Git

CMSC398W: Practical Tools For Efficient Development

Mohammad Durrani

June 22, 2026

Goals for Today

Objectives

- Model Git as a graph of snapshots.
- Use the three-stage workflow: Working Directory $\rightarrow$ Staging $\rightarrow$ Repository.
- See what a commit contains.
- View history, diff changes, and undo mistakes.
- Sync with remotes using push and pull.
- Manage branches and resolve merge conflicts.

Project 1 Overview

Overview

- Group project with 3-4 people
- Each given an individual branch to work off of and tasks to complete, write a mini-report, and then combine them together as a group
- Released (hopefully) Saturday, see Piazza for details

In Class

- Groups are due by the end of class (see Piazza post)
- Take 5 minutes to either find a group or exchange contact information with your group to make your life easier

What is Git?

Definition

- Git is a open-source distributed version control system (VCS) designed to track changes in files and source code during software development.
- Git != GitHub

History

- Created by Linus Torvalds in 2005 for Linux kernel development.
- Built for speed and data integrity in a distributed environment.

What is Git?

Definition

- Git is a open-source distributed version control system (VCS) designed to track changes in files and source code during software development.
- Git != GitHub

History

- Created by Linus Torvalds in 2005 for Linux kernel development.
- Built for speed and data integrity in a distributed environment.

Core Philosophy

Unlike SVN or CVS, Git gives every developer a complete copy of the entire history. It is a distributed version control system.

Why Git?

The Problem

- "It worked yesterday. What changed?"
- "I deleted that function and now I need it back."
- "My teammate's changes broke my code."

Why Git?

The Problem

- "It worked yesterday. What changed?"
- "I deleted that function and now I need it back."
- "My teammate's changes broke my code."

The Alternative (Don't do this!)

- `project_v1.zip`, `project_final.zip`, `project_FINAL_FINAL.zip`
- Emailing code to yourself.

Git replaces this with a snapshot of your project at every commit.

Two Ways to Start

`git init`

- Creates a hidden `.git/` folder.
- Use this to start tracking a new project from scratch.
- All history is stored locally in your folder.

`git clone <url>`

- Downloads a full copy of the repo and its entire history
- Automatically sets up `origin` pointing back to the source
- Use when joining an existing project

The Three Areas

The Three Areas

- Working Directory: Where you edit files. Changes here are not tracked until staged.
- Staging Area (Index): Intermediate area where you can decide what goes into the next commit.
- Repository: The permanent record, stored in `.git/`.

Discussion: The Staging Area

Question (Discuss at your tables)

Why doesn't Git just commit everything the moment you save a file? Why do we have to run `git add` first?

Discussion: The Staging Area

Question (Discuss at your tables)

Why doesn't Git just commit everything the moment you save a file? Why do we have to run `git add` first?

Answer

The staging area lets you group related changes together. You can fix five different things but commit them as five separate, logical units.

Workflow Example

Workflow Example

```
$ git status
On branch main
Changes not staged for commit:
  modified:   app.py

$ git add app.py
$ git status
On branch main
Changes to be committed:
  modified:   app.py

$ git commit -m "Fix login bug"
[main a1b2c3d] Fix login bug
 1 file changed, 1 insertion(+)
```

Think-Pair-Share: What Gets Committed?

The Setup

You edit two files: `app.py` and `README.md`. You run `git add app.py` then `git commit -m "fix login bug"`.

Question

What is inside the new commit?

Think-Pair-Share: What Gets Committed?

The Setup

You edit two files: `app.py` and `README.md`. You run `git add app.py` then `git commit -m "fix login bug"`.

