Comprehensive Guide to Version Control Systems (Day 2)

**Why Version Control is Essential in MLOps**

1. **Reproducibility Crisis Prevention**
   * Without versioning:
     + Rolling back to previous data preprocessing/model versions is impossible
     + Changes causing degraded performance can't be traced or reversed
   * Example: Failed model training → Rollback to last working preprocessing code
2. **Collaboration Enabler**
   * Allows global teams to:
     + Work simultaneously on same project
     + Merge contributions without conflicts
     + Maintain independent experimentation branches
3. **Experiment Tracking**
   * Compare model versions, parameters, and data dependencies

**Core Git Concepts & Workflow**

**The 4-Stage Lifecycle**



* **Untracked Files**: New files not yet added to Git (U status)
* **Staging Area**: "Preview zone" before permanent commits
* **Local Repository**: Version history on your machine
* **Remote Repository**: Centralized cloud storage (GitHub/GitLab)

**Essential Git Commands**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| git init | Initialize new repo | git init project\_ml |
| git status | Show changed/untracked files | git status -s (short format) |
| git add | Stage changes | git add preprocessing.py |
| git commit | Save staged changes | git commit -m "Fix feature scaling" |
| git log | View commit history | git log --oneline --graph |
| git diff | Compare changes | git diff HEAD~1 (vs. last commit) |
| git checkout | Restore/switch versions | git checkout a3f8d2 (commit ID) |

**Branching Strategy for ML Projects**

**Why Branch?**

* Isolate experiments (new features/models)
* Protect main branch from unstable changes
* Enable parallel development

**Branch Operations:**

# Create branch from specific commit

git branch feature-engineering 89a4c1d

# List all branches

git branch -a

# Switch branches

git checkout feature-engineering

# View all commits across branches

git log --oneline --all --graph

**Merging & Conflict Resolution**

**Safe Merge Process:**

1. Switch to target branch:

git checkout main

1. Merge feature branch:

git merge feature-engineering

**Handling Conflicts:**

When same file is modified in both branches:

1. Git pauses merge, marks conflict areas:

<<<<<<< HEAD

# Code from current branch

scaler = StandardScaler()

=======

# Code from feature-engineering branch

scaler = RobustScaler()

>>>>>>> feature-engineering

1. Manually edit file to resolve conflict
2. Finalize with:

git add resolved\_file.py

git commit -m "Merge feature engineering"

