Git & GitHub Complete Learning Guide

🚀 A comprehensive tutorial covering Git version control and GitHub collaboration from beginner to advanced level.

🗂️ What's Included

This repository contains a complete Git and GitHub learning resource designed to take you from zero to hero in version control and collaborative development.

📖 Main Tutorial

- **Git & GitHub Complete Guide** - The comprehensive tutorial covering all aspects of Git and GitHub

📋 Additional Resources

- **DEV LOGS - GitHub.md** - Development logs and additional GitHub-specific notes

🎯 Learning Objectives

By completing this guide, you will:

- ✅ Master fundamental Git concepts and commands
- ✅ Understand branching, merging, and conflict resolution
- ✅ Learn advanced Git techniques (rebase, cherry-pick, bisect)
- ✅ Master GitHub workflows and collaboration
- ✅ Implement CI/CD with GitHub Actions
- ✅ Follow industry best practices for version control
- ✅ Troubleshoot common Git problems
- ✅ Optimize repository performance

📖 Tutorial Structure

The main guide is organized into 6 comprehensive parts:

Part 1: Git Fundamentals 🌿

- Git basics and core concepts
- Repository setup (`init`, `clone`, `remote`)
- Basic workflow (`add`, `commit`, `push`, `pull`)
- Understanding `.gitignore`
- Branching fundamentals

Part 2: Intermediate Git 🔧

- Advanced branching strategies
- Merge vs. Rebase
- Stashing techniques

- Diff and log formatting
- Merge conflict resolution

Part 3: Advanced Git Techniques ⚡

- Interactive rebase for history cleanup
- Cherry-picking commits
- Git reset strategies
- Recovery with reflog
- Bug hunting with bisect
- Tagging and releases
- Git hooks automation
- Commit message best practices

Part 4: GitHub Mastery 🐙

- GitHub CLI (`gh`) mastery
- GitHub Actions and CI/CD
- Pull requests and code review
- Repository management
- Secrets and environment variables

Part 5: Advanced Workflows 🚀

- Maintaining clean Git history
- Branching strategies (Git Flow, GitHub Flow, Trunk-based)
- Safe force push techniques

Part 6: Troubleshooting & Best Practices 🛠️

- Common Git problems and solutions
- Configuration best practices
- Performance optimization
- Repository maintenance

🎓 Learning Paths

🚀 Fast Track (2-3 weeks)

For experienced developers who want to quickly master Git/GitHub:

- Focus on Parts 1, 3, 4, and 6
- Skip basic explanations, focus on advanced techniques
- Emphasize GitHub Actions and team workflows

📚 Standard Track (4-6 weeks)

For developers with some Git experience:

- Complete all parts in order

- Practice each concept with real projects
- Set up CI/CD pipelines

🌱 Beginner Track (6-8 weeks)

For complete beginners to version control:

- Start with Part 1 and practice extensively
- Use visual tools alongside command line
- Focus on understanding concepts before advanced techniques

🛠️ Prerequisites

- Basic command line knowledge
- Text editor familiarity
- GitHub account (free)
- Git installed on your system

🚀 Quick Start

1. **Install Git**: Download from git-scm.com
2. **Create GitHub Account**: Sign up at github.com
3. **Open the Guide**: Start with git-github-complete-guide.md
4. **Practice**: Create a test repository and follow along

📊 Progress Tracking

Use this checklist to track your learning progress:

Fundamentals

- [ ] Understand Git concepts (repository, commit, branch)
- [ ] Can create and clone repositories
- [ ] Master basic workflow (add, commit, push, pull)
- [ ] Understand branching and merging
- [ ] Can resolve simple merge conflicts

Intermediate

- [ ] Comfortable with rebasing
- [ ] Can use stashing effectively
- [ ] Understand different merge strategies
- [ ] Can create and manage pull requests
- [ ] Set up basic GitHub Actions

Advanced

- [ ] Master interactive rebase
- [ ] Can recover from complex Git problems
- [ ] Implement comprehensive CI/CD pipelines

- ☐ Follow team branching strategies
- ☐ Optimize repository performance

🎯 Key Features

- 📖 **Comprehensive Coverage**: From basics to advanced techniques
- 💻 **Practical Examples**: Real-world scenarios and commands
- 🔧 **Hands-on Practice**: Step-by-step tutorials
- 🚀 **Modern Workflows**: GitHub Actions, CLI tools, best practices
- 🛠️ **Troubleshooting**: Common problems and solutions
- 📊 **Visual Learning**: ASCII diagrams and clear explanations
- 🎓 **Multiple Learning Paths**: Beginner to advanced tracks

📚 Additional Resources

- **Official Git Documentation**: git-scm.com/doc
- **GitHub Documentation**: docs.github.com
- **Interactive Git Tutorial**: learngitbranching.js.org
- **Pro Git Book**: git-scm.com/book
- **GitHub CLI**: cli.github.com

🤝 Contributing

This is a learning resource. If you find errors or have suggestions for improvements, feel free to:

1. Create an issue describing the problem
2. Submit a pull request with fixes
3. Share your learning experience

📄 License

This educational content is provided for learning purposes. Feel free to use, share, and adapt for educational use.

Happy Learning! 🚀

Master Git and GitHub to become a more effective developer and collaborator.

Complete Git & GitHub Guide: From Beginner to Advanced

🎯 Complete Git & GitHub Learning Resource

Welcome to the most comprehensive Git and GitHub tutorial! This guide takes you from absolute beginner to advanced user with practical examples, real-world scenarios, and hands-on commands you can try immediately.

Table of Contents

Beginner Topics

Intermediate Topics

Advanced Topics

Beginner Topics

What is Git and GitHub?