Question

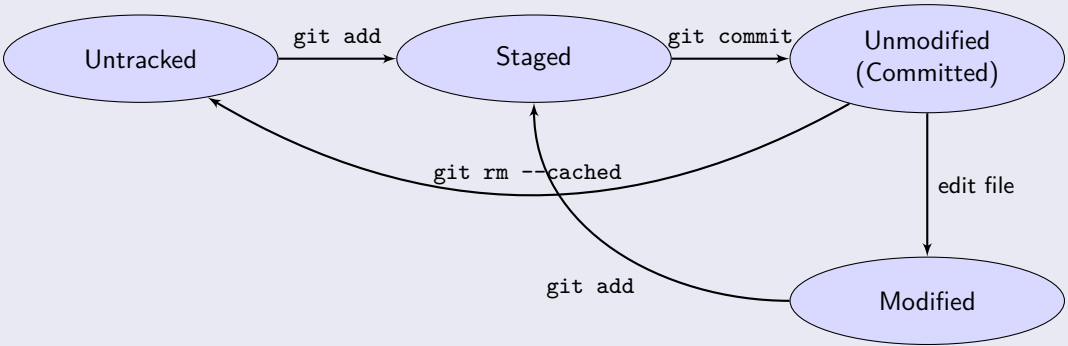
What is inside the new commit?

Answer

Only `app.py` is committed. Changes to `README.md` remain in your Working Directory. You must explicitly stage files to include them in a commit.

The Lifecycle of a File

Status Lifecycle

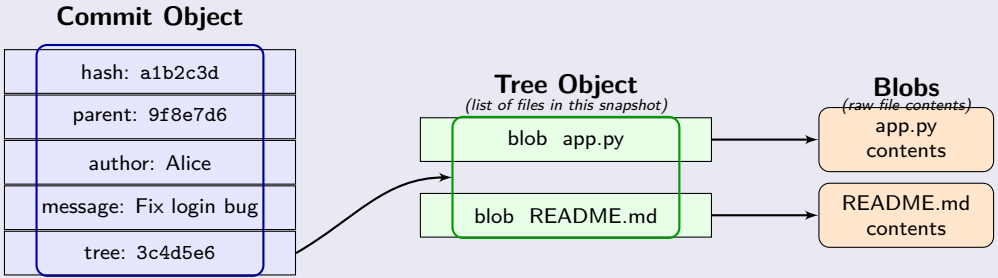


A new file starts as Untracked. Once committed, it cycles: Unmodified → Modified → Staged → Unmodified.

Anatomy of a Commit

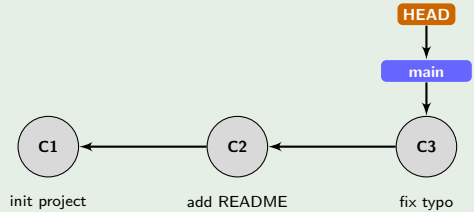
Commit Object

Git stores full snapshots, not diffs. Each commit is an object in the `.git/` database.



Commit Graph

The Graph



Note

Every commit points to its parent. `HEAD` marks your current position. The graph grows forward; you walk backward through history.

Inspecting Your State

Commands

Command	What It Shows
`git log`	Full commit history
`git log --oneline --graph --all`	Compact graph view of all branches
`git diff`	Changes not yet staged
`git diff --staged`	Changes staged for the next commit
`git show HEAD`	Full detail of the last commit

Who Changed This?

git blame <file>

Shows which user modified each line of a file, when they did it, and in which commit.

- Excellent for finding the context behind a line of code.
- Combine with `git show <hash>` to see the full change.

Revert: Safe Undo

git revert <hash>

Creates a new commit that is the inverse of the target commit. History is never rewritten, making it safe for shared branches.

Reset: Undo Local Commits

git reset

Moves the branch pointer to an earlier commit. Rewrites history; only use on unpushed commits.

Mode	Command	Side Effect (all will undo commits)
Soft	`git reset --soft HEAD~1`	Changes stay staged
Mixed (default)	`git reset HEAD~1`	Changes go to working directory
Hard	`git reset --hard HEAD~1`	Changes are discarded

Relative Refs

`HEAD~N` means N commits behind HEAD:

Ref	Meaning
`HEAD~1` or `HEAD~`	One commit back (parent)
`HEAD~2`	Two commits back (grandparent)

The Reflog

Using the Reflog

Git records every time HEAD moves, including destructive operations.

```
# Destroyed work with a hard reset
$ git reset --hard HEAD~2

# Find the lost commit
$ git reflog
a1b2c3d HEAD@{2}: commit: Add payment feature  <- this one!

# Restore it
$ git reset --hard a1b2c3d
```

Discussion: Reset or Revert?

Scenario

You pushed a commit to `main` that breaks the production server. Your team has already pulled it.

Question (Discuss at your tables)

Do you use `git reset --hard` or `git revert`? What happens to your teammates' history if you pick the wrong one?

