🚀 Model Training Pipeline - Complete Guide for Customers

Executive Summary

This document explains the **Model Training Pipeline** in simple, beginner-friendly terms. This GitHub Actions workflow automates the deployment of machine learning models from development through testing to production, ensuring code quality, security, and compliance at every step.

📋 Table of Contents

What is this Pipeline?

This is a **GitHub Actions workflow** - think of it as an automated robot that runs tasks whenever you make code changes. It ensures your machine learning model gets safely deployed from development → testing → production with proper quality checks and approvals.

File Location: `.github/workflows/model-training-pipeline.yml`

Purpose: Automate the entire deployment lifecycle for ML notebooks and data pipelines

---

When Does the Pipeline Run?

The pipeline can start in **three different ways**:

1. Pull Request Trigger (Automatic Testing)

```
pull_request:
  branches: [main]
  paths:
    - 'notebooks/model_training.Notebook/**'
    - 'pipelines/customer_analytics_pipeline.json'
```

When: Someone creates a Pull Request to merge their code
What it checks: Only runs if you changed files in `notebooks/` or `pipelines/`
Purpose: Test the code BEFORE it gets merged
Analogy: Quality inspection before accepting a delivery

2. Push Trigger (Automatic Deployment)

```
push:
  branches: [main]
  paths:
    - 'notebooks/model_training.Notebook/**'
```

When: Code gets merged to the `main` branch
What happens: Automatically deploys to DEV environment
Purpose: Keep DEV environment always up-to-date
Analogy: Auto-publish to staging area after approval

3. Manual Trigger (Controlled Deployments)

```
workflow_dispatch:
  inputs:
    environment: [dev, test, prod]
    deployment_reason: string
    change_ticket: string
```

When: You manually click "Run workflow" button in GitHub
What you choose: Which environment, reason for deployment, change management ticket
Purpose: Controlled deployments to TEST and PRODUCTION
Analogy: Pushing a button to start a process

Job 1: PR Validation (Quality Gates)

Purpose: Check if the code is good quality BEFORE merging

Runs when: Pull request is created

Step-by-Step Breakdown

Step 1: Get the Code

```
- name: 📥 Checkout code
  uses: actions/checkout@v4
```

What it does: Downloads your code from GitHub so the robot can examine it
Analogy: Taking a book off the shelf to read it

Step 2: Install Python

```
- name: 🐍 Set up Python
  uses: actions/setup-python@v5
  with:
    python-version: '3.10'
```

What it does: Installs Python 3.10
Analogy: Installing an app on your computer before you can use it

Step 3: Install Dependencies

```
- name: 📦 Install dependencies
  run: |
    pip install -r requirements.txt
    pip install black flake8 nbqa pytest
```

What it does: Installs all required Python libraries (pandas, numpy, scikit-learn, etc.)
Analogy: Installing plugins or extensions before using software
Why: Your code needs these libraries to run

Step 4: Check Code Formatting

```
- name: 🎨 Validate Python formatting
  run: black --check scripts/
```

What it does: Ensures code follows consistent style rules
Analogy: Grammar and spell-check for code
Why: Consistent code is easier to read and maintain

Step 5: Validate Notebook Structure

```
- name: 📓 Validate model_training notebook
  run: python scripts/validate_notebooks.py
```

What it does: Checks if your Jupyter notebook is properly formatted and not corrupted
Analogy: Making sure a document can be opened and read
Why: Prevents deploying broken notebooks

Step 6: Security Scan

```
- name: 🔐 Check for hardcoded secrets
  run: |
    if grep -r 'password|api_key|secret' ...; then
      exit 1
    fi
```

What it does: Searches for passwords, API keys, or secrets accidentally left in code
Analogy: Checking you didn't write your password on a sticky note attached to your laptop
Why: Prevents security breaches from exposed credentials

Step 7: Run Unit Tests

```
- name: 🧪 Run unit tests
  run: pytest scripts/tests/ -v
```

