoming changes**.

Manual resolution: For complex conflicts, edit the file directly. Git marks conflicting sections with markers:

```
<<<<<<< HEAD
Your current changes
=======
Incoming changes
>>>>>>> branch-name
```

Remove the markers and edit the content to keep the correct code. Select **Mark As Resolved** when finished.

After resolving all conflicts, select **Continue Merge** or **Continue Rebase** to complete the operation. If you made incorrect choices, select **Abort** to cancel and return to the previous state.

> **Tip**
>
> **Pull changes frequently** from the remote repository to catch conflicts early. Small, frequent merges are easier to resolve than large, complex ones.

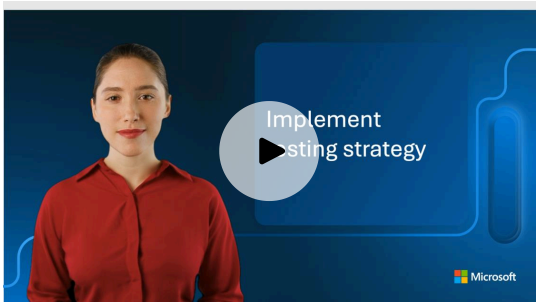
With these branching and conflict resolution skills, you can participate effectively in team development workflows. Your next step is understanding how these practices integrate with broader CI/CD processes for deploying code across environments.

4. Implement testing strategy

https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/4-implement-testing-strategy

Implement testing strategy

Completed

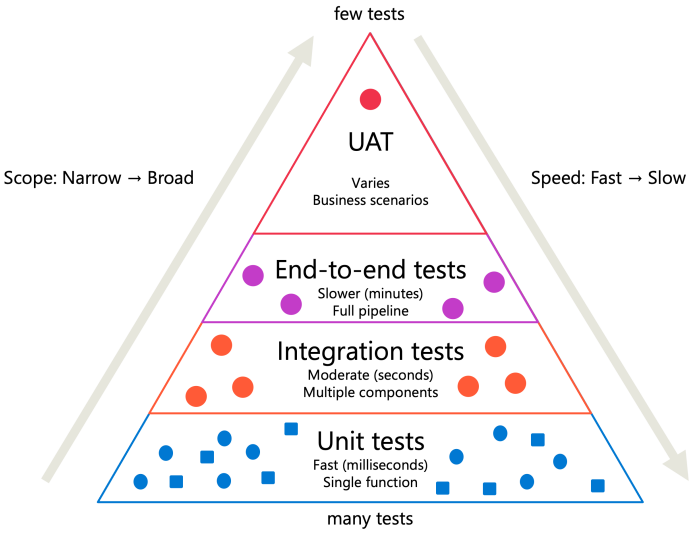


Testing forms a critical foundation of reliable data engineering solutions. When you implement a **comprehensive testing strategy**, you catch problems early, validate assumptions, and ensure your pipelines deliver consistent, trustworthy results. As data volumes grow and pipelines become more complex, **automated testing** becomes essential for maintaining quality and reducing the risk of production failures.

In this unit, you learn how to implement a testing strategy that covers **unit tests**, **integration tests**, **end-to-end tests**, and **user acceptance testing (UAT)** in Azure Databricks.

Understand the testing pyramid

A well-designed testing strategy follows the **testing pyramid** concept. At the base, you have many fast, isolated unit tests. Moving up, you have fewer integration tests that verify component interactions. At the top, you have a small number of comprehensive end-to-end tests and UAT scenarios.



This structure exists because different test types serve different purposes:

Test type	Purpose	Scope	Speed
Unit tests	Verify individual functions work correctly	Single function or class	Fast (milliseconds)

Test type	Purpose	Scope	Speed
Integration tests	Validate components work together	Multiple components	Moderate (seconds)
End-to-end tests	Confirm complete workflows produce expected results	Full pipeline	Slower (minutes)
UAT	Ensure solution meets business requirements	Business scenarios	Varies

Starting with unit tests, you build confidence in your code's fundamental building blocks. Integration tests then confirm those blocks connect properly. End-to-end tests validate the entire system delivers correct results. UAT ensures stakeholders approve the solution before production deployment.

Implement unit tests with pytest

Unit tests focus on testing **individual functions in isolation**. In Azure Databricks, the **pytest** framework provides a powerful way to write and run unit tests for your Python code.

Consider a data transformation function that filters records by country/region:

```
def filter_country_region(df, country_region="USA"):
    return df[df.iso_code == country_region]
```

To test this function, create a test file that follows pytest **naming conventions**. Files should start with `test_` or end with `_test.py`:

```
import pytest
import pandas as pd
from transforms import filter_country_region

@pytest.fixture
def sample_data():
    """Create test data that mimics production structure."""
    return pd.DataFrame({
        'iso_code': ['USA', 'USA', 'CAN', 'GBR'],
        'value': [100, 200, 150, 175]
    })

def test_filter_country_region_default(sample_data):
    result = filter_country_region(sample_data)
    assert len(result) == 2
    assert all(result.iso_code == 'USA')

def test_filter_country_region_specific(sample_data):
    result = filter_country_region(sample_data, country_region='CAN')
    assert len(result) == 1
    assert result.iloc[0]['value'] == 150
```

The `@pytest.fixture` decorator creates **reusable test data**. This approach protects production data by using **synthetic datasets** that mirror your actual data structure without exposing sensitive information.

To run these tests in a Databricks notebook, install pytest and execute it:

```
%pip install pytest

import pytest
retcode = pytest.main([".", "-v", "-p", "no:cacheprovider"])
assert retcode == 0, "Tests failed. Check the output above."
```

> **Tip**
>
> Design functions to return predictable, single-type outputs. A function that returns either a DataFrame or `False` becomes difficult to test. Instead, have it always return a DataFrame, even if empty.

Design integration tests

Integration tests verify that **multiple components work together** correctly. In data pipelines, these tests confirm that data flows properly between ingestion, transformation, and storage stages.

Unlike unit tests that use mocked data, integration tests often run against **actual Databricks resources**. You might read from a test table, apply transformations, and verify the output format:

```
def test_pipeline_integration(spark):
    """Test that transformation pipeline produces expected schema."""
    # Read from test table
    input_df = spark.sql("SELECT * FROM test_catalog.test_schema.raw_data")

    # Apply transformation pipeline
    result_df = transform_pipeline(input_df)

    # Verify output structure
    expected_columns = ['id', 'processed_date', 'category', 'amount']
    assert result_df.columns == expected_columns

    # Verify data types
    assert result_df.schema['amount'].dataType.simpleString() == 'decimal(10,2)'
```

Integration tests require more setup than unit tests. Create **dedicated test schemas or catalogs** that contain representative data samples. This isolation prevents tests from affecting production data while still validating real component interactions.

> **Important**

> Never run integration tests against production tables. Create separate test environments with data that mirrors production structure but contains no sensitive information.

Create end-to-end tests

End-to-end tests simulate **complete workflows** from start to finish. These tests validate that your entire pipeline produces the expected results when given specific inputs.

Structure end-to-end tests to cover **realistic scenarios**:

```
def test_daily_processing_pipeline():
    """Validate complete daily data processing workflow."""
    # Setup: Create test input files
    test_date = "2024-01-15"
    setup_test_input_files(test_date)

    # Execute: Run the complete pipeline
    run_daily_pipeline(test_date)

    # Verify: Check final output table
    result = spark.sql(f"""
        SELECT COUNT(*) as row_count,
               SUM(amount) as total_amount
        FROM production.daily_summary
        WHERE process_date = '{test_date}'
    """)

    row = result.first()
    assert row.row_count == 1000, f"Expected 1000 rows, got {row.row_count}"
    assert abs(row.total_amount - 50000.00) < 0.01

    # Cleanup: Remove test data
    cleanup_test_data(test_date)
```

End-to-end tests take longer to run than unit or integration tests. Schedule them to run during **off-peak hours** or as part of your **deployment pipeline** rather than on every code change.

Plan user acceptance testing

User acceptance testing (UAT) involves **stakeholders validating** that your solution meets **business requirements**. While the previous test types focus on technical correctness, UAT confirms the solution delivers business value.

