🔀 Day 8: Git and Version Control - Professional Code Collaboration

📚 Complete Learning Guide

🎯 Learning Objectives

By the end of Day 8, you will:

- Understand why version control is critical for data engineering teams
- Master Git fundamentals and professional workflows
- Set up collaborative development with GitHub
- Implement branching strategies for data projects
- Conduct code reviews for data pipelines
- Handle merge conflicts and rollback strategies

📊 Project for Day 8

Primary Project: Data Engineering Portfolio Setup

- **Repository Name**: `data-engineering-50-days`
- **Structure**: Organize all previous day's work with proper version control
- **Collaboration**: Set up for team development patterns
- **Use Case**: Professional portfolio that showcases Git skills

Files to Version Control:

- Python scripts from Day 2
- SQL queries from Days 3-4
- Cloud configurations from Day 6
- Linux scripts from Day 7
- Documentation and learning notes

Required Tools:

- **Git**: Command-line version control tool
- **GitHub Account**: Free at `github.com`
- **Text Editor**: VS Code, Sublime, or any editor with Git integration

- **Terminal**: For Git commands (from Day 7)

🎓 Conceptual Understanding First (60 minutes)

Why Version Control is Essential for Data Engineers

The Problem Without Version Control:

```
Monday: data_pipeline.py works perfectly
Tuesday: Modified for new requirement
Wednesday: Something breaks in production
Thursday: "What changed? How do we go back?"
Friday: Panic mode - manual comparison of files
```

The Solution with Git:

```
Monday: Commit working pipeline with message "Add customer segmentation"
Tuesday: Create feature branch "add-product-analytics"
Wednesday: Test fails, easy rollback to Monday's version
Thursday: Review changes, identify exact problem
Friday: Confident deployment with full change history
```

Real-World Data Engineering Scenarios:

1. **ETL Pipeline Evolution**: Track changes to data transformation logic
2. **SQL Query Optimization**: Compare performance before/after modifications
3. **Configuration Management**: Version control for database connections, API keys
4. **Team Collaboration**: Multiple data engineers working on same pipeline
5. **Production Rollbacks**: Quickly revert to working version when deployments fail

Git vs. Traditional File Management

Traditional Approach:

```
data_pipeline_v1.py
data_pipeline_v2.py
data_pipeline_v2_fixed.py
data_pipeline_v2_final.py
data_pipeline_v2_final_ACTUALLY_FINAL.py
```

Git Approach:

```
git log --oneline
abc1234 Add customer lifetime value calculation
def5678 Fix memory leak in data processing
ghi9012 Add error handling for API timeouts
jkl3456 Initial data pipeline implementation
```

Enterprise Git Workflows

How Major Companies Use Git:

- **Netflix**: 1000+ microservices, each with own Git repository
- **Spotify**: Feature branches for each new analytics capability
- **Uber**: Git hooks for automated testing of data pipeline changes
- **Airbnb**: Code reviews required for all ETL modifications

Common Data Engineering Git Patterns:

- **Feature Branches**: New data sources, transformations
- **Environment Branches**: dev, staging, production configurations
- **Hotfix Branches**: Quick fixes for production data issues
- **Release Branches**: Coordinated deployment of multiple features

---

🛠️ Git Installation and Setup (30 minutes)

Installing Git

Check if Git is Already Installed:

```bash
git --version
# If installed, you'll see version number like: git version 2.39.0
```

Installation by Operating System:

Linux (Amazon Linux/CentOS):

```bash
sudo yum install git -y
```

Linux (Ubuntu/Debian):

```bash
sudo apt-get update
sudo apt-get install git -y
```

macOS:

```bash
# Using Homebrew (recommended)
brew install git

# Or download from: https://git-scm.com/download/mac
```

Windows:

- Download from: `https://git-scm.com/download/windows`
- Use Git Bash terminal for commands
- Or use WSL2 with Linux installation

Initial Git Configuration

Global Configuration (One-time setup):

bash

```bash
# Set your identity (appears in all commits)
git config --global user.name "Your Full Name"
git config --global user.email "your.email@example.com"

# Set default branch name to 'main' (modern standard)
git config --global init.defaultBranch main

# Set default editor (optional)
git config --global core.editor "nano"  # or "vim" or "code"

# Improve Git output formatting
git config --global color.ui auto
git config --global core.autocrlf input  # For Unix systems
# Use 'true' instead of 'input' on Windows

# Verify configuration
git config --list
```

Understanding Git Configuration:

- `--global`: Settings apply to all repositories on your system
- `--local`: Settings apply only to current repository
- Configuration stored in `~/.gitconfig` (global) and `.git/config` (local)

📁 Creating Your First Repository (45 minutes)

Setting Up the Data Engineering Portfolio

Step 1: Create Project Directory:

```bash
# Create main project directory
mkdir data-engineering-50-days
cd data-engineering-50-days

# Initialize Git repository
git init
# Output: Initialized empty Git repository in /path/to/data-engineering-50-days/.git/

# Verify repository creation
ls -la
# You should see .git directory (this contains all Git data)
```

Step 2: Create Professional Project Structure:

bash

```
# Create organized directory structure
mkdir -p {day-01-introduction,day-02-python,day-03-sql,day-04-advanced-sql,day-05-data
mkdir -p {docs,scripts,configs,data,tests}

# Create main README file
cat > README.md << 'EOF'
# Data Engineering 50-Day Challenge

A comprehensive journey to master data engineering skills in 50 days.

## Project Structure

- `day-XX-topic/` - Daily learning materials and projects
- `docs/` - Documentation and learning notes
- `scripts/` - Reusable utility scripts
- `configs/` - Configuration files and templates
- `data/` - Sample datasets (gitignored for large files)
- `tests/` - Test scripts and validation

## Skills Covered

- [x] Python Fundamentals (Day 2)
- [x] SQL and Advanced SQL (Days 3-4)
- [x] Cloud Platforms - AWS (Day 6)
- [x] Linux Command Line (Day 7)
- [x] Git and Version Control (Day 8)
- [ ] Docker and Containerization (Day 9)
- [ ] And 41 more days of advanced topics...

## Learning Philosophy

**Understanding > Coding** - Focus on concepts before implementation.

## Resources

Each day includes:
- Conceptual explanations
- Hands-on projects with real datasets
- Production-ready examples
- Industry best practices

---
```

```
*Follow along at [LinkedIn](https://linkedin.com/in/yourprofile) for daily updates!*
EOF

# Create .gitignore file (essential for data projects)
cat > .gitignore << 'EOF'
# Data files (too large for Git)
*.csv
*.xlsx
*.json
data/raw/
data/processed/
*.parquet

# Credentials and secrets
*.env
config/credentials/
.aws/
*.pem
*.key

# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
.venv/
pip-log.txt
pip-delete-this-directory.txt

# Jupyter Notebooks checkpoints
.ipynb_checkpoints

# IDE files
.vscode/
.idea/
*.swp
*.swo

# OS files
.DS_Store
Thumbs.db
```

```
# Logs
*.log
logs/

# Temporary files
tmp/
temp/
*.tmp

# Database files
*.db
*.sqlite
*.sqlite3

# Compressed files
*.zip
*.tar.gz
*.rar

# Large model files
*.pkl
*.joblib
*.h5
*.hdf5
EOF
```

