

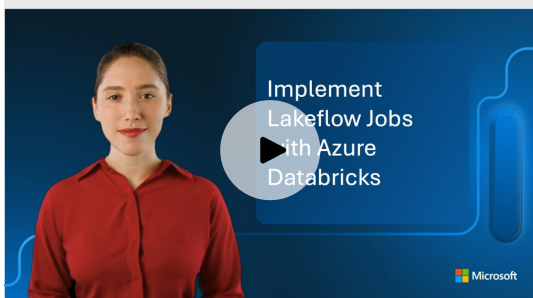
Implement Lakeflow Jobs with Azure Databricks

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.

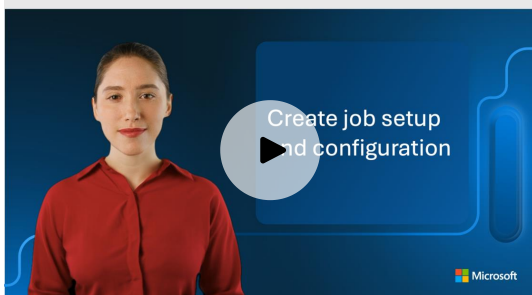
This module guides you through implementing these Lakeflow Jobs capabilities. You explore **job creation** and **task configuration**, event-based and scheduled triggers, alerting strategies, and **retry policies** that keep your data pipelines running smoothly. By mastering these concepts, you build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

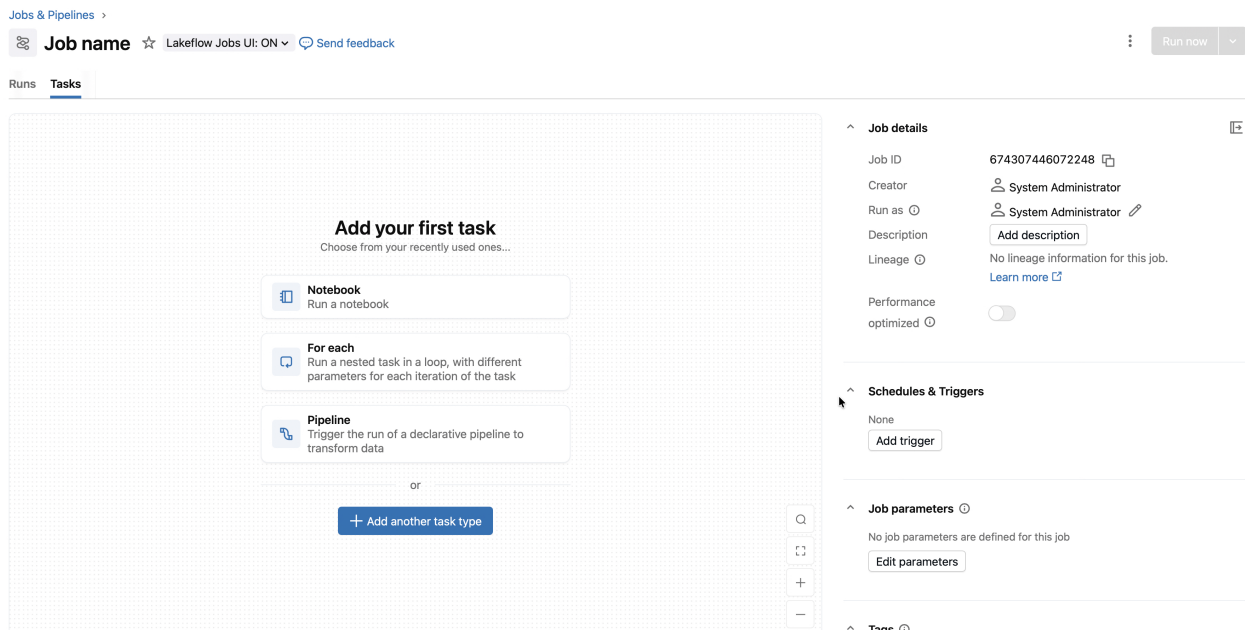
- A **task** containing the logic to run
- A **compute resource** to execute the task
- A **unique name** to identify the job