Effective UAT requires careful planning:

- **Define acceptance criteria** with stakeholders before development begins
- **Create a staging environment** that mirrors production
- **Prepare test scenarios** that reflect actual business use cases

- **Document expected outcomes** for each scenario
- **Establish a feedback process** for reporting issues

UAT often runs in a **staging environment** where business users can interact with the solution using realistic data. Consider creating notebooks that stakeholders can execute to validate specific scenarios:

```
# UAT Scenario: Monthly revenue reconciliation
# Expected outcome: Total matches finance system within 1%

finance_total = 1_250_000.00  # From finance system

pipeline_result = spark.sql("""
    SELECT SUM(revenue) as total_revenue
    FROM reporting.monthly_revenue
    WHERE month = '2024-01'
""").first()

variance = abs(pipeline_result.total_revenue - finance_total) / finance_total
print(f"Variance: {variance:.2%}")

if variance < 0.01:
    print("✓ UAT PASSED: Revenue totals match within tolerance")
else:
    print("✗ UAT FAILED: Revenue variance exceeds 1% threshold")
```

Organize your test structure

A **well-organized test structure** makes tests easier to maintain and run. Follow these conventions for your Azure Databricks projects:

```
project/
├── src/
│   └── transforms.py
├── tests/
│   ├── unit/
│   │   └── test_transforms.py
│   ├── integration/
│   │   └── test_pipeline.py
│   └── e2e/
│       └── test_daily_workflow.py
├── notebooks/
│   └── run_tests.py
└── requirements.txt
```

Databricks recommends storing functions and their unit tests **outside of notebooks** for Python projects. This approach enables better **code reuse** and makes testing more straightforward. Store your functions in `.py` files within a Git folder, and import them into notebooks as needed.

For running tests, create a dedicated notebook that executes pytest:

```
%pip install -r ../requirements.txt

import pytest
import sys

# Prevent pytest from caching to readonly filesystem
sys.dont_write_bytecode = True

# Run all tests with verbose output
retcode = pytest.main([
    "tests/",
    "-v",
    "-p", "no:cacheprovider"
])

assert retcode == 0, "Test suite failed"
```

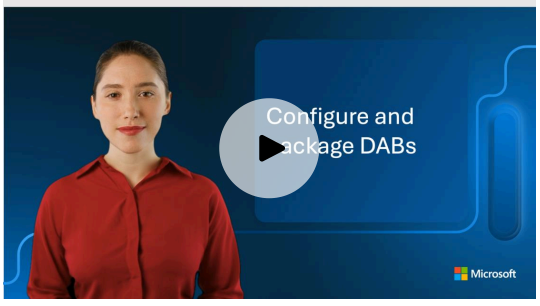
Automate test execution by creating a Databricks job that runs your test notebook before deploying changes to production. This automation ensures that code changes don't introduce **regressions**.

5. Configure and package DABs

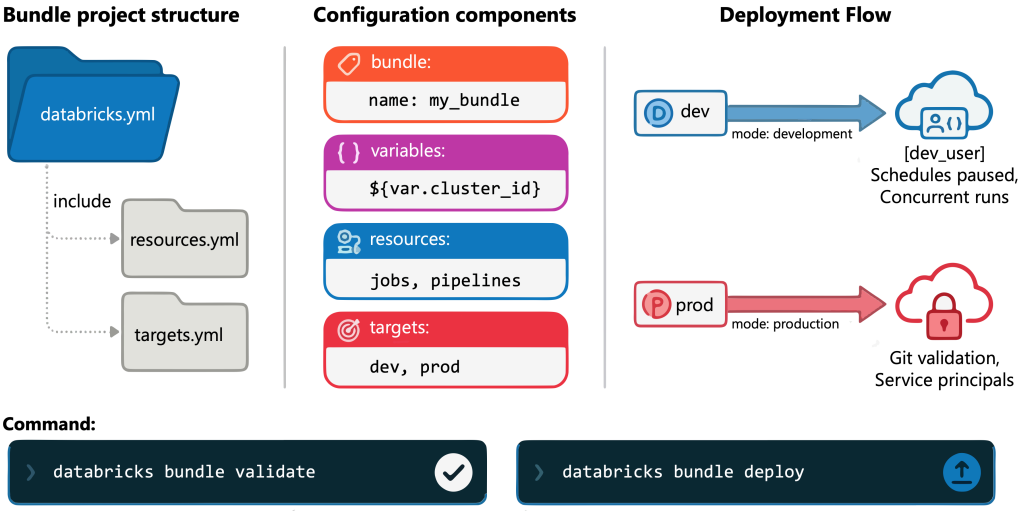
https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/5-configure-package-databricks-asset-bundles

Configure and package DABs

Completed



When you work on data engineering projects, you need a systematic way to define, package, and deploy your Databricks resources across environments. **Databricks Asset Bundles (DABs)** provide an **infrastructure-as-code** approach that lets you describe jobs, pipelines, and other resources in YAML configuration files. By configuring and customizing these bundles, you can **automate deployments** and ensure consistency between development and production workspaces.



Understand the bundle configuration structure

A Databricks Asset Bundle consists of **YAML configuration files** that define your project's resources and deployment settings. The primary configuration file, `databricks.yml`, must exist at the **root of your bundle project**. This file establishes the bundle's identity and can reference additional configuration files for better organization.

The `databricks.yml` file contains several key sections that control how your bundle behaves:

```
bundle:
  name: my-data-pipeline

include:
  - resources/*.yml

workspace:
  host: https://adb-1234567890123456.7.azuredatabricks.net

resources:
  jobs:
    daily-ingestion-job:
      name: daily-ingestion-job
      tasks:
        - task_key: ingest-data
          notebook_task:
            notebook_path: ./notebooks/ingest.py

targets:
  dev:
    default: true
  prod:
    workspace:
      host: https://adb-9876543210987654.3.azuredatabricks.net
```

The **bundle** mapping defines the bundle's programmatic name. The **resources** mapping specifies the Databricks resources you want to deploy, such as jobs and pipelines. The **targets** mapping defines different **deployment environments**, each potentially pointing to different workspaces with different configurations.

> **Tip**
>
> Split your configuration into separate files for resources and targets to keep configurations manageable. Use the `include` mapping to reference additional YAML files in your bundle.

Customize deployment parameters with variables

Variables make your bundle configurations flexible and reusable across environments. Rather than hardcoding values like cluster IDs or job names, you define variables that can be set differently for each deployment target.

To define a variable, add it to the **variables** section of your configuration:

```
variables:
  cluster_id:
    description: The cluster to use for job execution
    default: 1234-567890-abcde123
  environment:
    description: The deployment environment name
    default: development

resources:
  jobs:
    my-job:
      name: ${var.environment}-data-pipeline
      tasks:
        - task_key: process-data
          existing_cluster_id: ${var.cluster_id}
          notebook_task:
            notebook_path: ./notebooks/process.py
```

The **substitution syntax** `${var.variable_name}` inserts the variable's value wherever you reference it. This approach allows you to use the same bundle configuration while adjusting specific parameters for different environments.

You can provide variable values in several ways:

- Set default values in the variable definition
- Override values in target-specific configurations
- Pass values through command-line options: `--var="cluster_id=5678-901234-fghij567"`
- Use environment variables with the `BUNDLE_VAR_` prefix

When deploying to different targets, you can override variable values for each environment:

```
targets:
  dev:
    default: true
    variables:
      cluster_id: 1234-567890-abcde123
      environment: development
  prod:
    variables:
      cluster_id: 9876-543210-zyxwv987
      environment: production
```

Configure resources and task definitions

The **resources** section defines the Databricks objects your bundle manages. **Jobs** are the most common resource type, and you can configure various task types within each job.

A job definition includes the job name, tasks, and optional settings like schedules and notifications:

```
resources:
  jobs:
    etl-pipeline:
      name: etl-pipeline
      tasks:
        - task_key: extract-task
          notebook_task:
            notebook_path: ./notebooks/extract.py
        - task_key: transform-task
          depends_on:
            - task_key: extract-task
          notebook_task:
            notebook_path: ./notebooks/transform.py
            base_parameters:
              source_table: ${var.source_table}
```

Each task specifies its type, such as **notebook_task** for running notebooks or **spark_python_task** for Python scripts. The `depends_on` setting creates **task dependencies**, ensuring tasks run in the correct order. Parameters passed through `base_parameters` let you customize task behavior at runtime.

