Git & GitHub Cheat Sheet

Complete Reference Guide: Beginner to Advanced

1. Git Basics

Git is a distributed version control system that tracks changes in your code. Each developer has a complete copy of the repository history.

Key Concepts

Repository: Project folder tracked by Git

Commit: Snapshot of your code at a point in time

Branch: Independent line of development

Remote: Version of your repo hosted elsewhere

HEAD: Pointer to current branch/commit

Essential Commands

`git init`
Initialize a new Git repository

`git clone <url>`
Copy a remote repository locally

`git status`
Show working tree status

`git add <file>`
Stage file(s) for commit

`git add .`
Stage all changes in current directory

`git commit -m "message"`
Commit staged changes with message

`git log`
View commit history

`git log --oneline`
Compact commit history view

`git help <command>`
Get help for any Git command

2. Working With Files & Commits

The staging area is an intermediate layer between your working directory and repository. Changes must be staged before committing.

Workflow

- Modify files in working directory
- Stage changes with `git add`
- Commit staged changes with `git commit`

Commands

`git diff`
Show unstaged changes

`git diff --staged`
Show staged changes

`git restore <file>`
Discard changes in working directory

`git restore --staged <file>`
Unstage file (keep changes)

`git rm <file>`
Remove file and stage deletion

`git rm --cached <file>`
Remove from Git but keep locally

`git mv <old> <new>`
Rename/move file and stage change

`git commit --amend`
Modify last commit

> TIP: Use `git add -p` for interactive staging of partial changes

3. Branching & Merging

Branches allow parallel development. Merging integrates changes from different branches. Rebasing rewrites history for a cleaner commit log.

Branch Management

`git branch`
List all branches

`git branch <name>`
Create new branch

`git branch -d <name>`
Delete merged branch

`git branch -D <name>`
Force delete branch

`git checkout <branch>`
Switch to branch (older syntax)

`git checkout -b <name>`
Create and switch to new branch

`git switch <branch>`
Switch to branch (modern syntax)

`git switch -c <name>`
Create and switch to new branch

Merging & Rebasing

`git merge <branch>`
Merge branch into current branch

`git merge --abort`
Cancel merge with conflicts

`git rebase <branch>`
Rebase current branch onto target

`git rebase --continue`
Continue rebase after resolving conflicts

`git rebase --abort`
Cancel rebase operation

> WARNING: Never rebase commits pushed to shared branches

4. GitHub & Remote Repositories

Remote repositories enable collaboration. Push sends your commits to remote, pull fetches and merges changes, fetch only downloads changes.

Remote Setup

`git remote add origin <url>`
Add remote repository

`git remote -v`
List remote repositories

`git remote remove <name>`
Remove remote connection

Sync Commands

`git push`
Push commits to remote

`git push -u origin <branch>`
Push and set upstream tracking

`git push --force`
Force push (overwrites remote)

`git pull`
Fetch and merge remote changes

`git pull --rebase`
Fetch and rebase local commits

`git fetch`
Download remote changes (no merge)

`git fetch --all`
Fetch from all remotes

Forks & Pull Requests

- Fork: Copy of repo under your account
- Clone your fork locally
- Create feature branch for changes
- Push branch to your fork
- Open Pull Request on original repo
- Sync with upstream using `git fetch upstream`

5. Undoing, Fixing & Debugging

Reset (Rewrites History)

`git reset --soft HEAD~1`
Undo commit, keep changes staged

`git reset --mixed HEAD~1`
Undo commit, unstage changes

`git reset --hard HEAD~1`
⚠ Undo commit, discard changes

`git reset --hard origin/main`
⚠ Match local to remote exactly

> UNSAFE: --hard permanently deletes uncommitted work

Revert (Safe Alternative)

`git revert <commit>`
Create new commit that undoes changes

`git revert --no-commit <commit>`
Revert changes but don't commit yet

Stashing

`git stash`
Save uncommitted changes temporarily

`git stash pop`
Apply and remove latest stash

`git stash list`
Show all stashed changes

`git stash apply stash@{n}`
Apply specific stash

Cleanup

`git clean -n`
Preview untracked files to delete

`git clean -fd`
⚠ Delete untracked files/directories

6. Advanced Git

Interactive Rebase

`git rebase -i HEAD~3`
Edit last 3 commits interactively

Options: pick, reword, edit, squash, fixup, drop

Cherry-Picking

`git cherry-pick <commit>`
Apply specific commit to current branch

`git cherry-pick --continue`
Continue after resolving conflicts

Tagging

`git tag`
List all tags

`git tag v1.0.0`
Create lightweight tag

`git tag -a v1.0.0 -m "msg"`
Create annotated tag

`git push --tags`
Push all tags to remote

`git tag -d <tag>`
Delete local tag

Submodules

`git submodule add <url>`
Add repository as submodule

`git submodule update --init`
Initialize and update submodules

`git submodule update --remote`
Update submodules to latest commits

Aliases

`git config --global alias.co checkout`
Create shortcut: git co

`git config --global alias.st status`
Create shortcut: git st

7. Configuration & Settings

Git configuration exists at three levels: system, global (user), and local (repository). Local overrides global, global overrides system.