What it does: Runs automated tests to verify code works correctly
Analogy: Testing if a light switch turns the light on/off
Why: Catches bugs before they reach production

Step 8: Schema Change Detection

```
- name: 📊 Schema Change Detection
  run: python scripts/detect_schema_changes.py
```

What it does: Checks if you changed data columns in a way that would break Power BI reports
Example: If you delete a column "customer_name", any report using that column will break
What happens if breaking changes found:

- Pipeline posts a comment on your PR warning you
- Lists which columns changed
- Provides instructions to fix or update reports

Why: Prevents breaking production dashboards

Step 9: Create Summary

```
- name: 📊 PR Validation Summary
  run: echo "## ☑️ Model Training Notebook - PR Validation" >>
$GITHUB_STEP_SUMMARY
```

What it does: Creates a nice report card showing ☑ or ❌ for each check
Analogy: Getting a grade report after an exam
Why: Quick visual feedback on code quality

Job 2: Deploy to DEV

Purpose: Automatically deploy code to DEV environment after PR is merged

Runs when: Code is merged to `main` branch OR you manually select "dev"

Key Steps

Step 1-3: Setup Environment

- Get code from repository
- Install Python 3.10
- Install deployment tools

Purpose: Prepare the robot's toolbox

Step 4: Login to Azure

```
- name: 🔐 Authenticate to Azure
  uses: azure/login@v2
  with:
    creds: ${{ secrets.AZURE_CREDENTIALS }}
```

What it does: Logs into Azure using credentials stored securely in GitHub secrets
Analogy: Entering username/password to access a system
Why: Need permission to deploy to Fabric workspaces

Step 5: Get Fabric Access Token

```
- name: 🎫 Get Fabric Access Token
  run: |
    TOKEN=$(az account get-access-token --resource
https://analysis.windows.net/powerbi/api)
    echo "FABRIC_TOKEN=$TOKEN" >> $GITHUB_ENV
```

What it does: Gets a temporary security token to talk to Microsoft Fabric
Analogy: Getting a backstage pass at a concert
Technical note: Token masked with `::add-mask::` to prevent logging
Why: Fabric API requires authentication for all operations

Step 6: Deploy Notebook to DEV

```
- name: 🚀 Deploy model_training notebook to DEV
  run: |
    DEPLOYMENT_ID="dev-$(date +%Y%m%d-%H%M%S)"
    python scripts/deploy_to_fabric.py \
      --workspace-id "${{ secrets.FABRIC_DEV_WORKSPACE_ID }}" \
      --environment dev \
      --artifact-type notebooks \
      --artifacts-path notebooks/model_training.Notebook
```

What it does:

- Creates unique deployment ID (e.g., "dev-20260108-143045")
- Uploads `model_training` notebook to DEV Fabric workspace
- Uses Fabric REST API to create/update notebook

Analogy: Copying files to a server
Why: Makes your latest code available in DEV environment

Step 7: Deploy Pipeline

```
- name: 🔄 Deploy customer analytics pipeline
  run: |
    python scripts/deploy_to_fabric.py \
      --artifact-type pipelines \
      --artifacts-path pipelines/customer_analytics_pipeline.json
```

What it does: Uploads the data pipeline configuration to DEV workspace
Purpose: Data pipelines orchestrate when/how notebooks run
Why: Notebook and pipeline need to stay in sync

Step 8: Trigger Fabric Deployment Pipeline

```
- name: 🎯 Trigger Fabric Deployment Pipeline (DEV stage)
  continue-on-error: true
  run: |
    pwsh scripts/trigger-fabric-deployment-pipeline.ps1 \
      -PipelineName "pipeline1" \
      -TargetStage "Test"
```

What it does: Tells Fabric's UI-based deployment pipeline to promote changes from Dev → Test
Analogy: Pressing a "Copy to next environment" button automatically
Note: `continue-on-error: true` means if this fails, deployment continues (it's optional)
Why: Optional integration with Fabric's built-in deployment pipelines

Step 9: Run Smoke Tests

```
- name: ☑ Run smoke tests
  run: python scripts/run_smoke_tests.py
```

What it does: Quick basic tests to ensure deployment didn't break everything
Analogy: Turning on a light switch to see if electricity works
Examples:

- Can we connect to the workspace?
- Does the notebook exist?
- Can we read data from lakehouse?