> **Important**
>
> When configuring tasks, use paths relative to the bundle's root directory. The Databricks CLI resolves these paths when deploying the bundle to your workspace.

For compute configuration, you can reference existing clusters or define job clusters:

```
resources:
  jobs:
```

```
    my-job:
      job_clusters:
        - job_cluster_key: shared-cluster
          new_cluster:
            spark_version: 14.3.x-scala2.12
            node_type_id: Standard_DS3_v2
            num_workers: 2
      tasks:
        - task_key: my-task
          job_cluster_key: shared-cluster
          notebook_task:
            notebook_path: ./notebooks/analysis.py
```

Set up deployment targets

Targets represent different deployment environments, each with its own configuration overrides. A typical bundle includes development and production targets with distinct workspace URLs and settings.

Development mode automatically applies behaviors that simplify testing:

```
targets:
  dev:
    default: true
    mode: development
```

When you deploy with `mode: development`, the CLI **prefixes resource names** with `[dev username]`, **pauses all schedules**, and enables **concurrent job runs**. These defaults help prevent development resources from interfering with production workloads.

Production mode enforces stricter validation:

```
targets:
  prod:
    mode: production
    workspace:
      host: https://adb-prod.7.azuredatabricks.net
```

In production mode, the CLI validates that your local Git branch matches the specified branch and recommends using **service principals** for deployments. You can customize these behaviors with the **presets** mapping:

```
targets:
  staging:
    presets:
      name_prefix: staging_
      trigger_pause_status: PAUSED
      tags:
        environment: staging
```

Presets let you fine-tune deployment behaviors without relying solely on the development or production mode defaults. This flexibility is useful for staging environments or specialized deployment scenarios.

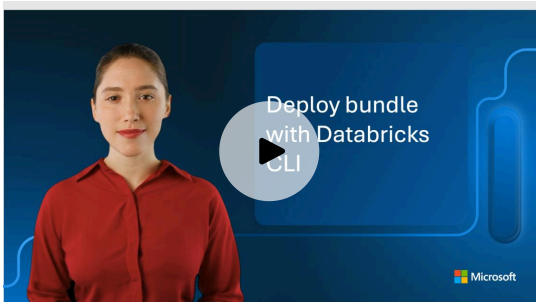
With your bundle configuration complete, you're ready to validate and deploy your resources. The `databricks bundle validate` command checks your configuration for errors before deployment, while `databricks bundle deploy` pushes your resources to the target workspace.

6. Deploy bundle with Databricks CLI

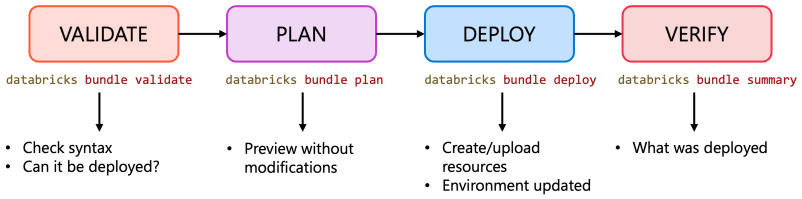
https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/6-deploy-bundle-databricks-cli

Deploy bundle with Databricks CLI

Completed



After your team packages a data asset bundle, deploying it to Azure Databricks becomes your next responsibility. The **Databricks CLI** provides a direct path from your local development environment to production workspaces. You control when and where your pipelines, jobs, and other resources go live. Understanding this **deployment workflow** enables you to move code reliably across environments while maintaining consistency.



Validate bundle configuration

Before deploying, confirm that your bundle configuration is **syntactically correct** and complete. The `bundle validate` command checks your configuration files and reports any issues that would prevent successful deployment.

```
databricks bundle validate
```

When validation succeeds, you see a summary of your bundle identity:

```
Name: my_data_pipeline
Target: dev
Workspace:
  Host: https://adb-1234567890123456.7.azuredatabricks.net
  User: someone@example.com
  Path: /Users/someone@example.com/.bundle/my_data_pipeline/dev

Validation OK!
```

This output confirms the **bundle name**, **target environment**, and **workspace details**. If your configuration contains errors, such as missing required fields or invalid property names, the command outputs warnings or errors that you need to address before proceeding.

> **Tip**
>
> Run `bundle validate` after making changes to your bundle configuration files. Catching syntax errors early saves time compared to discovering them during deployment.

Preview deployment changes

After validation, you can **preview exactly what the deployment will create, update, or remove** without making actual changes. The `bundle plan` command builds your bundle and displays the planned actions.

```
databricks bundle plan
```

The output shows what resources will be affected:

```
Building python_artifact...
create jobs.data_ingestion_job
create pipelines.transform_pipeline
```

This preview lets you verify that the deployment matches your expectations. If you see unexpected changes, such as resources being removed that you intended to keep, adjust your configuration before committing to the deployment. For targets other than the default, specify the target:

```
databricks bundle plan -t production
```

Deploy bundle to a target workspace

With validation complete and the plan reviewed, you deploy the bundle using the `bundle deploy` command. This command **uploads your artifacts** and **creates or updates resources** in the target workspace.

```
databricks bundle deploy
```

By default, deployment uses the **default target** defined in your bundle configuration. To deploy to a specific target such as development, staging, or production, use the `-t` flag:

```
databricks bundle deploy -t dev
```

During deployment, the CLI **tracks which resources it creates** by storing state in the workspace. This tracking enables several behaviors:

- **New resources** defined in your configuration are created in the workspace
- **Existing resources** that you previously deployed are updated to match your current configuration
- **Removed resources** that no longer appear in your configuration are deleted from the workspace

> **Important**
>
> Each bundle deployment has a unique identity based on the bundle name, target name, and the deploying user's identity. If multiple team members deploy the same bundle to the same target, their deployments will conflict. Coordinate with your team to establish who deploys to shared environments.

For automated deployments in **CI/CD pipelines**, add the `--auto-approve` flag to skip confirmation prompts:

```
databricks bundle deploy -t production --auto-approve
```

Verify deployed resources

After deployment completes, verify that your resources are available and configured correctly in the workspace. The `bundle summary` command outputs information about **deployed resources**, including **direct links** to view them in the Azure Databricks UI:

```
databricks bundle summary
```

```
Name: my_data_pipeline
Target: dev
```

```
Workspace:
  Host: https://adb-1234567890123456.7.azuredatabricks.net
  User: someone@example.com
  Path: /Users/someone@example.com/.bundle/my_data_pipeline/dev
Resources:
  Jobs:
    data_ingestion_job:
      Name: [dev someone] data_ingestion_job
      URL:  https://adb-1234567890123456.7.azuredatabricks.net/jobs/123456789?o=12
  Pipelines:
    transform_pipeline:
      Name: [dev someone] transform_pipeline
      URL:  https://adb-1234567890123456.7.azuredatabricks.net/pipelines/abc-123-de
```

You can also navigate directly to a specific resource using the `bundle open` command:

```
databricks bundle open data_ingestion_job
```

This command opens your browser to the resource in the Azure Databricks workspace.

Troubleshoot common deployment issues

Even with careful preparation, deployments sometimes encounter problems. Here are common issues and how to resolve them:

Authentication errors: If you see permission denied or authentication failures, verify that your CLI is configured with **valid credentials**. Check your profile configuration or reauthenticate with your workspace.

```
databricks auth login --host https://adb-1234567890123456.7.azuredatabricks.net
```

Lock conflicts: When a deployment is in progress, Azure Databricks acquires a **lock** to prevent concurrent modifications. If a previous deployment was interrupted, you might see lock errors. Use the `--force-lock` flag to override.

```
databricks bundle deploy --force-lock
```

Active run conflicts: If jobs or pipelines from your bundle are **currently running**, deployment fails by default to prevent disruption. You can choose to fail explicitly with the `--fail-on-active-runs` flag, or handle running resources in your deployment strategy.

Validation warnings: If `bundle validate` reports warnings about unknown properties, your configuration might reference features not available in your current CLI version or workspace. Update your CLI or review the property names against the current schema.

With these deployment commands, you can reliably move your data engineering work from development through staging to production environments, maintaining control over what gets deployed and when.

