Git Cheatsheet

Essential commands for version control and collaboration

This cheatsheet provides a quick reference to fundamental Git operations, syntax, and advanced features, ideal for both beginners and experienced developers for efficient version control and project management.

Repository Setup
Initialize and clone repositories

File Tracking
Stage and commit changes

Branch Management
Create and manage branches

History Analysis
View and analyze commit history

Remote Operations
Push, pull, and sync changes

Repository Setup & Configuration

Initialize Repository: `git init`

Create a new Git repository in the current directory.

```
# Initialize new repository
git init
# Initialize in new directory
git init project-name
# Initialize bare repository (no working directory)
git init --bare
# Use custom template directory
git init --template=path
```

Clone Repository: `git clone`

Create a local copy of a remote repository.

```
# Clone via HTTPS
git clone https://github.com/user/repo.git
# Clone via SSH
git clone git@github.com:user/repo.git
# Clone with custom name
git clone repo.git local-name
# Shallow clone (latest commit only)
git clone --depth 1 repo.git
```

Global Configuration: `git config`

Set up user information and preferences globally.

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
git config --global init.defaultBranch main
# View all configuration settings
git config --list
```

Local Configuration: `git config --local`

Set repository-specific configuration.

```
# Set for current repo only
git config user.name "Project Name"
# Project-specific email
git config user.email "project@example.com"
```

Remote Management: `git remote`

Manage connections to remote repositories.

```
# Add remote
git remote add origin https://github.com/user/repo.git
# List all remotes with URLs
git remote -v
# Show detailed remote info
git remote show origin
# Rename remote
git remote rename origin upstream
# Remove remote
git remote remove upstream
```

Credential Storage: `git config credential`

Store authentication credentials to avoid repeated login.

```
# Cache for 15 minutes
git config --global credential.helper cache
# Store permanently
git config --global credential.helper store
# Cache for 1 hour
git config --global credential.helper 'cache --
timeout=3600'
```

Repository Info & Status

Understand the current state and history of your repository.

01

Check Status: `git status`

Display the current state of the working directory and staging area.

```
# Full status information
git status
# Short status format
git status -s
# Machine-readable format
git status --porcelain
# Show ignored files too
git status --ignored
```

02

View Differences: `git diff`

Show changes between different states of your repository.

```
# Changes in working directory vs
staging
git diff
# Changes in staging vs last commit
git diff --staged
# All uncommitted changes
git diff HEAD
# Changes in specific file
git diff file.txt
```

03

View History: `git log`

Display commit history and repository timeline.

```
# Full commit history
git log
# Condensed one-line format
git log --oneline
# Show last 5 commits
git log -5
# Visual branch graph
git log --graph --all
```

Staging & Committing Changes

Stage Files: `git add`

Add changes to the staging area for the next commit.

```
# Stage specific file
git add file.txt
# Stage all changes in current directory
git add .
# Stage all changes (including deletions)
git add -A
# Stage all JavaScript files
git add *.js
# Interactive staging (patch mode)
git add -p
```

Commit Changes: `git commit`

Save staged changes to the repository with a descriptive message.

```
# Commit with message
git commit -m "Add user authentication"
# Stage and commit modified files
git commit -a -m "Update docs"
# Modify the last commit
git commit --amend
# Amend without changing message
git commit --no-edit --amend
```

Unstage Files: `git reset`

Remove files from the staging area or undo commits.

```
# Unstage specific file
git reset file.txt
# Unstage all files
git reset
# Undo last commit, keep changes staged
git reset --soft HEAD~1
# Undo last commit, discard changes
git reset --hard HEAD~1
```

Discard Changes: `git checkout` / `git restore`

Revert changes in working directory to last committed state.

```
# Discard changes in file (old syntax)
git checkout -- file.txt
# Discard changes in file (new syntax)
git restore file.txt
# Unstage file (new syntax)
git restore --staged file.txt
# Discard all uncommitted changes
git checkout .
```

Branch Operations

List Branches: `git branch`

View and manage repository branches.

```
# List local branches
git branch
# List all branches (local and remote)
git branch -a
# List only remote branches
git branch -r
# Show last commit on each branch
git branch -v
```

Create & Switch: `git checkout` / `git switch`

Create new branches and switch between them.