Why: Immediate feedback if deployment failed

Step 10: Deployment Summary

```
- name: 📊 DEV Deployment Summary
  run: echo "## 🚀 DEV Deployment Complete" >> $GITHUB_STEP_SUMMARY
```

What it does: Creates a report showing:

- Deployment ID
- Timestamp
- What was deployed
- Test results
- Next steps (how to promote to TEST)

Job 3: Deploy to TEST

Purpose: Promote code to TEST environment with additional validation

Runs when: You manually click "Run workflow" and select "test"

Requires: Manual approval (if GitHub environments configured)

Key Differences from DEV

Promotion via Fabric Deployment Pipeline

```
- name: 🚀 Promote DEV → TEST via Fabric Deployment Pipeline
  run: |
    python scripts/trigger_fabric_deployment_pipeline.py \
      --source-stage 0 \
      --target-stage 1 \
      --note "${{ github.event.inputs.deployment_reason }}"
```

What it does: Uses Fabric's deployment pipeline to copy artifacts from DEV (stage 0) to TEST (stage 1)
Records: Why you're deploying (e.g., "Testing bug fix for issue #3")
Why: Traceable, controlled promotion between environments

Data Quality Validation

```
- name: 🔍 Data quality validation
  run: python check_data_quality.py --environment test
```

What it does: Checks if the data looks good:

- No missing values in critical columns
- Correct data types
- Values within expected ranges
- No duplicates

Analogy: Food quality inspection
Why: Bad data = bad ML models

Integration Tests

```
- name: 🧪 Run integration tests
  run: python scripts/run_integration_tests.py
```

What it does: Tests if everything works together (not just individual pieces)
Examples:

- Can notebook read from lakehouse?
- Can pipeline trigger notebook?
- Does model training complete successfully?
- Are outputs in correct format?

Analogy: Testing if all parts of a car work together, not just the engine
Why: Individual pieces might work, but fail when combined

Deployment Validation

```
- name: ☑️ Validate deployment
  run: python scripts/validate_deployment.py
```

What it does: Double-checks everything deployed correctly
Checks:

- Notebook exists in workspace

- Notebook version matches source
- Pipeline configuration correct
- Dependencies available

Why: Catch deployment issues early

Job 4: Deploy to PRODUCTION

Purpose: Deploy to PRODUCTION with maximum safety controls

Runs when: You manually click "Run workflow" and select "prod"

Requires:

- Change management ticket
- Manager approval (if GitHub environments configured)
- Successful TEST deployment

Production-Specific Safety Measures

Step 1: Validate Change Ticket

```
- name: 🎫 Validate change ticket
  run: |
    if [ -z "${{ github.event.inputs.change_ticket }}" ]; then
      echo "❌ Change ticket is required for PROD deployment"
      exit 1
    fi
```

What it does: Ensures you provided a change ticket number (e.g., CHG-12345)
What happens if missing: Deployment stops immediately
Why:

- Audit trail required for compliance
- Change management board approval
- Incident tracking if something goes wrong

Analogy: Need approval from boss before making major changes

Step 2: Backup PRODUCTION

```
- name: 📸 Backup current PROD state
  run: |
    BACKUP_ID="backup-$(date +%Y%m%d-%H%M%S)"
    python scripts/backup_workspace.py \
      --workspace-id "${{ secrets.FABRIC_PROD_WORKSPACE_ID }}" \
      --backup-id "$BACKUP_ID"
```

What it does:

- Takes snapshot of PRODUCTION workspace before changing anything
- Creates backup ID like "backup-20260108-150000"
- Saves all notebook versions, pipeline configs, data schemas

