Gemini Chat

Here is a comprehensive, structured guide to Git, ranging from absolute basics to advanced internal manipulation.

🌳 Git Mastery: From Zero to Hero

🧱 Part 1: The Core Concepts & Architecture

Before typing commands, you must understand where your files live. Git has **three main states**.

1. **Working Directory:** The folder on your computer where you edit files.
2. **Staging Area (Index):** A draft space where you prepare files for the next commit.
3. **Local Repository (.git):** Where Git permanently stores the history (snapshots) on your computer.

⚙️ Part 2: Configuration (First-Time Setup)

You only need to do this once per computer.

Command	Description
`git config --global user.name "Your Name"`	Sets the name attached to your commits.
`git config --global user.email "email@example.com"`	Sets the email attached to your commits.
`git config --global core.editor "code --wait"`	Sets VS Code (or other editor) as the default text editor for Git.
`git config --list`	Lists all configuration settings.

🚀 Part 3: The Basics (Daily Workflow)

1. Initialization & Creation

Command	Description
`git init`	Turns the current directory into a new Git repository.
`git clone <url>`	Downloads an existing repository from GitHub/GitLab to your machine.

2. Staging & Committing (The Cycle)

Command	Description
`git status`	**Most used command.** Shows changed files and what is currently staged.
`git add <file>`	Moves a specific file from Working Directory to Staging Area.
`git add .`	Stages **all** changed and new files in the current directory.
`git commit -m "Message"`	Saves the staged changes to the Local Repository with a descriptive message.
`git commit -a -m "Message"`	Skip staging and commit all tracked files directly (shortcut).

3. Viewing History

Command	Description
`git log`	Shows the commit history (Hash, Author, Date, Message).
`git log --oneline`	Condensed history (one line per commit).
`git log --graph --oneline --all`	Visualizes the branch structure in the terminal.

🌿 Part 4: Branching & Merging

Branching allows you to work on new features without affecting the main code.

1. Managing Branches

Command	Description
`git branch`	Lists all local branches.
`git branch <name>`	Creates a new branch (but does not switch to it).
`git checkout <name>`	Switches to the specified branch.
`git switch <name>`	(Newer command) Switches to the specified branch.
`git checkout -b <name>`	Creates **and** switches to a new branch immediately.
`git branch -d <name>`	Deletes a branch (safe delete).
`git branch -D <name>`	Force deletes a branch (even if unmerged).

2. Merging Changes

Command	Description
`git merge <branch>`	Merges the specified `<branch>` into your **current** branch.
`git merge --abort`	Cancels a merge if conflicts occur.

> **Note on Conflicts:** If Git cannot merge automatically, you must open the files, fix the code inside the `<<<<<<<` and `>>>>>>>`

markers, run `git add`, and then `git commit`.

📡 Part 5: Remote Repositories (GitHub/GitLab)

Connecting your local computer to the cloud.

Command	Description
`git remote add origin <url>`	Connects your local repo to a remote server (aliases it as "origin").
`git remote -v`	Verifies the remote URLs.
`git push -u origin <branch>`	Uploads your branch to the remote. `-u` sets the upstream tracking (so you can just type `git push` next time).
`git pull`	Downloads changes from remote and **immediately merges** them into your local branch.
`git fetch`	Downloads changes from remote but **does not merge** them. Safer than pull.

📦 Part 6: Intermediate Tools

1. Stashing (Temporary Storage)

Useful when you aren't ready to commit but need to switch branches.

Command	Description
`git stash`	Temporarily saves modified tracked files and reverts to the last commit.
`git stash list`	Shows all stashed changes.
`git stash pop`	Restores the most recently stashed changes and deletes them from the stash list.
`git stash apply`	Restores the changes but keeps them in the stash list.

2. Comparisons

Command	Description
`git diff`	Shows differences between Working Directory and Staging Area.
`git diff --staged`	Shows differences between Staging Area and the Last Commit.

3. Ignoring Files

Create a file named `.gitignore`. Add file names or patterns (like `*.log`, `node_modules/`, `.env`) inside it. Git will completely ignore these files.

☠️ Part 7: Advanced Git (The Danger Zone)

Use these with caution, especially on shared branches.

1. Undoing & Resetting

Command	Description
`git restore <file>`	Discards changes in the working directory (undoes editing).
`git restore --staged <file>`	Unstages a file (moves it from Staging back to Working Directory).
`git commit --amend`	Edit the **most recent** commit message or add forgotten files to it.
`git reset --soft HEAD~1`	Undoes the last commit but keeps changes in the Staging Area.
`git reset --mixed HEAD~1`	(Default) Undoes commit, keeps changes in Working Directory (unstaged).
`git reset --hard HEAD~1`	**Destructive.** Undoes the commit and **deletes** all changes.

Command	Description
`git revert <commit-hash>`	Creates a *new* commit that does the exact opposite of a previous commit. **Safe for public repos.**

2. Rebasing (Rewriting History)

Rebasing moves your branch to start from the tip of another branch. It creates a linear history.

Command	Description
`git rebase main`	Applies your current branch's commits one by one on top of the `main` branch.
`git rebase -i HEAD~3`	**Interactive Rebase.** Allows you to squash, edit, reorder, or drop the last 3 commits.

3. Cherry Picking & Debugging

Command	Description
`git cherry-pick <commit-hash>`	Takes a specific commit from *any* branch and applies it to your current branch.
`git blame <file>`	Shows who modified which line of a file and when.
`git bisect start`	Starts a binary search to find which commit introduced a bug.

4. The Safety Net

Command	Description
`git reflog`	**The life-saver.** Shows a log of *every* action you've taken (even undoing resets and deleted branches). If you lost a commit, find its hash here and merge it back.

🧠 Summary of Workflow for a New Feature

1. `git pull origin main` (Get latest code)
2. `git checkout -b feature-login` (Create new branch)
3. *(Make changes)*
4. `git add .`
5. `git commit -m "Added login form"`
6. `git push -u origin feature-login`
7. *(Go to GitHub and open a Pull Request)*

Would you like me to explain **GitFlow** vs. **Trunk-Based Development** workflows next? This is crucial for your DevOps interest.