7. Exercise - Implement Development Lifecycle Processes in Azure Databricks

https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/7-exercise-implement-development-lifecycle-processes-in-azure-databricks

Exercise - Implement Development Lifecycle Processes in Azure Databricks

Completed

Now it's your chance to implement development lifecycle processes in Azure Databricks. In this lab, you implement a testing strategy for a data transformation pipeline using pytest, then package and deploy the pipeline as a Databricks Asset Bundle using the Databricks CLI.

> **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/implement-development-lifecycle-processes-in-azure-databricks/8-knowledge-check

Module assessment

Completed

9. Summary

https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/9-summary

Summary

Completed

Implementing development lifecycle processes transforms how your team builds and deploys data engineering solutions in Azure Databricks. Throughout this module, you explored **Git folder integration** for version control, **branching and pull request workflows** for collaboration, **comprehensive testing strategies** to ensure quality, and **Databricks Asset Bundles** with CLI deployment for consistent infrastructure management.

Effective version control through Git folders enables teams to track changes, collaborate through shared repositories, and maintain organized codebases. By following branching best practices and using pull requests for code review, you reduce integration conflicts and ensure code quality before merging. The **testing pyramid**—with unit tests at the base, integration tests in the middle, and end-to-end tests at the top—provides a structured approach to catching issues early and validating that pipelines deliver expected results.

Databricks Asset Bundles bring **infrastructure-as-code** principles to your deployment workflow. By defining jobs, pipelines, and resources in YAML configuration files, you create reproducible deployments across development, staging, and production environments. The Databricks CLI commands—**validate**, **plan**, and **deploy**—give you control over the deployment process and help troubleshoot issues before they impact production.

Apply these development lifecycle practices to your Azure Databricks projects to improve collaboration, reduce deployment risks, and deliver reliable data solutions that meet business requirements.

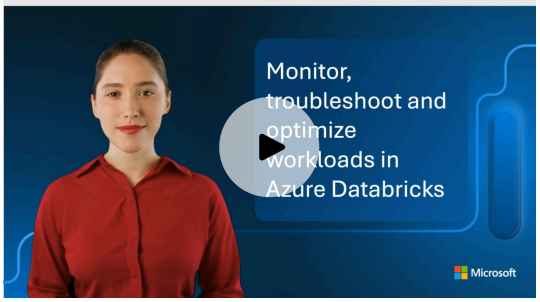
Monitor, troubleshoot and optimize workloads in Azure Databricks

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.

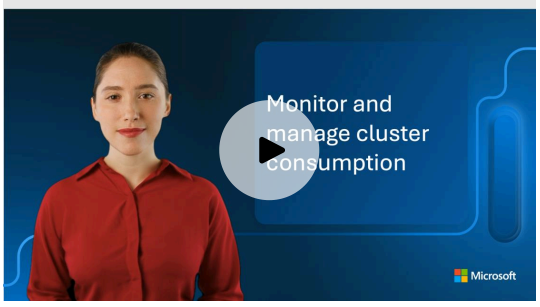
Throughout this module, you explore cluster consumption monitoring, Lakeflow Jobs troubleshooting, Spark job diagnostics, performance optimization techniques, and log streaming configuration to build the skills needed to keep your Azure Databricks workloads running reliably and cost-effectively.

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

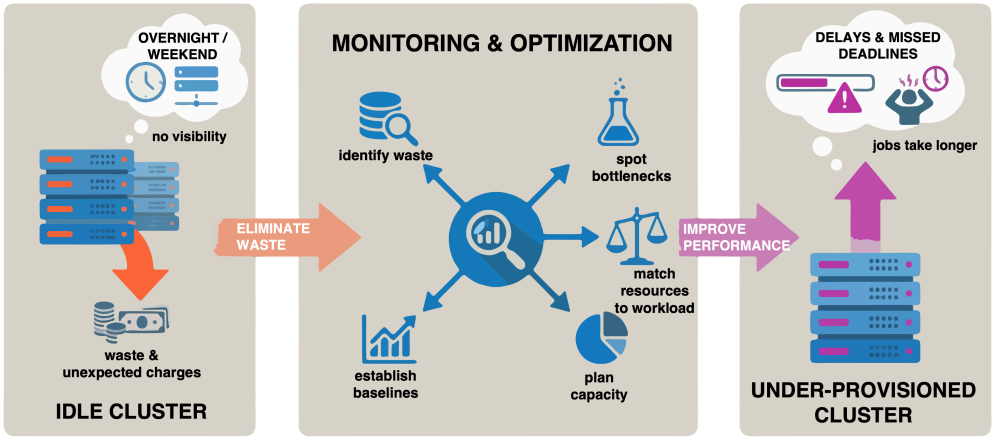


When clusters run longer than needed or consume more resources than necessary, costs can escalate quickly. As a data engineer, you're responsible for **monitoring cluster usage** and taking action to balance **performance requirements** with **cost efficiency**.

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.

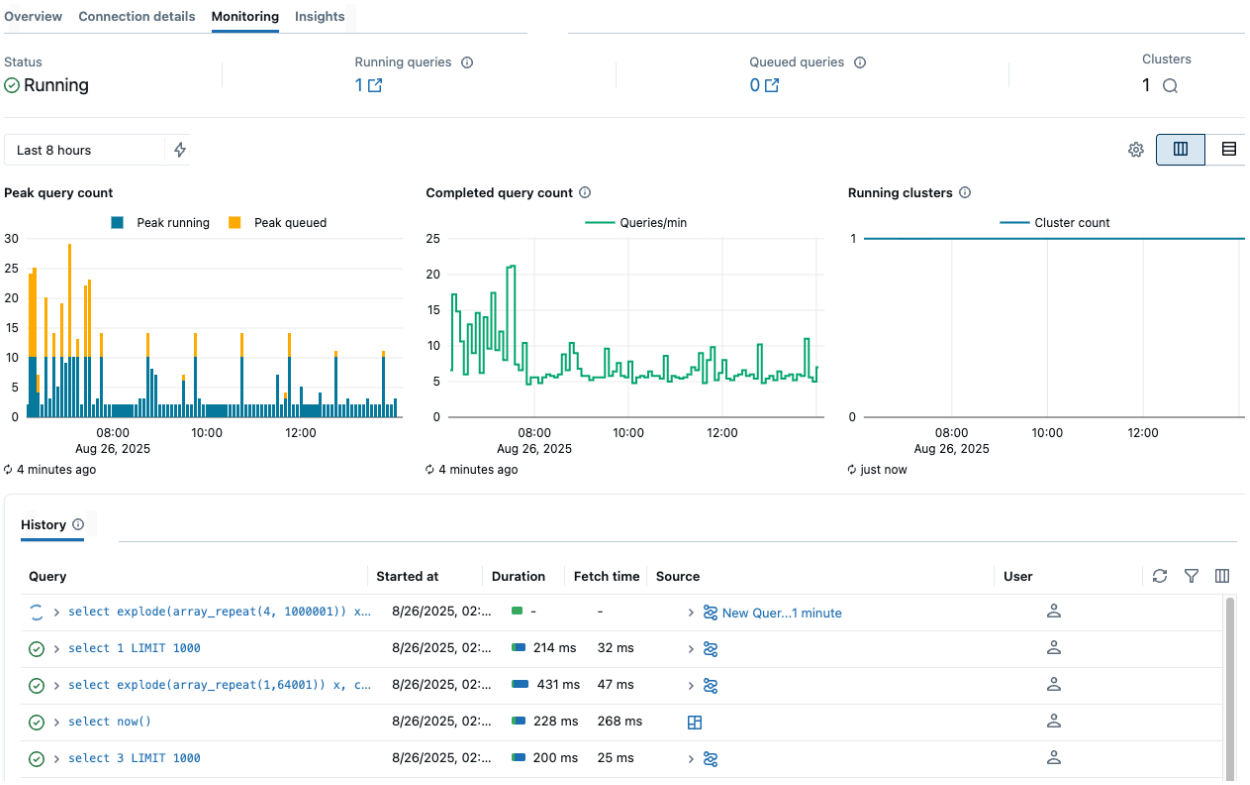
GPU metrics (available on Databricks Runtime ML 13.3 and later) track specialized compute utilization when running machine learning workloads.

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.

