Complete Git/GitHub Cheatsheet - Beginner to Advanced

Table of Contents

What is Git and GitHub?

Git is a distributed version control system that tracks changes in files and coordinates work among multiple people. It's like having a detailed history of every change made to your project.

GitHub is a cloud-based hosting service for Git repositories. Think of it as a social media platform for code where you can store, share, and collaborate on projects.

Key Differences:

- Git = The tool (works locally on your computer)
- GitHub = The service (cloud-based platform)

Installation and Setup

Installing Git

Windows:

```
# Download from https://git-scm.com/download/win
# Or use chocolatey
choco install git
```

macOS:

```
# Using Homebrew
brew install git
# Or download from https://git-scm.com/download/mac
```

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install git
```

Initial Configuration

```
# Set your name and email (required for commits)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

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

# Set default editor
git config --global core.editor "code --wait"  # VS Code
git config --global core.editor "nano"         # Nano

# Check your configuration
git config --list
git config user.name  # Check specific setting
```

Basic Git Concepts

The Three States

1. **Working Directory** - Where you modify files
2. **Staging Area (Index)** - Where you prepare files for commit
3. **Repository** - Where Git stores committed snapshots

Basic Workflow

```

```

```
Working Directory → Staging Area → Repository
     (edit)         (git add)      (git commit)
```

Repository Basics

Creating a Repository

```
# Create a new repository in current directory
git init

# Create a new repository in a specific directory
git init my-project
cd my-project

# Clone an existing repository
git clone https://github.com/username/repository-name.git
git clone https://github.com/username/repository-name.git my-folder

# Clone with SSH (more secure)
git clone git@github.com:username/repository-name.git
```

Repository Status

```
# Check repository status
git status

# Short status format
git status -s
# M  = modified
# A  = added
# D  = deleted
# R  = renamed
# C  = copied
# U  = updated but unmerged
```

File Operations

Adding Files to Staging Area

```
# Add specific file
git add filename.txt

# Add multiple files
```

```
git add file1.txt file2.txt

# Add all files in current directory
git add .

# Add all files in repository
git add -A

# Add files by pattern
git add *.txt
git add src/

# Add interactively (choose what to stage)
git add -i
git add -p  # Patch mode - stage parts of files
```

Removing Files

```

# Remove file from working directory and staging area
git rm filename.txt

# Remove file only from staging area (keep in working directory)
git rm --cached filename.txt

# Remove directory
git rm -r directory-name/

# Remove files matching pattern
git rm "*.log"
```

Moving/Renaming Files

```

# Rename/move file
git mv old-filename.txt new-filename.txt
git mv file.txt folder/file.txt

# This is equivalent to:
mv old-filename.txt new-filename.txt
git rm old-filename.txt
git add new-filename.txt
```

Commit Management

Creating Commits

```

```

```
# Commit with message
git commit -m "Add user authentication feature"

# Commit with detailed message
git commit -m "Add user authentication" -m "- Added login form validation
- Implemented JWT token handling
- Added password hashing"

# Add and commit in one step
git commit -am "Fix bug in user registration"

# Commit with editor for longer message
git commit

# Amend last commit (change message or add files)
git commit --amend -m "Better commit message"
git add forgotten-file.txt
git commit --amend --no-edit
```

Commit Message Best Practices

```
# Good commit message format:
# <type>: <description>
#
# [optional body]
#
# [optional footer]

git commit -m "feat: add user authentication system

- Implement login/logout functionality
- Add password hashing with bcrypt
- Create JWT token management

Closes #123"

# Common types:
# feat: new feature
# fix: bug fix
# docs: documentation
# style: formatting
# refactor: code restructuring
# test: adding tests
# chore: maintenance
```

Branching Fundamentals

Understanding Branches

A branch is a moveable pointer to a specific commit. The default branch is usually called `main` or `master`.

Basic Branch Operations

```
# List all branches
git branch          # Local branches
git branch -r       # Remote branches
git branch -a       # All branches