Why: If something goes wrong, you can restore the old version
Analogy: Creating a restore point before installing software

Step 3: Deploy to PRODUCTION Workspace

```
- name: 🚀 Deploy to PRODUCTION workspace
  run: |
    python scripts/deploy_to_fabric.py \
      --workspace-id "${{ secrets.FABRIC_PROD_WORKSPACE_ID }}" \
      --environment prod \
      --artifact-type notebooks
```

What it does:

- Uploads notebooks to PRODUCTION workspace
- Uploads pipelines to PRODUCTION workspace
- Records change ticket, reason, timestamp, and deployer

Note: Direct deployment (not via Fabric deployment pipeline) for better control
Why: Production stability is critical - we control exact artifacts deployed

Step 4: Validate PROD Deployment

```
- name: ☑ Validate PROD deployment
  run: python scripts/validate_deployment.py --environment prod
```

What it does: Verifies deployment succeeded
Checks:

- Notebooks deployed correctly
- Pipeline configuration matches expected
- No errors in deployment logs

Step 5: Run Smoke Tests in PROD

```
- name: 🩹 Run smoke tests in PROD
  run: python scripts/run_smoke_tests.py --environment prod
```

What it does: Quick tests in PRODUCTION to ensure basic functionality
Examples:

- Can connect to workspace
- Can read from production data
- Basic calculations work

Why: Immediate feedback if PROD is broken

Step 6: Log Deployment to Audit Trail

```
- name: 📝 Log deployment to audit trail
  run: |
    cat >> DEPLOYMENT_LOG.md << EOF
    ## 🎯 Production Deployment - $(date)
    - **Change Ticket**: ${{ github.event.inputs.change_ticket }}
    - **Deployed By**: @${{ github.actor }}
    - **Commit**: ${{ github.sha }}
    EOF
```

What it does: Writes deployment details to `DEPLOYMENT_LOG.md` file
Records:

- Who deployed
- When
- Why (change ticket)
- What was deployed (commit SHA)
- Deployment ID
- Backup ID

Why:

- Compliance requirements
- Troubleshooting history
- Audit trail for security reviews

Analogy: Captain's log on a ship

Step 7: Notify Stakeholders

```
- name: 📧 Notify stakeholders
  run: echo "📧 Sending deployment notifications..."
```

What it does: Sends email/Slack/Teams message to team
Message includes:

- Deployment completed successfully

- Change ticket number
- What was deployed
- Rollback instructions (if needed)

Why: Keep everyone informed of production changes

Job 5: Rollback (Emergency)

Purpose: Automatically undo PROD deployment if it fails

Runs when: PROD deployment job fails

How Rollback Works

```
rollback:
  if: failure() && github.event.inputs.environment == 'prod'
  needs: [deploy-prod]
  steps:
    - name: 🔙 Execute rollback
      run: python scripts/rollback_deployment.py
```

What it does:

1. Detects that PROD deployment failed
2. Automatically restores PRODUCTION to the backup taken earlier
3. Notifies team of rollback

Analogy: Pressing Ctrl+Z to undo
Why: Minimize downtime - get PROD working again immediately

Rollback Process:

1. Retrieves backup ID from previous step
2. Restores notebooks from backup
3. Restores pipeline configurations
4. Validates restoration succeeded
5. Logs rollback event

Key Concepts

1. Environments as Checkpoints

```
DEV (Playground) → TEST (Rehearsal) → PROD (Live Show)
```

Environment	Purpose	Testing Level	Who Uses

Environment	Purpose	Testing Level	Who Uses
DEV	Rapid iteration	Smoke tests	Developers
TEST	Quality validation	Integration tests	QA Team
PROD	Customer-facing	Smoke tests + monitoring	End Users

DEV: Where developers experiment safely - breaking things is OK
TEST: Where QA team validates everything works - must pass all tests
PROD: What customers see - must be perfect and stable

2. Secrets Management

```
${{ secrets.FABRIC_DEV_WORKSPACE_ID }}
${{ secrets.AZURE_CREDENTIALS }}
${{ secrets.AZURE_CLIENT_SECRET }}
```

What are secrets?
Sensitive information like:

- Passwords
- API keys
- Workspace IDs
- Access tokens

Where are they stored?
Securely in GitHub repository settings, NOT in code

Why?