Git: The Version Control System

Git is a distributed version control system that tracks changes in files and coordinates work among multiple people. Think of it as a sophisticated "save" system that:

- **Tracks every change** to your files over time
- **Allows multiple people** to work on the same project

- **Maintains a complete history** of all modifications
- **Enables branching** for parallel development
- **Works offline** - no internet required for most operations

GitHub: The Cloud Platform

GitHub is a cloud-based hosting service for Git repositories that provides:

- **Remote storage** for your Git repositories
- **Collaboration tools** like pull requests and issues
- **Project management** features
- **CI/CD integration** with GitHub Actions
- **Social coding** features like following and starring

Real-World Analogy

```
Git = Your local photo album with detailed history
GitHub = Google Photos (cloud storage + sharing)
```

Why Use Git?

- ✅ **Never lose work** - complete history preserved
- ✅ **Collaborate safely** - merge changes from multiple contributors
- ✅ **Experiment freely** - create branches for new features
- ✅ **Track responsibility** - see who changed what and when
- ✅ **Rollback easily** - return to any previous state

Getting Started: init, add, commit

Setting Up Git (First Time)

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

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

# Check your configuration
git config --list
```

Creating Your First Repository

Step 1: Initialize a Repository

```
# Create a new directory
mkdir my-first-repo
cd my-first-repo

# Initialize Git repository
git init

# What happened?
# - Creates a hidden .git folder
# - This folder contains all Git metadata
# - Your directory is now a Git repository
```

Step 2: Create and Add Files

```
# Create a simple file
echo "# My First Project" > README.md
echo "console.log('Hello, Git!');" > app.js

# Check repository status
git status

# Output explanation:
# - Untracked files: Git sees them but isn't tracking changes
# - Working directory: Your current file state
# - Staging area: Files prepared for commit (empty now)
```

Step 3: Stage Files (git add)

```
# Add specific file to staging area
git add README.md

# Add all files
git add .

# Add files by pattern
git add *.js

# Check status after adding
git status

# Now files are "staged" - ready to be committed
```

Step 4: Commit Changes (git commit)

```
# Commit with message
git commit -m "Initial commit: Add README and app.js"
```

```
# Commit with detailed message (opens editor)
git commit

# Add and commit in one step (for tracked files)
git commit -am "Update existing files"
```

The Git Workflow Diagram

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

[README.md]     →   [README.md]   →   [Commit: abc123]
[app.js]        →   [app.js]      →   ["Initial commit"]
(modified)          (staged)          (permanent)
```

Practical Exercise

```
# Try this complete workflow
mkdir git-practice
cd git-practice
git init

# Create multiple files
echo "# Practice Project" > README.md
echo "body { margin: 0; }" > style.css
echo "<h1>Hello World</h1>" > index.html

# Stage and commit step by step
git add README.md
git commit -m "Add project README"

git add style.css index.html
git commit -m "Add basic HTML and CSS files"

# Check your commit history
git log --oneline
```

Checking Status and History

git status: Your Repository's Current State

```
# Basic status check
git status
```

```
# Short format (more concise)
git status -s

# Show ignored files too
git status --ignored
```

Understanding Status Output:

```
# Example output and meanings:
On branch main
Changes to be committed:          # Staged files (green)
  new file:   feature.js
  modified:   README.md

Changes not staged for commit:    # Modified but not staged (red)
  modified:   app.js
  deleted:    old-file.txt

Untracked files:                  # New files Git doesn't know about
  temp.log
  config.local.js
```

git log: Exploring Your History

Basic Log Commands:

```
# Standard log (detailed)
git log

# One line per commit
git log --oneline

# Show last 5 commits
git log -5

# Show commits with file changes
git log --stat

# Show actual code changes
git log -p
```

Advanced Log Formatting:

```
# Custom format with colors
git log --pretty=format:"%C(yellow)%h%C(reset) - %C(green)(%cr)%C(reset) %s
%C(blue)<%an>%C(reset)"
```

```
# Graph view (great for branches)
git log --graph --oneline --all

# Filter by author
git log --author="John Doe"

# Filter by date
git log --since="2 weeks ago"
git log --until="2023-12-01"

# Filter by commit message
git log --grep="fix"

# Show commits that modified specific file
git log -- README.md
```

Practical Examples

Scenario: Investigating a Bug

```
# Find all commits in the last week
git log --since="1 week ago" --oneline

# Find commits that mention "bug" or "fix"
git log --grep="bug\|fix" --oneline

# See what changed in a specific commit
git show abc123

# Find who last modified a specific file
git log -1 --pretty=format:"%an %ad" -- problematic-file.js
```

Scenario: Code Review Preparation

```
# Show all commits since last release tag
git log v1.0.0..HEAD --oneline

# Show detailed changes for review
git log --stat --since="1 week ago"

# Create a changelog
git log --pretty=format:"- %s (%h)" v1.0.0..HEAD
```

Working with Remote Repositories

Understanding Remotes

A remote is a version of your repository hosted elsewhere (like GitHub). You can have multiple remotes, but `origin` is the conventional name for your main remote.