**Remote Repository Workflow**

**Connecting & Syncing:**

# Add remote origin

git remote add origin https://github.com/user/ml-project.git

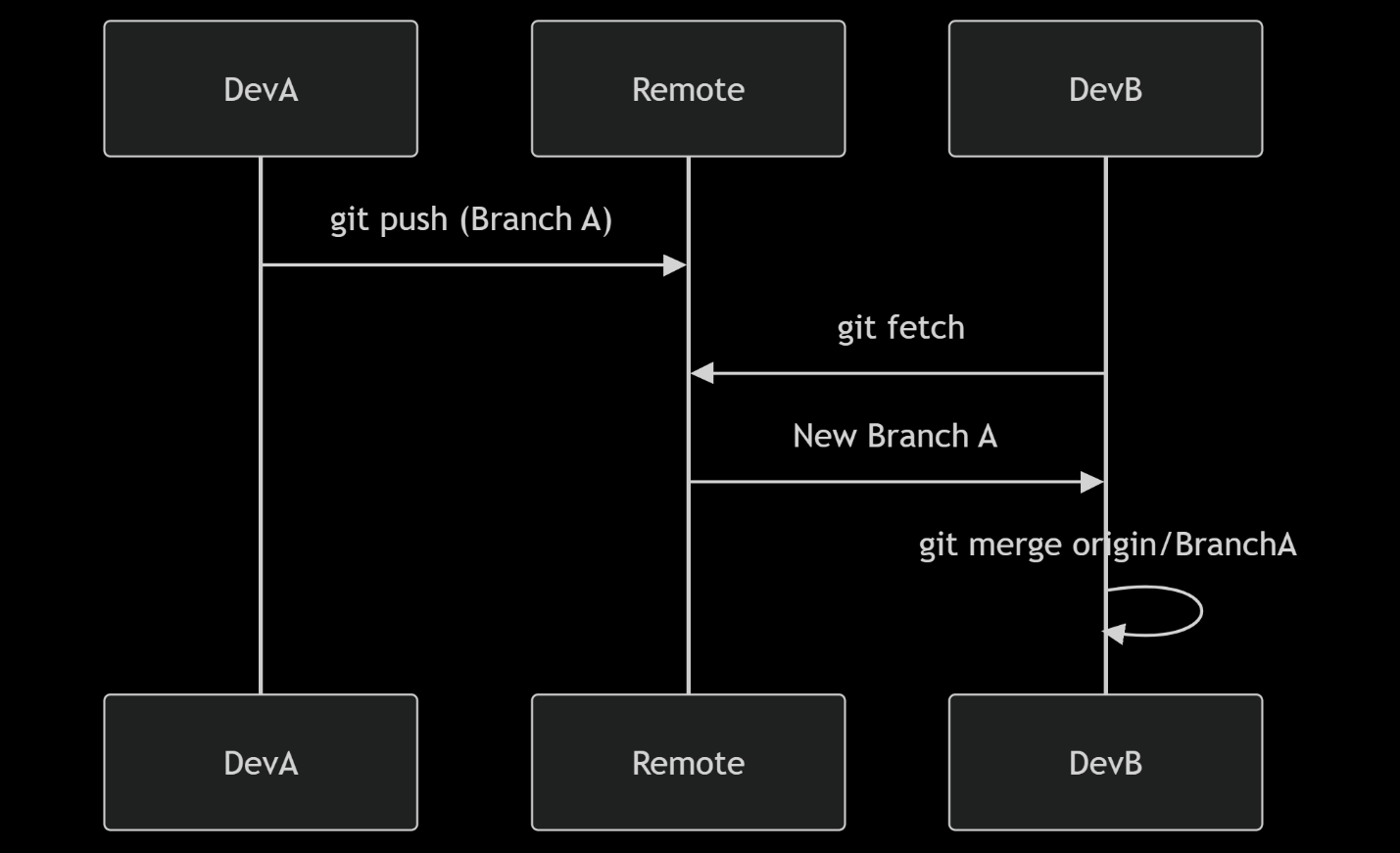
# First push to remote

git push -u origin main

# Subsequent pushes

git push

**Collaboration Pattern:**



**ML-Specific Versioning Best Practices**

1. **Data Versioning**:
   * Use .gitignore to exclude large datasets
   * Integrate DVC (Data Version Control) for data pipelines

dvc add dataset/

git add dataset.dvc .gitignore

git commit -m "Track dataset v1.2"

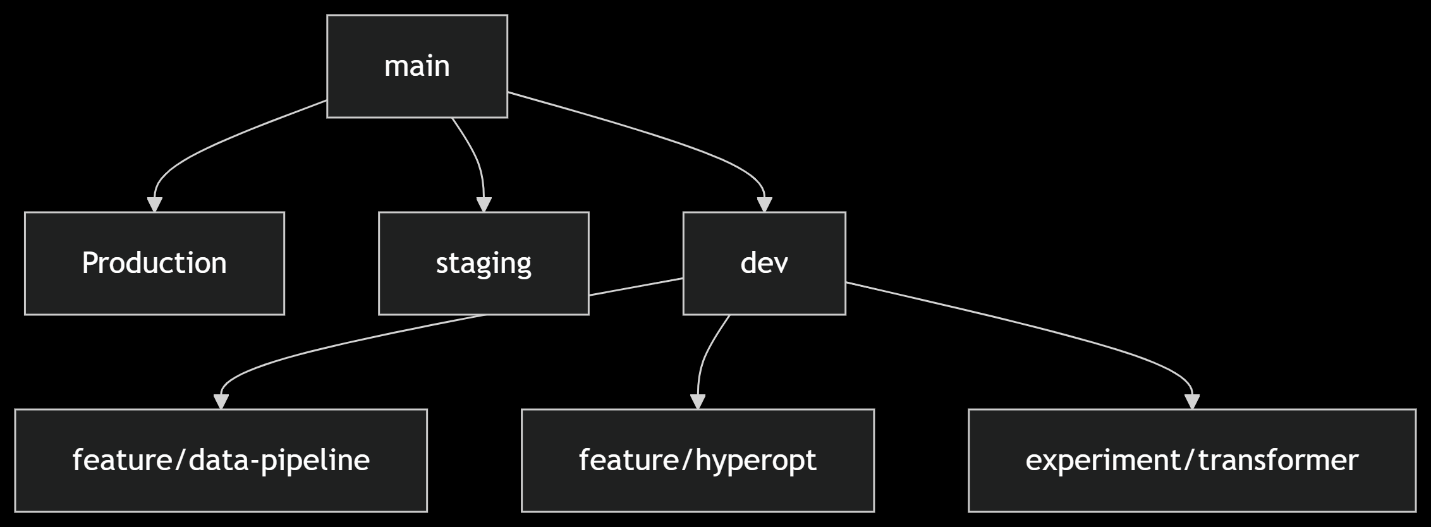
1. **Model Versioning**:
   * Store model binaries in cloud storage (S3/GCS)
   * Track metadata with MLflow:

import mlflow

mlflow.log\_model(model, "random\_forest\_v3")

1. **Config Versioning**:
   * Commit hyperparameter/config files (YAML/JSON)
   * Link configs to specific model versions

**Real-World MLOps Branching Strategy**



* **main**: Stable releases (production-ready)
* **staging**: Pre-production testing
* **dev**: Active development branch
* **feature/**: Isolated feature development
* **experiment/**: High-risk experimental work

**Advanced Techniques**

1. **Tagging Releases**:

git tag -a v1.3 -m "Model v1.3 with new features"

git push origin v1.3

1. **Rebasing vs Merging**:
   * Rebase: Clean linear history (for private branches)

git checkout feature

git rebase main

* + Merge: Preserves full history (for shared branches)

1. **Git Hooks for MLOps**:
   * Pre-commit: Run code formatting/tests
   * Post-merge: Trigger pipeline updates

**Common Pitfalls to Avoid**

1. Committing sensitive data (API keys/database credentials)
2. Pushing large files (>100MB) directly to Git
3. Long-lived branches causing merge nightmares
4. Undescriptive commit messages ("Fixed bug")

**Pro Tip**: Adopt Conventional Commits:  
feat: add feature scaling  
fix: resolve data leakage in preprocessing  
docs: update experiment methodology

**Learning Resources**

1. **Interactive Practice**: [Learn Git Branching](https://learngitbranching.js.org/)
2. **Cheat Sheet**: [GitHub Git Cheat Sheet](https://training.github.com/downloads/github-git-cheat-sheet/)
3. **ML Integration**: [DVC Documentation](https://dvc.org/doc)
4. **Collaboration Workflows**: [GitFlow](https://nvie.com/posts/a-successful-git-branching-model/" \t "_blank)

Mastering Git enables reproducible ML pipelines and seamless team collaboration - the foundation of successful MLOps implementations. Practice daily!