```
# Create and switch to new branch
git checkout -b feature-branch
# Create and switch (new syntax)
git switch -c feature-branch
# Switch to existing branch
git checkout main
# Switch to existing branch (new syntax)
git switch main
```

Merge Branches: `git merge`

Combine changes from different branches.

```
# Merge feature-branch into current branch
git merge feature-branch
# Force merge commit
git merge --no-ff feature-branch
# Squash commits before merging
git merge --squash feature-branch
```

Delete Branches: `git branch -d`

Remove branches that are no longer needed.

```
# Delete merged branch
git branch -d feature-branch
# Force delete unmerged branch
git branch -D feature-branch
# Delete remote branch
git push origin --delete feature-branch
```

Remote Repository Operations

Synchronize changes with remote repositories and collaborate with others.

Fetch Updates: `git fetch`

Download changes from remote repository without merging.

```
# Fetch from default remote
git fetch
# Fetch from specific remote
git fetch origin
# Fetch from all remotes
git fetch --all
# Fetch specific branch
git fetch origin main
```

Pull Changes: `git pull`

Download and merge changes from remote repository.

```
# Pull from tracking branch
git pull
# Pull from specific remote branch
git pull origin main
# Pull with rebase instead of merge
git pull --rebase
# Only fast-forward, no merge commits
git pull --ff-only
```

Push Changes: `git push`

Upload local commits to remote repository.

```
# Push to tracking branch
git push
# Push to specific remote branch
git push origin main
# Push and set upstream tracking
git push -u origin feature
# Force push safely
git push --force-with-lease
```

Track Remote Branches: `git branch --track`

Set up tracking between local and remote branches.

```
# Set tracking
git branch --set-upstream-to=origin/main main
# Track remote branch
git checkout -b local-branch origin/remote-branch
```

Stashing & Temporary Storage

Temporarily save work without committing.

Stash Changes: `git stash`

Temporarily save uncommitted changes for later use.

```
# Stash current changes
git stash
# Stash with message
git stash save "Work in progress on feature X"
# Include untracked files
git stash -u
# Stash only unstaged changes
git stash --keep-index
```

List Stashes: `git stash list`

View all saved stashes.

```
# Show all stashes
git stash list
# Show changes in latest stash
git stash show
# Show changes in specific stash
git stash show stash@{1}
```

Apply Stashes: `git stash apply`

Restore previously stashed changes.

```
# Apply latest stash
git stash apply
# Apply specific stash
git stash apply stash@{1}
# Apply and remove latest stash
git stash pop
# Delete latest stash
git stash drop
```

Stash Management: Advanced Operations

Advanced stash operations for complex workflows.

```
# Create branch from stash
git stash branch new-branch stash@{1}
# Delete all stashes
git stash clear
```

History & Log Analysis

View Commit History: `git log`

Explore repository history with various formatting options.

```
# Visual branch history
git log --oneline --graph --all
# Commits by specific author
git log --author="John Doe"
# Recent commits
git log --since="2 weeks ago"
# Search commit messages
git log --grep="bug fix"
```

Blame & Annotation: `git blame`

See who last modified each line of a file.

```
# Show line-by-line authorship
git blame file.txt
# Blame specific lines
git blame -L 10,20 file.txt
# Alternative to blame
git annotate file.txt
```

Search Repository: `git grep`

Search for text patterns across repository history.

```
# Search for text in tracked files
git grep "function"
# Search with line numbers
git grep -n "TODO"
# Search in staged files
git grep --cached "bug"
```

Show Commit Details: `git show`

Display detailed information about specific commits.

```
# Show latest commit details
git show
# Show previous commit
git show HEAD~1
# Show specific commit by hash
git show abc123
# Show commit with file statistics
git show --stat
```

Undoing Changes & History Editing

Correct mistakes and modify repository history safely.

Revert Commits: `git revert`

Create new commits that undo previous changes safely.

```
# Revert latest commit
git revert HEAD
# Revert specific commit
git revert abc123
# Revert range of commits
git revert HEAD~3..HEAD
# Revert without auto-commit
git revert --no-commit abc123
```

Reset History: `git reset`

Move branch pointer and optionally modify working directory.

