■ **Course:**	CSCI E-103: Reproducible Machine Learning
■ **Week:**	Lecture 13
■ **Instructors:**	Eric Gieseke & Ram Sriharsha
■ **Objective:**	Master the continuous improvement cycle: CI/CD pipelines, Infrastructure as Code (IaC), observability, data quality monitoring, and Databricks Asset Bundles (DABs)

Contents

1 Introduction: Beyond "Working Code"

Building a data pipeline that "works" is just the beginning. In enterprise environments, the real challenge is:

- How do you **deploy reliably** across environments (dev, staging, production)?
- How do you **automate testing** so bugs are caught early?
- How do you **monitor quality** so bad data doesn't corrupt downstream systems?
- How do you **manage costs** so your cluster doesn't run up a $100,000 bill?

This lecture bridges the gap between "data science" and "data engineering operations."

Lecture Overview

Key Topics:

1. **Software Development Lifecycle (SDLC):** Dev → Staging → Production
2. **Infrastructure as Code (IaC):** Terraform for reproducible infrastructure
3. **CI/CD Pipelines:** Automated testing and deployment
4. **Observability:** System tables, cost monitoring, data quality
5. **Databricks Asset Bundles (DABs):** Modern deployment packaging

2 Software Development Lifecycle (SDLC)

2.1 The Three-Environment Model

Never deploy code directly to production. Instead, code travels through three environments:

Aspect	Development	Staging	Production
Purpose	Experiment and implement	Test with realistic data	Serve real users
Data	Fake/sample data	Near-production volume	Real data
Compute	Single node, spot instances	Larger clusters	High-availability clusters
Credentials	Developer accounts	Service principals begin	**Only service principals**
Execution	Interactive (notebooks)	Transitioning to automated	**Fully automated**
Cost Focus	Minimize (use spot)	Balance cost/realism	Prioritize reliability

Table 1: *SDLC Environment Comparison*

Definition:

Service Principal A **Service Principal** is a "robot account"—an identity used by automated processes, not humans. In production, pipelines should never run under a developer's personal credentials.

Why? If a developer leaves the company and their account is deactivated, any pipeline running under their credentials will break.

2.2 Best Practices for Building Data Products

1. **Understand the Use Case**
   - Document business requirements and domain-specific challenges
   - Define functional requirements (what it does) and non-functional requirements (performance, scale)
   - Establish Service Level Agreements (SLAs) upfront
2. **Build for Scale from Day One**
   - A pipeline that works on 1GB will often fail on 1TB
   - Test with production-scale data in staging
3. **Use Repeatable Patterns**
   - Configuration-driven pipelines (not hardcoded values)
   - Document best practices for your team
4. **Optimize Early**
   - Prefer DataFrame APIs over custom UDFs (much faster)
   - Use binary formats (Delta, Parquet) over CSV
   - Enable caching during ML training
5. **Control Costs**
   - Enable auto-termination (clusters shut down after inactivity)
   - Use autoscaling (scale up when needed, scale down when idle)
   - Use spot instances for development and training
   - Tag resources for chargeback analysis

3 Service Level Agreements (SLAs)

3.1 SLA Types by Workload

Workload Type	**Key SLA Metric**
Batch Processing	Volume of data + Total processing time (e.g., "Process 1TB in 2 hours")
Streaming	Transactions Per Second (TPS) (e.g., "Sustain 10,000 TPS")
Business Intelligence	Query latency (e.g., "Dashboard loads in $<$ 1 second")
Concurrency	Number of simultaneous users (e.g., "Support 1,000 concurrent analysts")

Table 2: *SLA Metrics by Workload Type*

3.2 Availability and Disaster Recovery

Definition:

Availability (Uptime)

$$\text{Availability} = \frac{\text{Total Uptime}}{\text{Total Uptime} + \text{Total Downtime}} \times 100\%$$

Common Targets:

- **Five 9s (99.999%):** Only 5 minutes downtime per year (extremely demanding)
- **Four 9s (99.99%):** About 52 minutes downtime per year
- **Three 9s (99.9%):** About 8.7 hours downtime per year

Definition:

RTO and RPO **RTO (Recovery Time Objective):** How quickly must the system be restored after failure?

- Example: "RTO = 15 minutes" means system must be back online within 15 minutes of an outage

RPO (Recovery Point Objective): How much data can you afford to lose?

- Example: "RPO = 1 hour" means backups every hour; you might lose up to 1 hour of data

Key Information

Multi-Region Deployment:
For high availability, deploy across multiple cloud regions. If one data center goes down, traffic fails over to another region. This is expensive but essential for mission-critical applications.

