## ✅ Case Study 1: Managing Code, Data, and Model Versions Using Git & DVC

**Scenario:**

A team is building a machine learning model to classify images. The project involves:

- Python code (train.py)
- A large image dataset (data/)
- Trained model files (model.pkl)
- Multiple experiments with different hyperparameters

---

**Problem:**

- Git can track code but cannot handle large files like datasets or models.
- The team struggles to track experiments and share data.
- Reproducing old results becomes hard.

---

**Solution: Use Git + DVC (Data Version Control)**

| Tool | Purpose |
|---|---|
| Git | Tracks source code |
| DVC | Tracks data, models, and experiments |

---

**Steps to Implement:**

**Step 1: Initialize Git and DVC**

```
git init
dvc init
git commit -m "Initialize Git and DVC"
```

**Step 2: Add Dataset with DVC**

```
dvc add data/images/
git add data/images.dvc .gitignore
git commit -m "Track dataset with DVC"
```

**Step 3: Set Up Remote Storage (Google Drive, S3, etc.)**

```
dvc remote add -d myremote gdrive://<your-folder-id>
dvc push  # Push data to remote storage
```

**Step 4: Track Model Training Output**

```
dvc run -n train_model \
  -d train.py -d data/images \
  -o model.pkl -M metrics.json \
  python train.py
```

**Step 5: Run and Compare Experiments**

```
dvc exp run
dvc exp show
dvc exp apply <experiment_id>
```

**Benefits:**

- All team members get the same code + data
- Experiments are recorded and easy to compare
- Models and metrics are versioned
- No need to email large files

---

**Conclusion:**

Git + DVC is a simple way to manage everything in an ML project: code, data, models, and experiments. It improves collaboration, reproducibility, and version tracking

## Case Study 2: CI/CD for ML Projects Using Jenkins and GitHub

**Scenario:**

An ML team wants to automate:

- Running tests
- Training models
- Checking model accuracy
- Deploying the best models

They are using:

- **GitHub** for code hosting
- **Jenkins** for automation
- **Docker** for deployment

---

**Problem:**

- Developers must manually test, train, and deploy models.
- There's no check on model performance before deployment.
- Team wants faster feedback and safer deployments.

---

**Solution: Use Jenkins for CI/CD**

| Tool | Purpose |
|---|---|
| **GitHub** | Code hosting and trigger point |
| **Jenkins** | Automates the ML workflow |
| **DVC** | Pulls data/models |
| **Pytest** | Runs unit tests |
| **Docker** | Builds and runs the ML model app |

---

## Steps in the Jenkins Pipeline:

### ✅ Step 1: Code is pushed to GitHub

➞ Jenkins is notified via webhook

### ✅ Step 2: Jenkinsfile runs with these stages:

```
pipeline {
    agent any
    stages {
        stage('Install Requirements') {
            steps {
                sh 'pip install -r requirements.txt'
            }
        }
        stage('Run Tests') {
            steps {
                sh 'pytest tests/'  // Check code quality
            }
        }
        stage('Get Data & Model') {
            steps {
                sh 'dvc pull'  // Pull data/model files from remote
            }
        }
        stage('Train Model') {
            steps {
                sh 'python train.py'
            }
        }
        stage('Deploy Model') {
            steps {
                sh 'docker build -t ml-app .'
                sh 'docker run -d -p 5000:5000 ml-app'
            }
        }
    }
}
```

---

**What's Needed for This to Work:**

- train.py (your ML training script)
- metrics.json (stores model accuracy like {"accuracy": 0.91})
- requirements.txt (Python libraries)
- tests/ folder (unit tests using Pytest)
- Dockerfile (for deploying the model as an API)

---

**Benefits:**

- **Automatic training and testing** after each code change
- **Stops deployment if model is not good enough**
- **Saves time and reduces human errors**
- **Consistent deployments** with Docker

---

**Conclusion:**

Jenkins CI/CD pipeline makes ML workflows faster, safer, and repeatable. It ensures models are tested and meet accuracy goals before going live.