# Create new branch
git branch feature-login

# Create and switch to new branch
git checkout -b feature-login
git switch -c feature-login  # Newer syntax

# Switch between branches
git checkout main
git switch main      # Newer syntax

# Switch to previous branch
git checkout -
git switch -
```

Branch Management

```
# Rename current branch
git branch -m new-branch-name

# Rename any branch
git branch -m old-name new-name

# Delete branch (safe - prevents deletion if unmerged)
git branch -d feature-login

# Force delete branch (dangerous)
git branch -D feature-login

# Delete remote branch
git push origin --delete feature-login
```

Working with Branches

```
# Create branch from specific commit
git branch feature-x 9fceb02

# Create branch from another branch
git checkout -b feature-y feature-x
```

```
# Show branch information
git show-branch
git branch -v        # Show last commit on each branch
git branch --merged  # Show merged branches
git branch --no-merged  # Show unmerged branches
```

Remote Repositories

Setting Up Remotes

```
# Add remote repository
git remote add origin https://github.com/username/repo.git

# List remotes
git remote
git remote -v  # Show URLs

# Show remote information
git remote show origin

# Rename remote
git remote rename origin upstream

# Remove remote
git remote remove upstream
```

Pushing and Pulling

```
# Push to remote
git push origin main
git push origin feature-branch

# Push and set upstream
git push -u origin main
git push --set-upstream origin feature-branch

# Push all branches
git push origin --all

# Push tags
git push origin --tags

# Force push (dangerous - rewrites history)
git push --force origin main
git push --force-with-lease origin main  # Safer force push
```

```
# Fetch changes from remote (doesn't merge)
git fetch origin
git fetch --all

# Pull changes (fetch + merge)
git pull origin main
git pull  # If upstream is set

# Pull with rebase
git pull --rebase origin main

# Pull specific branch
git pull origin feature-branch
```

Tracking Branches

```
# Create local branch tracking remote branch
git checkout -b feature-x origin/feature-x
git checkout --track origin/feature-x

# Set upstream for existing branch
git branch --set-upstream-to=origin/main main

# Show tracking information
git branch -vv
```

Merging Strategies

Types of Merges

1. Fast-Forward Merge

- Happens when target branch hasn't diverged
- Simply moves pointer forward

2. Three-Way Merge

- Creates a new merge commit
- Combines changes from both branches

3. Squash Merge

- Combines all commits into one

Merge Commands

```
# Basic merge
git checkout main
git merge feature-branch

# Merge with custom message
git merge feature-branch -m "Merge feature-branch into main"

# No fast-forward merge (always create merge commit)
git merge --no-ff feature-branch

# Squash merge (combine all commits into one)
git merge --squash feature-branch
git commit -m "Add feature X"

# Abort merge if there are conflicts
git merge --abort
```

Merge vs Rebase Decision Tree

```
Use MERGE when:
- Working on a team
- Want to preserve exact history
- Working on public branches
- Want to show when features were integrated

Use REBASE when:
- Want clean, linear history
- Working on private branches
- Want to avoid merge commits
- Preparing feature branch for merge
```

Rebasing

What is Rebasing?

Rebasing replays commits from one branch onto another, creating a linear history.

Basic Rebase

```
# Rebase current branch onto main
git rebase main

# Rebase specific branch
git rebase main feature-branch

# Interactive rebase (edit commit history)
git rebase -i HEAD~3  # Last 3 commits
```

```
git rebase -i main    # All commits since main

# Continue rebase after resolving conflicts
git rebase --continue

# Skip current commit during rebase
git rebase --skip

# Abort rebase
git rebase --abort
```

Interactive Rebase Options

```
# During interactive rebase, you can:
pick f7f3f6d  # Keep commit as is
reword f7f3f6d  # Keep commit but edit message
edit f7f3f6d  # Stop to amend commit
squash f7f3f6d  # Combine with previous commit
fixup f7f3f6d  # Like squash but discard message
drop f7f3f6d  # Remove commit entirely

# Example interactive rebase:
git rebase -i HEAD~4
# This opens editor with last 4 commits
```

Rebase Examples

```
# Clean up commits before pushing
git rebase -i HEAD~3

# Update feature branch with latest main
git checkout feature-branch
git rebase main

# Rebase and push
git rebase main
git push --force-with-lease origin feature-branch
```

Conflict Resolution

Understanding Conflicts

Conflicts occur when Git can't automatically merge changes because the same lines were modified in different ways.

Conflict Markers

```
<<<<<<< HEAD
Your current branch changes
=======
Changes from other branch
>>>>>>> branch-name
```

Resolving Conflicts

```
# Check which files have conflicts
git status

# View conflicts
git diff

# After manually editing files to resolve conflicts:
git add resolved-file.txt
git commit  # For merge conflicts
git rebase --continue  # For rebase conflicts

