Git Deep Dive — Follow-Along Guide

Week 8 · CS 203 · Software Tools and Techniques for AI

Prof. Nipun Batra · IIT Gandhinagar

Spring 2026

How to Use This Guide

- Open `git_followalong.sh` in your editor (left half of screen)
- Open a terminal (right half of screen)
- Copy-paste each command, one at a time
- Compare your output with the expected output shown here
- **Type it yourself** — that's how you learn

Legend: `$` = command to type (don't type the `$`). `>>` = expected output. Blue boxes = look at the projector slide.

Projector: Slide 2 — "The Mess" (chaos image) Look at the projector. Recognize this? Five copies, no idea which is real.

Act 1: Life Without Git ~5 min

Create a project the way everyone starts:

```
$ mkdir ml-chaos && cd ml-chaos
```

Write a training script:

```
$ cat > train.py << 'EOF'
import numpy as np
from sklearn.linear_model import LogisticRegression

def train(X, y):
    model = LogisticRegression(max_iter=1000)
    model.fit(X, y)
    return model

print("Training model...")
EOF
```

Advisor wants changes. You're scared to break it:

```
$ cp train.py train_v2.py
$ cp train_v2.py train_v2_FINAL.py
$ cp train_v2_FINAL.py train_v2_FINAL_fixed.py
$ cp train_v2_FINAL_fixed.py train_v2_FINAL_fixed_actually_final.py
$ ls *.py
>> train.py  train_v2.py  train_v2_FINAL.py  ...  (5 files!)
```

Try to find what changed between v2 and FINAL:

```
$ diff train_v2.py train_v2_FINAL.py
```

Nothing! They're identical copies. You copied instead of editing. Now a teammate emails you *their* version:

```
$ cat > train_alice.py << 'EOF'
# ... Alice's version with different max_iter and a new function ...
EOF
$ ls *.py | wc -l
>> 6
```

Six files. Who has the right `max_iter`? How do you combine Alice's work with yours? **This is unsustainable.**

```
$ cd .. && rm -rf ml-chaos
```

Projector: Slides 3–4 — Letter Analogy + Three Areas Look at the projector. **edit** (write the letter) → **git add** (put in envelope) → **git commit** (mail it). Three areas: Working Directory → Staging Area → Repository.

Act 2: Starting Fresh With Git ~8 min

```
$ mkdir ml-project && cd ml-project
$ git init
>> Initialized empty Git repository in .../ml-project/.git/
```

`git init` creates a hidden `.git/` folder — that IS the repository.

```
$ ls -la
>> .git/     <-- this IS Git. Delete it and it's just a normal folder.
```

Configure your identity:

```
$ git config user.name "Your Name"
$ git config user.email "your.email@example.com"
```

Tip

Add `-global` to set this for ALL repos on your machine.

Create the same training script:

```
$ cat > train.py << 'EOF'
# ... same content as Act 1 ...
EOF
$ git status
>> Untracked files:   train.py   (RED)
```

Git sees it but isn't tracking it. Move it through the three areas:

```
$ git add train.py                  # Working Dir → Staging Area
$ git status
>> Changes to be committed:   train.py   (GREEN)
$ git commit -m "Add initial training script"   # Staging → Repository
>> [main (root-commit) abc1234] Add initial training script
$ git log --oneline
>> abc1234 (HEAD -> main) Add initial training script
```

One checkpoint. No copies. No _v2. The message explains *why*.

Prove the staging area matters — create two files, only stage one:

```
$ cat > utils.py << 'EOF'
import numpy as np
def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
EOF
$ cat > notes.txt << 'EOF'
TODO: ask advisor about learning rate (not ready to commit)
EOF
```

```
$ git add utils.py                  # stage ONLY utils.py
$ git status
>> Changes to be committed:     utils.py     (GREEN)
>> Untracked files:             notes.txt    (RED)
$ git commit -m "Add utils module with accuracy function"
```

Only `utils.py` was committed. `notes.txt` stayed out. **The staging area gives you fine-grained control.**

```
$ rm notes.txt
```

Projector: Slide 5 — "Acts 3–4: Changes & Time Travel" Brief recap slide on the projector. Then stay in terminal.

Act 3: Making Changes (No More Copies!) ~10 min

Advisor says "add an evaluation function." Old you: `cp train.py train_v2.py`. New you:

```
$ cat > train.py << 'EOF'
# ... add evaluate() function ...
EOF
$ git diff
>> +def evaluate(model, X, y):
>> +    acc = model.score(X, y)
>> +    print(f"Accuracy: {acc:.3f}")
>> -print("Training model...")
>> +print("Training and evaluating model...")
```

How to read diffs: `+` = added (green), `-` = removed (red), space = context.

```
$ git add train.py
$ git commit -m "Add evaluation function"
```