Discussion: Reset or Revert?

Scenario

You pushed a commit to `main` that breaks the production server. Your team has already pulled it.

Question (Discuss at your tables)

Do you use `git reset --hard` or `git revert`? What happens to your teammates' history if you pick the wrong one?

Answer

Use `git revert`. It creates a new "undo" commit that others can pull safely. Resetting rewrites history, which would cause your teammates' local repos to diverge and break their next pull.

Stashing: Save for Later

The Scenario

You're in the middle of a feature, but need to fix a bug on `main`. You don't want to commit half-broken work.

Commands

Command	Action
`git stash`	Saves all uncommitted changes and clears your WD
`git stash pop`	Restores stashed work and removes it from the stack
`git stash list`	View all stashed items
`git stash apply`	Restores stashed work but keeps it in the stack

Remotes: Another Copy of the Repo

Overview

A remote is another copy of the repository, on GitHub's servers or a teammate's machine. The graphs stay in sync by pushing and pulling commits.

Term	Meaning
`origin`	Default nickname for the remote URL
`upstream`	The original repo you forked from
`origin/main`	Local bookmark of where the remote's branch last was

Managing Remotes

Commands

Command	Action
`git remote add <name> <url>`	Connect local repo to a new server
`git remote -v`	List all configured remotes
`git remote remove <name>`	Delete a remote connection

- `origin` is just the default name for the first remote.
- You can have multiple remotes (e.g., `origin`, `upstream`).

Push, Fetch, Pull

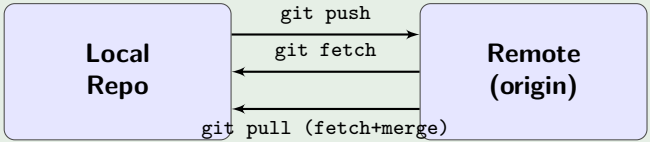
Commands

Command	What It Does
`git push <remote> <branch>`	Upload commits to a specific server/branch
`git push -u origin <branch>`	First push; links local branch to remote
`git fetch <remote>`	Download new commits; does not touch working files
`git pull <remote> <branch>`	`fetch` + `merge` in one step

- Plain `git pull` only works if a branch is "tracking" a remote (set with `-u`).
- Use `git fetch` to see what changed before merging.

Visualizing Push and Pull

Sync Flow



origin/main = local bookmark of remote state

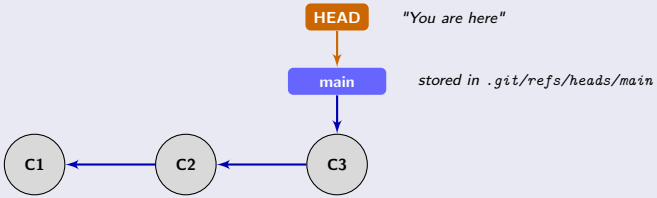
Note on -u

Without `-u` on the first push, Git errors: "The current branch has no upstream branch." Use `git push -u origin <branch>` once; after that, plain `git push` works.

Branches Are Pointers

What Is a Branch?

A branch is a pointer to a commit. Creating one copies no files. `HEAD` points to the current branch, which points to the tip commit.



Discussion: Branching Mental Model

Question (Discuss at your tables)

If a branch is just a pointer to a commit, what actually happens to the files on your hard drive when you run `git switch`? Does Git create a new folder for every branch?

Discussion: Branching Mental Model

Question (Discuss at your tables)

If a branch is just a pointer to a commit, what actually happens to the files on your hard drive when you run `git switch`? Does Git create a new folder for every branch?

Answer

Git updates the files in your Working Directory in-place to match the snapshot of the target commit. It does not create new folders; it swaps the contents of your existing folder based on where HEAD is pointing.

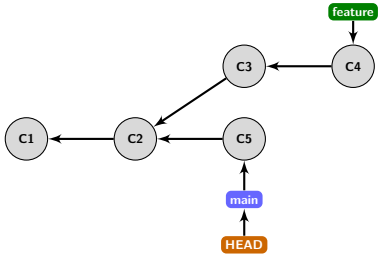
Why Branch?