# Use merge tools
git mergetool
git mergetool --tool=vimdiff
```

Conflict Resolution Tools

```
# Configure merge tool
git config --global merge.tool vimdiff
git config --global merge.tool vscode

# For VS Code
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'
```

Preventing Conflicts

```
# Regularly sync with main branch
git checkout main
git pull origin main
git checkout feature-branch
git rebase main

# Use smaller, focused commits
# Communicate with team about file changes
# Use feature flags for long-running features
```

Stashing

What is Stashing?

Stashing temporarily saves your uncommitted changes so you can work on something else, then come back and re-apply them.

Basic Stashing

```
# Stash current changes
git stash
git stash push -m "Work in progress on feature X"

# Stash including untracked files
git stash -u
git stash --include-untracked

# List stashes
git stash list

# Apply latest stash
git stash apply
git stash pop  # Apply and remove from stash list

# Apply specific stash
git stash apply stash@{2}
git stash pop stash@{1}
```

Advanced Stashing

```
# Stash only specific files
git stash push -m "Partial stash" file1.txt file2.txt

# Stash with pathspec
git stash push -m "Stash CSS changes" -- "*.css"

# Create branch from stash
git stash branch new-feature-branch stash@{1}

# Show stash contents
git stash show
git stash show -p stash@{1}  # Show diff

# Drop stash
git stash drop stash@{1}
git stash clear  # Remove all stashes
```

Stashing Use Cases

```
# Quick bug fix while working on feature
git stash -m "WIP: feature development"
git checkout main
git checkout -b hotfix-bug
# ... fix bug ...
git checkout feature-branch
git stash pop

# Switch branches with uncommitted changes
git stash
git checkout other-branch
# ... do work ...
git checkout original-branch
git stash pop
```

Tagging

Understanding Tags

Tags are references to specific commits, typically used to mark release points.

Types of Tags

- **Lightweight tags**: Just a pointer to a commit
- **Annotated tags**: Full objects with metadata

Creating Tags

```
# Lightweight tag
git tag v1.0

# Annotated tag (recommended)
git tag -a v1.0 -m "Version 1.0 release"

# Tag specific commit
git tag -a v0.9 -m "Beta release" 9fceb02

# Tag with GPG signature
git tag -s v1.0 -m "Signed version 1.0"
```

Managing Tags

```
# List tags
git tag
```

```
git tag -l "v1.*"  # List tags matching pattern

# Show tag information
git show v1.0

# Delete tag
git tag -d v1.0

# Push tags to remote
git push origin v1.0
git push origin --tags  # Push all tags

# Delete remote tag
git push origin --delete v1.0
```

Tag Best Practices

```
# Semantic versioning: MAJOR.MINOR.PATCH
git tag -a v1.2.3 -m "Release version 1.2.3"

# Pre-release tags
git tag -a v2.0.0-beta.1 -m "Beta release for v2.0.0"

# Build metadata
git tag -a v1.0.0+20230615 -m "Release with build metadata"
```

History and Logs

Basic Log Commands

```
# Show commit history
git log

# One line per commit
git log --oneline

# Show last n commits
git log -n 5
git log -5

# Show commits with file changes
git log --stat
git log --name-only
git log --name-status
```

Advanced Log Formatting

```
# Custom format
git log --pretty=format:"%h - %an, %ar : %s"
# %h = short hash, %an = author name, %ar = author date relative, %s = subject

# Graphical representation
git log --graph --oneline --all

# Show commits affecting specific file
git log -- filename.txt
git log -p -- filename.txt  # Show diffs

# Show commits by author
git log --author="John Doe"

# Show commits in date range
git log --since="2023-01-01" --until="2023-12-31"
git log --since="2 weeks ago"
```

Specialized Log Commands

```
# Show who changed each line of a file
git blame filename.txt
git blame -L 10,20 filename.txt  # Lines 10-20 only

# Show file history
git log --follow filename.txt

# Show commits that changed specific string
git log -S "function_name"
git log -G "regex_pattern"

# Show merge commits only
git log --merges

# Show non-merge commits only
git log --no-merges
```

Visual History Tools

```
# Built-in GUI
gitk
gitk --all

# Show commits in specific format
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit
```

Undoing Changes

Undoing Working Directory Changes

```
# Discard changes in working directory
git checkout -- filename.txt
git restore filename.txt  # Newer syntax