4 Infrastructure as Code (IaC)

4.1 Why Infrastructure as Code?

Example:

The Problem Without IaC **Scenario:** Your team needs 50 identical Databricks workspaces, each with specific network configurations, security groups, and cluster policies.
Manual Approach: Click through the AWS/Azure console 50 times. Hope you don't make mistakes. Document nothing. When someone asks "why is workspace 37 different?" you have no idea.
IaC Approach: Write code that describes the workspace. Run it 50 times. Every workspace is identical. Changes are version-controlled in Git.

4.2 IaC Principles

1. **Define Everything as Code:** Networks, storage, workspaces, clusters—all defined in configuration files

2. **Version Control:** Infrastructure code lives in Git, with full history
3. **Continuous Testing:** Validate infrastructure changes before applying
4. **Small, Independent Pieces:** Change one component without affecting others

4.3 Provisioning vs Configuration Management

Aspect	**Provisioning**	**Configuration Management**
Purpose	Create infrastructure from scratch	Modify existing infrastructure
Philosophy	Immutable (replace, don't modify)	Mutable (update in place)
Tools	Terraform, CloudFormation, Bicep	Ansible, Chef, Puppet
Example	"Create a VPC with these subnets"	"Update Python to 3.11 on all servers"
Agent Required?	No (uses APIs)	Yes (agents on each server)

Table 3: *Provisioning vs Configuration Management*

4.4 Terraform for Databricks

Terraform is the dominant provisioning tool, with strong Databricks support.

```
# Define the cloud provider
provider "azurerm" {
  features {}
}

# Create a resource group
resource "azurerm_resource_group" "rg" {
  name     = "databricks-rg"
  location = "East US"
}

# Create Databricks workspace
resource "azurerm_databricks_workspace" "workspace" {
  name                = "my-databricks"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = "premium"
}

# Output the workspace URL
output "workspace_url" {
  value = azurerm_databricks_workspace.workspace.workspace_url
}
```

Listing 1: Terraform Example: Databricks Workspace

```
# Initialize Terraform (download providers)
terraform init

# Preview what will be created (dry run)
terraform plan

# Apply changes (actually create resources)
terraform apply

# Destroy everything (clean up)
terraform destroy
```

Listing 2: Terraform Workflow

Key Information

Terraform State:
Terraform maintains a **state file** that tracks what resources exist. When you run `terraform apply`, it compares desired state (your code) vs. actual state (state file) and only changes what's different.

5 CI/CD: Continuous Integration and Continuous Delivery

5.1 What Is CI/CD?

Definition:

CI (Continuous Integration) Every time a developer commits code:

1. Code is automatically pulled from the repository
2. Build process runs (compile code, build packages)
3. Unit tests execute automatically
4. If tests fail, the developer is notified immediately

Definition:

CD (Continuous Delivery/Deployment) After CI passes:

1. Code is automatically deployed to staging
2. Integration tests run against staging
3. If approved (automatically or manually), deploy to production

Continuous Delivery: Requires manual approval before production deploy
Continuous Deployment: Fully automated to production (riskier but faster)

5.2 The CI/CD Pipeline Flow

`Commit` → `Build` → `Unit Test` → `Deploy to Staging` → `Integration Test` → `Deploy to Prod`

Warning

Always Have a Rollback Plan:
When deploying to production, have an automated way to roll back to the previous version if something goes wrong. This could be:

- Blue-green deployment (instant switch between versions)
- Canary deployment (gradually shift traffic to new version)
- Simple rollback script

5.3 CI/CD Tools

- **Source Control:** GitHub, GitLab, Bitbucket
- **CI/CD Servers:** Azure DevOps, Jenkins, GitHub Actions, GitLab CI
- **For Databricks:** Use Databricks REST APIs + Databricks Asset Bundles

6 Workspace Organization and Data Mesh

6.1 Workspace as Isolation Boundary

In Databricks, a **workspace** is the smallest isolation boundary:

- Own set of clusters, jobs, notebooks
- Own folder structure
- Own security perimeter

6.2 Workspace Organization Strategies

1. **Traditional (Simple):** Dev workspace, Staging workspace, Production workspace
2. **By Line of Business:** Marketing workspace, Sales workspace, Finance workspace
3. **By Project:** Each major project gets its own workspace
4. **Hybrid:** Core data in shared workspace, domain-specific in dedicated workspaces

Key Information

Enterprise Scale:
Mature organizations may have **hundreds to thousands** of workspaces. This is why IaC and standardized provisioning become essential—you can't manually configure 1,000 workspaces.

6.3 Data Mesh Architecture

Definition:

Data Mesh A decentralized approach to data architecture where:

1. **Domain Ownership:** Each business domain (Marketing, Sales) owns its data
2. **Data as Product:** Data is treated as a product with SLAs and documentation
3. **Self-Service Platform:** Domains can provision their own infrastructure
4. **Federated Governance:** Central policies, decentralized execution

Unity Catalog's Role: Even with decentralized domains, Unity Catalog provides the "glue" that enables data discovery, lineage tracking, and consistent governance across all domains.

