Databricks Jobs & Workflows

Complete Guide - Day 7

Yash Kavaiya

Databricks 14-Days AI Challenge

January 15, 2026

Agenda

- Introduction to Databricks Jobs
- Jobs vs Notebooks
- Multi-Task Workflows
- Parameters & Widgets
- Scheduling Jobs
- Error Handling & Retries
- Bronze→Silver→Gold Pipeline
- Best Practices

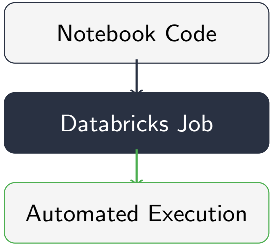
What is a Databricks Job?

Definition

A **Databricks Job** is a mechanism to run data processing workloads in a scheduled, automated, and production-ready manner.

Simple Analogy:

- ▷ **Notebook** = Manually cooking a meal step by step
- ▷ **Job** = Automated cooking machine that prepares meals at scheduled times



Why Do We Need Jobs?

Common Requirements:

- Run transformations at specific times
  - ▷ ▷ E.g., daily at 2 AM when load is low
- Ensure reliable processing
  - ▷ ▷ Without manual intervention
- Chain multiple processing steps

Jobs Provide:

- Handle failures gracefully
- Automatic retries
- Track execution history
- Monitor performance
- Production-grade execution

Jobs vs Notebooks: Core Differences

Aspect	Notebooks	Jobs
Purpose	Interactive development, exploration	Automated production execution
Execution	Manual (user clicks “Run”)	Scheduled, triggered, or API
Interaction	Required - active user	Not required - autonomous
Cluster	Often shared interactive	Job-specific, auto start/stop
State	Variables persist during session	Fresh state each run
Error Handling	Manual debugging	Automatic retries & alerts

When to Use What?

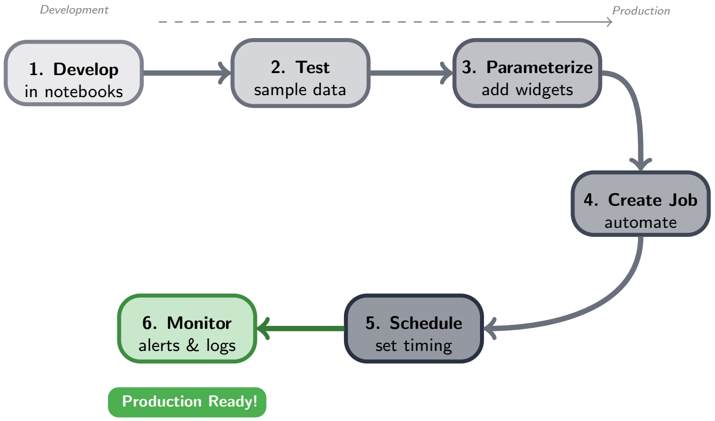
Use NOTEBOOKS when:

- Exploring new data sources
- Developing & testing transformations
- Creating visualizations
- Debugging pipeline issues
- Sharing analysis interactively

Use JOBS when:

- Running daily/weekly ETL
- Processing data on schedule
- Production workloads
- Multi-step workflows
- Automated, unattended execution

Development-to-Production Journey

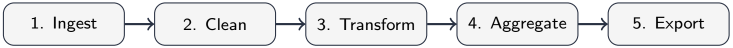


Multi-Task Workflows

Definition

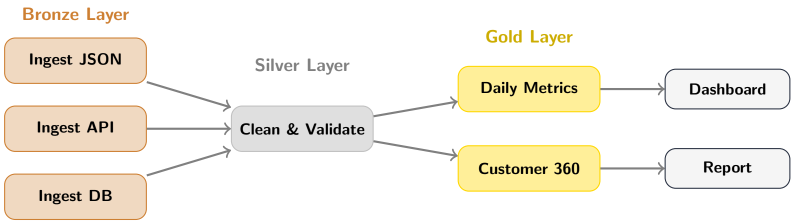
A **Multi-Task Workflow** is a job containing multiple tasks that can run in sequence, in parallel, or in complex dependency patterns.