- Prevents accidental exposure in version control
- Different values per environment (DEV vs PROD workspace IDs)
- Centralized credential rotation

How to access?
Only in workflow files using `${{ secrets.SECRET_NAME }}` syntax

3. Quality Gates

Each stage has progressively stricter checks:

PR Stage (Before Merge)

- ☑ Code formatting
- ☑ Security scan (no hardcoded secrets)
- ☑ Unit tests
- ☑ Schema validation
- ☑ Notebook structure

DEV Stage (After Merge)

- ☑ Deployment validation
- ☑ Smoke tests
- ☑ Basic connectivity

TEST Stage (Manual Promotion)

- ☑ Data quality validation
- ☑ Integration tests (end-to-end)
- ☑ Performance checks
- ☑ Schema compatibility

PROD Stage (Strict Controls)

- ☑ Change ticket required
- ☑ Backup before deployment
- ☑ Deployment validation
- ☑ Smoke tests
- ☑ Audit logging
- ☑ Automatic rollback on failure

4. Automation vs Manual Control

Action	Trigger	Approval Needed	Environment
PR Validation	Automatic (on PR)	No	N/A
Deploy to DEV	Automatic (on merge)	No	DEV
Deploy to TEST	Manual button press	Recommended	TEST
Deploy to PROD	Manual button press	Required	PROD

Philosophy:

- Automate what's safe (DEV)
- Require approval for what's risky (PROD)
- Always validate before promotion

5. Deployment Artifacts

What gets deployed?

1. **Notebooks** (`model_training.Notebook/`)
    - Python code for ML model training
    - Feature engineering logic
    - Model evaluation metrics
2. **Pipelines** (`pipelines/customer_analytics_pipeline.json`)

    - Orchestration configuration
    - Scheduling settings
    - Dependencies between steps

3. **Configuration** (via parameters)

    - Environment-specific settings
    - Connection strings
    - Feature flags

How?
Via Fabric REST API using Python scripts

6. Fabric Deployment Pipeline Integration

Two deployment methods:

Method 1: GitHub Actions Direct Deployment

- Uses Python scripts with Fabric REST API
- Deploys artifacts directly to workspace
- Full control over what gets deployed
- Used for: DEV and PROD

Method 2: Fabric UI Deployment Pipeline

- Uses Fabric's built-in deployment pipeline (created in UI)
- Promotes entire workspace: Dev (stage 0) → Test (stage 1) → Prod (stage 2)
- Handles dependencies automatically
- Used for: DEV → TEST promotion

Best practice: Combine both for maximum flexibility

Demo Talking Points

1. "This Pipeline Prevents Bad Code from Reaching Customers"

How?

- PR validation catches bugs BEFORE merge
- Progressive testing (dev → test → prod)
- Breaking changes detected automatically
- Schema validation prevents dashboard breaks

Customer benefit: Fewer production incidents, higher quality

2. "Complete Audit Trail for Compliance"

What's tracked?

- Every deployment logged with:
  - Who deployed
  - When (timestamp)
  - Why (change ticket + reason)
  - What (commit SHA, deployment ID)
  - Result (success/failure)

Customer benefit:

- Pass audits easily
- Troubleshoot issues faster
- Meet regulatory requirements (SOX, GDPR, HIPAA)

3. "Safety Mechanisms Protect Production"

Safety layers:

1. **Backups** before PROD changes
2. **Automatic rollback** on failure
3. **Manual approvals** for critical environments
4. **Change tickets** required for PROD
5. **Schema validation** prevents breaking reports

Analogy: Multiple airbags and seatbelts in a car

Customer benefit: Sleep better at night knowing PROD is protected

4. "Developer Productivity Increases"

Time savings:

- **Before**: Manual deployment = 2 hours, error-prone
- **After**: Automatic deployment = 5 minutes, consistent

How?

