Git Workshop

Advanced Programming

Dr Mojtaba Vahidi Asl

Ghaem Aliabadi - @Ghaem

Fall 2025

AP FALL 2025

Part 1

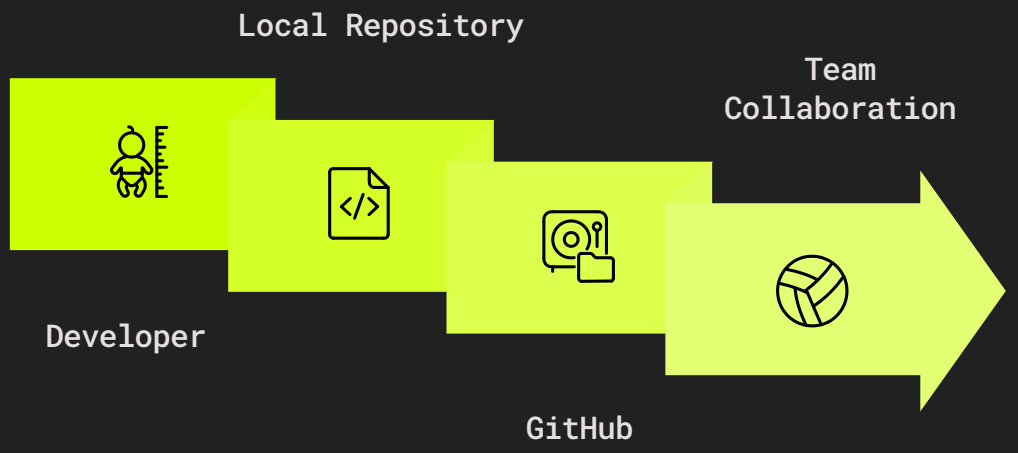
A comprehensive guide to understanding Git, the distributed version control system that revolutionized code collaboration and management.

Why Do We Need Git?

Modern software projects involve many developers working on the same codebase.

- Without version control, changes easily get lost or overwritten.
- Git helps you track every change, revert mistakes, and collaborate safely.
- You can experiment freely on branches without breaking the main project.

In short, Git makes teamwork, traceability, and rollback possible.



Introduction

Git = "Undo on steroids"

Enables code sharing and collaboration

Work together seamlessly with teams across the globe

Allows merging of different versions of code

Combine work from multiple developers effortlessly

Created by Linus Torvalds in April 2005

Born from the need for better Linux kernel development

Command-line version control tool

Distributed system (not centralized)

Cross-platform and open source

Distributed Version Control

No need for a central server

Every developer has a complete copy of the repository

Work offline (no internet required)

Commit, branch, and manage your code anywhere

No single point of failure

If one server goes down, work continues uninterrupted

Developers can work independently and merge later

Parallel development without stepping on each other's toes

Every repo copy can act as both server and client

Git tracks changes, not versions

Branches are collections of small change sets

What Is a Repository?

A repository (repo) is the storage space where your project's files and all change history are kept.

It can be:

Local repository

stored on your computer

Remote repository

stored on a platform like GitHub or GitLab

Every repo contains:

- A .git folder – internal database for commits, branches, and history
- Your project files (working directory)

Tip: Think of a repo as a timeline of your project.

Installation

Download from: **https://git-scm.com/downloads**

Commands:

```
git --version
git config --global user.name "Ghaem"
git config --global user.email "me@ghaem.me"
```

Repository (Repo) Basics

Initialize a repository:

```
git init
```

Change default branch name:

```
git config --global init.defaultBranch main
```

Check repo status:

```
git status
git status --short
```

List configs:

```
git config --list
git config user.name
```

Git Concepts

Each repo = collection of commits (snapshots)

HEAD points to the latest commit ("current version")

Git compares current changes to the HEAD commit

File States

Unchanged (Committed)

