

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>

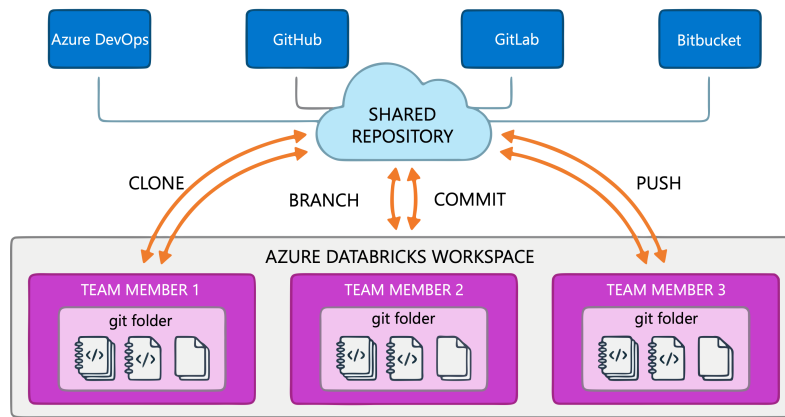
Apply Git version control best practices

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**.

Create Git folder



Git repository URL

https://example.com/organization/project.git

Git provider ⓘ

Select a Git provider

Git folder name

☐ Sparse checkout mode ⓘ

GitHub

GitHub Enterprise Server

Bitbucket Cloud

Bitbucket Server

GitLab

GitLab Enterprise Edition

Azure DevOps

Cancel

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**.

databricks-labs [Send feedback](#)



Branch: main

Create Branch

6312156



History

Pull

Changes

Settings

11 changed files

instructions/design-and-implement-data-pipeline
s/

- ☒ 01_setup_environment.ipynb A
- ☒ 02_clean_customers.ipynb A
- ☒ 03_clean_orders.ipynb A
- ☒ 04_create_gold_layer.ipynb A
- ☒ 05_notify_failure.ipynb A
- ☒ 06_compare_results.ipynb A

Added a lab on pipelines

Description (optional)

Commit & Push

instructions/design-and-implement-data-pipelines/01_setup_environment.ipynb

Code

Raw file

Note: This notebook may have outputs, but they are not included because the workspace administrator disabled committing outputs.

Cmd 1

```
+ # Create a dedicated schema for this lab
+ schema_name = "pipeline_lab"
+ spark.sql(f"CREATE SCHEMA IF NOT EXISTS {schema_name}")
+ spark.sql(f"USE {schema_name}")
+
+ print(f"Schema '{schema_name}' created and selected.")
```

Cmd 2

```
+ from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType,
+ from datetime import date
+
+ # Define customer data (simulating raw ingested data)
+ customers_data = [
+     (1, "Alice Johnson", "alice@email.com", "US-East", "2026-01-15"),
+     (2, "Bob Smith", "bob@email.com", "US-West", "2026-01-16")
+ ]
```

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.