User Identity

```
git config --global user.name "Your Name"
```
Set your name for commits

```
git config --global user.email "you@example.com"
```
Set your email for commits

Editor & Tools

```
git config --global core.editor "code --wait"
```
Set VS Code as default editor

```
git config --global merge.tool vimdiff
```
Set merge conflict tool

View & Modify

```
git config --list
```
Show all configuration settings

```
git config --global --list
```
Show global settings only

```
git config --global --edit
```
Edit global config file

Useful Settings

```
git config --global init.defaultBranch main
```
Set default branch name

```
git config --global color.ui auto
```
Enable colored output

```
git config --global pull.rebase true
```
Use rebase by default for pulls

For more info, visit @digitalProgramLife

8. Tips, Best Practices & Workflow

Commit Messages

- Use present tense: "Add feature" not "Added"
- Keep first line under 50 characters
- Capitalize first letter
- No period at end of subject
- Add detailed body if needed (blank line after subject)
- Reference issues: "Fix #123"

```
Good: "Add user authentication module"
Bad: "stuff", "fixed bug", "updates"
```

Branching Strategy

- `main/master` : Production-ready code
- `develop` : Integration branch
- `feature/*` : New features
- `hotfix/*` : Urgent production fixes
- `release/*` : Release preparation
- Delete branches after merging
- Keep branches short-lived

Common Mistakes to Avoid

- Never commit sensitive data (passwords, keys)
- Don't commit large binary files
- Avoid `git add .` without reviewing changes
- Don't rewrite published history
- Never force push to shared branches
- Don't commit directly to main/master
- Avoid mixing unrelated changes in one commit

Pull Request Workflow

1. Create Branch	2. Make Changes	3. Commit	4. Push	5. Open PR	6. Review & Merge
git checkout -b feature/name	Implement feature	git commit -m "msg"	git push -u origin branch	On GitHub	After approval

PRO TIP: Use `.gitignore` to exclude files. Add `.env`, `node_modules/`, `*.log`, IDE configs. Template: github.com/github/gitignore

Merge Conflict Resolution Guide

Beginner | Intermediate

What are merge conflicts? They occur when Git can't automatically combine changes from different branches because the same lines of code were modified differently. Don't panic - conflicts are normal and solvable!

1 Conflict Detected!

Git stops the merge/rebase and shows: `CONFLICT (content): Merge conflict in <file>`
Your repository is now in a "conflicted" state.

↓

2 Identify Conflicted Files

Run: `git status`
Files with conflicts will be listed under "Unmerged paths" in red.
Look for files marked with "both modified" or "both added".

↓

3 Open Conflicted Files

Open each conflicted file in your editor. Git marks conflicts with:
`<<<<<<< HEAD` (your current branch changes)
`=======` (separator)
`>>>>>>> branch-name` (incoming branch changes)

↓

Decision: How to Resolve?

Choose your resolution strategy →

Option A: Manual Resolution

- Edit the file directly
- Remove conflict markers (<<<, ===, >>>)
- Keep what you want from both sides
- Combine changes if needed
- Save the file

Option B: Accept One Side

- `git checkout --ours <file>`
- Keeps YOUR changes (current branch)
- `git checkout --theirs <file>`
- Keeps THEIR changes (merging branch)
- Use when one side is clearly correct

PRO TIP: Use a merge tool for complex conflicts: `git mergetool` opens a visual diff tool (VS Code, KDiff3, P4Merge). Configure with: `git config --global merge.tool <toolname>`

Git & GitHub Cheat Sheet

ConflictResolution Flowchart(cont.)

4 Stage Resolved Files

After resolving conflicts: `git add <resolved-file>`
Verify: `git status` should show files in green

↓

Were You Merging or Rebasing?

↓

If MERGING:

`git commit`
(Git auto-generates merge message)
Or: `git commit -m "Merge: resolve conflicts"`

If REBASING:

`gitrebase --continue`
Git will apply remaining commits.
May need to resolve more conflicts.

↓

5 Verify & Test

- Run tests: `npm test` / `pytest`
- Check compilation/execution
- Review: `git log --oneline -5`

⚠ **EMERGENCY EXITS:** Abort and start over:
• Merge: `git merge --abort` • Rebase: `git rebase --abort`

Common Conflict Scenarios

Same Line Modified
Problem: Two branches changed same line.
Fix: Manually combine or choose best version.

File Deleted vs Modified
Problem: One deleted, other modified.
Fix: `git rm <file>` OR keep modified.

Semantic Conflicts
Problem: No conflicts shown, code breaks.
Fix: Review all changes, run tests.

Binary File Conflicts
Problem: Images/PDFs can't merge.
Fix: Use `--ours` / `--theirs`.

BEST PRACTICES:
✓ Keep branches short-lived • ✓ Pull frequently • ✓ Small focused commits
✓ Communicate before big merges • ✓ Run tests after resolution

Prevention Strategies
- Pull daily: `git pull origin main`
- Short feature branches (2-5 days)
- Coordinate on shared files
- Use `git fetch` before merge

Useful Tools
- `git mergetool` - Visual merge tool
- VS Code built-in merge editor
- KDiff3, P4Merge, Beyond Compare
- `git diff --check` - Find conflicts