7 Observability and Monitoring

7.1 System Tables for Platform Monitoring

Databricks provides **system tables** in a special `system` catalog that contain:

Schema	Contents
`system.billing`	Cost and usage data (SKUs, usage quantities, custom tags)
`system.compute`	Cluster information (who owns it, instance types, CPU usage)
`system.access`	Audit logs (who accessed what data, when)

Table 4: *Key System Tables*

```sql
-- Find top 10 most expensive jobs this month
SELECT
    custom_tags:project AS project,
    SUM(usage_quantity) AS total_dbus,
    SUM(usage_quantity * list_price) AS estimated_cost
FROM system.billing.usage
WHERE usage_date >= DATE_TRUNC('month', CURRENT_DATE())
GROUP BY custom_tags:project
ORDER BY estimated_cost DESC
LIMIT 10;
```

Listing 3: Querying System Tables for Cost Analysis

Important:

Tag Your Resources! Custom tags are essential for cost allocation. Tag all clusters and jobs with:

- Project name
- Team/department
- Environment (dev/staging/prod)
- Cost center

Without tags, you cannot answer "Which project is costing us the most?"

7.2 Data Quality Monitoring

Data teams often have SLAs on **delivery** (get data by 8 AM) but users expect **quality**. When quality issues arise, teams are reactive rather than proactive.

7.2.1 Anomaly Detection

One-click setup that uses AI to detect:

- **Freshness:** Data hasn't been updated when expected
- **Completeness:** Sudden drops in row counts

The system learns patterns automatically and alerts when deviations occur.

7.2.2 Lakehouse Monitoring

More detailed monitoring at the table level:

- **Profile Metrics:** Min, max, mean, nulls, distinct counts
- **Drift Detection:** Statistical tests (KS test, chi-square) to detect distribution changes
- **Automatic Dashboards:** Visual representation of data quality over time

Example:

Drift Detection Use Case **Scenario:** A column `preferred_payment_method` suddenly has 90% NULL values (normally 5%).
Without Monitoring: You discover this 2 weeks later when a downstream ML model starts making bad predictions.
With Lakehouse Monitoring: Alert fires within hours. Investigation reveals an upstream API change that stopped sending payment data. Fix deployed before ML model is affected.

7.3 Lineage for Root Cause Analysis

When something breaks, lineage helps you answer:

- **What upstream process caused this?** (trace back to source)
- **What downstream systems are affected?** (impact analysis)

```
-- This table is used by these downstream tables/dashboards
-- Visible in Unity Catalog UI under "Lineage" tab
-- Shows: source tables -> transformations -> target tables -> dashboards
```

Listing 4: Using Lineage for Impact Analysis

7.4 DQX: Code-Based Quality Rules

For more explicit quality rules, use DQX (Data Quality X):

```yaml
# Define quality rules in YAML
rules = """
checks:
  - column: id
    rule: not_null_and_not_empty
    criticality: error
  - column: age
    rule: value_in_range
    params: {min: 18, max: 120}
    criticality: warning
  - column: country
    rule: is_in_list
    params: {allowed: [Germany, France, USA]}
    criticality: warning
"""

# Apply rules to DataFrame
from dqx import DQEngine
engine = DQEngine()
valid_df, invalid_df, errors = engine.apply_checks(df, rules)

# valid_df contains rows passing all checks
# invalid_df contains rows that failed
# errors contains detailed failure information
```

Listing 5: DQX Quality Rules Example

8 Databricks Asset Bundles (DABs)

8.1 The Problem DABs Solve

Moving projects between environments traditionally required:

- Manual export/import of notebooks
- Recreating jobs with different configurations
- Updating cluster references
- Lots of API scripting

DABs package everything together in a single, portable bundle.

Definition:

Databricks Asset Bundle A **DAB** is a project-level packaging format that includes:

- Notebooks and Python files
- Job definitions

- Cluster configurations
- Variables that differ by environment
- Target environments (dev, staging, prod)

One command deploys everything: `databricks bundle deploy`

8.2 DAB Structure

```yaml
bundle:
  name: my_project

variables:
  catalog:
    default: dev_catalog
  cluster_id:
    lookup:
      cluster: my-cluster

resources:
  jobs:
    etl_pipeline:
      name: "ETL Pipeline"
      tasks:
        - task_key: bronze
          notebook_task:
            notebook_path: ./src/create_bronze.py
        - task_key: silver
          depends_on:
            - task_key: bronze
          notebook_task:
            notebook_path: ./src/create_silver.py

targets:
  development:
    mode: development
    default: true
    workspace:
      host: https://dev.cloud.databricks.com

  production:
    mode: production
    workspace:
      host: https://prod.cloud.databricks.com
    variables:
      catalog: prod_catalog
    resources:
      jobs:
        etl_pipeline:
          name: "Production ETL Pipeline"
```