Step 3: Add Previous Work:

bash

```
# Copy Python work from Day 2
mkdir day-02-python/src
cat > day-02-python/src/data_processor.py << 'EOF'
#!/usr/bin/env python3
"""
Day 2: Python Fundamentals for Data Engineering
E-commerce transaction data processor
"""

import pandas as pd
import json
from datetime import datetime
import os

class EcommerceDataProcessor:
    """Process e-commerce transaction data with error handling."""

    def __init__(self, data_file):
        self.data_file = data_file
        self.df = None

    def load_data(self):
        """Load data with comprehensive error handling."""
        try:
            if self.data_file.endswith('.csv'):
                self.df = pd.read_csv(self.data_file)
            elif self.data_file.endswith('.json'):
                self.df = pd.read_json(self.data_file)
            else:
                raise ValueError("Unsupported file format")

            print(f"✅ Successfully loaded {len(self.df)} records")
            return True

        except FileNotFoundError:
            print(f"❌ Error: File {self.data_file} not found")
            return False
        except Exception as e:
            print(f"❌ Error loading data: {e}")
            return False

    def analyze_data(self):
        """Perform basic data analysis."""
```

```
        if self.df is None:
            print("No data loaded")
            return

        print("\n=== Data Analysis Summary ===")
        print(f"Dataset shape: {self.df.shape}")
        print(f"Memory usage: {self.df.memory_usage(deep=True).sum() / 1024:.2f} KB")
        print("\nColumn information:")
        print(self.df.info())

if __name__ == "__main__":
    # Example usage
    processor = EcommerceDataProcessor("sample_data.csv")
    if processor.load_data():
        processor.analyze_data()
EOF

# Copy SQL work from Day 3
mkdir day-03-sql/queries
cat > day-03-sql/queries/customer_analysis.sql << 'EOF'
-- Day 3: SQL Fundamentals
-- Customer analysis queries for Superstore dataset

-- Customer summary with key metrics
SELECT
    customer_name,
    customer_id,
    segment,
    region,
    COUNT(DISTINCT order_id) as total_orders,
    SUM(sales) as total_sales,
    SUM(profit) as total_profit,
    AVG(sales) as avg_order_value,
    MAX(order_date) as last_order_date
FROM superstore
GROUP BY customer_name, customer_id, segment, region
HAVING SUM(sales) > 1000
ORDER BY total_sales DESC
LIMIT 50;

-- Regional performance analysis
SELECT
    region,
    COUNT(DISTINCT customer_id) as unique_customers,
```

```
    COUNT(DISTINCT order_id) as total_orders,
    SUM(sales) as total_revenue,
    AVG(sales) as avg_order_value,
    SUM(profit) / SUM(sales) * 100 as profit_margin_pct
FROM superstore
GROUP BY region
ORDER BY total_revenue DESC;
EOF

# Copy Linux scripts from Day 7
mkdir day-07-linux/scripts
cat > day-07-linux/scripts/log_analyzer.sh << 'EOF'
#!/bin/bash
# Day 7: Linux Command Line
# Production log analyzer script

LOG_FILE=${1:-"access.log"}
REPORT_FILE="log_analysis_$(date +%Y%m%d_%H%M%S).txt"

echo "=== Log Analysis Report ===" > $REPORT_FILE
echo "Generated: $(date)" >> $REPORT_FILE
echo "Log file: $LOG_FILE" >> $REPORT_FILE
echo >> $REPORT_FILE

if [ ! -f "$LOG_FILE" ]; then
    echo "Error: Log file $LOG_FILE not found"
    exit 1
fi

# Basic statistics
echo "=== Basic Statistics ===" >> $REPORT_FILE
echo "Total lines: $(wc -l < $LOG_FILE)" >> $REPORT_FILE
echo "File size: $(ls -lh $LOG_FILE | awk '{print $5}')" >> $REPORT_FILE
echo >> $REPORT_FILE

# Error analysis
echo "=== Error Analysis ===" >> $REPORT_FILE
grep -c "ERROR" $LOG_FILE >> $REPORT_FILE
echo >> $REPORT_FILE

echo "Analysis complete: $REPORT_FILE"
EOF

chmod +x day-07-linux/scripts/log_analyzer.sh
```

```
# Create documentation structure
mkdir docs/daily-notes
cat > docs/daily-notes/README.md << 'EOF'
# Daily Learning Notes

This directory contains detailed learning notes for each day of the challenge.

## Template

Each day should include:
- Key concepts learned
- Practical exercises completed
- Real-world applications
- Challenges overcome
- Resources used

## Progress Tracking

- [ ] Day 1: Introduction and Setup
- [x] Day 2: Python Fundamentals
- [x] Day 3: SQL Fundamentals
- [x] Day 4: Advanced SQL
- [ ] Day 5: Data Modeling
- [x] Day 6: Cloud Platforms (AWS)
- [x] Day 7: Linux Command Line
- [x] Day 8: Git and Version Control
EOF
```

Your First Git Commits

Step 4: Stage and Commit Files:

```bash
# Check repository status
git status
# Shows untracked files in red

# Add specific files
git add README.md
git add .gitignore

# Check status again
git status
# Shows staged files in green

# Make your first commit
git commit -m "Initial commit: Add project structure and README

- Set up organized directory structure for 50-day challenge
- Add comprehensive .gitignore for data engineering projects
- Create main README with project overview and progress tracking"

# Add remaining files
git add day-02-python/
git add day-03-sql/
git add day-07-linux/
git add docs/

# Commit with descriptive message
git commit -m "Add initial work from Days 2, 3, and 7

- Python data processor with error handling (Day 2)
- SQL customer analysis queries (Day 3)
- Linux log analyzer script (Day 7)
- Documentation structure and progress tracking"

# View commit history
git log --oneline
```

Understanding Git Commands:

- `git add`: Stage files for commit (prepare changes)
- `git commit`: Save changes to repository with message
- `git status`: Show current state of working directory

- `git log`: View commit history

🌐 GitHub Integration and Remote Repositories (45 minutes)

Setting Up GitHub

Step 1: Create GitHub Account:

- Go to `github.com`
- Sign up with same email used in Git config
- Choose free plan (sufficient for learning)
- Verify email address

Step 2: Create Repository on GitHub:

- Click "New repository" (green button)
- Repository name: `data-engineering-50-days`
- Description: `Complete 50-day journey to master data engineering skills`
- Set to Public (for portfolio visibility)
- **Don't** initialize with README (we already have one)
- Click "Create repository"

Step 3: Connect Local Repository to GitHub:

```bash
# Add GitHub as remote origin
git remote add origin https://github.com/YOUR_USERNAME/data-engineering-50-days.git

# Verify remote configuration
git remote -v
# Should show origin with your GitHub URL

# Push local commits to GitHub
git push -u origin main
# -u sets upstream tracking for future pushes

# Verify on GitHub
# Go to your repository URL - you should see all files and commits
```

Understanding Remote Repositories

Local vs Remote:

- **Local Repository**: Git files on your computer (`.git` directory)
- **Remote Repository**: Git files on GitHub servers
- **Origin**: Default name for primary remote repository
- **Upstream**: Reference branch that local branch tracks

Common Remote Operations:

```bash
# Push changes to GitHub
git push origin main

# Pull changes from GitHub (when working with others)
git pull origin main

# Clone repository to new location
git clone https://github.com/YOUR_USERNAME/data-engineering-50-days.git

# Fetch latest changes without merging
git fetch origin
```

🌿 Branching and Professional Workflows (60 minutes)

