

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

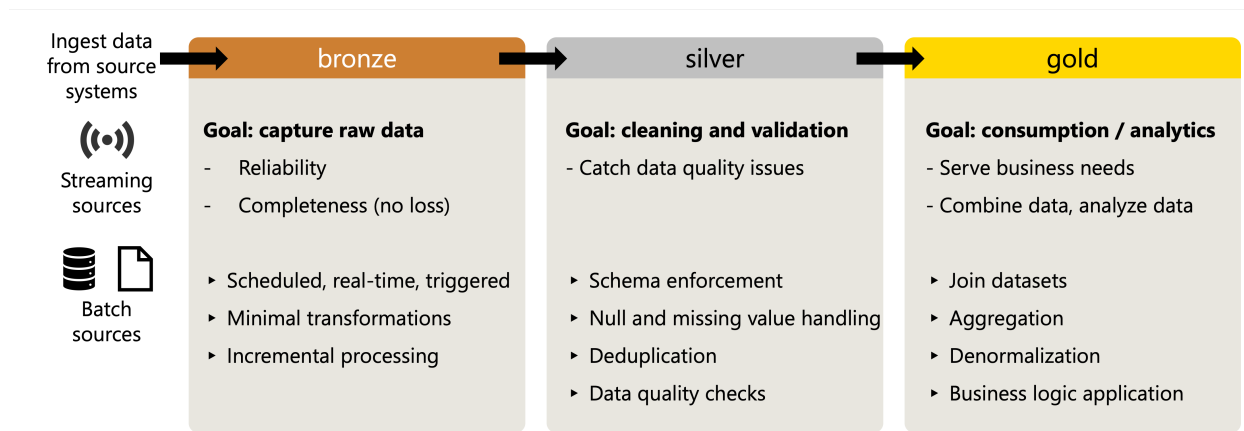
<https://learn.microsoft.com/en-us/training/modules/design-implement-data-pipelines/2-design-order-operations-pipeline>