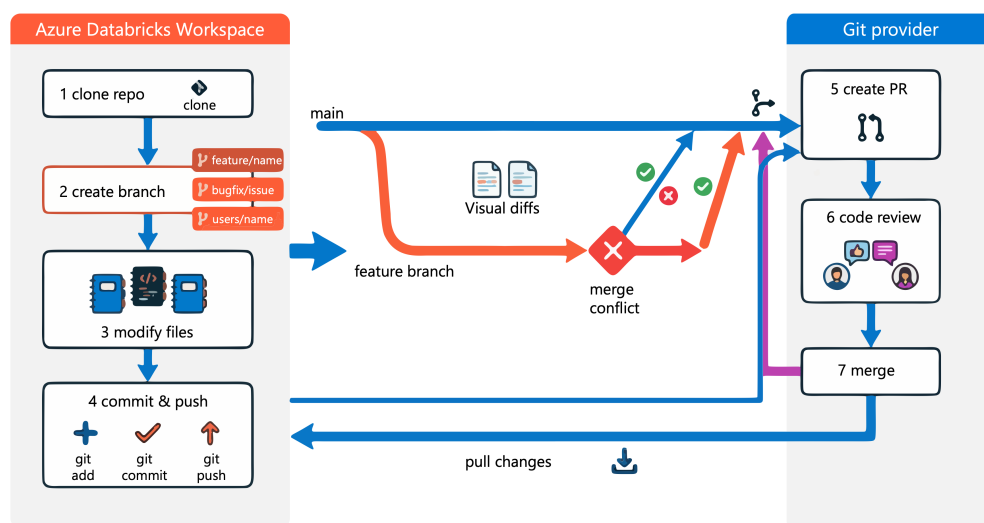
3. Manage branching and pull requests

<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.



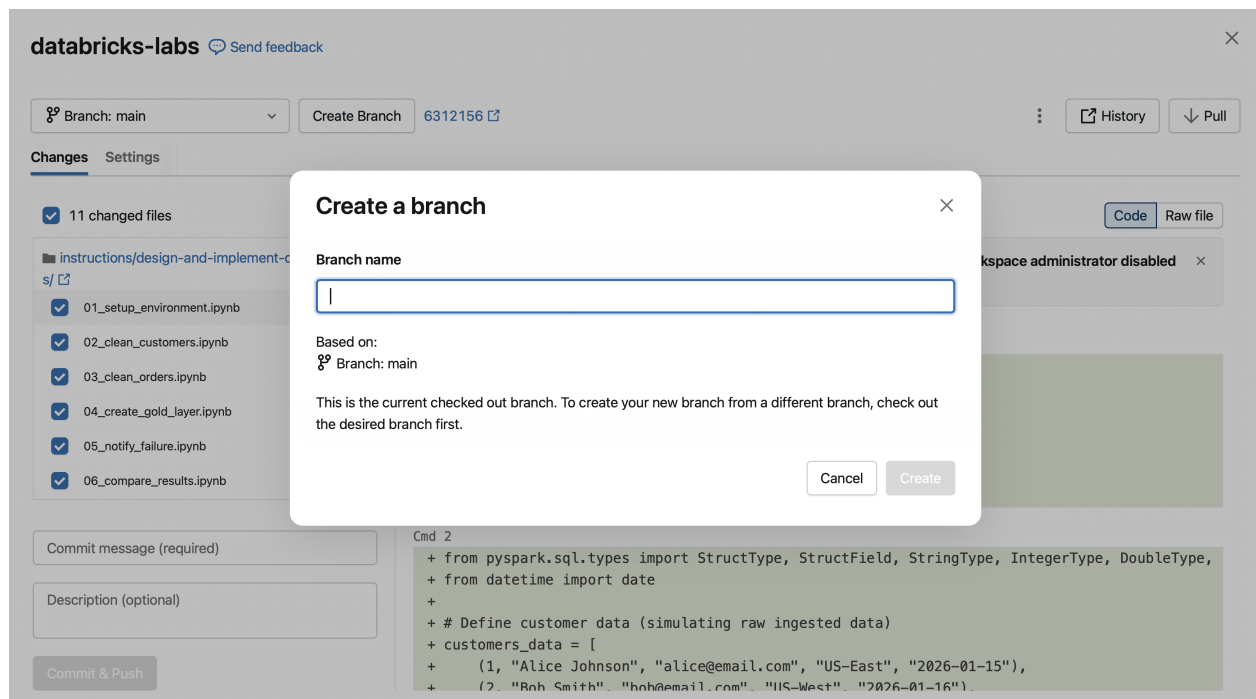
Azure Databricks Git folders provide a visual Git client that integrates directly with your workspace. This integration lets you perform common Git operations—branching, committing, and merging—without leaving the Azure Databricks environment.