Untracked

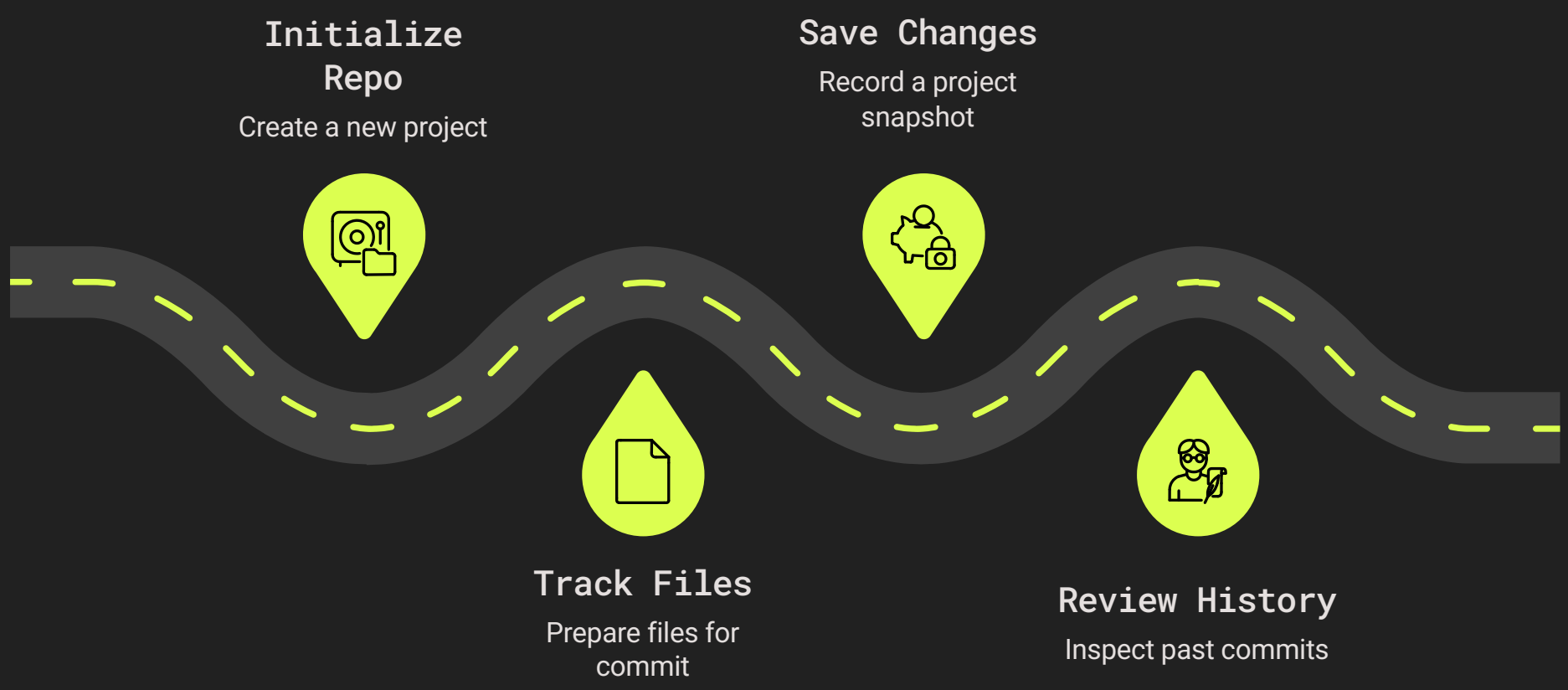
Modified

Deleted

AP FALL 2025

Git Workflow Overview (Step by Step)

Understanding the core Git workflow is essential for effective version control. This process typically involves these key stages:



AP FALL 2025

Git Commands Reference

01

Initialize a repo

`git init` → creates a new repository

02

Track files

`git add <file>` → moves files to the "staging area"

03

Save changes

`git commit -m "message"` → records a snapshot

04

Review history

`git log` → view all commits

05

Branch and merge

`git branch, git merge` → manage parallel work

06

Push to remote

`git push` → share changes with others

AP FALL 2025

Moving Files Between States

```
git add                 # Stage a file
git add .               # Stage all modified/untracked files
git add --all            # Stage all files
git commit -m "message"     # Commit staged changes
git commit               # Commit with editor message
git commit -a -m "message"   # Add + Commit all tracked files
```

What Is a Log (git log)?

A log is a chronological list of commits in a repository.

Each entry shows:

- Commit hash (unique ID)
- Author name and email
- Date and time
- Commit message

Commands:

```
git log       # full detailed history
git log --oneline  # compact summary
```

Use logs to track changes, debug issues, and understand who changed what.