# Discard all changes in working directory
git checkout -- .
git restore .

# Restore file to specific commit
git checkout 9fceb02 -- filename.txt
git restore --source=HEAD~2 filename.txt
```

Undoing Staged Changes

```
# Unstage file (keep changes in working directory)
git reset HEAD filename.txt
git restore --staged filename.txt  # Newer syntax

# Unstage all files
git reset HEAD
git restore --staged .
```

Undoing Commits

```
# Undo last commit (keep changes in working directory)
git reset --soft HEAD~1

# Undo last commit (keep changes in staging area)
git reset --mixed HEAD~1
git reset HEAD~1  # Same as --mixed

# Undo last commit (discard all changes)
git reset --hard HEAD~1

# Undo multiple commits
git reset --hard HEAD~3

# Reset to specific commit
git reset --hard 9fceb02
```

Safe Undoing with Revert

```
# Create new commit that undoes previous commit
git revert HEAD
git revert 9fceb02

# Revert merge commit
git revert -m 1 HEAD

# Revert multiple commits
git revert HEAD~3..HEAD
```

Emergency Recovery

```
# Show reference log (recent actions)
git reflog

# Recover lost commits
git reflog
git checkout 9fceb02  # From reflog

# Recover deleted branch
git reflog
git checkout -b recovered-branch 9fceb02
```

GitHub Specific Features

Repository Management

```
# Create repository on GitHub using CLI
gh repo create my-new-repo --public
gh repo create my-new-repo --private

# Clone your own repository
gh repo clone username/repository-name

# Fork repository
gh repo fork original-owner/repo-name
```

Pull Requests

```
# Create pull request
gh pr create --title "Add new feature" --body "Description of changes"

# List pull requests
gh pr list
```

```
# View pull request
gh pr view 123

# Checkout pull request locally
gh pr checkout 123

# Merge pull request
gh pr merge 123 --merge
gh pr merge 123 --squash
gh pr merge 123 --rebase
```

Issues

```
# Create issue
gh issue create --title "Bug report" --body "Description"

# List issues
gh issue list

# View issue
gh issue view 123

# Close issue
gh issue close 123
```

GitHub Actions

```
# View workflow runs
gh run list

# View specific run
gh run view 123456

# Re-run workflow
gh run rerun 123456
```

Advanced Git Operations

Cherry-picking

```
# Apply specific commit to current branch
git cherry-pick 9fceb02

# Cherry-pick multiple commits
```

```
git cherry-pick 9fceb02 1234567

# Cherry-pick range of commits
git cherry-pick 9fceb02..1234567

# Cherry-pick without committing
git cherry-pick --no-commit 9fceb02
```

Bisecting (Finding Bugs)

```
# Start bisecting
git bisect start
git bisect bad          # Current commit is bad
git bisect good v1.0    # v1.0 was good

# Git will checkout middle commit
# Test and mark as good or bad
git bisect good   # or git bisect bad

# Continue until bug is found
git bisect reset  # Return to original state
```

Subtrees and Submodules

```
# Add subtree
git subtree add --prefix=lib/mylib https://github.com/user/mylib.git main --squash

# Update subtree
git subtree pull --prefix=lib/mylib https://github.com/user/mylib.git main --
squash

# Add submodule
git submodule add https://github.com/user/mylib.git lib/mylib

# Initialize submodules after cloning
git submodule init
git submodule update
# Or in one command:
git submodule update --init --recursive
```

Advanced Merging

```
# Merge with custom strategy
git merge -X theirs feature-branch    # Prefer their changes
git merge -X ours feature-branch      # Prefer our changes
```

```
# Merge specific files only
git checkout feature-branch -- specific-file.txt
git commit -m "Merge specific file from feature-branch"

# Merge without checkout
git merge --no-commit feature-branch
```

Best Practices

Commit Best Practices

```
# Make atomic commits (one logical change per commit)
git add user-auth.js
git commit -m "Add user authentication logic"

git add user-auth.test.js
git commit -m "Add tests for user authentication"

# Write clear commit messages
# Good:
git commit -m "Fix null pointer exception in user login"

# Bad:
git commit -m "Fix bug"
```

Branching Strategy

```
# Git Flow example
git checkout -b develop           # Development branch
git checkout -b feature/login     # Feature branch
# Work on feature
git checkout develop
git merge --no-ff feature/login
git branch -d feature/login