Adding Remote Repositories

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

# Add a remote with SSH (more secure)
git remote add origin git@github.com:username/repository.git

# List all remotes
git remote -v

# Show detailed remote information
git remote show origin
```

Pushing Changes (git push)

```
# Push to remote for the first time
git push -u origin main
# -u sets upstream tracking (only needed once)

# Regular push (after upstream is set)
git push

# Push specific branch
git push origin feature-branch

# Push all branches
git push --all

# Push tags
git push --tags
```

Pulling Changes (git pull)

```
# Pull latest changes from remote
git pull

# Pull from specific remote and branch
git pull origin main

# Pull with rebase (cleaner history)
git pull --rebase

# Fetch without merging (safer)
```

```
git fetch
git merge origin/main
```

Real-World Workflow Example

Scenario: Working on a Team Project

```
# Morning routine: Get latest changes
git pull origin main

# Work on your feature
echo "new feature code" >> feature.js
git add feature.js
git commit -m "Add new feature functionality"

# Before pushing: Check for new remote changes
git fetch
git status  # Check if your branch is behind

# If behind, pull latest changes
git pull --rebase origin main

# Push your changes
git push origin main
```

Scenario: Handling Push Rejection

```
# If push is rejected (someone else pushed first)
git push origin main
# Error: Updates were rejected

# Solution 1: Pull and merge
git pull origin main
git push origin main

# Solution 2: Pull with rebase (cleaner)
git pull --rebase origin main
git push origin main
```

Ignoring Files with .gitignore

Why Use .gitignore?

Some files shouldn't be tracked by Git:

- **Build artifacts** (compiled code, dist folders)

- **Dependencies** (node_modules, vendor)
- **Environment files** (API keys, local configs)
- **IDE files** (editor settings)
- **Temporary files** (logs, cache)

Creating .gitignore

```
# Create .gitignore file
touch .gitignore

# Edit with your preferred editor
nano .gitignore
```

Common .gitignore Patterns

Basic Syntax:

```
# Comments start with #

# Ignore specific file
secret.txt

# Ignore all files with extension
*.log
*.tmp

# Ignore directory
node_modules/
build/

# Ignore files in any directory
**/temp

# Ignore files except specific ones
*.env
!.env.example

# Ignore files only in root
/config.local.js
```

Language-Specific Examples:

Node.js Project:

```
# Dependencies
node_modules/
npm-debug.log*
```

```
yarn-debug.log*
yarn-error.log*

# Environment variables
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

# Build output
build/
dist/

# IDE
.vscode/
.idea/

# OS
.DS_Store
Thumbs.db
```

Python Project:

```
# Byte-compiled / optimized files
__pycache__/
*.py[cod]
*$py.class

# Virtual environments
venv/
env/
ENV/

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

# Testing
.pytest_cache/
.coverage

# Distribution / packaging
build/
dist/
*.egg-info/
```

Advanced .gitignore Techniques

```
# Check if file is ignored
git check-ignore -v filename.txt

# Add already tracked file to .gitignore
echo "config.local.js" >> .gitignore
git rm --cached config.local.js
git commit -m "Stop tracking config.local.js"

# Temporarily ignore tracked file
git update-index --skip-worktree config.js

# Undo temporary ignore
git update-index --no-skip-worktree config.js
```

Global .gitignore

```
# Create global gitignore for all repositories
git config --global core.excludesfile ~/.gitignore_global

# Add common patterns to global ignore
echo ".DS_Store" >> ~/.gitignore_global
echo "Thumbs.db" >> ~/.gitignore_global
echo "*.swp" >> ~/.gitignore_global
```

Creating and Cloning Repositories

Creating Repositories on GitHub

Method 1: GitHub Web Interface

1. Go to github.com and sign in
2. Click "+" → "New repository"
3. Fill in repository details:
    - **Name**: my-awesome-project
    - **Description**: A brief description
    - **Visibility**: Public or Private
    - **Initialize**: Add README, .gitignore, license
4. Click "Create repository"

Method 2: GitHub CLI

```
# Install GitHub CLI first
# Then create repository
gh repo create my-awesome-project --public --description "My awesome project"
```

```
# Create with README and .gitignore
gh repo create my-awesome-project --public --add-readme --gitignore node

# Create private repository
gh repo create my-awesome-project --private
```

Cloning Repositories

Basic Cloning:

```
# Clone with HTTPS
git clone https://github.com/username/repository.git

# Clone with SSH (recommended for frequent use)
git clone git@github.com:username/repository.git

# Clone to specific directory
git clone https://github.com/username/repository.git my-local-name

# Clone specific branch
git clone -b develop https://github.com/username/repository.git

# Shallow clone (faster, less history)
git clone --depth 1 https://github.com/username/repository.git
```

After Cloning:

```
# Navigate to cloned repository
cd repository

# Check remote configuration
git remote -v

# Check current branch
git branch

# See all branches (including remote)
git branch -a
```

Connecting Local Repository to GitHub

Scenario: You have local code, want to put it on GitHub

```
# Step 1: Create repository on GitHub (empty, no README)
# Step 2: In your local directory
git init
```

```
git add .
git commit -m "Initial commit"

# Step 3: Add remote and push
git remote add origin https://github.com/username/repository.git
git branch -M main  # Rename branch to main if needed
git push -u origin main
```

SSH Setup for GitHub (Recommended)

```
# Generate SSH key
ssh-keygen -t ed25519 -C "your.email@example.com"

# Start SSH agent
eval "$(ssh-agent -s)"

# Add key to agent
ssh-add ~/.ssh/id_ed25519

# Copy public key to clipboard (macOS)
pbcopy < ~/.ssh/id_ed25519.pub

# Copy public key to clipboard (Linux)
cat ~/.ssh/id_ed25519.pub | xclip -selection clipboard

# Add to GitHub:
# 1. Go to GitHub Settings → SSH and GPG keys
# 2. Click "New SSH key"
# 3. Paste your public key

# Test connection
ssh -T git@github.com
```

Branching Basics

What are Branches?

Branches allow you to diverge from the main line of development and work on features, experiments, or fixes in isolation.