- Automatic DEV deployments (no manual steps)
- Clear instructions for next steps
- Standardized process (no guessing)
- Parallel testing (faster feedback)

Customer benefit:

- Developers focus on building features, not deploying
- Faster time-to-market
- Consistent quality

5. "Built for Enterprise Scale"

Enterprise features:

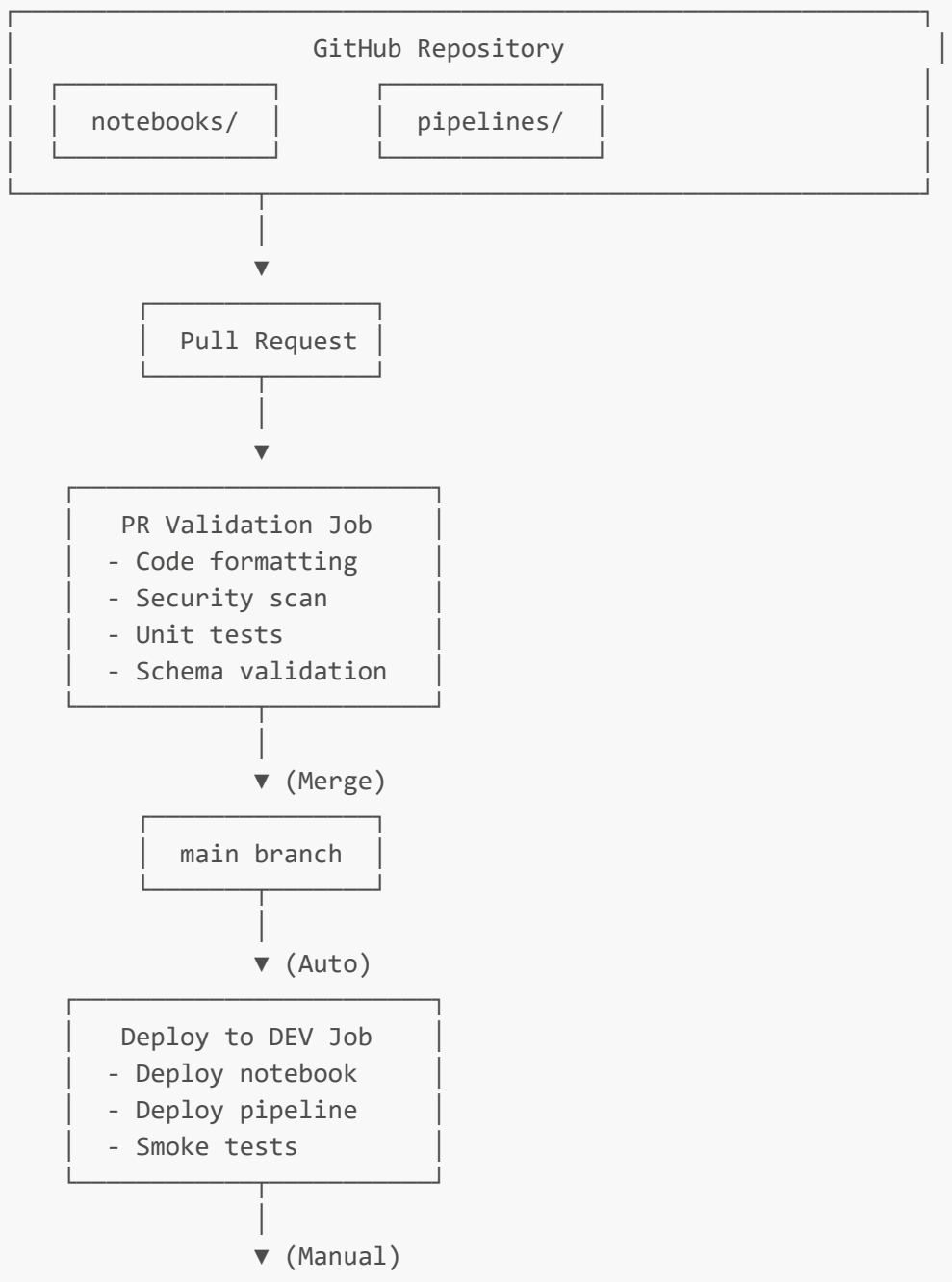
- Multi-environment strategy (dev/test/prod)