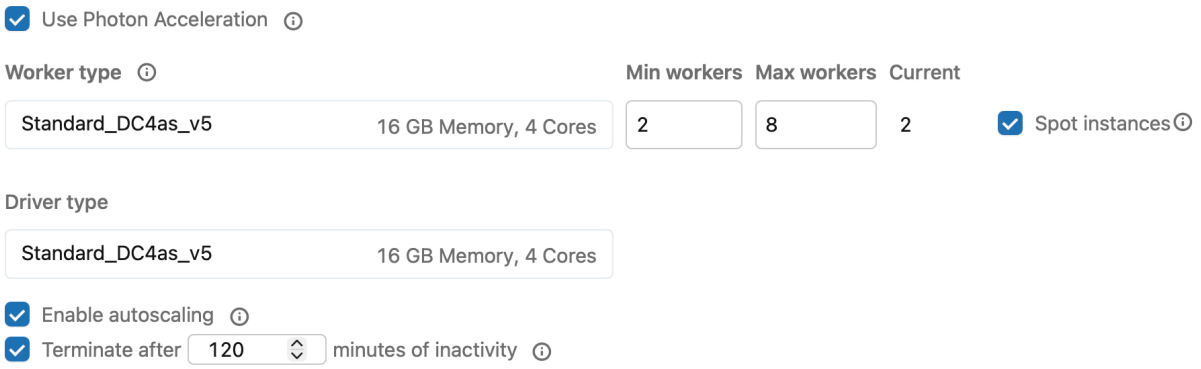
The **running clusters** chart shows how many clusters were allocated over time. If you've enabled **autoscaling**, you'll see this number fluctuate based on query load. Clusters that remain at maximum capacity for extended periods suggest you might need to increase the maximum cluster count.

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.



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.

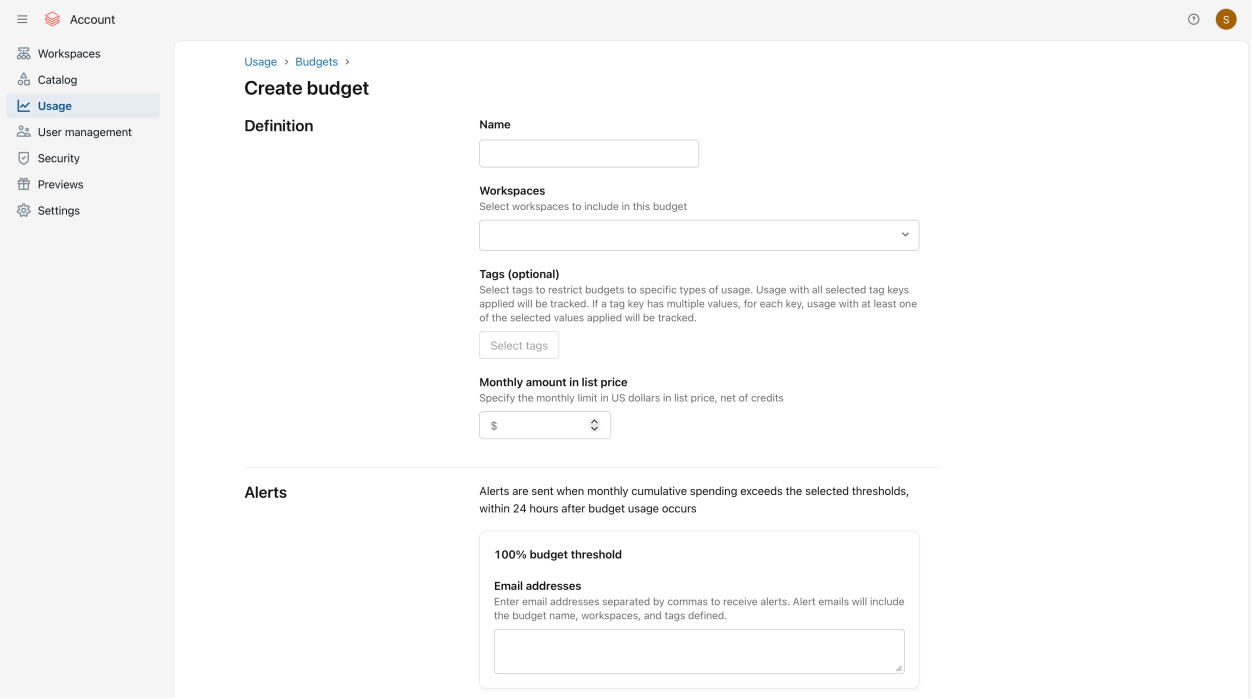
Autoscaling dynamically adjusts the number of worker nodes based on workload demand. Configure **minimum and maximum node counts** based on your workload analysis. The cluster adds workers during intensive processing and removes them during lighter periods, reducing costs by **20-40%** compared to fixed-size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

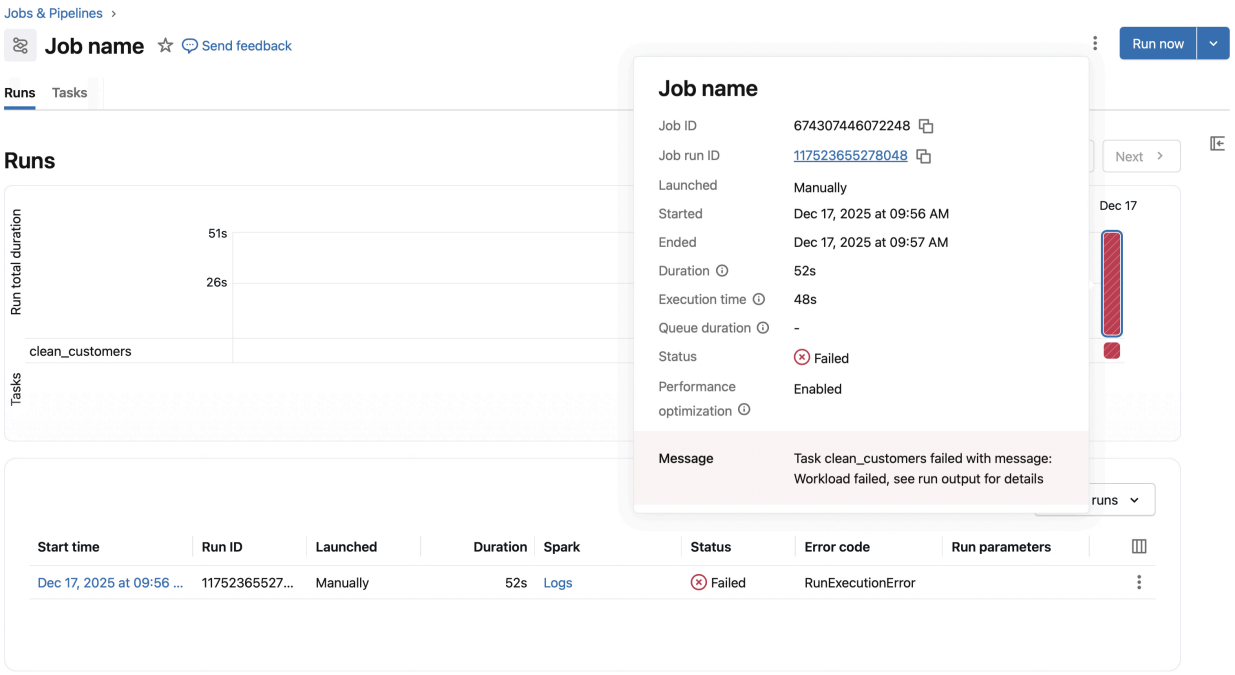
Understanding these states helps you prioritize your response. A failed leaf task requires immediate attention, while a skipped run might indicate a scheduling conflict that resolves itself.

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.