```
Main branch:    A---B---C---F---G
                     \
                      D---E  (feature branch)
```

Basic Branch Commands

```
# List all branches
git branch

# List all branches (including remote)
git branch -a

# Create new branch
git branch feature-login

# Create and switch to new branch
git checkout -b feature-login
# or (newer syntax)
git switch -c feature-login

# Switch to existing branch
git checkout main
# or
git switch main

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

# Force delete branch
git branch -D feature-login
```

Modern Git: switch vs checkout

git switch (Git 2.23+) - Recommended for branch operations:

```
# Switch to existing branch
git switch main
git switch feature-branch

# Create and switch to new branch
git switch -c new-feature

# Switch to previous branch
git switch -
```

git checkout - Still useful for other operations:

```
# Checkout specific commit
git checkout abc123

# Checkout specific file from another branch
git checkout main -- README.md
```

```
# Checkout file from specific commit
git checkout abc123 -- src/app.js
```

Branch Workflow Examples

Feature Development Workflow:

```
# Start from main branch
git switch main
git pull origin main

# Create feature branch
git switch -c feature-user-authentication

# Work on feature
echo "auth code" > auth.js
git add auth.js
git commit -m "Add user authentication"

# More commits...
echo "more auth code" >> auth.js
git commit -am "Improve authentication security"

# Push feature branch
git push -u origin feature-user-authentication

# When feature is complete, merge back to main
git switch main
git merge feature-user-authentication
git push origin main

# Clean up
git branch -d feature-user-authentication
git push origin --delete feature-user-authentication
```

Hotfix Workflow:

```
# Critical bug found in production
git switch main
git pull origin main

# Create hotfix branch
git switch -c hotfix-critical-bug

# Fix the bug
echo "bug fix" > fix.js
git add fix.js
git commit -m "Fix critical security vulnerability"
```

```
# Push and merge immediately
git push -u origin hotfix-critical-bug
# (Create PR for review, then merge)

git switch main
git merge hotfix-critical-bug
git push origin main

# Clean up
git branch -d hotfix-critical-bug
git push origin --delete hotfix-critical-bug
```

Branch Naming Conventions

```
# Feature branches
feature/user-authentication
feature/payment-integration
feat/shopping-cart

# Bug fix branches
bugfix/login-error
fix/memory-leak

# Hotfix branches
hotfix/security-patch
hotfix/critical-bug

# Release branches
release/v1.2.0
release/2023-12-01

# Experimental branches
experiment/new-ui
spike/performance-test
```

🟡 Intermediate Topics

Merge vs Rebase

Understanding the Difference

Merge: Combines branches by creating a new commit that has two parents **Rebase**: Replays commits from one branch onto another, creating a linear history

Visual Comparison

Before (both scenarios):

```
Main:     A---B---C
               \
               D---E  (feature)
```

After Merge:

```
Main:     A---B---C-------F  (merge commit)
               \       /
               D---E---   (feature)
```

After Rebase:

```
Main:     A---B---C---D'---E'  (linear history)
```

Merge Commands and Examples

Basic Merge:

```
# Switch to target branch
git switch main

# Merge feature branch
git merge feature-branch

# Push merged changes
git push origin main
```

Merge Types:

```
# Fast-forward merge (when possible)
git merge feature-branch

# Force merge commit (even when fast-forward possible)
git merge --no-ff feature-branch

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

Rebase Commands and Examples

Basic Rebase:

```
# Switch to feature branch
git switch feature-branch

# Rebase onto main
git rebase main

# If conflicts occur, resolve them, then:
git add .
git rebase --continue

# Switch to main and merge (fast-forward)
git switch main
git merge feature-branch
```

Interactive Rebase:

```
# Rebase last 3 commits interactively
git rebase -i HEAD~3

# Rebase from specific commit
git rebase -i abc123

# Interactive rebase options:
# pick = use commit as-is
# reword = change commit message
# edit = stop to amend commit
# squash = combine with previous commit
# fixup = like squash but discard message
# drop = remove commit
```

When to Use Each

Use Merge When:

- ✅ Working on public/shared branches
- ✅ Want to preserve exact history
- ✅ Feature branch has multiple contributors
- ✅ Want to see when features were integrated

Use Rebase When:

- ✅ Working on private/local branches
- ✅ Want clean, linear history
- ✅ Preparing feature branch for merge
- ✅ Cleaning up commit history

Practical Examples

Scenario 1: Feature Development with Rebase

```
# Create and work on feature
git switch -c feature-api
echo "api code" > api.js
git add api.js
git commit -m "Add API endpoint"

echo "more api code" >> api.js
git commit -am "Improve API validation"

# Meanwhile, main branch has new commits
# Before merging, rebase to get latest changes
git fetch origin
git rebase origin/main

# Now merge with clean history
git switch main
git merge feature-api  # Fast-forward merge
```

Scenario 2: Team Integration with Merge

```
# Multiple developers worked on feature
# Preserve their individual contributions
git switch main
git merge --no-ff feature-team-effort

# This creates a merge commit showing:
# - When feature was integrated
# - Who integrated it
# - All individual commits preserved
```

Handling Rebase Conflicts

```
# Start rebase
git rebase main

# If conflicts occur:
# 1. Git pauses and shows conflicted files
# 2. Edit files to resolve conflicts
# 3. Stage resolved files
git add conflicted-file.js

# 4. Continue rebase
git rebase --continue

# If you want to abort rebase
git rebase --abort
```

```
# If you want to skip a commit during rebase
git rebase --skip
```

Git Stash Management

What is Git Stash?