Setting Commit Message Editor

You can configure which editor Git uses for commit messages:

```
git config --global core.editor "nano"
git config --global core.editor "code --wait"
git config core.editor
```

Popular editor options:

`nano`

Simple terminal editor

`code --wait`

Visual Studio Code

`vim`

Advanced terminal editor

Navigation tip: Use **Enter** to scroll in git log, **Q** to quit

Undoing Changes

Unstage a file:

```
git restore --staged
```

Restore modified/deleted file:

```
git restore
```

Restore both staging and working dir:

```
git restore --staged --worktree
```

Tags

Tags mark specific points in your repository's history (like releases).

Types:

Annotated

includes author, date, message

Lightweight

only points to a commit

Tag Commands

Essential commands for working with tags:

Creating tags:

```
git tag v1.0                # lightweight tag
git tag -a v1.0 -m "Release 1.0"  # annotated tag
```

Managing tags:

```
git tag              # list all tags
git show v1.0        # show tag details
git tag -d v1.0      # delete local tag
```

Sharing tags:

```
git push origin v1.0     # push specific tag
git push origin --tags   # push all tags
git push origin --delete tag v1.0  # delete remote tag
```

Undoing Commits

Undo changes made in a specific commit:

```
git revert <commit-hash>
```

→ Creates a new commit that reverses previous changes.

This is the **safe** way to undo commits because it preserves history and doesn't break collaboration with others.

Git Ignore

Create .gitignore file:

```
touch .gitignore
```

Common rules:

```
*.txt      # ignore all txt files
hello/     # ignore hello folder everywhere
/hello/    # ignore hello folder in root only
```

Wildcards supported (*, !)

Place .gitignore in subfolders for specific rules

Reference: w3schools.com/git/git_ignore.asp

Part 2

Branches

- Git stores snapshots of files at each commit
- Git commits form a DAG (Directed Acyclic Graph)
- Branches, Tags, and HEAD are just pointers to commits
- The default branch (master or main) is created after the first commit

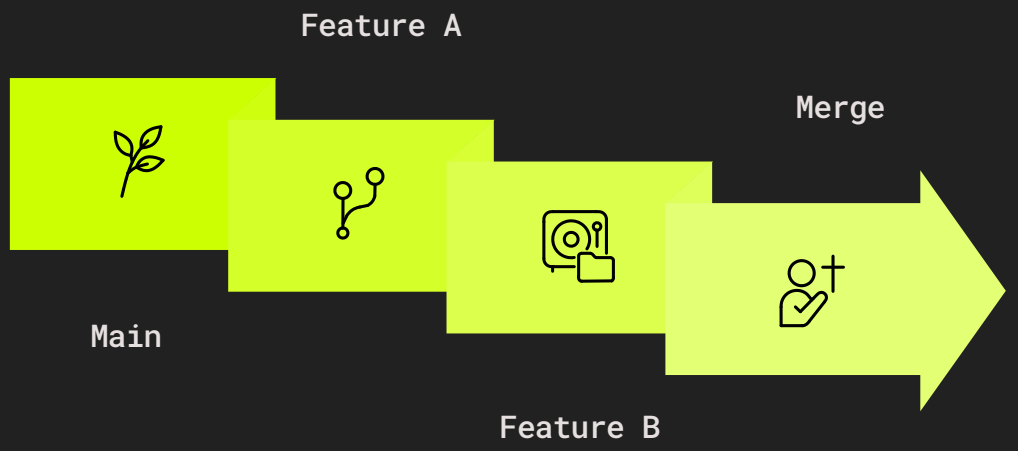
---

Branch Commands

```
git branch              # list all branches
git branch      # create new branch
git checkout -b  # create + switch to branch
git branch -d   # delete branch
git log --all           # show all commits and branches
```

AP FALL 2025

Understanding Branches (Detailed)



01

A branch is a pointer to a specific commit.

02

Used to develop features, fix bugs, or experiment independently.

03

Each branch has its own timeline and history.

04

Main (or master) is the default branch.

05

Merge branches to integrate changes into main.

Branch Commands

Essential commands for working with branches:

```
git branch new-feature      # create new branch
git checkout new-feature    # switch to branch
git checkout -b new-feature # create and switch in one command
git merge main              # merge main into current branch
git branch -d feature-name  # delete branch (safe)
git branch -D feature-name  # force delete branch
```

💡 **Tip:** Think of branches as parallel universes of your code - you can experiment freely without affecting the main project.

Reset vs Checkout

`git reset`

Moves the branch pointer

Updates the branch reference

`git checkout`

Moves the HEAD pointer

Changes the current working commit

Understanding Reset vs Rebase (in depth)

Reset moves the branch pointer backward or forward in history.

```
git reset --hard <commit>   # deletes commits after <commit>
```

Use when you want to discard unwanted commits.