Listing 6: Example databricks.yml

8.3 DAB Commands

```
# Initialize a new bundle from template
databricks bundle init

# Validate bundle configuration (catch errors before deploy)
databricks bundle validate

# Deploy to development environment
databricks bundle deploy -t development

# Run a job in development
databricks bundle run -t development etl_pipeline

# Deploy to production (uses production overrides)
databricks bundle deploy -t production
```

Listing 7: DAB CLI Commands

Key Information

DAB vs Terraform:

- **Terraform:** Creates the *infrastructure* (workspace, networks, storage)
- **DAB:** Deploys your *project* within that infrastructure (notebooks, jobs, pipelines)

Both are complementary. Use Terraform to create workspaces, then DAB to deploy code to them.

8.4 DAB in CI/CD Pipelines

DABs integrate seamlessly into CI/CD:

```
# .azure-pipelines.yml
trigger:
  - main

stages:
  - stage: Test
    jobs:
      - job: ValidateBundle
        steps:
          - script: pip install databricks-cli
          - script: databricks bundle validate

  - stage: DeployDev
    jobs:
      - job: DeployToDevEnvironment
        steps:
```

```
          - script: databricks bundle deploy -t development
          - script: databricks bundle run -t development etl_pipeline

  - stage: DeployProd
    condition: succeeded()
    jobs:
      - deployment: DeployToProduction
        environment: production
        strategy:
          runOnce:
            deploy:
              steps:
                - script: databricks bundle deploy -t production
```

Listing 8: Azure DevOps Pipeline with DABs

9 MLOps: Special Considerations for Machine Learning

9.1 Deploy Model vs Deploy Code

Approach	Deploy Model	Deploy Code
What Moves	Trained model artifact	Training code
Training Location	Development environment	Each environment
Data Used	Dev data	Environment-specific data
Best For	Simple models, same data everywhere	Models that need environment-specific training

Table 5: *MLOps Deployment Strategies*

Warning

Data Differences Matter:
A model trained on development data (which may be sampled or anonymized) might perform differently on production data. Consider retraining in production with full data.

10 Cost Optimization Strategies

1. **Use Spot Instances for Development**
   - 60-90% cheaper than on-demand
   - May be reclaimed by cloud provider
   - Never use for production-critical workloads
2. **Enable Auto-Termination**
   - Clusters shut down after N minutes of inactivity (e.g., 30 minutes)
   - Prevents "weekend clusters" that run unused for 48 hours
3. **Use Autoscaling**

- Scale up when workload increases
- Scale down (even to 0) when idle

4. **Move to Serverless**
   - Pay only for what you use
   - No cluster management overhead
   - Faster startup (seconds vs minutes)
5. **Write Efficient Code**
   - Use DataFrame APIs over UDFs
   - Use binary formats (Delta/Parquet) over CSV
   - Leverage caching appropriately
6. **Monitor and Alert**
   - Set up alerts for unusual spending
   - Review system tables weekly
   - Tag resources for attribution

11 Quick Summary: One-Page Review

Key Summary

Key Takeaways from Lecture 13:

1. **SDLC Environments:**
   - Development → Staging → Production
   - Use service principals (not personal accounts) in production
   - Production should be fully automated
2. **Infrastructure as Code (IaC):**
   - Use Terraform for reproducible infrastructure
   - Version control all infrastructure code
   - Immutable infrastructure: replace, don't modify
3. **CI/CD:**
   - Automate testing on every commit
   - Deploy through pipelines, not manually
   - Always have a rollback plan
4. **Observability:**
   - System tables for cost and audit monitoring
   - Anomaly detection for freshness/completeness
   - Lakehouse monitoring for detailed quality metrics
   - Lineage for root cause and impact analysis
5. **Databricks Asset Bundles:**
   - Package entire projects (notebooks, jobs, configs)

- Environment-specific overrides via targets
- `databricks bundle deploy` for one-command deployment

6. **Cost Optimization:**
   - Spot instances for dev, auto-termination, autoscaling
   - Tag everything for chargeback
   - Consider serverless for variable workloads

12 Glossary

Term	Definition
Service Principal	A "robot account" for automated processes
Spot Instance	Discounted compute that can be reclaimed by the cloud provider
RTO	Recovery Time Objective: how quickly to restore after failure
RPO	Recovery Point Objective: maximum acceptable data loss
IaC	Infrastructure as Code: defining infrastructure in version-controlled files
CI/CD	Continuous Integration / Continuous Delivery: automated testing and deployment
DAB	Databricks Asset Bundle: project packaging for cross-environment deployment
Data Mesh	Decentralized data architecture with domain ownership
Drift	Changes in data distributions over time
Lineage	Tracking data's journey from source to destination

Table 6: *Key Terms*