# Release branch
git checkout -b release/1.0 develop
# Bug fixes only
git checkout main
git merge --no-ff release/1.0
git tag -a v1.0 -m "Version 1.0"
git checkout develop
git merge --no-ff release/1.0
```

Workflow Best Practices

```
# Always pull before starting work
git checkout main
git pull origin main
git checkout -b feature/new-feature

# Regularly sync feature branch
git checkout main
git pull origin main
git checkout feature/new-feature
git rebase main

# Clean up before pushing
git rebase -i HEAD~3  # Clean up commits
git push -u origin feature/new-feature
```

.gitignore Best Practices

```
# Create .gitignore file
echo "node_modules/" >> .gitignore
echo "*.log" >> .gitignore
echo ".env" >> .gitignore

# Common patterns:
# Dependencies
node_modules/
vendor/

# Build outputs
dist/
build/
*.o
*.exe

# Environment files
.env
.env.local

# IDE files
.vscode/
.idea/
*.swp

# OS files
.DS_Store
Thumbs.db

# Logs
*.log
logs/
```

Troubleshooting Common Issues

Common Problems and Solutions

```
# Problem: Accidental commit to wrong branch
# Solution: Move commit to correct branch
git log --oneline -n 5  # Find commit hash
git checkout correct-branch
git cherry-pick <commit-hash>
git checkout wrong-branch
git reset --hard HEAD~1

# Problem: Need to change last commit message
# Solution: Amend commit
git commit --amend -m "Corrected commit message"

# Problem: Pushed wrong commit
# Solution: Revert (safe for shared repositories)
git revert HEAD
git push origin main

# Problem: Merge conflicts
# Solution: Resolve manually
git status  # See conflicted files
# Edit files to resolve conflicts
git add resolved-file.txt
git commit

# Problem: Lost commits
# Solution: Use reflog
git reflog
git checkout <lost-commit-hash>
git checkout -b recovery-branch
```

Performance Issues

```
# Large repository cleanup
git gc --aggressive
git prune

# Reduce repository size
git filter-branch --tree-filter 'rm -rf large-folder' HEAD
# Modern alternative:
git filter-repo --path large-folder --invert-paths

# Shallow clone for large repositories
git clone --depth 1 https://github.com/user/large-repo.git
```

Configuration Issues

```
# Fix line ending issues
git config --global core.autocrlf true   # Windows
git config --global core.autocrlf input  # Mac/Linux

# Fix permission issues
git config --global core.filemode false

# Fix SSL issues
git config --global http.sslverify false  # Not recommended for production
```

Quick Reference

Essential Commands

```
# Setup
git init
git clone <url>
git config --global user.name "Name"
git config --global user.email "email"

# Basic workflow
git status
git add <file>
git commit -m "message"
git push origin main
git pull origin main

# Branching
git branch                    # List branches
git branch <name>             # Create branch
git checkout <branch>         # Switch branch
git checkout -b <branch>      # Create and switch
git merge <branch>            # Merge branch
git branch -d <branch>        # Delete branch

# History
git log
git log --oneline
git show <commit>
git diff

# Undo
git checkout -- <file>        # Discard changes
git reset HEAD <file>         # Unstage
git reset --hard HEAD~1       # Undo commit
git revert <commit>           # Safe undo
```

Useful Aliases

```
# Add to ~/.gitconfig or use git config --global alias.<name> "<command>"
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual '!gitk'
git config --global alias.tree 'log --graph --pretty=format:"%h - %an, %ar : %s"'
```

Keyboard Shortcuts (Terminal)

```
# Command history
Ctrl+R          # Search command history
!!              # Repeat last command
!git            # Repeat last git command

# Navigation
Ctrl+A          # Beginning of line
Ctrl+E          # End of line
Ctrl+W          # Delete word backward
Ctrl+U          # Delete line backward
```

Summary

This cheatsheet covers Git and GitHub from basic concepts to advanced techniques. Remember:

1. **Start simple** - Master basic commands before moving to advanced features
2. **Practice regularly** - Use Git daily to build muscle memory
3. **Read commit messages** - They tell the story of your project
4. **Use branches** - They're cheap and powerful
5. **Backup important work** - Push to remote repositories regularly
6. **Learn from mistakes** - Git's reflog can save you from most disasters

Keep this cheatsheet handy and refer to specific sections as needed. The more you use Git, the more natural these commands will become!