Understanding Branches

What are Branches?: Think of branches as parallel universes for your code:

- `main` branch: Production-ready, stable code
- `feature` branches: Experimental work, new features
- `bugfix` branches: Fixes for specific issues
- `hotfix` branches: Urgent production fixes

Why Use Branches?:

- Work on features without breaking main code
- Collaborate without conflicts
- Test changes safely

- Easy rollback if something goes wrong

Feature Branch Workflow

Scenario: Add a new data validation module

Step 1: Create Feature Branch:

```bash
# Create and switch to new branch
git checkout -b feature/add-data-validation

# Verify you're on new branch
git branch
# * feature/add-data-validation (current branch marked with *)
#   main

# Alternative: create branch without switching
git branch feature/add-data-validation
git checkout feature/add-data-validation
```

Step 2: Develop Feature:

bash

```
# Create new validation module
mkdir day-08-git/data-validation
cat > day-08-git/data-validation/validator.py << 'EOF'
#!/usr/bin/env python3
"""
Data Validation Module for ETL Pipelines
Ensures data quality before processing
"""

import pandas as pd
import numpy as np
from typing import Dict, List, Any
import logging

class DataValidator:
    """Comprehensive data validation for ETL pipelines."""

    def __init__(self):
        self.validation_results = {}
        self.logger = logging.getLogger(__name__)

    def validate_completeness(self, df: pd.DataFrame, required_columns: List[str]) -> I
        """Check for missing required columns and null values."""
        results = {
            'test': 'completeness',
            'passed': True,
            'issues': []
        }

        # Check required columns exist
        missing_columns = set(required_columns) - set(df.columns)
        if missing_columns:
            results['passed'] = False
            results['issues'].append(f"Missing required columns: {missing_columns}")

        # Check for null values in required columns
        for col in required_columns:
            if col in df.columns:
                null_count = df[col].isnull().sum()
                if null_count > 0:
                    results['passed'] = False
                    results['issues'].append(f"Column '{col}' has {null_count} null va
```

```
        return results

    def validate_data_types(self, df: pd.DataFrame, expected_types: Dict[str, str]) ->
        """Validate column data types match expectations."""
        results = {
            'test': 'data_types',
            'passed': True,
            'issues': []
        }

        for col, expected_type in expected_types.items():
            if col in df.columns:
                actual_type = str(df[col].dtype)
                if expected_type not in actual_type:
                    results['passed'] = False
                    results['issues'].append(f"Column '{col}' expected {expected_type}

        return results

    def validate_ranges(self, df: pd.DataFrame, range_checks: Dict[str, Dict]) -> Dict
        """Validate numeric columns are within expected ranges."""
        results = {
            'test': 'ranges',
            'passed': True,
            'issues': []
        }

        for col, checks in range_checks.items():
            if col in df.columns:
                if 'min' in checks:
                    below_min = (df[col] < checks['min']).sum()
                    if below_min > 0:
                        results['passed'] = False
                        results['issues'].append(f"Column '{col}' has {below_min} valu

                if 'max' in checks:
                    above_max = (df[col] > checks['max']).sum()
                    if above_max > 0:
                        results['passed'] = False
                        results['issues'].append(f"Column '{col}' has {above_max} valu

        return results

    def run_full_validation(self, df: pd.DataFrame, validation_config: Dict) -> Dict[s
```

```
        """Run complete validation suite."""
        all_results = {
            'timestamp': pd.Timestamp.now(),
            'total_records': len(df),
            'validation_passed': True,
            'tests': []
        }

        # Run completeness checks
        if 'required_columns' in validation_config:
            completeness_result = self.validate_completeness(df, validation_config['re
            all_results['tests'].append(completeness_result)
            if not completeness_result['passed']:
                all_results['validation_passed'] = False

        # Run data type checks
        if 'expected_types' in validation_config:
            types_result = self.validate_data_types(df, validation_config['expected_ty
            all_results['tests'].append(types_result)
            if not types_result['passed']:
                all_results['validation_passed'] = False

        # Run range checks
        if 'range_checks' in validation_config:
            ranges_result = self.validate_ranges(df, validation_config['range_checks']
            all_results['tests'].append(ranges_result)
            if not ranges_result['passed']:
                all_results['validation_passed'] = False

        return all_results

# Example usage and testing
if __name__ == "__main__":
    # Create sample data for testing
    sample_data = pd.DataFrame({
        'customer_id': [1, 2, 3, None, 5],
        'sales': [100.50, 250.75, -10.00, 150.25, 500.00],
        'quantity': [1, 2, 3, 1, 5],
        'order_date': pd.date_range('2025-01-01', periods=5)
    })

    # Define validation configuration
    config = {
        'required_columns': ['customer_id', 'sales', 'quantity'],
```

```
        'expected_types': {
            'customer_id': 'int',
            'sales': 'float',
            'quantity': 'int'
        },
        'range_checks': {
            'sales': {'min': 0, 'max': 10000},
            'quantity': {'min': 1, 'max': 100}
        }
    }

    # Run validation
    validator = DataValidator()
    results = validator.run_full_validation(sample_data, config)

    print("=== Data Validation Results ===")
    print(f"Validation passed: {results['validation_passed']}")
    print(f"Total records: {results['total_records']}")
    print("\nTest Results:")
    for test in results['tests']:
        print(f"- {test['test']}: {'PASS' if test['passed'] else 'FAIL'}")
        if test['issues']:
            for issue in test['issues']:
                print(f"  ❌ {issue}")
EOF

# Add test file
cat > day-08-git/data-validation/test_validator.py << 'EOF'
#!/usr/bin/env python3
"""
Tests for DataValidator module
"""

import pandas as pd
import pytest
from validator import DataValidator

def test_completeness_validation():
    """Test completeness validation functionality."""
    validator = DataValidator()

    # Test data with missing values
    df = pd.DataFrame({
        'id': [1, 2, None],
```

```
        'name': ['A', 'B', 'C'],
        'value': [10, 20, 30]
    })

    required_cols = ['id', 'name', 'value']
    result = validator.validate_completeness(df, required_cols)

    assert not result['passed']  # Should fail due to null in 'id'
    assert len(result['issues']) > 0

def test_data_type_validation():
    """Test data type validation."""
    validator = DataValidator()

    df = pd.DataFrame({
        'id': [1, 2, 3],
        'value': [1.0, 2.0, 3.0]
    })

    expected_types = {'id': 'int', 'value': 'float'}
    result = validator.validate_data_types(df, expected_types)

    assert result['passed']  # Should pass

if __name__ == "__main__":
    test_completeness_validation()
    test_data_type_validation()
    print("All tests passed!")
EOF

# Create documentation for the feature
cat > day-08-git/data-validation/README.md << 'EOF'
# Data Validation Module

A comprehensive data validation system for ETL pipelines.

## Features

- **Completeness Validation**: Check for missing columns and null values
- **Data Type Validation**: Ensure columns match expected types
- **Range Validation**: Verify numeric values are within acceptable ranges
- **Comprehensive Reporting**: Detailed validation results with specific issues

## Usage
```

```python
from validator import DataValidator

# Configure validation rules
config = {
    'required_columns': ['customer_id', 'sales'],
    'expected_types': {'customer_id': 'int', 'sales': 'float'},
    'range_checks': {'sales': {'min': 0, 'max': 10000}}
}

# Run validation
validator = DataValidator()
results = validator.run_full_validation(df, config)

if results['validation_passed']:
    print("✅ Data validation passed")
else:
    print("❌ Data validation failed")
    for test in results['tests']:
        if not test['passed']:
            print(f"Issues in {test['test']}: {test['issues']}")
```