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

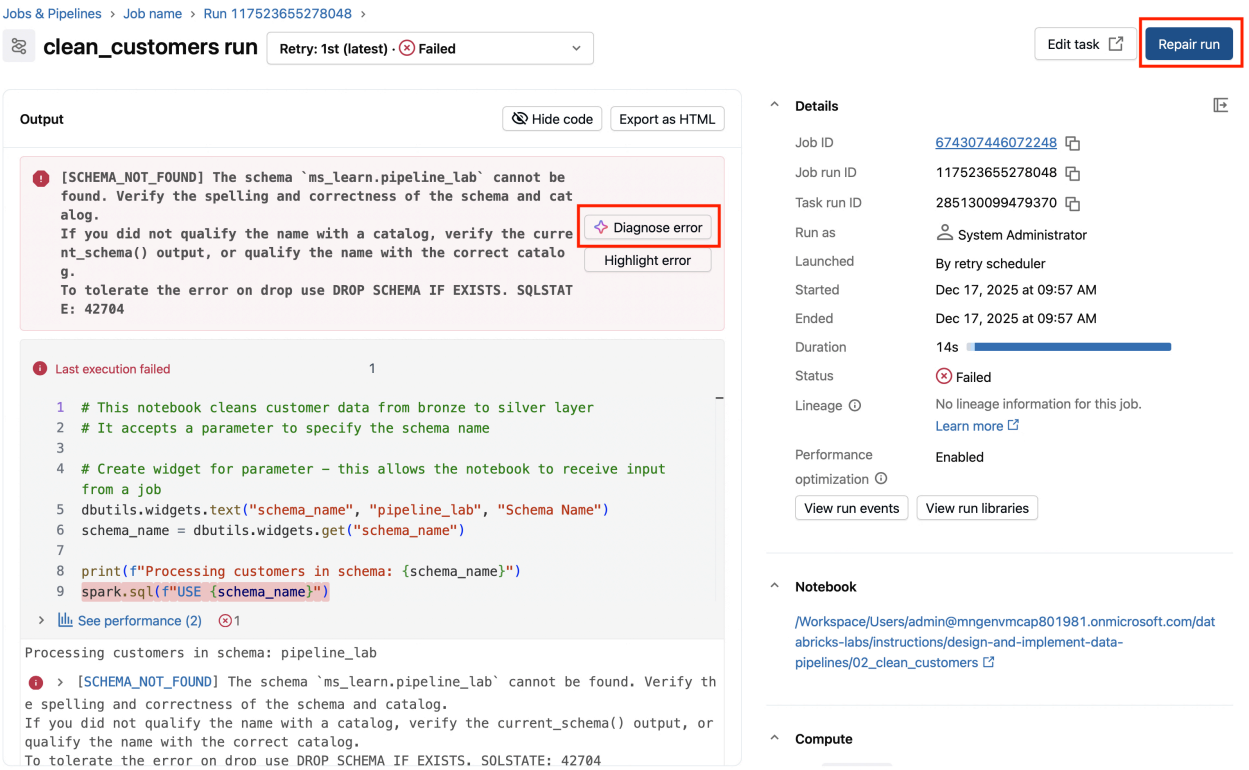
can use the repair feature to re-run the job without starting from scratch.

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.



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).

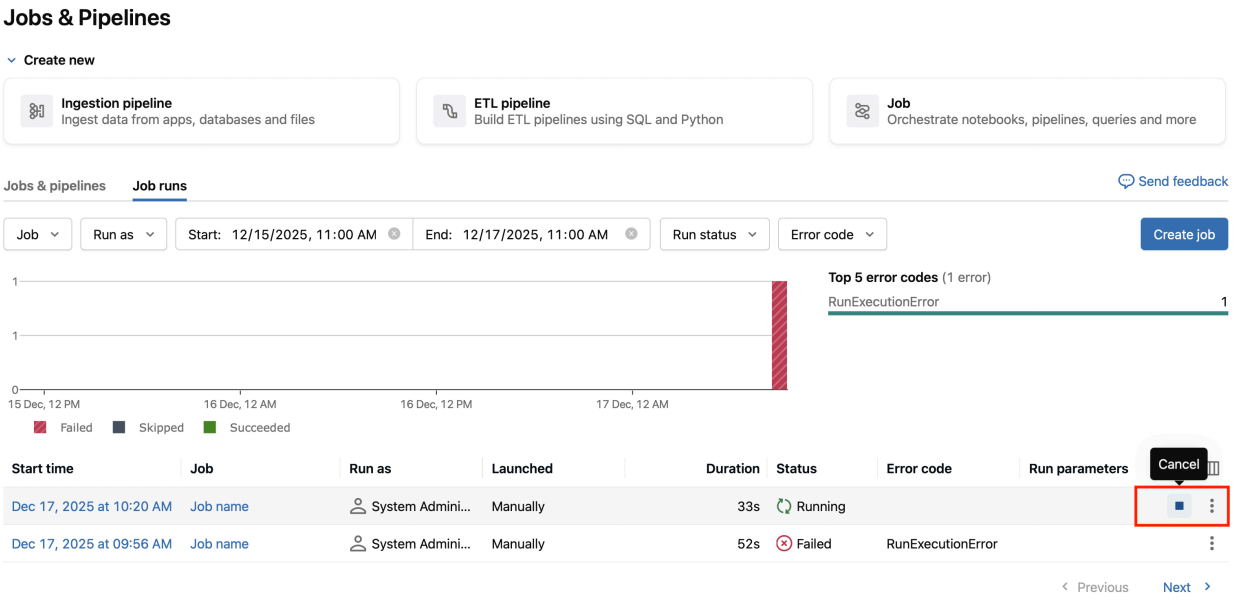
> **Note**
>
> Repair is available only for jobs with two or more tasks. If tasks share a job cluster, the repair creates a new cluster instance (for example, `my_job_cluster_v1`) so you can compare settings between the original and repaired runs.

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.



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

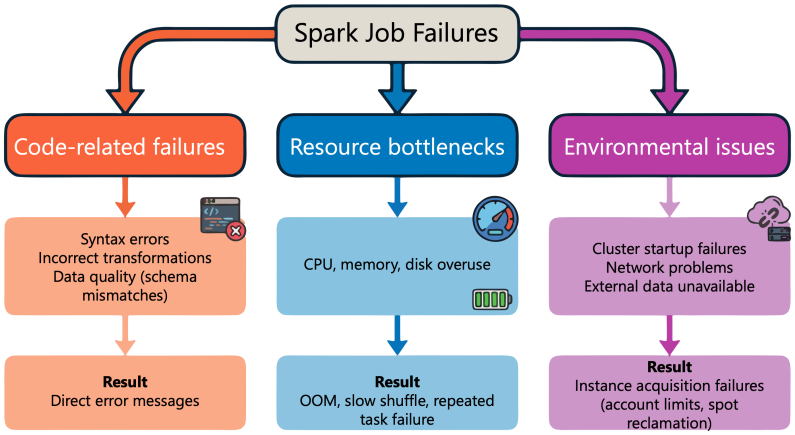
Troubleshoot Spark jobs and notebooks

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.



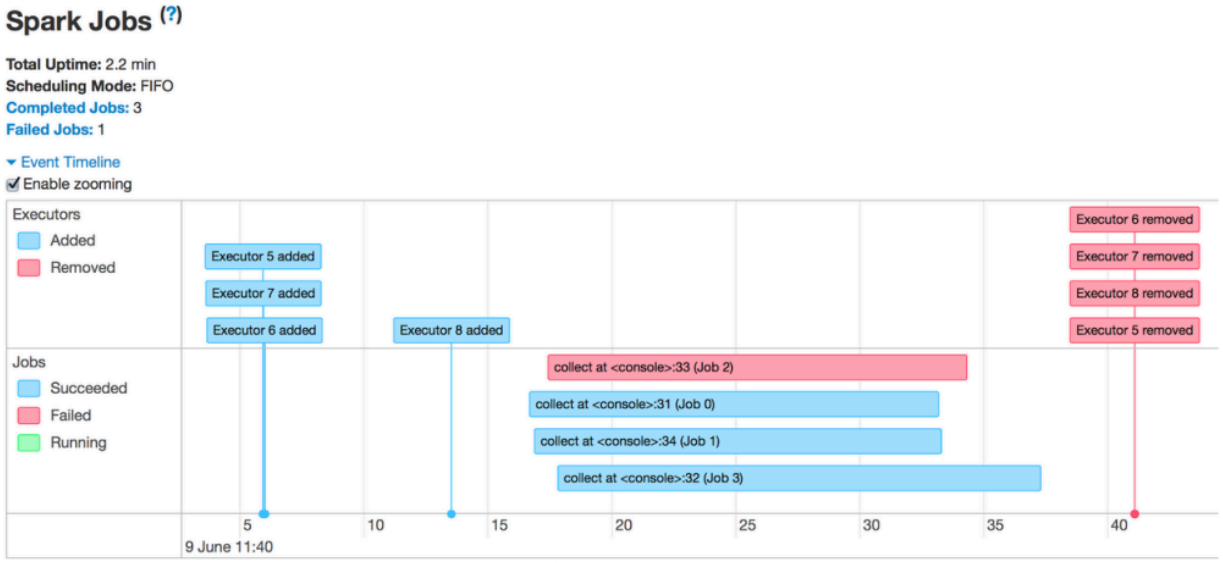
Code-related failures include syntax errors in notebooks, incorrect transformations, or data quality issues like schema mismatches. These failures typically produce error messages that point directly to the problematic code.