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
```

The screenshot shows the 'Create task' form in the Azure Databricks interface. At the top, a workflow diagram shows two tasks: 'clean_customers' and 'clean_orders'. The 'clean_orders' task is selected, and its configuration form is displayed below. The form includes the following fields:

- Task name***: clean_orders
- Type***: Notebook
- Source***: Workspace
- Path***: s/instructions/design-and-implement-data-pipelines/03_clean_orders
- Compute***: Serverless (Autoscaling)
- Depends on**: clean_customers
- Run if dependencies**: All succeeded
- Environment and Libraries***: Notebook Environment
- Parameters**: UI/JSON

At the bottom right, there are 'Cancel' and 'Create task' buttons.

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.

Parameters ⓘ UI JSON

Key	Value
date	{{job.trigger.time.iso_date}} {}

[+ Add](#)

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.

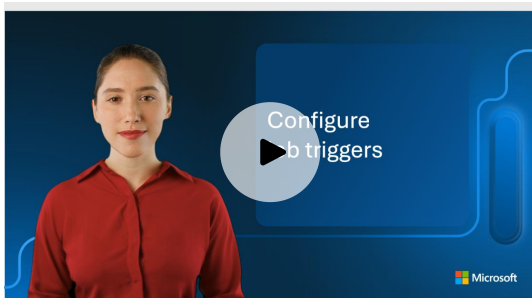
With your job created, tasks configured, dependencies set, and permissions assigned, you're ready to run your workflow. The next step is understanding how to monitor job execution and handle run outcomes.

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.

Schedules & Triggers

×

Trigger type

None (manual) ▾

Scheduled

File arrival

Table update

Continuous ⓘ

Cancel

Save

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/
```

Schedules & Triggers



Trigger Status

- ☒ Active
☐ Paused

Trigger type

File arrival



This job does not have any failure notifications. Consider adding email or webhook notifications to alert if the trigger evaluation fails.

File arrival triggers monitor cloud storage paths for new files. These paths are either Unity Catalog volumes or external locations managed through Unity Catalog.

Storage location ⓘ

e.g. '/Volumes/mycatalog/myschema/myvolume/path_within_volume/' or 'abfss://filesystem@accountname.dfs.core.windows.net/path/'

Advanced ^

Minimum time between triggers ⓘ

00h 00m



Wait after last change ⓘ

00h 00m



Test trigger

Cancel

Save

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.

Schedules & Triggers



Trigger Status

- ☒ Active
☐ Paused

Trigger type

Continuous ⓘ

A continuous job ensures that there is always one active run. A new run is created as soon as the previous one finished (due to failure or success) or when there are none.

Advanced ^

Task retry mode

- ☒ On failure ☐ Never

When On failure is selected, a failed task will retry a few times before a new job run is created.

Cancel

Save

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.

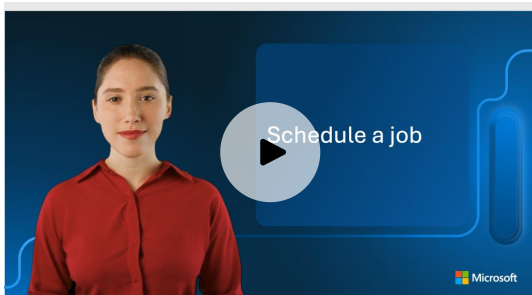
Now that you understand how to configure triggers that automate job execution based on data events, you can build responsive pipelines that process data as it becomes available.

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

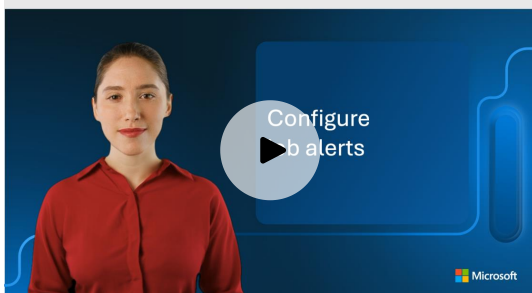
- Schedule jobs with buffer time between dependent pipelines
- Don't rely on sub-minute precision for business-critical timing
- Monitor job run history to identify scheduling patterns or delays
- Use the minimum 10-second interval rule when planning high-frequency jobs

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.

Create a new destination



Type

 Webhook 

Email

Slack

Webhook 

PagerDuty

Microsoft Teams

Working with external endpoints, Databricks
username and password for each configured



URL

Username (optional)

Password (optional)

Cancel

Create

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 notifications



Supported destinations: Email, Microsoft Teams, PagerDuty, Slack, Webhook

Destination	Start	Success	Failure	Duration warning	Streaming backlog	
MS Teams	☐	☐	☒	☐	☐	
someone@example.com	☒	☒	☒	☐	☐	

☐ Mute notifications for skipped runs

☐ Mute notifications for canceled runs

+ Add notification

Cancel

Save

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



Supported destinations: Email, Microsoft Teams, PagerDuty, Slack, Webhook

Destination	Start	Success	Failure	Duration warning	Streaming backlog	
 MS Teams	☐	☒	☒	☐	☐	