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

Basic Stash Operations

```
# Stash current changes
git stash

# Stash with custom message
git stash push -m "Work in progress on login feature"

# Stash including untracked files
git stash -u

# Stash including ignored files
git stash -a

# List all stashes
git stash list

# Apply most recent stash
git stash apply

# Apply and remove most recent stash
git stash pop

# Apply specific stash
git stash apply stash@{2}

# Remove specific stash
git stash drop stash@{1}

# Clear all stashes
git stash clear
```

Advanced Stash Management

Partial Stashing:

```
# Stash only specific files
git stash push -m "Stash only CSS changes" -- style.css theme.css

# Interactive stashing (choose what to stash)
git stash -p

# Stash only staged changes
git stash --staged
```

Stash Inspection:

```
# Show stash contents
git stash show

# Show detailed stash contents
git stash show -p

# Show specific stash
git stash show stash@{1} -p

# List stashes with details
git stash list --stat
```

Creating Branches from Stashes:

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

# This is useful when:
# - Stash conflicts with current branch
# - Want to continue work in separate branch
```

Real-World Stash Scenarios

Scenario 1: Emergency Bug Fix

```
# Working on feature, but urgent bug reported
echo "half-finished feature" > feature.js
git add feature.js

# Stash work in progress
git stash push -m "WIP: new feature development"

# Switch to main and fix bug
git switch main
echo "bug fix" > bugfix.js
```

```
git add bugfix.js
git commit -m "Fix critical bug"

# Return to feature work
git switch feature-branch
git stash pop

# Continue working on feature
echo "completed feature" >> feature.js
git commit -am "Complete new feature"
```

Scenario 2: Switching Contexts

```
# Working on multiple features
# Feature A - half done
echo "feature A progress" > featureA.js
git stash push -m "Feature A: authentication logic"

# Feature B - different approach needed
echo "feature B attempt 1" > featureB.js
git stash push -m "Feature B: first attempt"

# Try different approach for Feature B
echo "feature B attempt 2" > featureB.js
git add featureB.js
git commit -m "Feature B: implement with new approach"

# Go back to Feature A
git stash list
git stash apply stash@{1}  # Feature A stash
```

Scenario 3: Code Review Preparation

```
# Clean up working directory before review
git stash -u  # Stash everything including untracked

# Review code with clean state
git log --oneline -10
git diff HEAD~5

# Restore work after review
git stash pop
```

Stash Best Practices

```
# Always use descriptive messages
git stash push -m "Login form validation - needs testing"
```

```
# Regularly clean up old stashes
git stash list
git stash drop stash@{5}  # Remove old stashes

# Use stash show before applying
git stash show stash@{0} -p  # Review what you're about to apply

# Create aliases for common stash operations
git config --global alias.sl 'stash list'
git config --global alias.sp 'stash pop'
git config --global alias.ss 'stash show -p'
```

Understanding Differences

git diff: Comparing Changes

Basic Diff Commands:

```
# Show unstaged changes
git diff

# Show staged changes
git diff --staged
# or
git diff --cached

# Show all changes (staged + unstaged)
git diff HEAD

# Compare specific files
git diff README.md
git diff --staged app.js
```

Comparing Branches and Commits

```
# Compare two branches
git diff main..feature-branch

# Compare current branch with main
git diff main

# Compare specific commits
git diff abc123..def456

# Compare with previous commit
git diff HEAD~1
git diff HEAD~2..HEAD
```

```
# Show changes in specific commit
git show abc123
```

Advanced Diff Options

Formatting and Filtering:

```
# Show only file names that changed
git diff --name-only

# Show file names with status (A=added, M=modified, D=deleted)
git diff --name-status

# Show statistics (lines added/removed)
git diff --stat

# Ignore whitespace changes
git diff -w

# Show word-level differences
git diff --word-diff

# Show differences with more context lines
git diff -U10  # 10 lines of context instead of default 3
```

File and Directory Filtering:

```
# Compare specific directory
git diff main..feature -- src/

# Compare specific file types
git diff main..feature -- "*.js"

# Exclude specific files
git diff main..feature -- . ':(exclude)package-lock.json'

# Compare only staged changes in specific files
git diff --staged -- src/app.js src/utils.js
```

Practical Diff Examples

Code Review Preparation:

```
# See all changes in your feature branch
git diff main..HEAD
```

```
# Get summary of changes
git diff main..HEAD --stat

# See changes in specific files
git diff main..HEAD -- src/

# Generate patch file for review
git diff main..HEAD > feature-changes.patch
```

Debugging Workflow:

```
# Compare with last working version
git diff HEAD~5..HEAD

# See what changed in specific commit
git show abc123

# Compare two versions of same file
git diff HEAD~3:app.js HEAD:app.js

# See changes between tags
git diff v1.0.0..v1.1.0
```

Pre-commit Review:

```
# Review what you're about to commit
git diff --staged

# Review specific files before committing
git diff --staged -- src/critical-file.js

# Make sure no debug code is included
git diff --staged | grep -i "console.log\|debugger\|TODO"
```

External Diff Tools

```
# Configure external diff tool (VS Code)
git config --global diff.tool vscode
git config --global difftool.vscode.cmd 'code --wait --diff $LOCAL $REMOTE'

# Use external tool
git difftool
git difftool main..feature-branch

# Configure for other editors
# Vim
```

```
git config --global diff.tool vimdiff

# Beyond Compare
git config --global diff.tool bc3
git config --global difftool.bc3.path "c:/Program Files/Beyond Compare
3/bcomp.exe"
```

Advanced Logging

Pretty Formatting

```
# Custom format with colors and symbols
git log --pretty=format:"%C(red)%h%C(reset) - %C(green)(%cr)%C(reset) %s %C(blue)
<%an>%C(reset)%C(yellow)%d%C(reset)"

# One-line with graph
git log --oneline --graph --all

# Detailed with file changes
git log --stat --graph

# Show actual code changes
git log -p --max-count=5
```

Filtering and Searching

By Author and Date:

```
# Commits by specific author
git log --author="John Doe"
git log --author="john@example.com"

# Multiple authors
git log --author="John\|Jane"

# Date ranges
git log --since="2023-01-01"
git log --until="2023-12-31"
git log --since="2 weeks ago"
git log --since="yesterday"