- Integration with change management (tickets)
- Approval workflows (GitHub environments)
- Notification system (stakeholder alerts)
- Rollback capabilities (disaster recovery)
- Audit logging (compliance)

Scales to:

- Multiple teams
- Multiple projects
- Multiple regions
- Regulatory requirements

Architecture Diagram



```
        ┌──────────────────────────┐
        │   Deploy to TEST Job     │
        │  - Integration tests     │
        │  - Data quality check    │
        │  - Validate deploy       │
        └──────────┬───────────────┘
                   │
                   ▼ (Manual + Approval)
        ┌──────────────────────────┐
        │   Deploy to PROD Job     │
        │  - Validate ticket       │
        │  - Backup workspace      │
        │  - Deploy artifacts      │
        │  - Smoke tests           │
        │  - Audit logging         │
        └──────────┬───────────────┘
                   │
                   ▼ (On Failure)
             ┌─────────────────┐
             │  Rollback Job   │
             │  - Restore      │
             │  - Notify       │
             └─────────────────┘
```

Troubleshooting Guide

Common Issues

Issue 1: "Secrets not found"

Symptom: Pipeline fails with "FABRIC_DEV_WORKSPACE_ID not found"
Solution: Add secrets in GitHub repository settings
How: Settings → Secrets and variables → Actions → New repository secret

Issue 2: "Schema validation fails"

Symptom: PR validation fails with breaking schema changes
Solution: Either revert changes or update Power BI reports
Prevention: Test schema changes in DEV first

Issue 3: "Deployment to Fabric fails"

Symptom: "Authentication failed" or "Workspace not found"
Solution:

1. Check Azure credentials are current
2. Verify service principal has permissions
3. Confirm workspace ID is correct

Issue 4: "Tests timeout"

Symptom: Integration tests run forever
Solution: Check if Fabric workspace is responsive
Debug: Add `continue-on-error: true` temporarily

Best Practices

1. Always Create PR First

Never push directly to `main` - always create PR for code review

2. Write Meaningful Deployment Reasons

Bad: "update"
Good: "Fix division by zero bug in feature engineering (Issue #3)"

3. Test in DEV and TEST Before PROD

Never skip environments - always validate in TEST before PROD

4. Include Change Tickets for PROD

Required for audit trail and compliance

5. Monitor After Deployment

Check logs, metrics, and dashboards after PROD deployment

6. Keep Backup IDs

Save backup IDs from PROD deployments for quick rollback

Summary

This pipeline provides **enterprise-grade CI/CD for machine learning** with:

- ☑ **Quality**: Automated testing and validation at every stage
- ☑ **Security**: Secret scanning, secure credential management
- ☑ **Compliance**: Complete audit trail, change management integration
- ☑ **Safety**: Backups, rollbacks, approval workflows
- ☑ **Productivity**: Automated deployments, clear next steps
- ☑ **Reliability**: Progressive testing, smoke tests, validation

Result: Safer, faster, more reliable ML deployments

Additional Resources

- GitHub Actions Documentation: https://docs.github.com/actions

- Microsoft Fabric Documentation: https://learn.microsoft.com/fabric
- Repository: https://github.com/sautalwar/nvrcicddemo1
- Pull Request #14: Bug fix example with full pipeline execution

---

Document Version: 1.0
Last Updated: January 8, 2026
Author: Development Team
For: Customer Demo / Training