- ☐ Mute notifications for skipped runs
- ☐ Mute notifications for canceled runs
- ☒ Mute notifications until the last retry

+ Add notification

Cancel

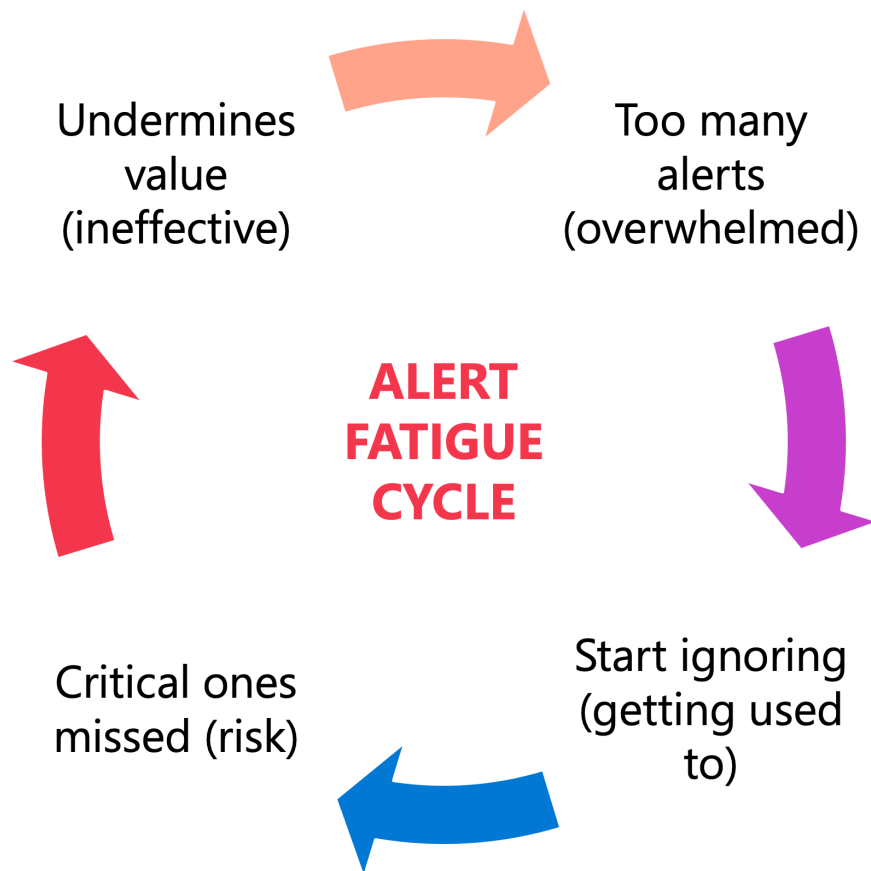
Save

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.