AP FALL 2025

Understanding Rebase

Rebase rewrites history by replaying commits on top of another branch.

```
git rebase main    # moves your branch to the latest main
```

Benefits:

- Makes history cleaner and linear
- Eliminates unnecessary merge commits
- Creates a more readable project timeline

Main differences:

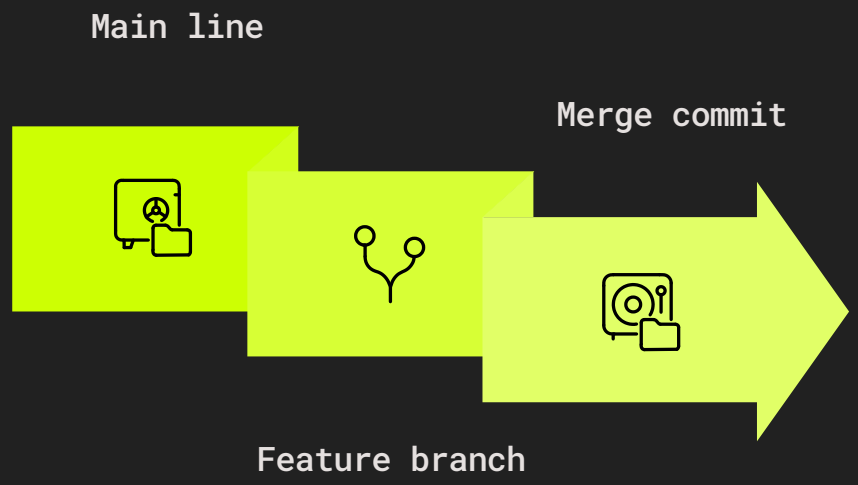
- **Reset** changes where your branch points
- **Rebase** changes the commit history structure itself

Merge vs Rebase Visualization

Understanding the difference between merge and rebase through visual examples:

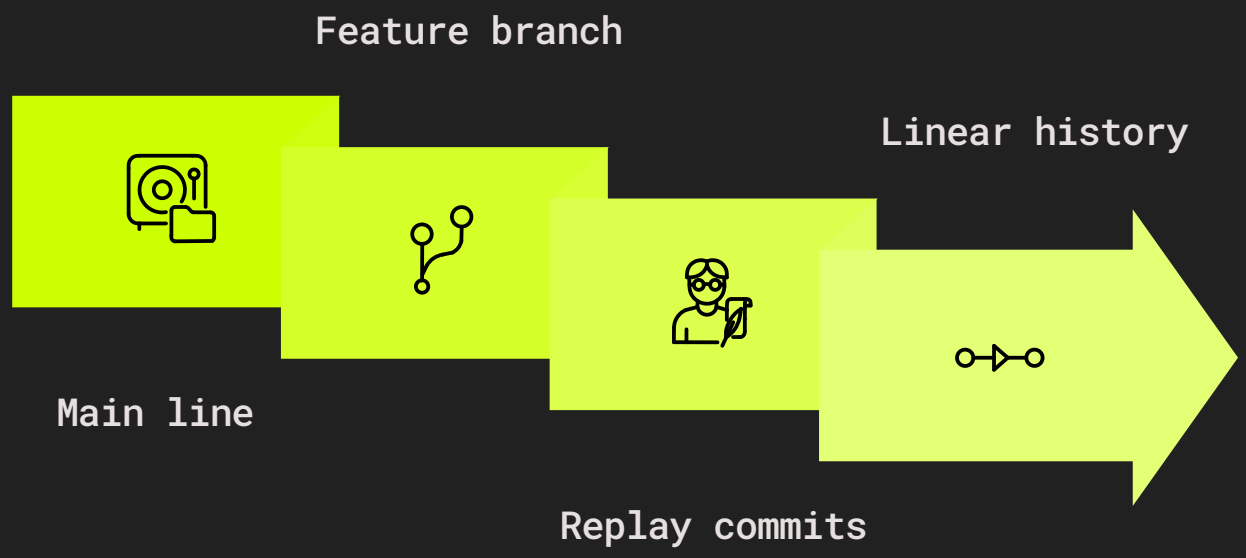
Standard Merge (diverging history):

Shows how branches diverge and then merge back together, creating a non-linear history with merge commits.



Rebase (linear history):

Shows how commits are replayed on top of the main branch, creating a clean, linear progression without merge commits.