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.



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.

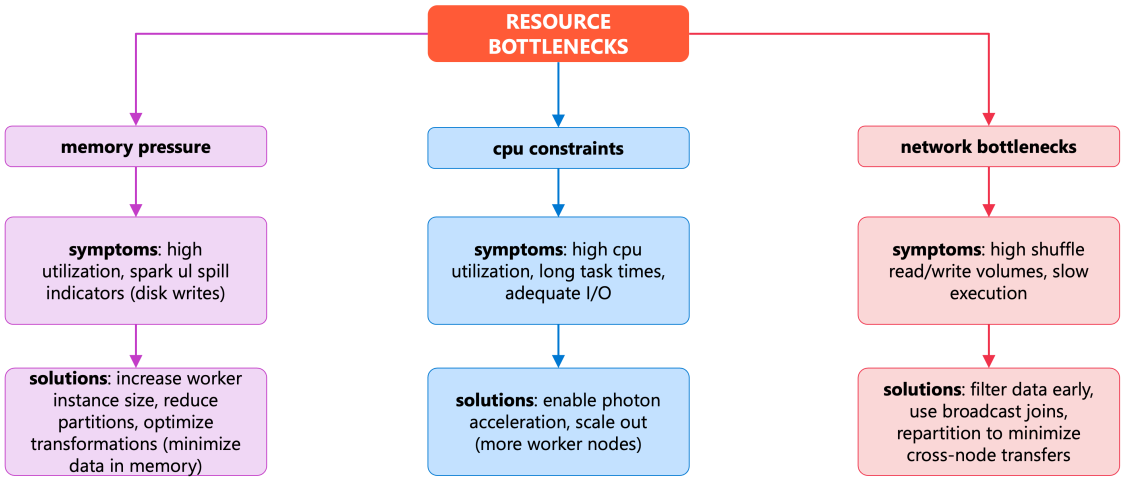
Gaps in execution represent periods where no Spark work occurred. Short gaps are normal as the driver coordinates tasks, but gaps longer than a minute in the middle of a pipeline suggest the driver might be overloaded or a cluster malfunction occurred.

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.

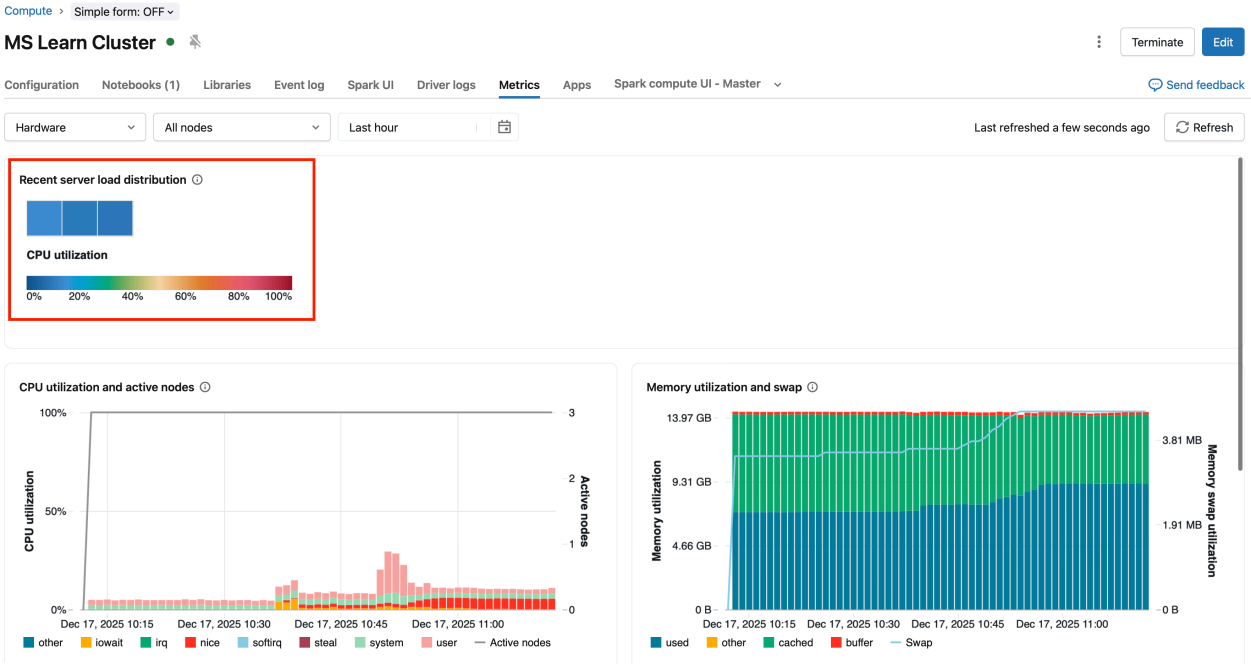


Memory pressure appears as high memory utilization across workers or the driver. In the Spark UI, look for spill indicators showing data being written to disk because memory is insufficient. You can address memory issues by increasing worker instance sizes, reducing partition counts, or optimizing transformations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

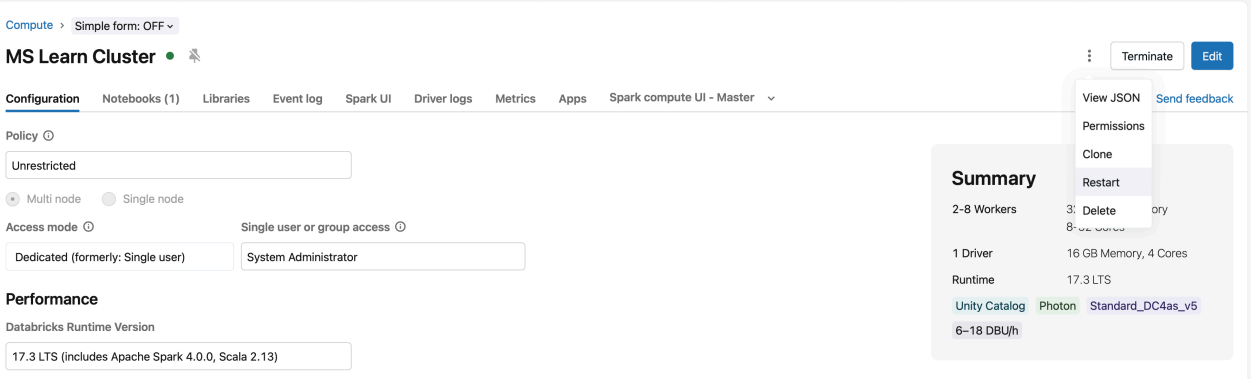


Restart clusters to resolve environmental issues

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.

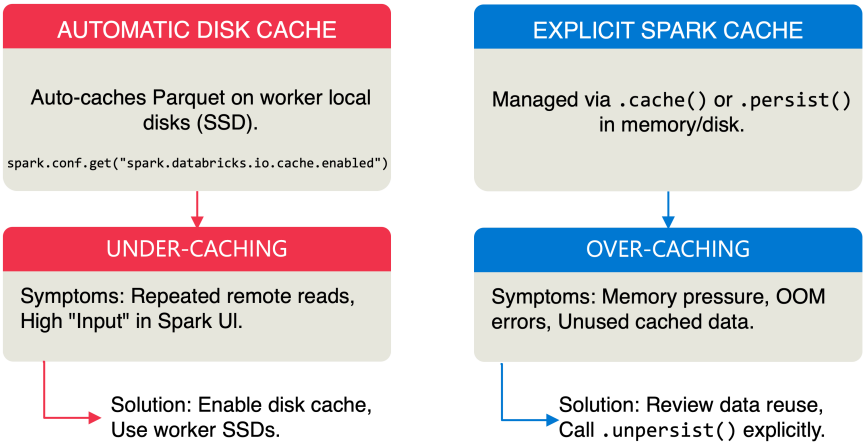
Investigate caching issues

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.