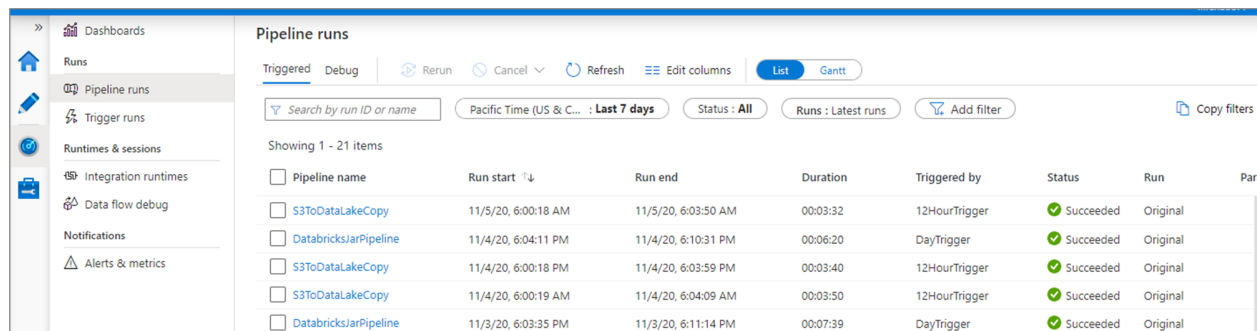
Webhook payloads include the event type, workspace ID, job information, and run details:

```
{
  "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.



Pipeline name	Run start	Run end	Duration	Triggered by	Status	Run
S3ToDataLakeCopy	11/5/20, 6:00:18 AM	11/5/20, 6:03:50 AM	00:03:32	12HourTrigger	Succeeded	Original
DatabricksJarPipeline	11/4/20, 6:04:11 PM	11/4/20, 6:10:31 PM	00:06:20	DayTrigger	Succeeded	Original
S3ToDataLakeCopy	11/4/20, 6:00:18 PM	11/4/20, 6:03:59 PM	00:03:40	12HourTrigger	Succeeded	Original
S3ToDataLakeCopy	11/4/20, 6:00:19 AM	11/4/20, 6:04:09 AM	00:03:50	12HourTrigger	Succeeded	Original
DatabricksJarPipeline	11/3/20, 6:03:35 PM	11/3/20, 6:11:14 PM	00:07:39	DayTrigger	Succeeded	Original

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.

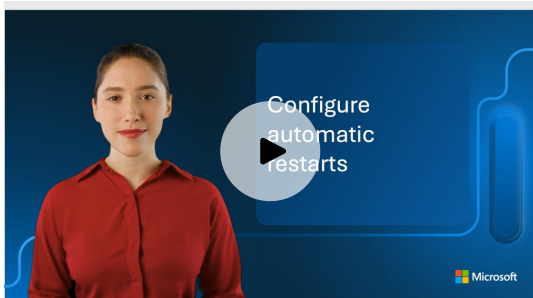
With alerts configured at the right granularity and sent to the right channels, you can respond to issues quickly while avoiding the distraction of unnecessary notifications. As you deploy more jobs to production, revisit your alerting strategy to ensure it scales with your workload.

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

Retry Policy ⓘ