Why Multi-Task? Real pipelines have multiple steps:



Each step depends on the previous one completing successfully!

Workflow Architecture: Medallion Pipeline

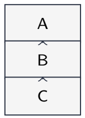


Task Types Available

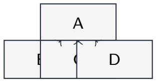
Task Type	Description	Use Case
Notebook	Runs a Databricks notebook	Data transformations, ETL
Python Script	Executes Python file	Standalone applications
SQL	Runs SQL queries	Aggregations, transforms
JAR	Java/Scala JAR files	Spark applications
Delta Live Tables	DLT pipelines	Declarative ETL
If/else	Conditional branching	Dynamic workflow logic
For each	Loop over items	Processing multiple tables

Dependency Patterns

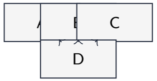
Sequential:



Fan-out (Parallel):



Fan-in:



Dependency Conditions:

- **All succeeded** - Run only if ALL upstream tasks completed
- **At least one succeeded** - Run if ANY upstream succeeded
- **None failed** - Run if no upstream failed (includes skipped)
- **All done** - Run regardless of outcomes

Understanding Parameterization

What is Parameterization?

Making code flexible by accepting inputs at runtime rather than hardcoding values.

Why Parameterize?

- Reuse same notebook for different scenarios
- Run pipelines for different dates
- Test with different configurations
- Promote code from dev to prod

Databricks Widgets create input controls that accept values either:

- **Interactively** - in the notebook UI
- **Programmatically** - when run as a job

Widget Types

Type	Method	Best For
Text	`widgets.text()`	File paths, table names, custom strings
Dropdown	`widgets.dropdown()`	Environment selection, layer names
Combobox	`widgets.combobox()`	Options with occasional custom values
Multiselect	`widgets.multiselect(`	Processing multiple tables/-columns

Widget Implementation

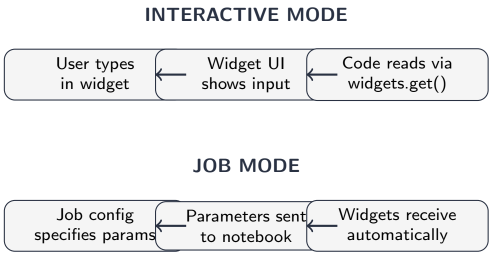
Creating Widgets:

```
# Text Widget
dbutils.widgets.text(
    "source_path",
    "/mnt/raw/data",
    "Source Data Path"
)

# Dropdown Widget
dbutils.widgets.dropdown(
    "layer",
    "bronze",
    ["bronze", "silver", "gold"],
    "Processing Layer"
)
```

Retrieving Values:

```
# Get single value
source = dbutils.widgets.get(
    "source_path"
)
layer = dbutils.widgets.get(
    "layer"
)

# Multiselect returns comma-
# separated string
tables = dbutils.widgets.get(
    "tables"
).split(",")
```

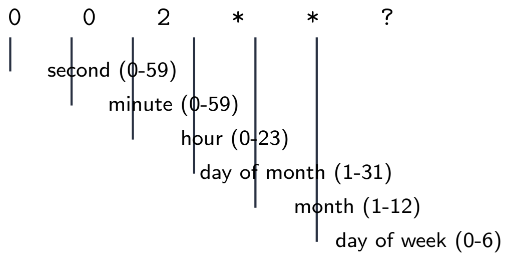
Parameter Flow: Interactive vs Job


INTERACTIVE MODE
User types in widget
Widget UI shows input
Code reads via widgets.get()
JOB MODE
Job config specifies params
Parameters sent to notebook
Widgets receive automatically

Scheduling Concepts

Type	Description	Use Case
None (Manual)	Only when manually triggered	Testing, ad-hoc execution
Scheduled	Time-based (Cron)	Regular ETL, daily reports
Continuous	Runs continuously, restarts after completion	Real-time/streaming
File Arrival	Triggers on new files	Event-driven processing