- Keep work off `main` until it's done
- Run multiple features in parallel
- Each Pull Request is one branch

Visualizing Branches

```
$ git switch -c feature
# Make commits C3, C4
$ git switch main
# Make commit C5

$ git log --oneline --graph --all
  * a1b2c3d (HEAD -> main) C5
  | * e5f6g7h (feature) C4
  | * i8j9k0l C3
  |/
  * m1n2o3p C2
  * q4r5s6t C1
```



Branch Commands

Branch Commands

Command	Action
`git branch <name>`	Create a new pointer at HEAD
`git switch <name>`	Move HEAD to that branch
`git switch -c <name>`	Create and switch in one step
`git branch -d <name>`	Delete a merged branch pointer
`git branch -D <name>`	Force-delete even if unmerged
`git branch -v`	List branches with their latest commit
`git push origin --delete <name>`	Delete a branch on the remote

Branch Commands

Branch Commands

Command	Action
`git branch <name>`	Create a new pointer at HEAD
`git switch <name>`	Move HEAD to that branch
`git switch -c <name>`	Create and switch in one step
`git branch -d <name>`	Delete a merged branch pointer
`git branch -D <name>`	Force-delete even if unmerged
`git branch -v`	List branches with their latest commit
`git push origin --delete <name>`	Delete a branch on the remote

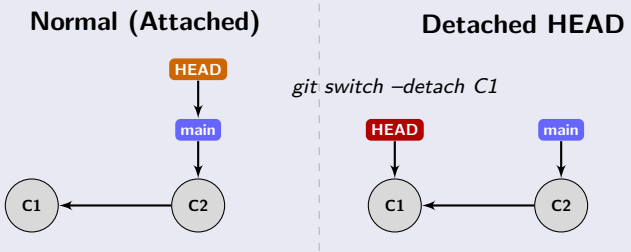
Note on `git checkout`

Older tutorials use `git checkout -b <name>`, which does the same thing. `git switch` was added in Git 2.23 to make this explicit.

Detached HEAD

Detached HEAD State

`git switch --detach <hash>` puts you in detached HEAD state: `HEAD` points directly to a commit, not a branch name.



Discussion: Detached HEAD Rescue

Scenario

You used `git switch --detach` to look at an old version of the code. You found a bug, fixed it, and made a commit. Then, you ran `git switch main` to go back to work. Your fix is now "invisible" because it has no branch label.

Question (Discuss at your tables)

How do you find that commit hash to save your work?

Discussion: Detached HEAD Rescue

Scenario

You used `git switch --detach` to look at an old version of the code. You found a bug, fixed it, and made a commit. Then, you ran `git switch main` to go back to work. Your fix is now "invisible" because it has no branch label.

Question (Discuss at your tables)

How do you find that commit hash to save your work?

Answer

Use `git reflog`. It records every movement of HEAD, including commits made in a detached state. Find the hash in the log and run `git switch -c fix-branch <hash>`.

Fast-Forward Merge

Concept

If `main` hasn't moved since you branched off, Git just moves the pointer forward. No merge commit.

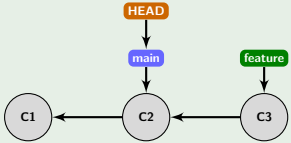
Fast-Forward Merge

Concept

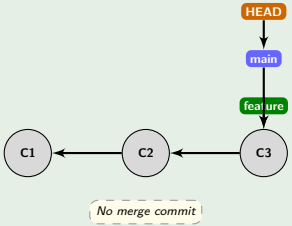
If `main` hasn't moved since you branched off, Git just moves the pointer forward. No merge commit.

Result

Before



After `git merge feature`



Merge Commit

Diverged History

When both branches have diverged, Git ties them together with a merge commit that has two parents.

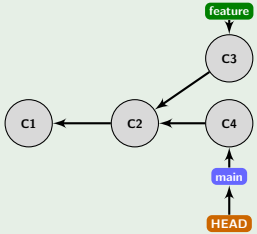
Merge Commit

Diverged History

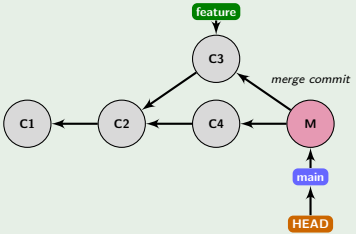
When both branches have diverged, Git ties them together with a merge commit that has two parents.