Integration with ETL Pipelines

This validator should be used at key points in data pipelines:

1. **Data Ingestion**: Validate raw data quality
2. **Transformation**: Ensure transformations don't break data integrity
3. **Loading**: Final validation before loading to warehouse

Testing

Run tests with:

```
bash

python test_validator.py
```

EOF

Stage and commit the feature

git add day-08-git/
git commit -m "Add comprehensive data validation module

Features:

- Completeness validation (missing columns, null values)
- Data type validation with configurable expectations
- Range validation for numeric columns
- Comprehensive reporting with detailed issue tracking
- Unit tests and documentation
- Example usage patterns for ETL pipelines

This module can be integrated into data pipelines to ensure
data quality at ingestion, transformation, and loading stages."

```
**Step 3: Push Feature Branch**:
```bash
# Push feature branch to GitHub
git push origin feature/add-data-validation

# View branches on GitHub
# Go to your repository, click "branches" to see all branches
```

Pull Requests and Code Reviews

Step 4: Create Pull Request:

1. **On GitHub**:
   - Go to your repository
   - Click "Compare & pull request" (appears after pushing branch)
   - Or click "Pull requests" tab → "New pull request"
2. **Pull Request Configuration**:
```

```
Title: Add comprehensive data validation module for ETL pipelines

Base branch: main
Compare branch: feature/add-data-validation

Description:
## Summary
Adds a professional data validation module for ensuring data quality in ETL
pipelines.

## Features
- ✅ Completeness validation (missing columns, null values)
- ✅ Data type validation with configurable expectations
- ✅ Range validation for numeric columns
- ✅ Comprehensive reporting with detailed issue tracking
- ✅ Unit tests and documentation

## Testing
- All unit tests pass
- Example usage included
- Documentation complete

## Integration
Ready for integration into existing data pipelines. Can be used at:
- Data ingestion points
- Transformation validation
- Pre-loading quality checks

## Review Notes
Please review the validation logic and test coverage.
```

3. **Submit Pull Request**:
   - Click "Create pull request"
   - This opens the PR for review and discussion

Step 5: Self-Review Process (Simulating Team Review):

```bash
# Switch to main branch to review changes
git checkout main

# View differences between branches
git diff main..feature/add-data-validation

# View file changes
git log --oneline main..feature/add-data-validation

# Test the feature branch locally
git checkout feature/add-data-validation
cd day-08-git/data-validation
python test_validator.py
cd ../../

# If everything looks good, merge via GitHub or command line
```

Step 6: Merge Pull Request:

On GitHub:

- Click "Merge pull request"
- Choose merge type:
  - **Create merge commit**: Preserves branch history
  - **Squash and merge**: Combines all commits into one
  - **Rebase and merge**: Replays commits on main branch
- Add merge commit message
- Click "Confirm merge"
- Delete feature branch (cleanup)

Step 7: Update Local Repository:

```bash
# Switch to main branch
git checkout main

# Pull latest changes from GitHub
git pull origin main

# Delete local feature branch (cleanup)
git branch -d feature/add-data-validation

# Verify merge
git log --oneline -5
```

🔧 Handling Merge Conflicts (30 minutes)

Understanding Merge Conflicts

What Causes Conflicts?:

- Two branches modify the same lines in a file
- One branch deletes a file while another modifies it
- Different branches add files with same name

Simulating a Merge Conflict:

Step 1: Create Conflicting Branches:

bash

```
# Create first feature branch
git checkout -b feature/update-readme-v1

# Modify README.md
cat >> README.md << 'EOF'

## Current Progress

### Week 1: Foundations
- [x] Day 2: Python fundamentals with pandas and data processing
- [x] Day 3: SQL basics with PostgreSQL and data analysis
- [x] Day 4: Advanced SQL with window functions and CTEs
- [x] Day 6: AWS cloud fundamentals with S3 and IAM
- [x] Day 7: Linux command line for data engineering
- [x] Day 8: Git version control and collaboration

### Achievements
- Built real data processing pipelines
- Analyzed 50K+ records with SQL
- Set up production cloud infrastructure
- Mastered professional development workflows
EOF

git add README.md
git commit -m "Update progress tracking with detailed achievements"

# Switch back to main and create conflicting branch
git checkout main
git checkout -b feature/update-readme-v2

# Make different changes to same section
cat >> README.md << 'EOF'

## Learning Journey

### Foundations Complete ✅
- Python data processing with error handling
- SQL analytics from basic to advanced
- Cloud infrastructure on AWS
- Linux automation and scripting
- Professional Git workflows

### Key Skills Acquired
```

```
- ETL pipeline development
- Database query optimization
- Cloud resource management
- Data validation and quality assurance
EOF

git add README.md
git commit -m "Add learning journey section with skills summary"
```

Step 2: Attempt Merge (This Will Conflict):

```
bash

# Try to merge first branch into second
git merge feature/update-readme-v1
# Output: CONFLICT (content): Merge conflict in README.md
# Automatic merge failed; fix conflicts and then commit the result.
```

Step 3: Resolve Merge Conflict:

```
bash

# Check conflict status
git status
# Shows files with conflicts

# View the conflict in README.md
cat README.md
# You'll see conflict markers:
# <<<<<<< HEAD
# (current branch content)
# =======
# (merging branch content)
# >>>>>>> feature/update-readme-v1
```

Manual Conflict Resolution:

bash

```
# Edit README.md to resolve conflict
# Remove conflict markers and combine content meaningfully
cat > temp_readme.md << 'EOF'
# Data Engineering 50-Day Challenge

A comprehensive journey to master data engineering skills in 50 days.

## Project Structure

- `day-XX-topic/` - Daily learning materials and projects
- `docs/` - Documentation and learning notes
- `scripts/` - Reusable utility scripts
- `configs/` - Configuration files and templates
- `data/` - Sample datasets (gitignored for large files)
- `tests/` - Test scripts and validation

## Skills Covered

- [x] Python Fundamentals (Day 2)
- [x] SQL and Advanced SQL (Days 3-4)
- [x] Cloud Platforms - AWS (Day 6)
- [x] Linux Command Line (Day 7)
- [x] Git and Version Control (Day 8)
- [ ] Docker and Containerization (Day 9)
- [ ] And 41 more days of advanced topics...

## Learning Philosophy

**Understanding > Coding** - Focus on concepts before implementation.

## Resources

Each day includes:
- Conceptual explanations
- Hands-on projects with real datasets
- Production-ready examples
- Industry best practices

---

*Follow along at [LinkedIn](https://linkedin.com/in/yourprofile) for daily updates!*

## Learning Journey & Progress
```

```
### Foundations Complete ✅
- [x] Day 2: Python fundamentals with pandas and data processing
- [x] Day 3: SQL basics with PostgreSQL and data analysis
- [x] Day 4: Advanced SQL with window functions and CTEs
- [x] Day 6: AWS cloud fundamentals with S3 and IAM
- [x] Day 7: Linux command line for data engineering
- [x] Day 8: Git version control and collaboration

### Key Skills Acquired
- ETL pipeline development and data processing
- Database query optimization and analytics
- Cloud resource management and infrastructure
- Data validation and quality assurance
- Professional development workflows
- Linux automation and scripting

### Achievements
- Built real data processing pipelines with error handling
- Analyzed 50K+ records with advanced SQL techniques
- Set up production-ready cloud infrastructure
- Mastered professional Git collaboration patterns
- Created comprehensive data validation systems
EOF

# Replace README.md with resolved version
mv temp_readme.md README.md

# Stage the resolved file
git add README.md

# Complete the merge
git commit -m "Resolve merge conflict in README.md

Combined progress tracking and learning journey sections.
Merged detailed achievements with skills summary for
comprehensive project documentation."

# Verify merge completion
git log --oneline -3
```