# Combine filters
git log --author="John" --since="1 month ago" --oneline
```

By Content and Message:

```
# Search commit messages
git log --grep="fix"
git log --grep="bug\|error" --oneline

# Search code content
git log -S "function_name"  # When function was added/removed
git log -G "regex_pattern"  # When regex pattern changed

# Search in specific files
git log -- path/to/file.js
git log --follow -- renamed-file.js  # Follow renames
```

By File Changes:

```
# Commits that modified specific files
git log -- src/app.js
git log -- "*.css"

# Commits that added or removed files
git log --diff-filter=A  # Added files
git log --diff-filter=D  # Deleted files
git log --diff-filter=M  # Modified files
git log --diff-filter=R  # Renamed files
```

Advanced Log Techniques

Branch and Merge Analysis:

```
# Show commits unique to branch
git log main..feature-branch

# Show commits in main but not in feature
git log feature-branch..main

# Show commits in either branch but not both
git log main...feature-branch

# Show merge commits only
git log --merges

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

Performance and Statistics:

```

```

```
# Show commit frequency by author
git shortlog -sn

# Show commit activity by date
git log --format="%ad" --date=short | sort | uniq -c

# Show largest commits
git log --stat | grep -E "files? changed" | sort -nr

# Show commits with most file changes
git log --pretty=format:"%h %s" --stat | grep -B1 "files changed"
```

Custom Log Aliases

```
# Create useful aliases
git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset
-%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

git config --global alias.lol "log --graph --decorate --pretty=oneline --abbrev-
commit"

git config --global alias.lola "log --graph --decorate --pretty=oneline --abbrev-
commit --all"

git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph
--date=short"

# Usage
git lg
git lol
git hist
```

Practical Logging Scenarios

Release Notes Generation:

```
# Changes since last release
git log v1.0.0..HEAD --pretty=format:"- %s (%h)"

# Group by type (if using conventional commits)
git log v1.0.0..HEAD --pretty=format:"%s" | grep "^feat:" | sed 's/feat: /- /'
git log v1.0.0..HEAD --pretty=format:"%s" | grep "^fix:" | sed 's/fix: /- /'
```

Code Archaeology:

```
# Find when bug was introduced
git log -S "problematic_code" --source --all

# See evolution of specific function
git log -L :function_name:file.js

# Find commits that touched specific lines
git log -L 15,25:src/app.js
```

Team Productivity Analysis:

```
# Commits per author in date range
git log --since="1 month ago" --pretty=format:"%an" | sort | uniq -c | sort -nr

# Most active files
git log --name-only --pretty=format: | grep -v "^$" | sort | uniq -c | sort -nr |
head -10

# Commit activity by day of week
git log --date=format:"%A" --pretty=format:"%ad" | sort | uniq -c
```

Resolving Merge Conflicts

Understanding Merge Conflicts

Conflicts occur when Git can't automatically merge changes because:

- Same lines were modified in both branches
- File was deleted in one branch, modified in another
- File was renamed differently in both branches

Conflict Markers Explained

```
// Example conflict in code file
<<<<<<< HEAD (current branch)
function calculateTotal(price, tax) {
    return price + (price * tax);
}
=======
function calculateTotal(price, taxRate, discount) {
    const subtotal = price - discount;
    return subtotal + (subtotal * taxRate);
}
>>>>>>> feature-branch
```

Conflict Markers:

- `<<<<<<< HEAD`: Start of current branch changes
- `=======`: Separator between conflicting changes
- `>>>>>>> branch-name`: End of incoming branch changes

Resolving Conflicts Step by Step

Step 1: Identify Conflicts

```
# Attempt merge
git merge feature-branch

# Git shows conflict message
# Auto-merging file.js
# CONFLICT (content): Merge conflict in file.js
# Automatic merge failed; fix conflicts and then commit the result.

# Check status
git status
# Shows files with conflicts
```

Step 2: Resolve Conflicts

```
# Open conflicted files in editor
# Remove conflict markers
# Choose which changes to keep
# Or combine both changes

# Example resolution:
function calculateTotal(price, taxRate, discount = 0) {
    const subtotal = price - discount;
    return subtotal + (subtotal * taxRate);
}
```

Step 3: Mark as Resolved

```
# Stage resolved files
git add file.js

# Check status
git status
# Should show "All conflicts fixed but you are still merging"

# Complete the merge
git commit
# Git opens editor with default merge message
# Save and close to complete merge
```

Conflict Resolution Tools

Built-in Merge Tool:

```
# Use Git's built-in merge tool
git mergetool

# Configure merge tool (VS Code)
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'

# Configure merge tool (Vim)
git config --global merge.tool vimdiff

# Configure merge tool (Beyond Compare)
git config --global merge.tool bc3
git config --global mergetool.bc3.path "c:/Program Files/Beyond Compare
3/bcomp.exe"
```

Manual Resolution Strategies:

```
# Accept all changes from current branch
git checkout --ours conflicted-file.js
git add conflicted-file.js

# Accept all changes from incoming branch
git checkout --theirs conflicted-file.js
git add conflicted-file.js

# For entire merge, accept one side
git merge -X ours feature-branch    # Prefer current branch
git merge -X theirs feature-branch  # Prefer incoming branch
```

Advanced Conflict Scenarios

Binary File Conflicts:

```
# Binary files can't be merged automatically
# Choose which version to keep
git checkout --ours image.png
# or
git checkout --theirs image.png

# Then stage the chosen version
git add image.png
```

Rename Conflicts:

```
# When same file renamed differently
# Git shows both versions
# Choose one name and delete the other
mv file_version1.js final_name.js
git rm file_version2.js
git add final_name.js
```