Result

Before



After `git merge feature`



Conflict Resolution

When Conflicts Happen

A conflict happens when both branches changed the same line. Git can't pick for you.

1. Git stops and writes conflict markers into the file.
2. Open the file, delete the markers, keep what you want.
3. `git add <file>` to mark it resolved.
4. `git commit` to finish.

Changed your mind? `git merge --abort` puts everything back.

Conflict Markers Resolved

Before: Conflict Markers

Git writes markers showing both sides. `HEAD` is your branch:

```
<<<<< HEAD
return total * 1.08
=======
return total * 1.10
>>>>> feature/tax-update
```

Conflict Markers Resolved

Before: Conflict Markers

Git writes markers showing both sides. `HEAD` is your branch:

```
<<<<< HEAD
return total * 1.08
=======
return total * 1.10
>>>>> feature/tax-update
```

After: Delete Markers, Keep What You Want

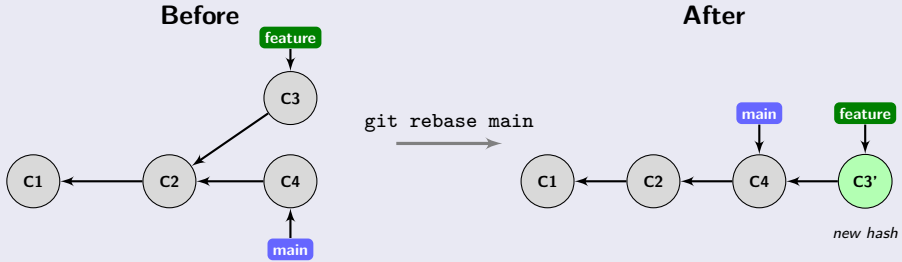
```
return total * 1.10
```

Then: `git add <file>` → `git commit`. VSCode: click *Resolve in Merge Editor* to avoid editing by hand.

Rebase: Replaying on a New Base

The Idea

Rebase takes your commits and replays them on top of another branch. The result is a straight-line history with no merge commit.




```
# While on your feature branch:
git rebase main
```

Each replayed commit gets a new hash. Same changes, different parent.

Discussion: Merge vs. Rebase Philosophy

Question (Discuss at your tables)

Rebasing creates a "clean" linear history, while merging preserves the reality of when work actually overlapped. In a workplace setting, which is more important: a readable history or a more accurate audit trail?

Discussion: Merge vs. Rebase Philosophy

Question (Discuss at your tables)

Rebasing creates a "clean" linear history, while merging preserves the reality of when work actually overlapped. In a workplace setting, which is more important: a readable history or a more accurate audit trail?

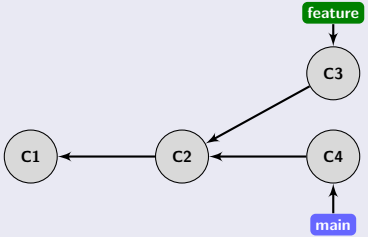
Answer

It depends on team policy!

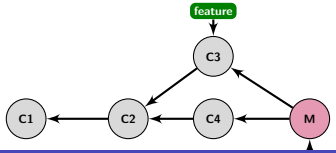
- Rebase: Easier to read/debug (`git bisect`), but "lies" about when work was done.
- Merge: Clearer record of parallel work and integration points, but can lead to messy history.

Merge vs. Rebase

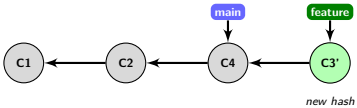
Before: Diverged History




```
$ git switch main
$ git merge feature
```




```
$ git switch feature
$ git rebase main
```



Interactive Rebase: Clean History

git rebase -i <base>

Opens your `$EDITOR` with the last N commits listed. Change the keyword on each line, save, and quit.

```
git rebase -i HEAD~5   # open the last 5 commits
```

Vim tip: `i` to edit, `Esc :wq` to save. To use nano instead: `git config --global core.editor "nano"`.