Cron Expression Format



Common Schedules:

- `0 0 2 * * ?` - Every day at 2:00 AM
- `0 0 * * * ?` - Every hour
- `0 */15 * * * ?` - Every 15 minutes
- `0 0 9 ?  * MON-FRI` - Weekdays at 9 AM

Scheduling Best Practices

DO:

- Avoid peak hours (schedule 2-4 AM)
- Stagger similar jobs
- Consider dependencies with buffer time
- Be explicit about time zones
- Monitor execution duration

DON'T:

- Schedule all jobs at midnight
- Ignore DST transitions
- Schedule without buffer time
- Forget to set alerts for long-running jobs

Error Handling: Why It Matters

Production Reality

Sources become unavailable, data formats change, clusters run out of memory. Proper error handling ensures **reliability**, **visibility**, **recovery**, and **debugging**.

Types of Failures:

Type	Examples	Solution
Transient	Network timeout, temp unavailable	Retry automatically
Resource	OOM, disk full	Scale up, retry
Data	Schema mismatch, corrupt files	Fix data, alert team
External	Source down, API rate limited	Wait and retry

Retry Configuration

Job-Level Retries:

- **Max Retries**: 0-3 recommended
- **Default**: 0 (no retries)
- **Retry on Timeout**: Enable for resource constraints

Task-Level Settings:

- **Max Retries**: 1-3 for most tasks
- **Min Retry Interval**: 30-60 seconds
- **Max Retry Interval**: 300-600 seconds

Error Handling Patterns

Pattern: Exit Codes for Job Control

```python
try:
    df = spark.read.format("delta").load("/path/to/data")
    record_count = df.count()

    if record_count == 0:
        dbutils.notebook.exit('{"status": "success", "records": 0}')

    processed = transform_and_write(df)
    dbutils.notebook.exit(f'{{"status": "success", "records": {processed}}}')

except Exception as e:
    error_msg = str(e).replace('"', '\\"')
    dbutils.notebook.exit(f'{{"status": "failed", "error": "{error_msg}"}}')
    raise  # Trigger job retry
```

Alerting Configuration

Alert Type	Trigger	Notification
On Failure	Any task fails (after retries)	Email, Slack, PagerDuty
On Success	Job completes successfully	Email (optional)
On Duration	Exceeds expected duration	Monitoring system
SLA Breach	Doesn't complete on time	PagerDuty

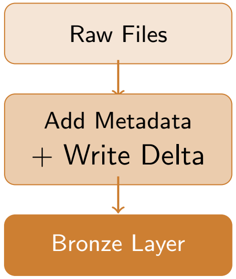
Bronze Layer: Raw Ingestion

Characteristics:

- Raw data as-is from source
- Added metadata for lineage
- No business transformations
- Preserved original schema

Metadata Added:

- `_ingestion_timestamp`
- `_source_file`
- `_process_date`
- `_row_hash` (SHA-256)



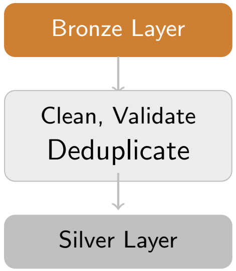
Silver Layer: Cleansed Data

Characteristics:

- Cleaned and validated data
- Standardized formats
- Deduplication applied
- Schema enforced
- Data quality checks passed

Quality Rules:

- Remove nulls in required fields
- Standardize strings (trim)
- Remove duplicates
- Validate value ranges



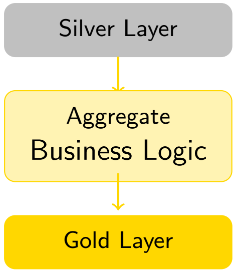
Gold Layer: Business Aggregations

Aggregation Types:

- **Daily Summary**: Totals by category
- **Customer Metrics**: Lifetime value, segmentation
- **Product Performance**: Revenue rankings
- **Regional Analysis**: Revenue by region

Customer Segments:

- Platinum: LTV $\geq$ \$10,000
- Gold: LTV $\geq$ \$5,000
- Silver: LTV $\geq$ \$1,000
- Bronze: LTV $<$ \$1,000



Job Configuration: Daily ETL Pipeline

General Settings:

- **Job Name**: daily_etl_pipeline
- **Max Concurrent**: 1
- **Timeout**: 4 hours

Schedule:

- **Cron**: 0 0 2 * * ?
- **Time Zone**: America/New_York
- Daily at 2:00 AM Eastern

Cluster Config:

- **Workers**: 2-8 (autoscaling)
- **Spark**: 12.2.x-scala2.12
- **Auto-terminate**: 10 min

Alerts:

- On Failure: Email team
- On Duration $>$ 2h: Warn team

Job Design Best Practices

Category	Best Practice	Rationale
Idempotency	Safely re-runnable	Retry without data corruption
Parameterize	Use params for env values	Same code in dev/staging/prod
Small Tasks	Break into focused tasks	Easier debugging, parallelism
Logging	Meaningful log messages	Aid troubleshooting
Exit Codes	Use notebook.exit()	Enable downstream decisions
Checkpoints	Save progress periodically	Recovery without restart

Cluster & Scheduling Best Practices

Cluster Configuration:

- Use **Job Clusters** for production
  - ▷ ▷ Cost-efficient, isolated
- Enable **Autoscaling**
  - ▷ ▷ Min 2, max based on volume
- Use **Spot Instances**
  - ▷ ▷ 60-90% cost reduction
- Use **LTS Spark versions**

Scheduling Tips:

- Avoid scheduling at midnight
- Use buffer time between jobs
- Be explicit about time zones
- Set duration alerts
- Stagger similar jobs

Troubleshooting Common Issues

Issue 1: Job Fails to Start

- **Causes**: Quota exceeded, no capacity, invalid config, permissions
- **Solution**: Check quotas, try different region/instance type

Issue 2: Task Timeout

- **Diagnostic**: Check data volume, Spark UI, data skew
- **Solution**: Increase timeout, optimize queries, add workers

Issue 3: Parameter Not Passed

- **Cause**: Parameter name mismatch between notebook and job
- **Solution**: Ensure exact name match (case-sensitive)

Troubleshooting: Retries Not Working

Common Causes:

- `notebook.exit()` called with error status (counts as success!)
- Exception not raised after exit
- Retry configured at wrong level

Correct Pattern:

```
try:
    result = process_data()
    dbutils.notebook.exit(f'{{"status": "success", "count": {result}}}')
except Exception as e:
    print(f"ERROR: {str(e)}")
    dbutils.notebook.exit(f'{{"status": "failed", "error": "{str(e)}"}}')
    raise  # IMPORTANT: Raise to trigger retry!
```

Key Takeaways

Core Concepts:

1. **Jobs vs Notebooks**: Dev vs Production
2. **Multi-Task Workflows**: Chain with dependencies
3. **Parameters**: Widgets for flexibility
4. **Scheduling**: Cron expressions

Production Patterns:

1. **Error Handling**: Retries & alerts
2. **Medallion**: Bronze→Silver→Gold
3. **Idempotency**: Safe to re-run
4. **Monitoring**: Track & alert

Implementation Checklist

- ☐ Parameterize notebooks with widgets
- ☐ Create separate notebooks per layer
- ☐ Set up multi-task job with dependencies
- ☐ Configure retry settings
- ☐ Set up failure alerting
- ☐ Schedule at appropriate time
- ☐ Test end-to-end with sample data
- ☐ Document the pipeline

Ready to build production pipelines!

Thank You!

Day 7: Databricks Jobs & Workflows

linkedin.com/in/yashkavaiya

easy-ai-labs.lovable.app

Gen AI Guru