Add more files in separate commits:

```
$ cat > config.yaml << 'EOF'
model:
  type: logistic_regression
  max_iter: 1000
  learning_rate: 0.01
data:
  train_path: data/train.npy
  test_path: data/test.npy
EOF
$ git add config.yaml && git commit -m "Add model config file"

$ cat > utils.py << 'EOF'
# ... add confusion_matrix() and normalize() ...
EOF
$ git add utils.py && git commit -m "Add confusion matrix and normalization to utils"

$ cat > preprocess.py << 'EOF'
# ... preprocessing pipeline using normalize() ...
EOF
$ git add preprocess.py && git commit -m "Add preprocessing pipeline"
```

Check your history:

```
$ git log --oneline
>> f6f7g8h Add preprocessing pipeline
>> e5e6f7g Add confusion matrix and normalization to utils
>> d4d5e6f Add model config file
>> c3c4d5e Add evaluation function
```

```
>> b2b3c4d Add utils module with accuracy function
>> a1a2b3c Add initial training script
```

Six checkpoints. No copies. Each explains what changed and why.

Staged vs unstaged changes:

```
$ echo "# TODO: add validation" >> train.py
$ git add train.py
$ echo "# TODO: add early stopping" >> train.py
$ git diff --staged        # shows what's STAGED (validation TODO)
$ git diff                 # shows what's NOT staged (early stopping TODO)
$ git restore --staged train.py && git restore train.py   # clean up
```

Tip

The Rhythm: edit → `git status` → `git diff` → `git add` → `git commit`
`git diff -staged` = "What am I about to commit?"

Act 4: Going Back in Time ~8 min

Advisor: "What did the code look like before preprocessing?"

```
$ git show HEAD~1                          # what did the last commit change?
$ git show HEAD~5:train.py                 # train.py at the first commit
$ git diff HEAD~5 HEAD -- train.py         # all changes across 5 commits
```

Searching history:

```
$ git log --oneline --grep="preprocessing"    # search commit messages
$ git log -S "normalize"                      # find when code was added
$ git log --oneline -- utils.py               # history of one file
$ git blame train.py                          # who changed each line?
```

Time travel:

```
$ git checkout HEAD~5
>> You are in 'detached HEAD' state...
$ ls
>> train.py                  # only train.py exists at this point!
$ git checkout main           # return to the present
$ ls
>> config.yaml  preprocess.py  train.py  utils.py   # all back!
```

Set up an alias (you'll use this constantly):

```
$ git config --global alias.lg "log --oneline --graph --all --decorate"
$ git lg
```

Projector: Slide 6 — Undo Operations (reset modes diagram) Look at the projector. Three modes: `-soft` (keep staged), `-mixed` (keep unstaged), `-hard` (delete all).

Act 5: "I Messed Up" — Undo Operations ~8 min

Level 1 — Discard file changes (not staged):

```
$ echo "BAD LINE" >> train.py
$ git diff train.py                    # see the damage
$ git restore train.py                 # gone!
```

Warning

`git restore` permanently discards uncommitted changes. No undo for this undo!

Level 2 — Unstage a file:

```
$ echo "debug = True" >> utils.py
$ git add utils.py                     # oops, didn't mean to stage
$ git restore --staged utils.py        # unstaged (change still in file)
$ git restore utils.py                 # discard the change too
```

Level 3 — Undo a commit:

```
$ echo "# TODO: fix" >> train.py
$ git add train.py && git commit -m "Bad commit"
$ git reset --soft HEAD~1              # commit gone, changes still staged
$ git restore --staged train.py && git restore train.py   # clean up
```

Level 4 — Undo a pushed commit (safe):

```
$ echo "# debug" >> train.py
$ git add train.py && git commit -m "Accidental debug"
$ git revert HEAD --no-edit            # creates a new "undo" commit
>> Revert "Accidental debug"
$ cat train.py                         # debug line is gone
```

Stash — save work for later:

```
$ echo "# WIP" >> train.py
$ git stash                            # hide changes
$ git status                           # clean!
$ git stash pop                        # get them back
$ git restore train.py                 # clean up
```

Tip

`git stash` → switch branches → do your thing → switch back → `git stash pop`

Projector: Slides 7–8 — Parallel Universes + Branch Pointer Look at the projector. A branch is just a label pointing at a commit. Creating one is instant and costs nothing.

Act 6: Working on a Feature (Branching) ~8 min

Advisor: "Try augmentation, but don't break what we have."

```
$ git checkout -b feature/augmentation
>> Switched to a new branch 'feature/augmentation'
$ git branch
>>   main
>> * feature/augmentation
```

Build the feature (create `augment.py`, update `train.py`, commit). Then the magic:

```
$ git checkout main
$ ls *.py
>> preprocess.py  train.py  utils.py
```

`augment.py` vanished! It's safely on `feature/augmentation`. Git swapped your working directory.

```
$ cat train.py                          # old version! no augmentation
$ git checkout feature/augmentation
$ ls *.py
>> augment.py  preprocess.py  train.py  utils.py    # it's back!
$ git checkout main
```

Two parallel universes. One folder. Your advisor can demo the clean `main` while your experiment is safe on its branch.

Projector: Slide 9 — Diverging Branches Look at the projector. When both branches have new commits, Git creates a **merge commit** with two parents.