Complex Multi-file Conflicts:

```
# See all conflicted files
git diff --name-only --diff-filter=U

# Resolve each file individually
for file in $(git diff --name-only --diff-filter=U); do
    echo "Resolving $file"
    # Open in editor, resolve, then:
    git add "$file"
done
```

Preventing Conflicts

Best Practices:

```
# Keep branches up to date
git switch feature-branch
git rebase main  # Regularly rebase on main

# Use smaller, focused commits
git commit -m "Add user validation" -- user.js
git commit -m "Add user tests" -- user.test.js

# Communicate with team about file changes
# Use .gitattributes for merge strategies
echo "*.generated merge=ours" >> .gitattributes
```

Rebase vs Merge for Conflict Handling:

```
# Rebase: Resolve conflicts commit by commit
git rebase main
# Resolve conflict in first commit
git add .
git rebase --continue
# Resolve conflict in second commit
```

```
git add .
git rebase --continue

# Merge: Resolve all conflicts at once
git merge main
# Resolve all conflicts
git add .
git commit
```

Aborting and Retrying

```
# Abort merge and return to pre-merge state
git merge --abort

# Abort rebase and return to pre-rebase state
git rebase --abort

# Reset to specific commit if needed
git reset --hard HEAD~1

# Try different merge strategy
git merge -s recursive -X patience feature-branch
git merge -s recursive -X ignore-space-change feature-branch
```

Pull Requests and Code Review

Understanding Pull Requests

A Pull Request (PR) is a method of submitting contributions to a project. It's a request to "pull" your changes into the main codebase after review.

Benefits:

- ✅ Code review before merging
- ✅ Discussion and collaboration
- ✅ Automated testing integration
- ✅ Documentation of changes
- ✅ Knowledge sharing

Creating Pull Requests

Method 1: GitHub Web Interface

```
# 1. Push your feature branch
git push origin feature-user-auth

# 2. Go to GitHub repository
```

```
# 3. Click "Compare & pull request" button
# 4. Fill in PR details:
#    - Title: Clear, descriptive
#    - Description: What, why, how
#    - Reviewers: Team members
#    - Labels: bug, feature, etc.
#    - Milestone: Release version
```

Method 2: GitHub CLI

```
# Create PR from command line
gh pr create --title "Add user authentication" --body "Implements login/logout
functionality with JWT tokens"

# Create PR with reviewers
gh pr create --title "Fix memory leak" --reviewer john,jane --assignee @me

# Create draft PR
gh pr create --draft --title "WIP: New dashboard design"
```

PR Best Practices

Good PR Title Examples:

```
✅ Add user authentication with JWT
✅ Fix memory leak in image processing
✅ Update dependencies to latest versions
✅ Refactor database connection logic

❌ Fix stuff
❌ Updates
❌ Changes
❌ WIP
```

Good PR Description Template:

```
## What

Brief description of changes

## Why

Reason for the changes (issue link, business requirement)

## How

Technical approach taken
```

```
## Testing

- [ ] Unit tests added/updated
- [ ] Manual testing completed
- [ ] Edge cases considered

## Screenshots (if UI changes)

[Before/After images]

## Checklist

- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Documentation updated
- [ ] No breaking changes (or documented)

Closes #123
```

Code Review Process

As a Reviewer:

```
# Checkout PR locally for testing
gh pr checkout 123

# Or manually:
git fetch origin pull/123/head:pr-123
git switch pr-123

# Test the changes
npm test
npm run build

# Review code changes
git diff main..HEAD
```

Review Guidelines:

- 🔍 **Functionality**: Does it work as intended?
- 🏗️ **Architecture**: Is the approach sound?
- 🧹 **Code Quality**: Is it readable and maintainable?
- 🔒 **Security**: Any security implications?
- ⚡ **Performance**: Any performance impacts?
- 📚 **Documentation**: Is it adequately documented?
- 🧪 **Testing**: Are there sufficient tests?

Review Comments Types:

```
# Blocking issues (must fix)

**Issue**: This could cause a memory leak

# Suggestions (nice to have)

**Suggestion**: Consider using const instead of let here

# Questions (seeking clarification)

**Question**: Why did you choose this approach over X?

# Praise (positive feedback)

**Nice**: Great error handling here!
```

Handling Review Feedback

Making Changes:

```
# Make requested changes
echo "improved code" >> file.js
git add file.js
git commit -m "Address review feedback: improve error handling"

# Push updates
git push origin feature-branch

# PR automatically updates with new commits
```

Responding to Comments:

```
# In GitHub PR comments:

✅ Fixed in commit abc123
✅ Good point, updated the documentation
✅ Refactored as suggested
❓ Could you clarify what you mean by...?
💭 I chose this approach because...
```

Advanced PR Workflows

Draft PRs for Early Feedback:

```
# Create draft PR for work in progress
gh pr create --draft --title "WIP: Redesign user dashboard"

# Convert to ready for review
gh pr ready 123
```

Auto-merge Setup:

```
# Enable auto-merge after approvals
gh pr merge 123 --auto --squash

# Set up branch protection rules:
# - Require PR reviews
# - Require status checks
# - Require up-to-date branches
# - Restrict pushes to main
```

PR Templates:

```
# Create PR template
mkdir .github
cat > .github/pull_request_template.md << 'EOF'
## Description
Brief description of changes

## Type of Change
- [ ] Bug fix
- [ ] New feature
- [ ] Breaking change
- [ ] Documentation update

## Testing
- [ ] Tests added/updated
- [ ] Manual testing completed

## Checklist
- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Documentation updated
EOF
```

GitHub PR Features

Linking Issues:

```
# In PR description, use keywords:

Closes #123
Fixes #456
Resolves #789

# These automatically close issues when PR merges
```