Step 4: Clean Up:

```bash
# Switch to main and merge the resolved branch
git checkout main
git merge feature/update-readme-v2

# Delete feature branches
git branch -d feature/update-readme-v1
git branch -d feature/update-readme-v2

# Push updated main branch
git push origin main
```

Advanced Conflict Resolution Tools

Using Git Merge Tools:

```bash
# Configure merge tool (VS Code example)
git config --global merge.tool code

# Use merge tool for conflicts
git mergetool

# Alternative: manual resolution with diff
git diff --name-only --diff-filter=U  # Show conflicted files
git show :1:filename  # Common ancestor version
git show :2:filename  # Current branch version
git show :3:filename  # Merging branch version
```

📋 Professional Git Workflows (45 minutes)

Gitflow Workflow for Data Engineering

Branch Types in Data Engineering:

1. **main**: Production-ready ETL pipelines
2. **develop**: Integration branch for features
3. **feature/**: New data sources, transformations
4. **release/**: Preparing production deployments
5. **hotfix/**: Emergency fixes for production issues

Setting Up Gitflow:

bash

```
# Create develop branch
git checkout -b develop
git push origin develop

# Set develop as default branch for features
git checkout develop

# Example feature workflow
git checkout -b feature/add-kafka-integration

# Work on feature...
cat > day-08-git/kafka-integration/consumer.py << 'EOF'
#!/usr/bin/env python3
"""
Kafka Consumer for Real-time Data Ingestion
Integrates with existing data validation pipeline
"""

import json
import logging
from kafka import KafkaConsumer
from datetime import datetime
import pandas as pd

class DataStreamConsumer:
    """Real-time data consumer with validation integration."""

    def __init__(self, topic, bootstrap_servers='localhost:9092'):
        self.topic = topic
        self.bootstrap_servers = bootstrap_servers
        self.consumer = None
        self.logger = logging.getLogger(__name__)

    def connect(self):
        """Establish connection to Kafka cluster."""
        try:
            self.consumer = KafkaConsumer(
                self.topic,
                bootstrap_servers=self.bootstrap_servers,
                value_deserializer=lambda x: json.loads(x.decode('utf-8')),
                auto_offset_reset='latest',
                group_id='data-engineering-group'
            )
```

```
            self.logger.info(f"Connected to Kafka topic: {self.topic}")
            return True
        except Exception as e:
            self.logger.error(f"Failed to connect to Kafka: {e}")
            return False

    def process_stream(self, batch_size=100):
        """Process streaming data in batches."""
        if not self.consumer:
            self.logger.error("Consumer not connected")
            return

        batch = []

        try:
            for message in self.consumer:
                # Add message to batch
                batch.append({
                    'timestamp': datetime.now(),
                    'offset': message.offset,
                    'data': message.value
                })

                # Process batch when full
                if len(batch) >= batch_size:
                    self._process_batch(batch)
                    batch = []

        except KeyboardInterrupt:
            self.logger.info("Shutting down consumer...")
            if batch:  # Process remaining messages
                self._process_batch(batch)

    def _process_batch(self, batch):
        """Process a batch of messages."""
        self.logger.info(f"Processing batch of {len(batch)} messages")

        # Convert to DataFrame for processing
        df = pd.DataFrame([msg['data'] for msg in batch])

        # Here you would integrate with the DataValidator
        # from our previous feature branch

        # Example processing
```

```
        processed_count = len(df)
        self.logger.info(f"Successfully processed {processed_count} records")

# Example configuration
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)

    consumer = DataStreamConsumer('ecommerce-transactions')
    if consumer.connect():
        consumer.process_stream(batch_size=50)
EOF

mkdir -p day-08-git/kafka-integration
# Move the file to correct location
mv day-08-git/kafka-integration/consumer.py day-08-git/kafka-integration/

# Commit feature
git add day-08-git/kafka-integration/
git commit -m "Add Kafka integration for real-time data streaming

Features:
- Real-time consumer with batch processing
- Integration points for data validation
- Configurable batch sizes and connection settings
- Comprehensive logging and error handling
- Ready for integration with existing validation pipeline"

# Merge to develop
git checkout develop
git merge feature/add-kafka-integration
git branch -d feature/add-kafka-integration

# Create release branch
git checkout -b release/v1.0.0

# Prepare release (update version, documentation)
cat > CHANGELOG.md << 'EOF'
# Changelog

All notable changes to this project will be documented in this file.

## [1.0.0] - 2025-01-15

### Added
```

```
- Complete project structure for 50-day data engineering challenge
- Python data processing module with error handling
- SQL query collection for customer and sales analysis
- Linux automation scripts for log analysis
- AWS cloud infrastructure documentation
- Comprehensive data validation system for ETL pipelines
- Kafka integration for real-time data streaming
- Professional Git workflow documentation

### Features
- Data validation with completeness, type, and range checking
- Real-time stream processing with batch handling
- Production-ready error handling and logging
- Comprehensive test coverage
- Professional documentation standards

### Infrastructure
- Organized project structure for scalability
- Proper .gitignore for data engineering projects
- CI/CD ready configuration
- Environment separation patterns
EOF

git add CHANGELOG.md
git commit -m "Prepare v1.0.0 release

- Add comprehensive changelog
- Document all features and improvements
- Ready for production deployment"

# Merge release to main
git checkout main
git merge release/v1.0.0

# Tag the release
git tag -a v1.0.0 -m "Release v1.0.0: Complete data engineering foundation

Includes:
- Data processing and validation systems
- Cloud infrastructure setup
- Real-time streaming capabilities
- Professional development workflows"

# Merge back to develop
```

```
git checkout develop
git merge release/v1.0.0

# Clean up release branch
git branch -d release/v1.0.0

# Push everything
git push origin main
git push origin develop
git push origin v1.0.0
```