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

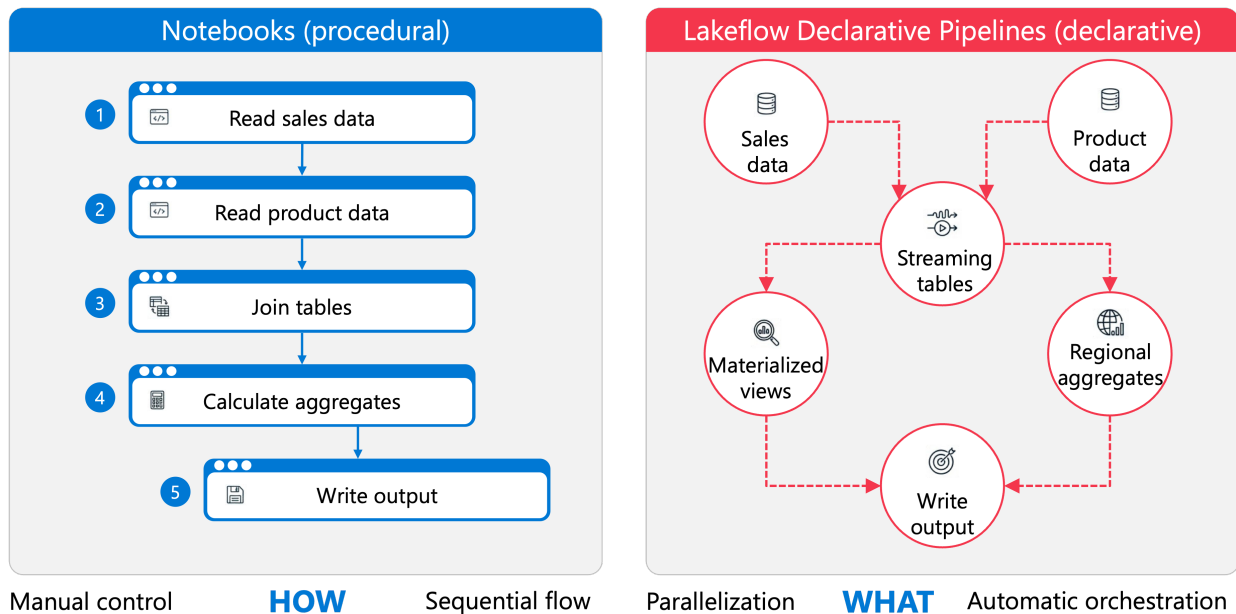
Completed

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

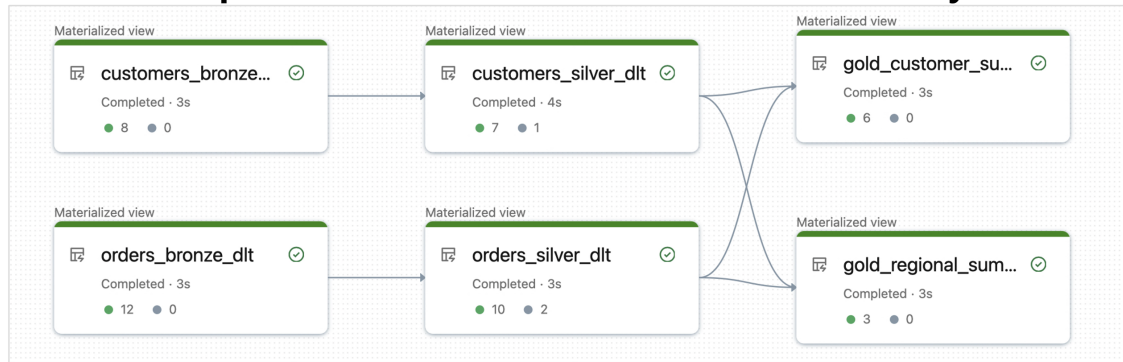
When you design task logic for a **Lakeflow Job**, you make decisions that affect how your workflow executes, scales, and responds to changing conditions. A well-designed job balances performance, cost, and maintainability while meeting business requirements.

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

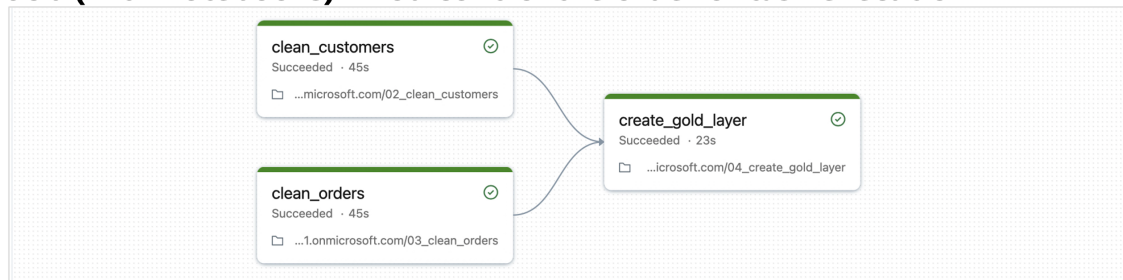
Understand task relationships in a job

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.

Declarative Pipeline – Databricks builds the DAG automatically



Job (with notebooks) – You control the order of task execution



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:

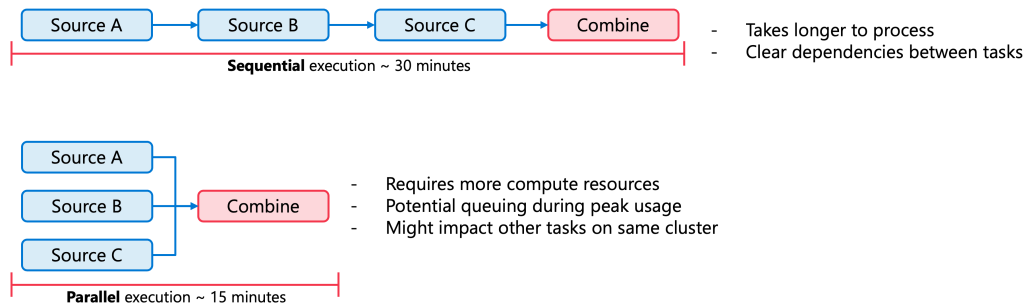
Dependency condition	Behavior
All succeeded	Runs only when all upstream tasks complete successfully
At least one succeeded	Runs when any upstream task succeeds
None failed	Runs when no upstream tasks failed, even if some are skipped
All done	Runs after all upstream tasks finish, regardless of outcome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

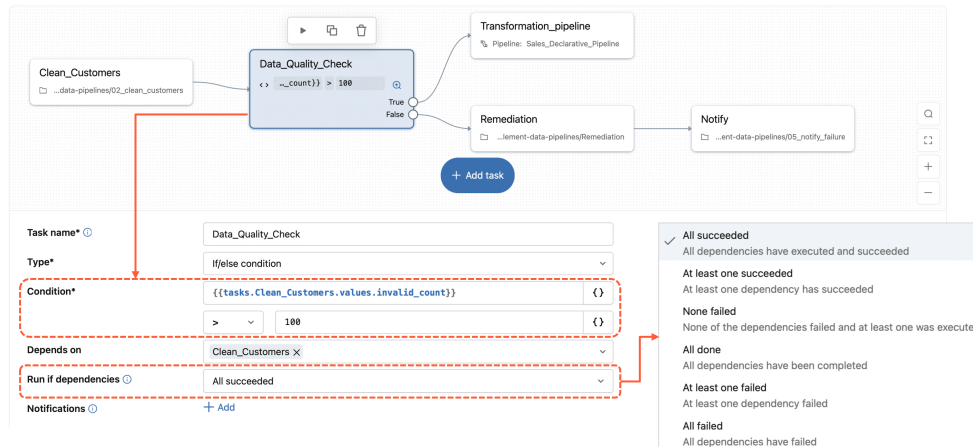
Use **If/else condition** tasks when you need to choose between different processing paths based on values or state. The condition evaluates an expression using task values, job parameters, or dynamic references.

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.

The screenshot displays a workflow editor interface. At the top, a task named 'Get_regions' is connected to a task named 'Process_files_for_each_region'. The 'Process_files_for_each_region' task is highlighted with a dashed blue box. Below the tasks, a configuration panel for the 'Process_files_for_each_region' task is shown. The 'Inputs*' field is highlighted with a dashed red box and contains the expression `{{tasks.Get_regions.values.regions_to_process}}`. Other fields include 'Task name*' (set to 'Process_files_for_each_region'), 'Type*' (set to 'For each'), 'Concurrency (optional)' (set to 1), 'Depends on' (set to 'Get_regions'), and 'Run if dependencies' (set to 'All succeeded').

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

Design error handling in pipelines and jobs

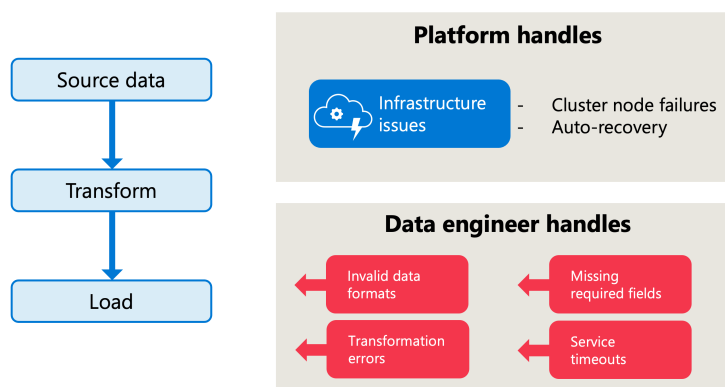
Completed

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:

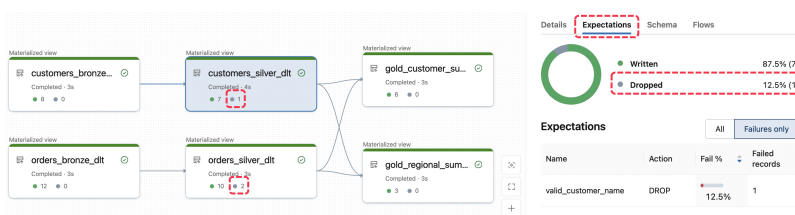
- Source data contains invalid formats or missing required fields

- 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.

Notifications ⓘ

✉ user@example.com

On failure

Edit notifications

Retries ⓘ

1 min delay, at most 3x (4 total attempts) ✎

Metric thresholds ⓘ

Duration warning: 5m 0s ✎

Duration timeout: 10m 0s ✎

Edit metric thresholds

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>

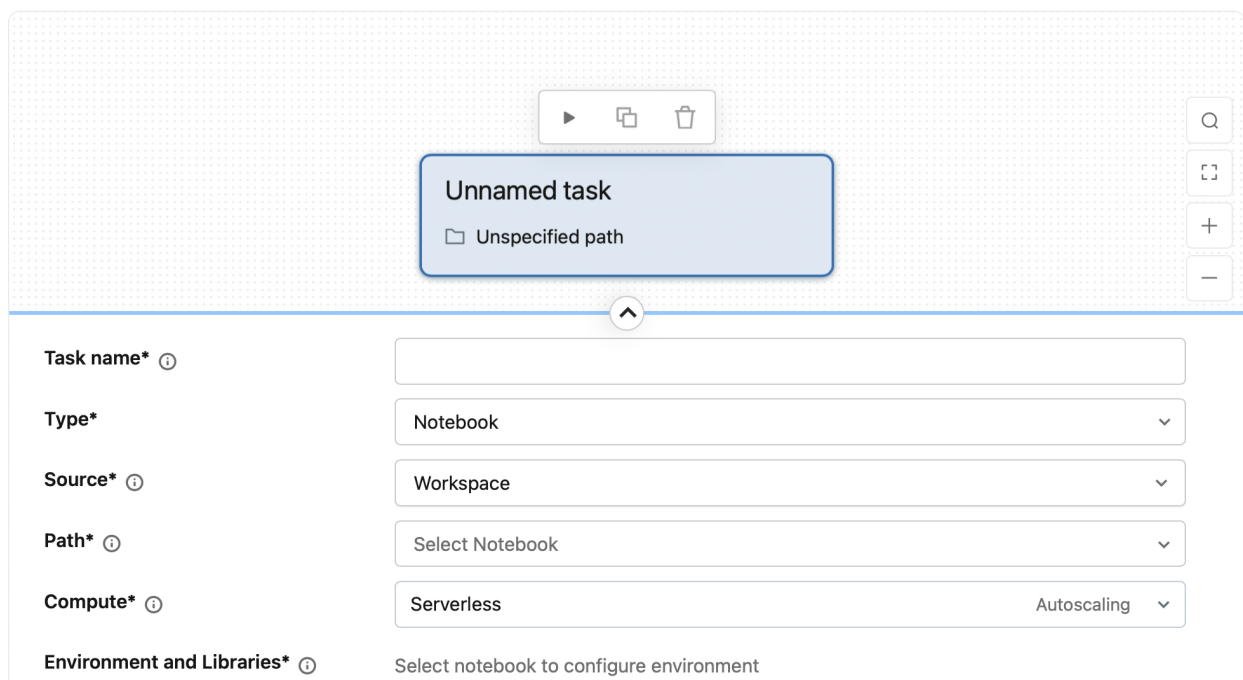
Create pipeline with notebook

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.