Act 7: Merging ~10 min

Fast-forward merge (main hasn't changed since you branched):

```
$ git merge feature/augmentation
>> Fast-forward
$ git branch -d feature/augmentation   # clean up the label
```

Three-way merge (both branches have new commits):

```
$ git checkout -b feature/evaluation
# ... create evaluate.py, commit ...
$ git checkout main
# ... create README.md, commit ...
$ git lg                                # branches have DIVERGED
$ git merge feature/evaluation -m "Merge evaluation module"
>> Merge made by the 'ort' strategy.
$ ls                                    # both files are here!
$ git branch -d feature/evaluation
```

Act 8: Merge Conflicts ~10 min

Two branches change the **same line** in `config.yaml`:

```
$ git checkout -b experiment/high-lr
# ... change max_iter to 2000, lr to 0.01, commit ...
$ git checkout main && git checkout -b experiment/low-lr
# ... change max_iter to 500, lr to 0.001, commit ...
$ git checkout main
$ git merge experiment/high-lr -m "Merge high-lr"     # fine
$ git merge experiment/low-lr                           # CONFLICT!
>> CONFLICT (content): Merge conflict in config.yaml
```

Open `config.yaml` — you'll see conflict markers:

```
<<<<<<< HEAD
  max_iter: 2000
  learning_rate: 0.01
=======
  max_iter: 500
  learning_rate: 0.001
>>>>>>> experiment/low-lr
```

Resolve: edit the file, remove all markers, pick values. Then:

```
$ git add config.yaml
$ git commit -m "Resolve conflict: compromise on hyperparameters"
```

Tip

Overwhelmed? `git merge -abort` cancels the merge.
For code conflicts: `git checkout -ours <file>` or `git checkout -theirs <file>` picks a whole side.

Act 9: .gitignore ~5 min

```
$ echo "SECRET_API_KEY=sk-abc123" > .env
$ dd if=/dev/zero bs=1024 count=100 of=model.pkl 2>/dev/null
$ mkdir -p __pycache__ data
$ git status
>> .env   __pycache__/   data/   model.pkl    (all RED!)
```

Create `.gitignore`:

```
__pycache__/
*.pyc
.env
*.pkl
*.h5
data/*.csv
data/*.parquet
.vscode/
.DS_Store
```

```
$ git status                          # only .gitignore shows up!
$ git add .gitignore && git commit -m "Add .gitignore for ML project"
```

Warning

If you already committed a secret, `.gitignore` won't remove it from history. **Rotate the key immediately.**

Projector: Slide 10 — Local ↔ Remote Look at the projector. Everything is local until you `push` or `pull`. Only three commands talk to the remote.

Act 10: Sharing Your Work (Remotes) ~10 min

```
$ cd ..
$ git init --bare fake-github.git      # simulate GitHub
$ cd ml-project
$ git remote add origin ../fake-github.git
$ git push -u origin main              # upload everything!
```

A collaborator clones, makes changes, pushes:

```
$ cd .. && git clone fake-github.git collaborator-clone
$ cd collaborator-clone
$ git config user.name "Alice"
# ... Alice edits README.md, commits, pushes ...
```

You pull their changes:

```
$ cd ../ml-project
$ git pull origin main
$ tail -5 README.md                    # Alice's changes are here!
```

What if you both push without pulling?

```
$ git push
>> ! [rejected] (fetch first)          # Git protects you!
$ git pull                             # get their changes first
$ git push                             # now it works
```

Tip

Golden rule: Always pull before you push.

Projector: Slide 11 — Quick Reference Leave this slide up while students ask questions.

Quick Reference

I want to...	Command
See what changed	`git status`
See exact changes	`git diff` / `git diff --staged`
Stage files	`git add <files>`
Save a checkpoint	`git commit -m "message"`
View history	`git lg` (after alias)
Create a branch	`git checkout -b <name>`
Switch branches	`git checkout <name>`
Merge a branch	`git merge <branch>`
Delete merged branch	`git branch -d <name>`
Discard file changes	`git restore <file>`
Unstage a file	`git restore --staged <file>`
Undo last commit	`git reset --soft HEAD~1`
Undo pushed commit	`git revert <hash>`
Stash WIP	`git stash` / `git stash pop`
Push to remote	`git push`
Pull from remote	`git pull`
Who changed this?	`git blame <file>`
Search messages	`git log --grep="text"`
Search code	`git log -S "code"`
File history	`git log -- <file>`
Cancel a merge	`git merge --abort`
Take our side	`git checkout --ours <file>`
Take their side	`git checkout --theirs <file>`

Commit message rule: Complete *"If applied, this commit will _____."*

- **Good:** "Add data augmentation for training images"
- **Good:** "Fix off-by-one error in batch loader"
- **Bad:** "fix", "stuff", "update", "asdfgh"

Best Practices:

1. Commit early, commit often
2. Write meaningful commit messages
3. Never commit secrets (`.env`, API keys) — add to `.gitignore`
4. Never commit large files (models, data) — use DVC or Git LFS
5. Branch for every experiment
6. Always `git status` before `git add`
7. Pull before you push
8. Delete branches after merging