Hotfix Workflow

Emergency Production Fix:

bash

```
# Create hotfix from main
git checkout main
git checkout -b hotfix/fix-validation-memory-leak

# Fix critical issue
cat > day-08-git/data-validation/validator_patch.py << 'EOF'
#!/usr/bin/env python3
"""
Memory optimization patch for DataValidator
Fixes memory leak in large dataset processing
"""

import gc
import psutil
import os

class OptimizedDataValidator:
    """Memory-optimized version of DataValidator."""

    def __init__(self):
        self.validation_results = {}
        self.max_memory_usage = 0.8  # 80% of available memory

    def _check_memory_usage(self):
        """Monitor memory usage and trigger cleanup if needed."""
        process = psutil.Process(os.getpid())
        memory_percent = process.memory_percent()

        if memory_percent > self.max_memory_usage * 100:
            gc.collect()  # Force garbage collection
            return True
        return False

    def validate_large_dataset(self, df, chunk_size=10000):
        """Process large datasets in chunks to prevent memory issues."""
        total_rows = len(df)
        results = []

        for start_idx in range(0, total_rows, chunk_size):
            end_idx = min(start_idx + chunk_size, total_rows)
            chunk = df.iloc[start_idx:end_idx]

            # Process chunk
```

```
            chunk_result = self._process_chunk(chunk)
            results.append(chunk_result)

            # Check memory and cleanup if needed
            if self._check_memory_usage():
                del chunk  # Explicit cleanup

        return self._combine_results(results)

    def _process_chunk(self, chunk):
        """Process individual chunk with memory awareness."""
        # Implementation here
        return {"processed": len(chunk)}

    def _combine_results(self, results):
        """Combine chunk results into final result."""
        total_processed = sum(r["processed"] for r in results)
        return {"total_processed": total_processed}
EOF

# Update main validator to include fix
sed -i '/class DataValidator:/a\    def __init__(self):\n        self.validation_resul

git add day-08-git/data-validation/
git commit -m "HOTFIX: Fix memory leak in data validation

Critical fix for production memory issues:
- Add memory monitoring and garbage collection
- Implement chunked processing for large datasets
- Prevent out-of-memory errors in ETL pipelines
- Maintain backward compatibility

Impact: Prevents production pipeline failures on large datasets"

# Merge to main
git checkout main
git merge hotfix/fix-validation-memory-leak

# Tag hotfix
git tag -a v1.0.1 -m "Hotfix v1.0.1: Fix critical memory leak in data validation"

# Merge to develop
git checkout develop
git merge hotfix/fix-validation-memory-leak
```

```bash
# Clean up
git branch -d hotfix/fix-validation-memory-leak

# Push all changes
git push origin main
git push origin develop
git push origin v1.0.1
```

🔄 Advanced Git Operations (30 minutes)

Git History and Inspection

Viewing Commit History:

```bash
# Basic log
git log --oneline

# Detailed log with changes
git log --stat

# Graphical representation
git log --graph --oneline --all

# Search commit messages
git log --grep="validation"

# Show commits by author
git log --author="Your Name"

# Show commits in date range
git log --since="2025-01-01" --until="2025-01-15"

# Show file history
git log --follow -- day-08-git/data-validation/validator.py
```

Inspecting Changes:

bash

```bash
# Show changes in specific commit
git show abc1234

# Compare two commits
git diff abc1234..def5678

# Show changes between branches
git diff main..develop

# Show changes in specific file
git diff HEAD~1 HEAD -- README.md

# Show who changed each line (blame)
git blame README.md
```

Undoing Changes

Different Types of Undo:

bash

```bash
# Undo working directory changes (before staging)
git checkout -- filename.py

# Unstage files (after git add, before commit)
git reset HEAD filename.py

# Undo last commit (keep changes in working directory)
git reset --soft HEAD~1

# Undo last commit (discard changes - DANGEROUS)
git reset --hard HEAD~1

# Revert a commit (creates new commit that undoes changes)
git revert abc1234

# Interactive rebase to edit history
git rebase -i HEAD~3
```

Safe Rollback Strategies:

```bash
# Create backup branch before major operations
git branch backup-before-rebase

# Use revert for shared branches (main, develop)
git revert problematic-commit-hash

# Use reset only for local, unshared branches
git reset --hard origin/main  # Reset to remote state
```

Git Stash for Work-in-Progress

Temporary Storage:

```bash
# Save current work without committing
git stash

# Save with description
git stash save "WIP: adding new feature X"

# List all stashes
git stash list

# Apply most recent stash
git stash apply

# Apply specific stash
git stash apply stash@{2}

# Apply and remove from stash
git stash pop

# Create branch from stash
git stash branch new-feature-branch stash@{1}
```

Real-World Stash Usage:

bash

```bash
# Scenario: Working on feature, need to fix urgent bug
git stash save "WIP: data validation enhancement"
git checkout main
git checkout -b hotfix/urgent-fix

# ... fix the bug ...
git add .
git commit -m "Fix urgent production issue"
git checkout main
git merge hotfix/urgent-fix

# Return to original work
git checkout feature-branch
git stash pop  # Continue where you left off
```

🔒 Git Security and Best Practices (20 minutes)

Protecting Sensitive Information

Never Commit These:

bash

```bash
# Update .gitignore for security
cat >> .gitignore << 'EOF'

# Security - Never commit these!
*.env
.env.*
config/secrets/
credentials/
*.pem
*.key
*.p12
*.pfx
id_rsa*
password.txt
secrets.yaml
api_keys.json

# Database connections
database.ini
db_config.py
connection_strings.txt

# AWS/Cloud credentials
.aws/credentials
.azure/
.gcp/
terraform.tfstate
terraform.tfvars

# Application secrets
secret_key.txt
jwt_secret
oauth_tokens.json
EOF
```

Environment Variables Pattern:

bash

```
# Create example configuration
cat > day-08-git/configs/database_config.example.py << 'EOF'
#!/usr/bin/env python3
"""
Database configuration template
Copy to database_config.py and fill in real values
"""

import os

# Database connection settings
DATABASE_CONFIG = {
    'host': os.getenv('DB_HOST', 'localhost'),
    'port': os.getenv('DB_PORT', 5432),
    'database': os.getenv('DB_NAME', 'your_database'),
    'username': os.getenv('DB_USER', 'your_username'),
    'password': os.getenv('DB_PASSWORD', 'your_password'),  # Never hardcode!
}

# AWS configuration
AWS_CONFIG = {
    'access_key_id': os.getenv('AWS_ACCESS_KEY_ID'),
    'secret_access_key': os.getenv('AWS_SECRET_ACCESS_KEY'),
    'region': os.getenv('AWS_DEFAULT_REGION', 'us-east-1'),
    'bucket_name': os.getenv('S3_BUCKET_NAME'),
}

# API keys
API_KEYS = {
    'external_api': os.getenv('EXTERNAL_API_KEY'),
    'data_source_api': os.getenv('DATA_SOURCE_API_KEY'),
}

# Usage in your code:
# from configs.database_config import DATABASE_CONFIG
# connection = psycopg2.connect(**DATABASE_CONFIG)
EOF

# Create environment template
cat > .env.example << 'EOF'
# Environment Variables Template
# Copy to .env and fill in real values
```

```
# Database Configuration
DB_HOST=localhost
DB_PORT=5432
DB_NAME=your_database_name
DB_USER=your_username
DB_PASSWORD=your_secure_password

# AWS Configuration
AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key
AWS_DEFAULT_REGION=us-east-1
S3_BUCKET_NAME=your-bucket-name

# External APIs
EXTERNAL_API_KEY=your_api_key
DATA_SOURCE_API_KEY=another_api_key

# Application Settings
SECRET_KEY=your_secret_key_for_app
DEBUG=false
ENVIRONMENT=production
EOF

git add .gitignore day-08-git/configs/ .env.example
git commit -m "Add security configuration templates

- Update .gitignore to prevent credential commits
- Add database configuration template with environment variables
- Provide .env.example for secure credential management
- Follow security best practices for data engineering projects"
```