```
# Undo commit, keep changes staged
git reset --soft HEAD~1
# Undo commit and staging
git reset --mixed HEAD~1
# Undo commit, staging, and working dir
git reset --hard HEAD~1
```

Interactive Rebase: `git rebase -i`

Edit, reorder, or squash commits interactively.

```
# Interactive rebase last 3 commits
git rebase -i HEAD~3
# Rebase current branch onto main
git rebase -i main
# Continue after resolving conflicts
git rebase --continue
# Cancel rebase operation
git rebase --abort
```

Cherry-pick: `git cherry-pick`

Apply specific commits from other branches.

```
# Apply specific commit to current branch
git cherry-pick abc123
# Apply range of commits
git cherry-pick abc123..def456
# Cherry-pick without committing
git cherry-pick -n abc123
```

Conflict Resolution

Handle and resolve merge conflicts effectively.

Merge Conflicts: Resolution Process

Steps to resolve conflicts during merge operations.

```
# Check conflicted files
git status
# Mark conflict as resolved
git add resolved-file.txt
# Complete the merge
git commit
# Cancel merge and return to previous state
git merge --abort
```

Merge Tools: `git mergetool`

Launch external tools to help resolve conflicts visually.

```
# Launch default merge tool
git mergetool
# Set default merge tool
git config --global merge.tool vimdiff
# Use specific tool for this merge
git mergetool --tool=meld
```

Conflict Markers: Understanding the Format

Interpret Git's conflict markers in files.

```
<<<<<<< HEAD
Current branch content
=======
Incoming branch content
>>>>>>> feature-branch

# Edit file to resolve, then:
git add conflicted-file.txt
git commit
```

Diff Tools: `git difftool`

Use external diff tools for better conflict visualization.

```
# Launch diff tool for changes
git difftool
# Set default diff tool
git config --global diff.tool vimdiff
```

Tagging & Releases

Mark important points in repository history.

Create Tags: `git tag`

Mark specific commits with version labels.

```
# Create lightweight tag
git tag v1.0
# Create annotated tag
git tag -a v1.0 -m "Version 1.0 release"
# Tag specific commit
git tag -a v1.0 abc123
# Create signed tag
git tag -s v1.0
```

List & Show Tags: `git tag -l`

View existing tags and their information.

```
# List all tags
git tag
# List tags matching pattern
git tag -l "v1.*"
# Show tag details
git show v1.0
```

Push Tags: `git push --tags`

Share tags with remote repositories.

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

Delete Tags: `git tag -d`

Remove tags from local and remote repositories.

```
# Delete local tag
git tag -d v1.0
# Delete remote tag
git push origin --delete tag v1.0
# Alternative delete syntax
git push origin :refs/tags/v1.0
```

Git Configuration & Aliases

Customize Git behavior and create shortcuts for efficiency.

View Configuration: `git config --list`

Display current Git configuration settings.

```
# Show all config settings
git config --list
# Show global settings only
git config --global --list
# Show repository-specific settings
git config --local --list
# Show specific setting
git config user.name
```

Create Aliases: `git config alias`

Set up shortcuts for frequently used commands.

```
# git st = git status
git config --global alias.st status
# git co = git checkout
git config --global alias.co checkout
# git br = git branch
git config --global alias.br branch
# git ci = git commit
git config --global alias.ci commit
```

Advanced Aliases: Complex Commands

Create aliases for complex command combinations.

```
git config --global alias.lg "log --oneline --graph --all"
git config --global alias.unstage "reset HEAD --"
git config --global alias.last "log -1 HEAD"
git config --global alias.visual "!gitk"
```

Editor Configuration: `git config core.editor`

Set preferred text editor for commit messages and conflicts.

```
# VS Code
git config --global core.editor "code --wait"
# Vim
git config --global core.editor "vim"
# Nano
git config --global core.editor "nano"
```

Performance & Optimization

Repository Maintenance: `git gc`

Optimize repository performance and storage.

```
# Standard garbage collection
git gc
# More thorough optimization
git gc --aggressive
# Run only if needed
git gc --auto
# Check repository integrity
git fsck
```

Large File Handling: `git lfs`

Manage large binary files efficiently with Git LFS.