Jobs that fail are retried a number of times based on the following policy. You can specify a maximum number of attempts for a run and a minimal interval between attempts. On serverless, automatic optimization can affect retries. To disable all retries, turn off automatic optimization.

Retry at most and wait between retries

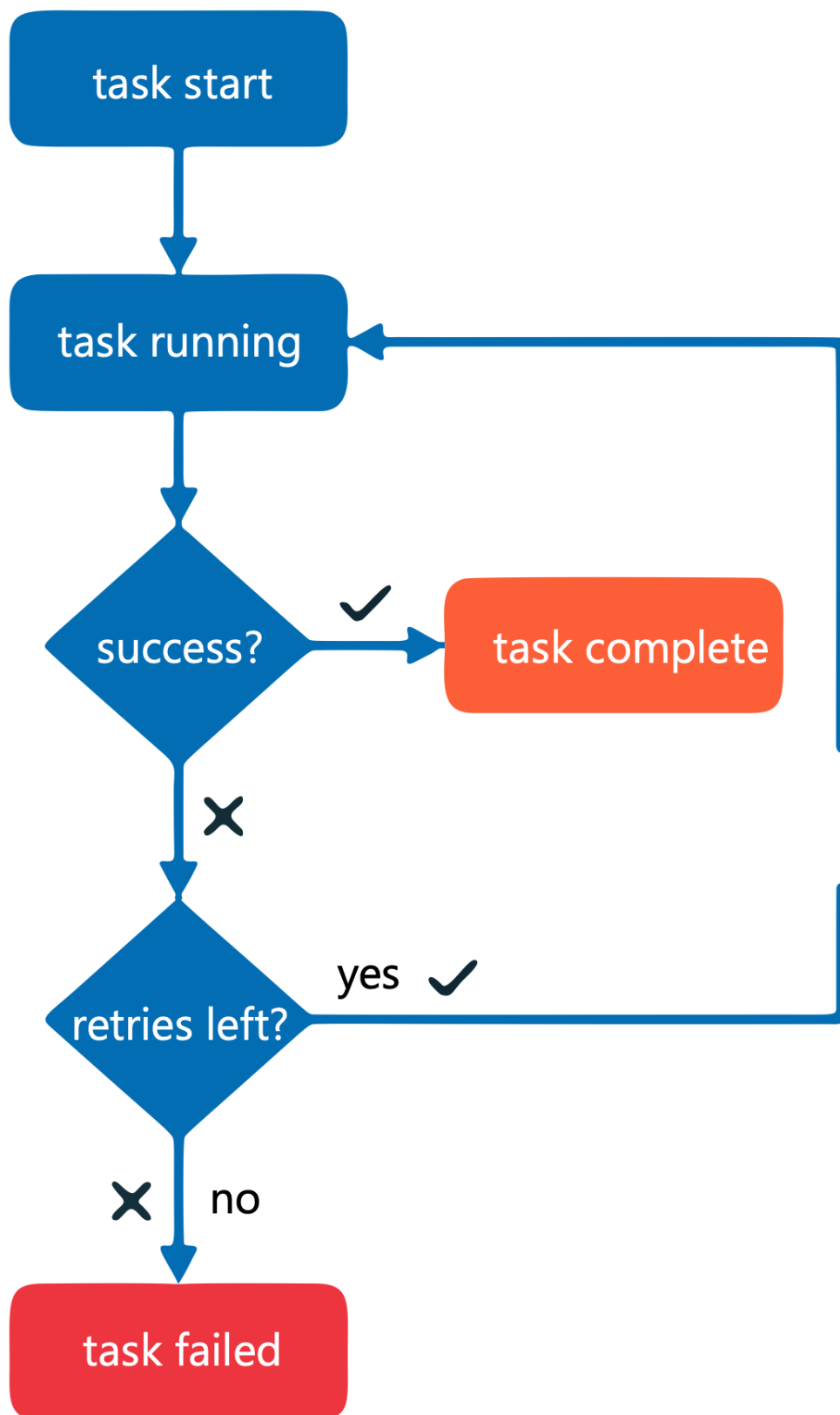
☐ Retry on timeout

☒ Enable serverless auto-optimization (may include at most 3 retries)

Cancel

Confirm

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.



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**

Schedules & Triggers

Trigger Status

☒ Active

☐ Paused

Trigger type

Continuous ⓘ

A continuous job ensures that there is always one active run. A new run is created as soon as the previous one finished (due to failure or success) or when there are none.

Advanced ^

Task retry mode

☒ On failure ☐ Never

When On failure is selected, a failed task will retry a few times before a new job run is created.

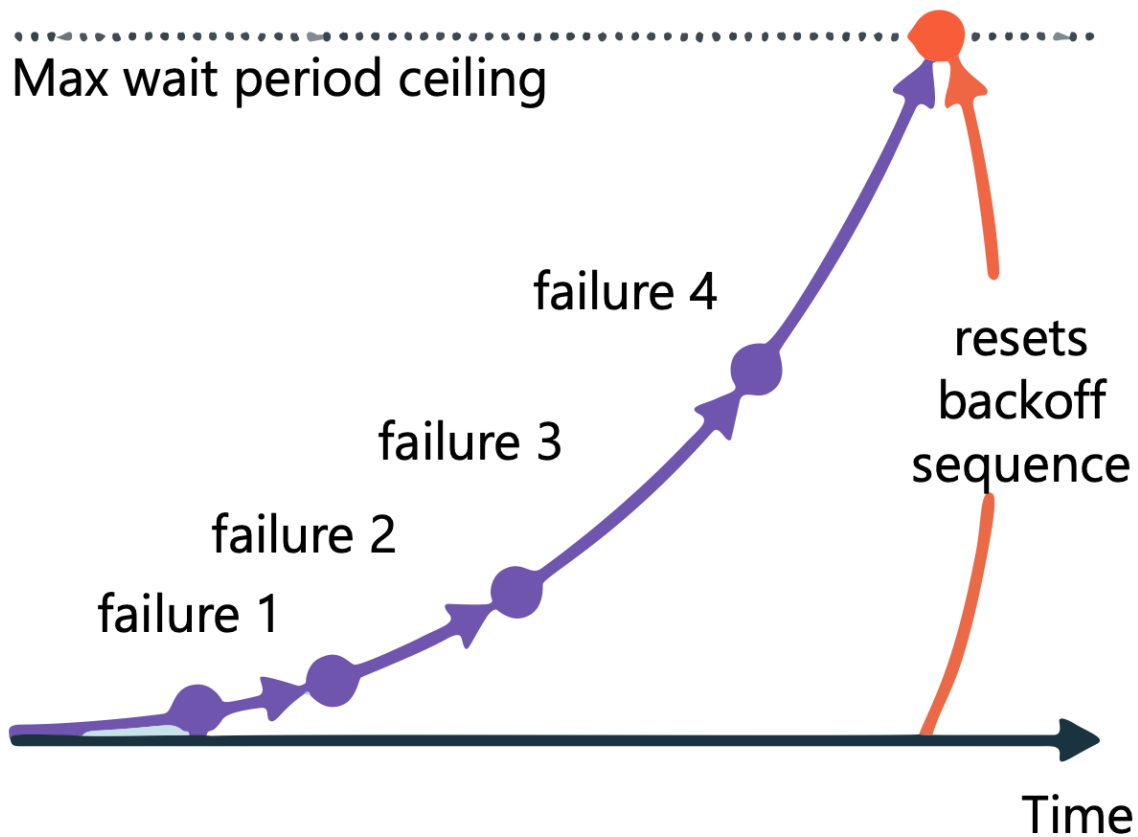
Cancel

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

Metric thresholds



"clean_customers" task

Metric	Warning threshold ⓘ	Timeout threshold ⓘ
Run duration ▼	00h 03m 00s ✕	00h 08m 00s ✕
Cancel Save Save and edit notifications		

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.