Commit Message Best Practices

Professional Commit Format:

```bash
# Good commit message structure:
# <type>(<scope>): <subject>
#
# <body>
#
# <footer>

# Examples:
git commit -m "feat(validation): add email format validation

Add regex-based email validation to DataValidator class.
Supports common email formats and provides detailed
error messages for invalid inputs.

Closes #123"

git commit -m "fix(pipeline): resolve memory leak in batch processing

- Implement proper cleanup of DataFrame objects
- Add memory monitoring and garbage collection
- Reduce memory usage by 60% in large dataset processing

Breaking change: batch_size parameter now required"

git commit -m "docs(readme): update installation instructions

Add detailed setup steps for development environment
including Python dependencies and database configuration."

# Commit types:
# feat: new feature
# fix: bug fix
# docs: documentation
# style: formatting, no code change
# refactor: code restructuring
# test: adding tests
# chore: maintenance
```

Git Hooks for Automation

Pre-commit Hook Example:

bash

```bash
# Create pre-commit hook
mkdir -p .git/hooks
cat > .git/hooks/pre-commit << 'EOF'
#!/bin/bash
# Pre-commit hook: Run tests and security checks

echo "Running pre-commit checks..."

# Check for sensitive information
if grep -r "password\s*=" --include="*.py" --include="*.sql" .; then
    echo "❌ Error: Found hardcoded passwords in code"
    echo "Use environment variables instead"
    exit 1
fi

# Check for AWS keys
if grep -r "AKIA[0-9A-Z]{16}" --include="*.py" --include="*.sql" .; then
    echo "❌ Error: Found AWS access keys in code"
    echo "Use IAM roles or environment variables"
    exit 1
fi

# Run Python tests if available
if [ -f "day-08-git/data-validation/test_validator.py" ]; then
    echo "Running Python tests..."
    cd day-08-git/data-validation
    if ! python test_validator.py; then
        echo "❌ Tests failed"
        exit 1
    fi
    cd ../..
fi

echo "✅ All pre-commit checks passed"
EOF

chmod +x .git/hooks/pre-commit

# Test the hook
git add .git/hooks/pre-commit
git commit -m "Add pre-commit hook for security and quality checks"
```

📊 Real-World Data Engineering Git Workflows (30 minutes)

Setting Up CI/CD Ready Repository

GitHub Actions Configuration:

bash

```
mkdir -p .github/workflows
cat > .github/workflows/data-pipeline-ci.yml << 'EOF'
name: Data Pipeline CI/CD

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8, 3.9, 3.10]

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v3
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pandas numpy pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi

    - name: Run data validation tests
      run: |
        cd day-08-git/data-validation
        python test_validator.py

    - name: Check code quality
      run: |
        # Add linting, security checks, etc.
        echo "Code quality checks passed"

    - name: Test SQL queries
      run: |
```

```
        # Add SQL syntax validation
        echo "SQL validation passed"

  security-scan:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3

    - name: Run security scan
      run: |
        # Check for hardcoded secrets
        if grep -r "password\s*=" --include="*.py" .; then
          echo "Security issue: hardcoded passwords found"
          exit 1
        fi
        echo "Security scan passed"

  deploy:
    needs: [test, security-scan]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v3

    - name: Deploy to production
      run: |
        echo "Deploying data pipelines to production..."
        # Add deployment steps here
EOF

# Create requirements file
cat > requirements.txt << 'EOF'
# Data Processing
pandas>=1.5.0
numpy>=1.24.0

# Database connectivity
psycopg2-binary>=2.9.0
sqlalchemy>=2.0.0

# Cloud services
boto3>=1.26.0
awscli>=1.27.0
```

```
# Real-time processing
kafka-python>=2.0.0

# Testing
pytest>=7.0.0

# Data validation
jsonschema>=4.0.0

# Monitoring and logging
structlog>=22.0.0

# Development tools
black>=23.0.0
flake8>=6.0.0
EOF

git add .github/ requirements.txt
git commit -m "Add CI/CD pipeline configuration

- GitHub Actions workflow for automated testing
- Multi-Python version testing matrix
- Security scanning for hardcoded credentials
- Automated deployment on main branch merges
- Requirements.txt for dependency management"
```

Data Engineering Project Documentation

Professional README.md:

```
bash

cat > PROJECT_TEMPLATE.md << 'EOF'
# Data Engineering Project Template

## Overview
Brief description of the data pipeline or project purpose.

## Architecture
```

Data Sources → Ingestion → Transformation → Storage → Analytics
↓ ↓ ↓ ↓ ↓

APIs/DBs → Kafka → Spark → S3 → Redshift

```
## Setup

### Prerequisites
- Python 3.8+
- Docker and Docker Compose
- AWS CLI configured
- PostgreSQL (for local development)

### Installation
```bash
git clone <repository-url>
cd <project-name>
pip install -r requirements.txt
cp .env.example .env  # Configure your environment variables
```

Configuration

1. Update `.env` with your credentials
2. Configure database connections
3. Set up AWS permissions

Usage

Running Locally

```bash
python src/main.py --config config/local.yaml
```

Running Tests

```bash
pytest tests/
```

Deploying

```bash
docker-compose up -d
```
```

Data Pipeline Components

1. Data Ingestion

- **Purpose**: Extract data from various sources
- **Technologies**: Kafka, REST APIs, Database connectors
- **Location**: `src/ingestion/`

2. Data Transformation

- **Purpose**: Clean, validate, and transform data
- **Technologies**: Apache Spark, Pandas
- **Location**: `src/transformation/`

3. Data Storage

- **Purpose**: Store processed data for analytics
- **Technologies**: AWS S3, Redshift, PostgreSQL
- **Location**: `src/storage/`

Development Workflow

Branch Strategy

- `main`: Production-ready code
- `develop`: Integration branch
- `feature/*`: New features
- `hotfix/*`: Production fixes

Code Review Process

1. Create feature branch from `develop`
2. Implement changes with tests
3. Submit pull request
4. Code review and approval
5. Merge to `develop`
6. Deploy via `main` branch

Monitoring and Alerting

Metrics

- Data processing latency
- Error rates and data quality
- Resource utilization

Alerts

- Pipeline failures
- Data quality issues
- Performance degradation

Troubleshooting

Common Issues

1. **Memory errors**: Increase batch sizes or add more workers
2. **Connection timeouts**: Check network connectivity and credentials
3. **Data quality failures**: Review validation rules and source data

Logs

- Application logs: `logs/application.log`
- Pipeline logs: `logs/pipeline.log`
- Error logs: `logs/errors.log`

Contributing

1. Fork the repository
2. Create feature branch
3. Add tests for new functionality
4. Ensure all tests pass
5. Submit pull request

License

[Your License Here]
EOF

git add PROJECT_TEMPLATE.md
git commit -m "Add comprehensive project documentation template

Professional template for data engineering projects including:

- Architecture diagrams and component descriptions
- Setup and deployment instructions
- Development workflow and branching strategy
- Monitoring and troubleshooting guides
- Contributing guidelines for team collaboration"

✅ Success Metrics and Assessment (15 minutes)

Day 8 Mastery Checklist

Git Fundamentals ✅:
- [ ] Initialize repositories and understand Git structure
- [ ] Stage, commit, and push changes effectively
- [ ] Write professional commit messages
- [ ] Navigate Git history and inspect changes

Branching and Merging ✅:
- [ ] Create and manage feature branches
- [ ] Merge branches with proper workflow
- [ ] Resolve merge conflicts manually
- [ ] Understand different merge strategies

Remote Collaboration ✅:
- [ ] Set up GitHub repository integration
- [ ] Create and manage pull requests
- [ ] Conduct code reviews effectively
- [ ] Synchronize local and remote repositories