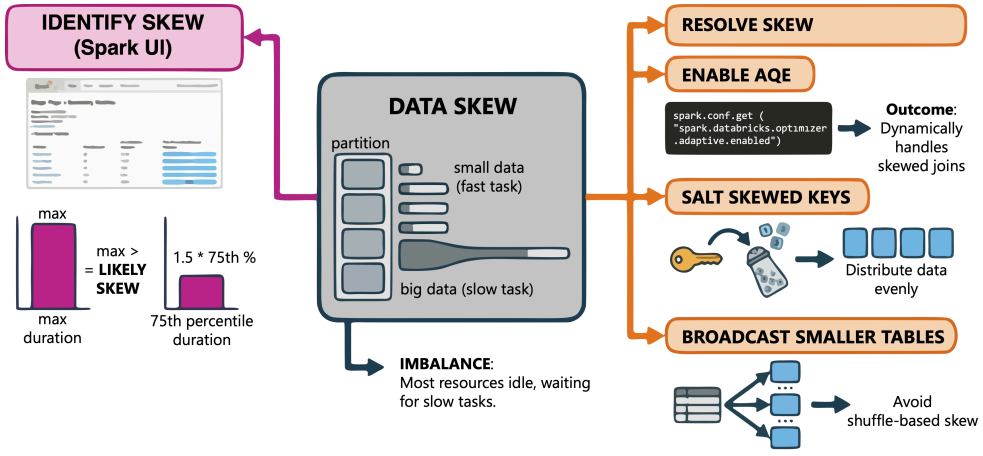
Over-caching consumes memory that Spark needs for processing. If you see memory pressure or out-of-memory errors, review whether cached data is actually being reused. Spark cache (using `.cache()` or `.persist()`) requires explicit management, unlike automatic disk caching.

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:

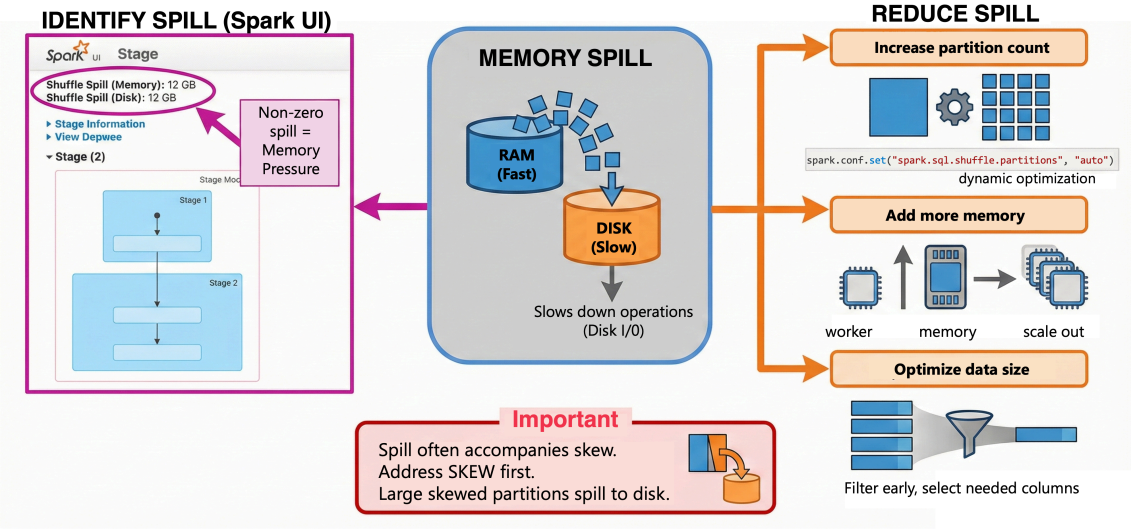
- **Enable Adaptive Query Execution (AQE)**, which dynamically handles skewed joins. Verify it's enabled:

```
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")
```

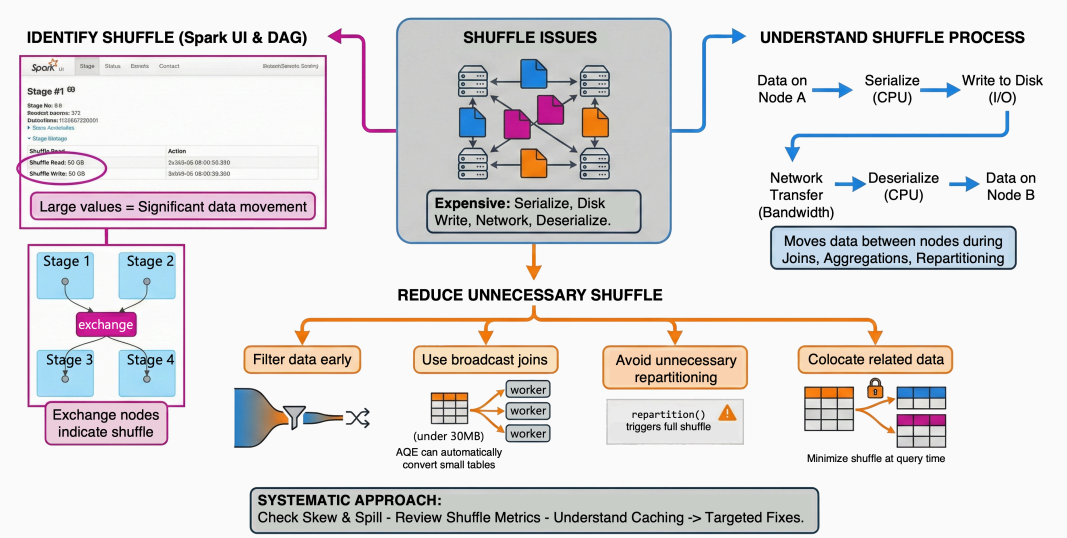
- **Add more memory** by choosing worker nodes with higher memory-to-core ratios or increasing the number of workers.
- **Optimize data size** by selecting only needed columns and filtering early in your pipeline. Reducing the data processed at each stage lowers memory requirements.

> **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

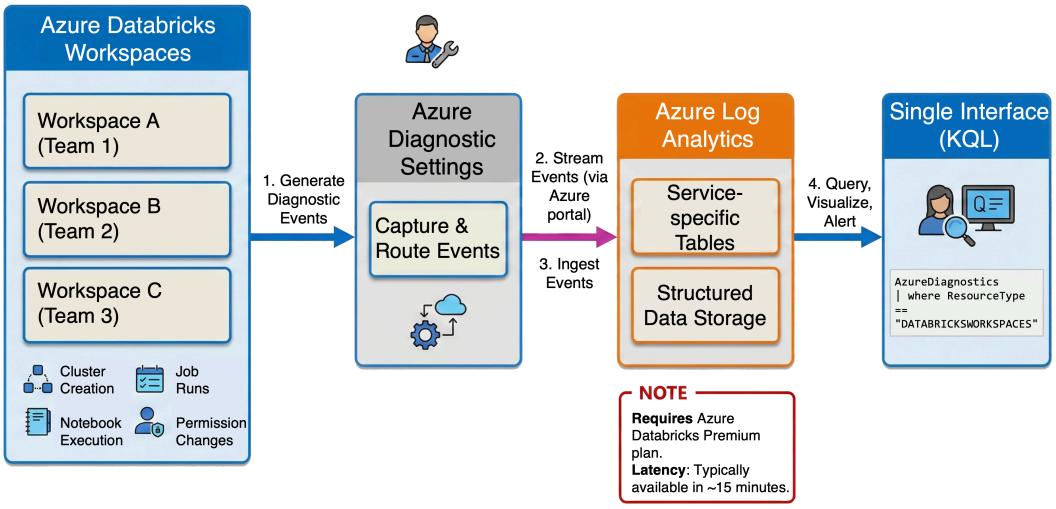
Log streaming sends **diagnostic data** from Azure Databricks to Azure Log Analytics in **near real-time**. This integration uses **Azure diagnostic settings**, which capture events from your Azure Databricks workspace and route them to a Log Analytics workspace.

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

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