The screenshot shows the Azure Databricks interface for adding a new task to a job. At the top, there's a toolbar with icons for play, copy, and delete. Below it, a blue box labeled "Unnamed task" contains a folder icon and the text "Unspecified path". To the right of this box is a vertical toolbar with icons for search, zoom in, zoom out, and a minus sign. Below the task box is a horizontal line with an upward arrow icon. The main form has several fields: "Task name*" with a help icon, "Type*" set to "Notebook", "Source*" set to "Workspace", "Path*" set to "Select Notebook", "Compute*" set to "Serverless" with an "Autoscaling" dropdown, and "Environment and Libraries*" with a note "Select notebook to configure environment".

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
<code>EXPECT</code> (default)	Retains invalid records, tracks violation counts in metrics
<code>ON VIOLATION DROP ROW</code>	Removes invalid records from the output
<code>ON VIOLATION FAIL UPDATE</code>	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

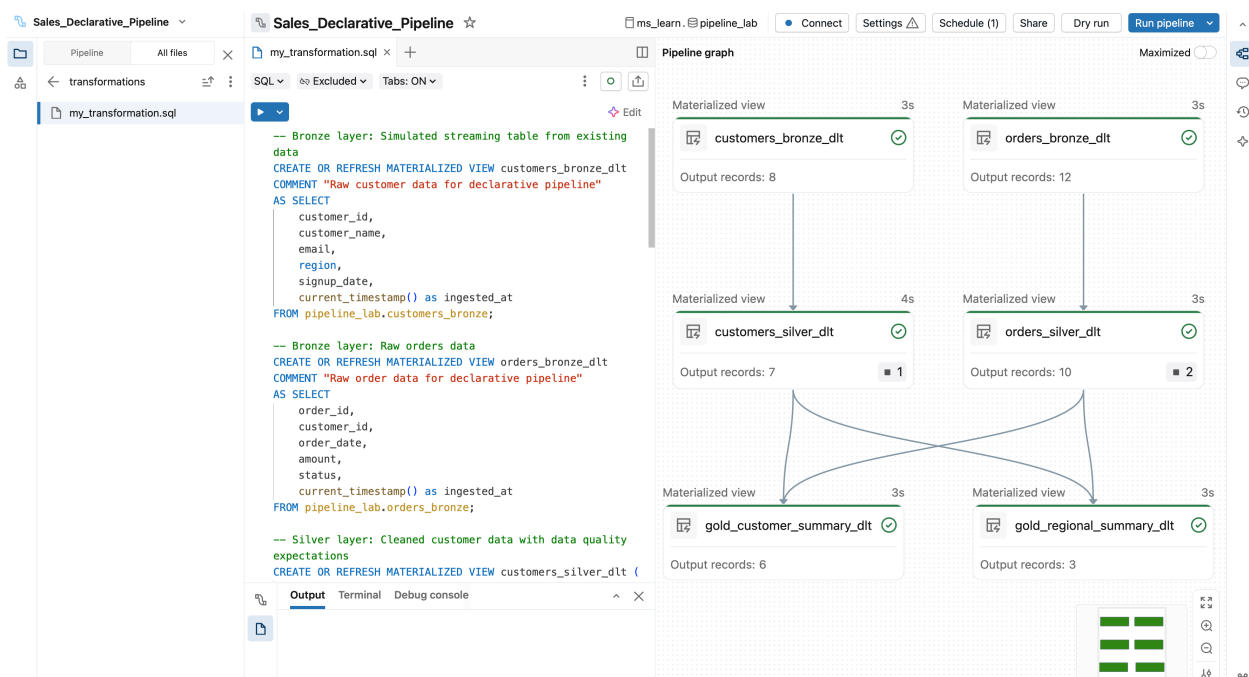
The Lakeflow Spark Declarative Pipelines **Advanced** product edition is required to use expectations. If your pipeline includes expectations with the Core or Pro editions, you receive an error.

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.