Common Commands

Keyword	Action
`pick`	Keep the commit as is
`reword`	Edit the commit message
`squash`	Combine with commit above
`drop`	Delete the commit entirely

```
pick  a1b2c3d add login
squash d4e5f6g fix typo
reword k0l1m2n add payment
drop  n3o4p5q debug print
```

Discussion: Squash for PRs

Scenario

Your branch has 5 commits: `"fix"`, `"typo"`, `"asdf"`, `"wip"`, `"finally"`. You need to open a PR.

Question (Discuss at your tables)

How do you squash these into one commit? What command and what keyword?

Discussion: Squash for PRs

Scenario

Your branch has 5 commits: `"fix"`, `"typo"`, `"asdf"`, `"wip"`, `"finally"`. You need to open a PR.

Question (Discuss at your tables)

How do you squash these into one commit? What command and what keyword?

Answer

`git rebase -i HEAD~5`. First line: `pick`. Rest: `squash`. Git folds them into one.

Rebase Warning

The Rule

Never rebase commits that have been pushed.

Rebasing rewrites hashes. If a teammate pulled your branch, their history diverges from yours and `git pull` breaks for them.

Rebase Warning

The Rule

Never rebase commits that have been pushed.

Rebasing rewrites hashes. If a teammate pulled your branch, their history diverges from yours and `git pull` breaks for them.

When to Use

Safe on branches that only exist locally.

Discussion: Parallel Universe Conflict

Scenario

Alice and Bob both branch off `main` at the same time. Alice finishes her `UI-fix` and merges it into `main`. Bob is still working on his `db-refactor`.

Question (Discuss at your tables)

1. Does Bob have Alice's UI changes yet?
2. If Bob needs Alice's changes to continue his work, what are his two options to get them?
3. Which option should he pick if he wants a clean, linear history?

Discussion: Parallel Universe Conflict

Scenario

Alice and Bob both branch off `main` at the same time. Alice finishes her `UI-fix` and merges it into `main`. Bob is still working on his `db-refactor`.

Question (Discuss at your tables)

1. Does Bob have Alice's UI changes yet?
2. If Bob needs Alice's changes to continue his work, what are his two options to get them?
3. Which option should he pick if he wants a clean, linear history?

Answer

1. No, Bob's branch is isolated.
2. Bob can either `git merge main` into his branch OR `git rebase main`.
3. He should `rebase` on top of `main`.

Cherry-Pick

The Command

Copies one specific commit from another branch and applies it to your current branch.

```
git cherry-pick <hash>
```

Creates a new commit with the same changes but a different hash.

Cherry-Pick

The Command

Copies one specific commit from another branch and applies it to your current branch.

```
git cherry-pick <hash>
```

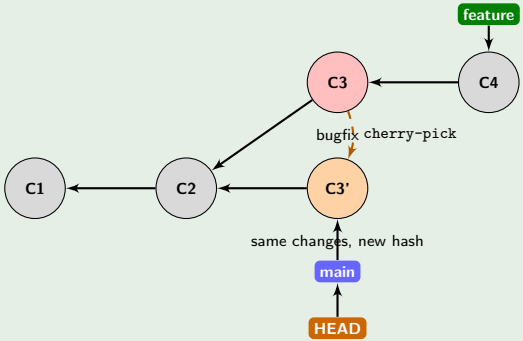
Creates a new commit with the same changes but a different hash.

When to Use It

Scenario	Why Cherry-Pick?
Hotfix on `main` needed on `release`	Avoid merging all of `main`
One useful commit from an abandoned branch	No need to revive the whole branch
Backport a fix to an older version	Target exactly one change

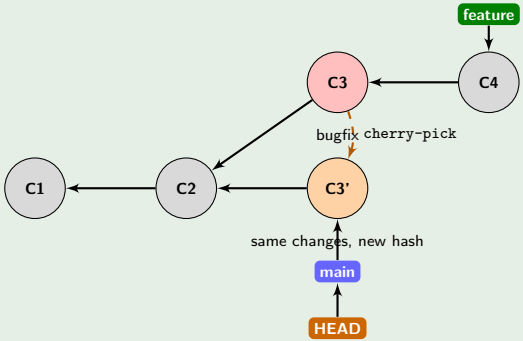
Visualizing Cherry-Pick

The Graph



Visualizing Cherry-Pick

The Graph



Key Detail

C3 and C3' contain the same changes but are independent commits with different hashes.