```
# Install LFS in repository
git lfs install
# Track PDF files with LFS
git lfs track "*.pdf"
# List files tracked by LFS
git lfs ls-files
# Migrate existing files
git lfs migrate import --include="*.zip"
```

Shallow Clones: Reducing Repository Size

Clone repositories with limited history for faster operations.

```
# Latest commit only
git clone --depth 1 https://github.com/user/repo.git
# Last 10 commits
git clone --depth 10 repo.git
# Convert shallow to full clone
git fetch --unshallow
```

Sparse Checkout: Working with Subdirectories

Check out only specific parts of large repositories.

```
git config core.sparseCheckout true
echo "src/*" > .git/info/sparse-checkout
# Apply sparse checkout
git read-tree -m -u HEAD
```

Git Installation & Setup

Install and configure Git for your development environment.

Package Managers: `apt`, `yum`, `brew`

Install Git using system package managers.

```
# Ubuntu/Debian
sudo apt install git
# CentOS/RHEL
sudo yum install git
# macOS with Homebrew
brew install git
# Windows with winget
winget install Git.Git
```

Download & Install: Official Installers

Use official Git installers for your platform.

```
# Download from https://git-
scm.com/downloads
# Verify installation
git --version
# Show Git executable path
which git
```

First-Time Setup: User Configuration

Configure Git with your identity for commits.

```
git config --global user.name
"Your Full Name"
git config --global user.email
"your.email@example.com"
git config --global
init.defaultBranch main
# Set merge behavior
git config --global pull.rebase
false
```

Git Workflows & Best Practices

Adopt proven workflows for team collaboration and project management.

Feature Branch Workflow

Standard workflow for feature development with isolated branches.

```
# Start from main branch
git checkout main
# Get latest changes
git pull origin main
# Create feature branch
git checkout -b feature/user-auth
# ... make changes and commits ...
# Push feature branch
git push -u origin feature/user-auth
# ... create pull request ...
```

Git Flow: Structured Branching Model

Systematic approach with dedicated branches for different purposes.

```
# Initialize Git Flow
git flow init
# Start feature
git flow feature start new-feature
# Finish feature
git flow feature finish new-feature
# Start release branch
git flow release start 1.0.0
```

Commit Message Conventions

Follow conventional commit format for clear project history.

```
# Format: <type>(<scope>): <subject>
git commit -m "feat(auth): add user login functionality"
git commit -m "fix(api): resolve null pointer exception"
git commit -m "docs(readme): update installation
instructions"
git commit -m "refactor(utils): simplify date formatting"
```

Atomic Commits: Best Practices

Create focused, single-purpose commits for better history.

```
# Stage changes interactively
git add -p
# Specific change
git commit -m "Add validation to email field"
# Avoid: git commit -m "Fix stuff" # Too vague
# Good:  git commit -m "Fix email validation regex
pattern"
```

Troubleshooting & Recovery

Resolve common Git issues and recover from mistakes.

Reflog: Recovery Tool

Use Git's reference log to recover lost commits.

```
# Show reference log
git reflog
# Show HEAD movements
git reflog show HEAD
# Recover lost commit
git checkout abc123
# Create branch from lost commit
git branch recovery-branch abc123
```

Corrupted Repository: Repair

Fix repository corruption and integrity issues.

```
# Check repository integrity
git fsck --full
# Aggressive cleanup
git gc --aggressive --prune=now
# Rebuild index if corrupted
rm .git/index; git reset
```

Authentication Issues

Resolve common authentication and permission problems.

```
# Use token
git remote set-url origin
https://token@github.com/user/repo.git
# Add SSH key to agent
ssh-add ~/.ssh/id_rsa
# Windows credential manager
git config --global credential.helper manager-core
```

Performance Issues: Debugging

Identify and resolve repository performance problems.

```
# Show repository size
git count-objects -vH
# Count total commits
git log --oneline | wc -l
# Count branches
git for-each-ref --format='%(refname:short)' | wc -l
```

Reference: This cheatsheet covers essential Git commands and modern practices for efficient version control and collaboration in software development workflows.