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

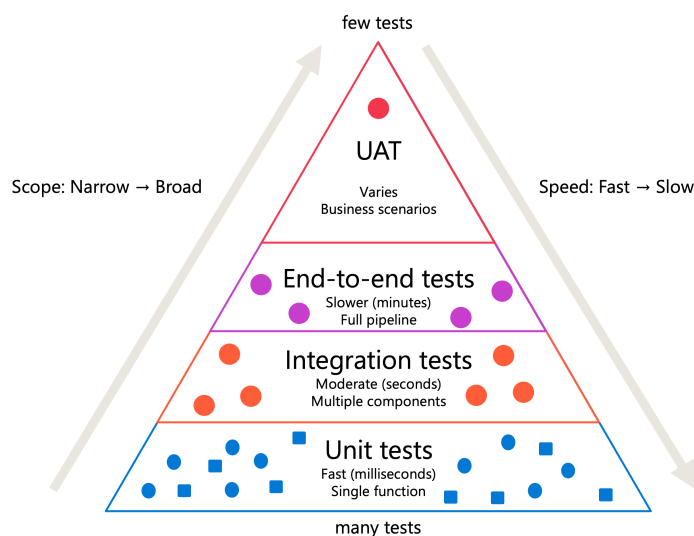
Completed

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("X 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

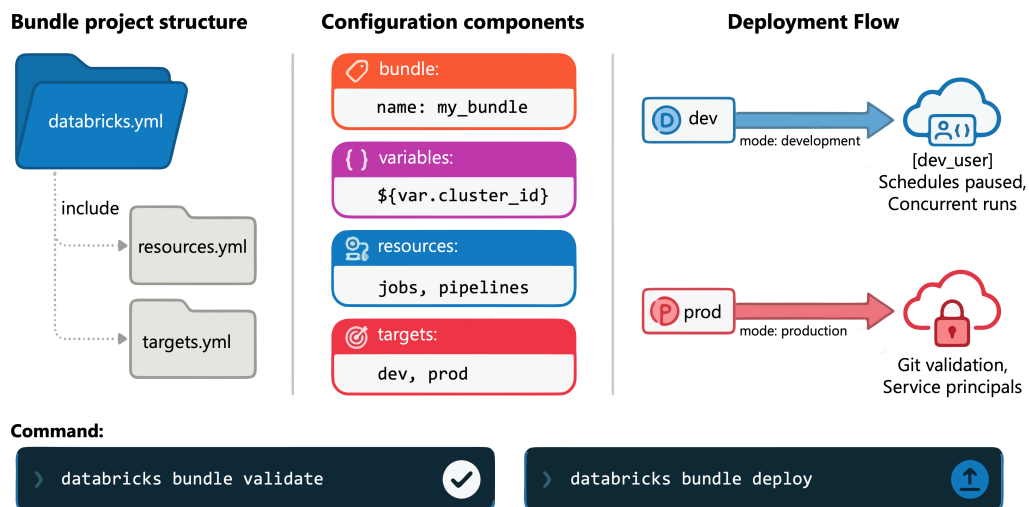
<https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/5-configure-package-databricks-asset-bundles>

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/*.yaml

workspace:
  host: https://adb-1234567890123456.7.azure.databricks.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.azure.databricks.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

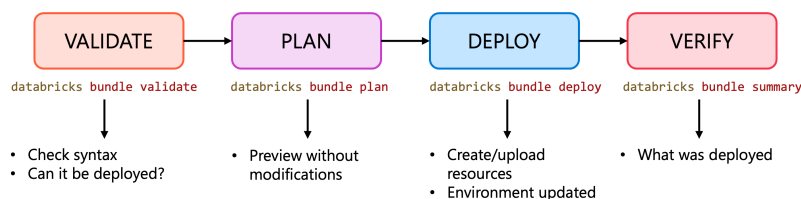
<https://learn.microsoft.com/en-us/training/modules/implement-development-lifecycle-processes-in-azure-databricks/6-deploy-bundle-databricks-cli>

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=123
  Pipelines:
    transform_pipeline:
      Name: [dev someone] transform_pipeline
      URL: https://adb-1234567890123456.7.azuredatabricks.net/pipelines/abc-123-d
```

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.