Merging

```
git merge
```

Types:

01

Fast-forward merge

→ branch head simply moves forward

02

Three-headed merge

→ combines two different histories

Example:

- If Branch B2 commits are descendants of Branch B1:

```
git merge B2
```

→ Fast-forward occurs

Merge Conflicts

Conflicts happen when both branches modify overlapping content.

Git is smart – conflicts are not based on line numbers only.

If conflict resolution goes wrong:

```
git merge --abort
```

Important notes:

- Conflicts can also appear during pull operations
- Always resolve, test, and then commit the merge
- Take time to understand what each change does before deciding

What Are Conflicts?

A conflict happens when Git cannot automatically merge two versions of a file.

Example:

- Two people edit the same line in the same file
- Git doesn't know whose change to keep

To resolve:

1. Manually edit the file
2. Keep the correct version
3. Stage and commit again

Conflict markers appear in the file:

```
<<<<<<< HEAD
Your changes
=======
Incoming changes
>>>>>>> branch-name
```

Tip: Conflicts are normal — they mean teamwork is happening!

What Is GitHub? How Is It Different from Git?

Git is a local version control tool that tracks changes in code.

GitHub is a cloud platform for hosting and sharing Git repositories.

Key differences:

Feature	Git	GitHub
Type	Tool	Platform
Works Offline?	✅ Yes	❌ Needs internet
Purpose	Track and manage versions	Collaborate, share, review code
Example Command	git commit	git push origin main

GitHub = Social + Remote layer built on top of Git.

GitHub (Remote Repositories)

Commands:

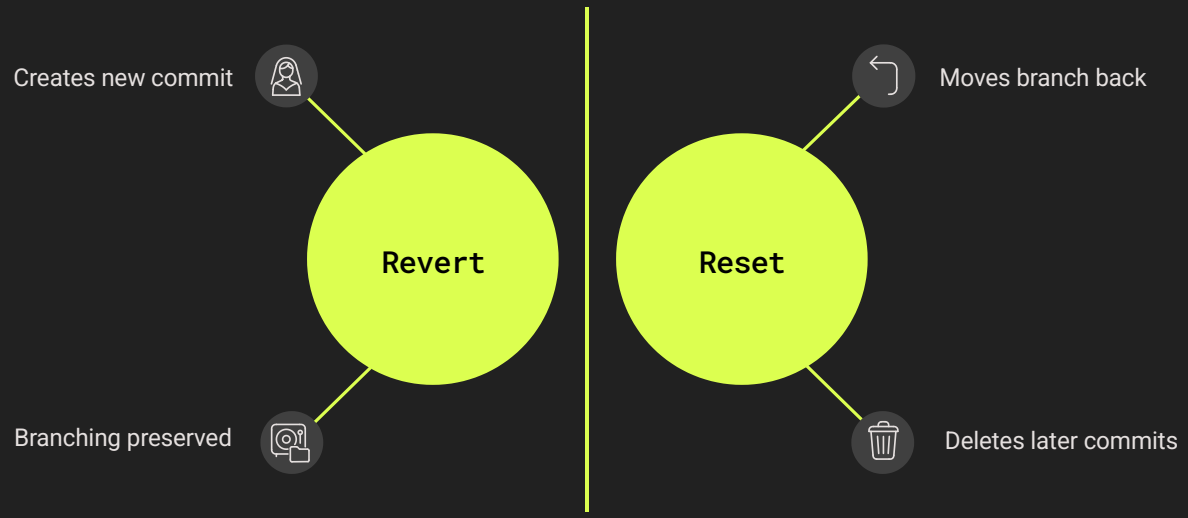
```
git remote add origin    # connect local repo to GitHub
git push origin master      # upload branch (can be any branch)
git clone            # download entire repo
git pull origin master     # fetch and merge remote updates
```

💡 git pull = git fetch + git merge

Final Tips

- Use git help and documentation often
- Don't panic — re-clone if things get messy
- Test new commands in a toy repo
- Avoid committing large or private files
- Never rewrite public history
- Always pull before push (and ideally before coding)

Revert vs Reset



Revert

Creates a new commit to undo changes (preserves history)

Reset

Moves the branch back and deletes later commits

Commands:

```
git revert <commit-hash>
git reset --hard <commit-hash>
git log --oneline  # view commit hashes
```

💡 You can still recover commits if you know their hash (even after reset)