Professional Workflows ✅:
- [ ] Implement Gitflow workflow patterns
- [ ] Handle hotfix and release branches
- [ ] Use Git tags for version management
- [ ] Set up automated CI/CD pipelines

Security and Best Practices ✅:
- [ ] Protect sensitive information with .gitignore
- [ ] Use environment variables for credentials
- [ ] Implement pre-commit hooks
- [ ] Follow commit message conventions

Knowledge Self-Assessment

Rate Your Confidence (1-10):
- Basic Git operations (add, commit, push, pull): ___/10
- Branching strategies and merge conflict resolution: ___/10
- GitHub collaboration and pull request workflows: ___/10

- Professional Git workflows (Gitflow, CI/CD): ___/10
- Git security and best practices: ___/10

Practical Challenges

Complete These Git Tasks:
1. Create a feature branch, implement a new data processing module, and merge via pull request
2. Simulate and resolve a complex merge conflict between two feature branches
3. Set up a complete Gitflow workflow with develop, feature, and release branches
4. Implement a pre-commit hook that validates Python code and SQL syntax
5. Configure GitHub Actions for automated testing of data pipeline code

---

🔗 Essential Resources for Continued Learning

Git Documentation and Guides

Official Documentation:
- Git Official Documentation: `git-scm.com/doc`
- GitHub Guides: `guides.github.com`
- Atlassian Git Tutorials: `atlassian.com/git/tutorials`

Interactive Learning:
- Learn Git Branching: `learngitbranching.js.org`
- GitHub Learning Lab: `lab.github.com`
- Git Immersion: `gitimmersion.com`

Advanced Git Techniques

Professional Workflows:
- Gitflow Workflow: `nvie.com/posts/a-successful-git-branching-model/`
- GitHub Flow: `guides.github.com/introduction/flow/`
- GitLab Flow: `docs.gitlab.com/ee/topics/gitlab_flow.html`

Advanced Git Operations:
- Interactive Rebase: `git-scm.com/book/en/v2/Git-Tools-Rewriting-History`
- Git Hooks: `git-scm.com/book/en/v2/Customizing-Git-Git-Hooks`
- Submodules: `git-scm.com/book/en/v2/Git-Tools-Submodules`

Data Engineering Specific Git Practices

Repository Organization:

- Cookiecutter Data Science: `drivendata.github.io/cookiecutter-data-science/`
- ML/Data Project Structure: `github.com/drivendata/cookiecutter-data-science`
- DBT Git Workflows: `docs.getdbt.com/docs/collaborate/git-workflow`

Version Control for Data:
- DVC (Data Version Control): `dvc.org`
- Git LFS for Large Files: `git-lfs.github.io`
- Delta Lake Versioning: `delta.io`

🚀 Tomorrow's Preview: Day 9 - Docker Fundamentals

What You'll Learn Tomorrow

Core Focus: Containerization for data engineering applications
- Docker fundamentals and container concepts
- Creating Docker images for data pipelines
- Docker Compose for multi-service applications
- Container orchestration basics

Why Docker Matters for Data Engineers:
- **Environment Consistency**: Same runtime everywhere (dev, staging, prod)
- **Dependency Management**: Bundle all requirements with your application
- **Scalability**: Easy horizontal scaling of data processing jobs
- **Isolation**: Prevent conflicts between different data tools
- **Deployment**: Simplified deployment to any Docker-capable platform

Tomorrow's Preparation

Tools to Install:
- Docker Desktop (for Windows/Mac) or Docker Engine (Linux)
- Docker Compose (usually included with Docker Desktop)
- Your Day 8 Git repository (we'll containerize the applications)

Concepts to Review:
- The difference between virtual machines and containers
- Why containerization revolutionized software deployment
- How containers solve "it works on my machine" problems

Real-World Applications We'll Build:
- Containerized data validation service
- Multi-container data pipeline with database
- Automated testing environment with Docker

- Production-ready container orchestration

🎉 Congratulations on Mastering Git!

What You've Accomplished Today

You've gained professional-level version control skills that are essential for any data engineering team. You now understand:

Technical Skills:
- Complete Git workflow from initialization to production deployment
- Professional branching strategies and merge conflict resolution
- GitHub collaboration patterns used by enterprise teams
- CI/CD pipeline setup for automated testing and deployment
- Security best practices for protecting sensitive data

Collaboration Skills:
- Code review processes that prevent production issues
- Pull request workflows for team development
- Communication through commit messages and documentation
- Project organization and documentation standards

Production-Ready Patterns:
- Gitflow workflow for complex data engineering projects
- Hotfix procedures for emergency production fixes
- Automated testing and quality gates
- Environment-based configuration management

Your Version Control Foundation is Solid

You're now equipped with the same Git skills used by data engineering teams at major technology companies. These skills enable you to:
- Collaborate safely on complex data pipelines
- Track and rollback changes in production systems
- Maintain multiple environments (dev, staging, production)
- Automate testing and deployment processes

Progress: 16% (8/50 days) | **Next**: Day 9 - Docker Fundamentals
Skills Mastered: Python ✅ + SQL ✅ + Advanced SQL ✅ + Cloud Fundamentals ✅ + Linux CLI ✅ + Git/Version Control ✅

Portfolio Project Status

Your `data-engineering-50-days` repository now includes:
- **Professional Structure**: Organized, scalable project layout
- **Real Applications**: Working data processing and validation modules
- **Documentation**: Comprehensive README and learning notes
- **CI/CD Ready**: GitHub Actions and automated testing setup
- **Security**: Proper credential management and .gitignore configuration
- **Version History**: Clean commit history demonstrating professional practices

This repository serves as both a learning journey and a professional portfolio that showcases your data engineering and software development skills to potential employers.

Learning Journal Template

```markdown
# Day 8: Git and Version Control - Learning Notes

## Version Control Skills Mastered
- Git fundamentals: staging, committing, branching, merging
- GitHub collaboration: pull requests, code reviews, remote repositories
- Professional workflows: Gitflow, hotfixes, release management
- Security practices: credential protection, pre-commit hooks

## Real-World Applications
- Built comprehensive data engineering portfolio repository
- Implemented professional branching strategy for team collaboration
- Set up automated CI/CD pipeline with GitHub Actions
- Created security-focused development workflow

## Key Insights
- Version control is essential for any team development
- Professional workflows prevent production issues
- Good commit messages serve as project documentation
- Security must be built into development processes from day one

## Tomorrow's Goals
- Learn Docker containerization fundamentals
- Containerize existing data engineering applications
- Understand container orchestration for scalable deployments
- Apply DevOps practices to data pipeline deployment
```

Final Git Command Reference

Daily Git Workflow:

```bash
# Start new feature
git checkout develop
git pull origin develop
git checkout -b feature/new-feature

# Work and commit
git add .
git commit -m "descriptive message"

# Push and create PR
git push origin feature/new-feature
# Create pull request on GitHub

# After merge, cleanup
git checkout develop
git pull origin develop
git branch -d feature/new-feature
```

Emergency Hotfix:

```bash
# Quick production fix
git checkout main
git checkout -b hotfix/urgent-fix
# ... make fixes ...
git commit -m "HOTFIX: description"
git push origin hotfix/urgent-fix
# Merge via GitHub, then:
git checkout main && git pull
git checkout develop && git pull
git branch -d hotfix/urgent-fix
```

You've now mastered the foundation skills that every professional data engineer needs. Tomorrow, we'll learn Docker to package these applications for consistent deployment across any environment!