Review Requests:

```
# Request review from specific users
gh pr create --reviewer john,jane

# Request review from teams
gh pr create --reviewer @org/frontend-team

# Re-request review after changes
gh pr review 123 --request-changes
```

Status Checks:

```
# .github/workflows/pr-checks.yml
name: PR Checks
on:
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run tests
        run: npm test
      - name: Check code style
        run: npm run lint
```

Forking and Upstream Management

Understanding Forks

A fork is a personal copy of someone else's repository. It allows you to:

- Experiment without affecting the original
- Contribute to projects you don't have write access to
- Maintain your own version of a project

Forking Workflow

Step 1: Fork the Repository

```
# Fork via GitHub web interface
# Click "Fork" button on repository page

# Or use GitHub CLI
gh repo fork owner/repository

# Fork and clone in one step
gh repo fork owner/repository --clone
```

Step 2: Clone Your Fork

```
# Clone your fork
git clone https://github.com/yourusername/repository.git
cd repository

# Check remotes
git remote -v
# origin  https://github.com/yourusername/repository.git (fetch)
# origin  https://github.com/yourusername/repository.git (push)
```

Step 3: Add Upstream Remote

```
# Add original repository as upstream
git remote add upstream https://github.com/originalowner/repository.git

# Verify remotes
git remote -v
# origin    https://github.com/yourusername/repository.git (fetch)
# origin    https://github.com/yourusername/repository.git (push)
# upstream  https://github.com/originalowner/repository.git (fetch)
# upstream  https://github.com/originalowner/repository.git (push)
```

Keeping Fork in Sync

Regular Sync Process:

```
# Fetch latest changes from upstream
git fetch upstream

# Switch to main branch
git switch main
```

```
# Merge upstream changes
git merge upstream/main

# Push updates to your fork
git push origin main
```

Automated Sync with GitHub CLI:

```
# Sync fork with upstream
gh repo sync yourusername/repository

# Sync specific branch
gh repo sync yourusername/repository --branch develop
```

Sync Script for Regular Use:

```
#!/bin/bash
# save as sync-fork.sh

echo "Syncing fork with upstream..."
git fetch upstream
git switch main
git merge upstream/main
git push origin main
echo "Fork synced successfully!"

# Make executable
chmod +x sync-fork.sh

# Use it
./sync-fork.sh
```

Contributing to Upstream

Complete Contribution Workflow:

```
# 1. Sync your fork
git fetch upstream
git switch main
git merge upstream/main
git push origin main

# 2. Create feature branch
git switch -c fix-typo-in-docs

# 3. Make changes
echo "Fixed typo" >> README.md
```

```
git add README.md
git commit -m "Fix typo in installation instructions"

# 4. Push to your fork
git push origin fix-typo-in-docs

# 5. Create PR to upstream
gh pr create --repo originalowner/repository --title "Fix typo in docs"
```

Managing Multiple Remotes

Working with Multiple Forks:

```
# Add multiple remotes
git remote add upstream https://github.com/original/repo.git
git remote add colleague https://github.com/colleague/repo.git
git remote add company https://github.com/company/repo.git

# Fetch from all remotes
git fetch --all

# See all branches from all remotes
git branch -a

# Create branch from colleague's work
git switch -c test-colleague-feature colleague/feature-branch
```

Pushing to Different Remotes:

```
# Push to your fork
git push origin feature-branch

# Push to company fork
git push company feature-branch

# Push to specific remote and branch
git push upstream feature-branch:proposed-feature
```

Advanced Fork Management

Rebasing Fork on Upstream:

```
# Instead of merging, rebase for cleaner history
git fetch upstream
git switch main
```

```
git rebase upstream/main
git push --force-with-lease origin main
```

Cherry-picking from Upstream:

```
# Pick specific commits from upstream
git fetch upstream
git cherry-pick upstream/main~3  # Pick 3rd commit from top
git push origin main
```

Handling Diverged Forks:

```
# When your fork has diverged significantly
git fetch upstream
git switch main

# Option 1: Hard reset to upstream (loses your changes)
git reset --hard upstream/main
git push --force-with-lease origin main

# Option 2: Create new branch from upstream
git switch -c main-updated upstream/main
git push origin main-updated

# Option 3: Interactive rebase to clean history
git rebase -i upstream/main
```

Fork-specific GitHub Features

Comparing Forks:

```
# Compare your fork with upstream
# GitHub URL: https://github.com/original/repo/compare/main...yourusername:main

# Using GitHub CLI
gh pr create --repo original/repo --head yourusername:feature --base main
```

Fork Network Insights:

```
# See fork network on GitHub
# Go to: https://github.com/original/repo/network

# List forks with GitHub CLI
gh repo list original --fork
```

Maintaining Fork Visibility:

```
# Keep fork active with regular commits
# GitHub archives inactive forks

# Add fork-specific documentation
echo "# My Fork" > FORK_README.md
echo "This fork includes:" >> FORK_README.md
echo "- Custom configuration" >> FORK_README.md
echo "- Additional features" >> FORK_README.md
git add FORK_README.md
git commit -m "Add fork-specific documentation"
git push origin main
```

Advanced Topics

Interactive Rebase

What is Interactive Rebase?

Interactive rebase allows you to modify commit history by:

- Reordering commits
- Combining commits (squashing)
- Editing commit messages
- Splitting commits
- Removing commits

Basic Interactive Rebase

```
# Rebase last 3 commits interactively
git rebase -i HEAD~3

# Rebase from specific commit
git rebase -i abc123

# Rebase entire branch onto main
git rebase -i main
```

Interactive Rebase Commands:

```
pick abc123 Add user authentication
pick def456 Fix login bug
pick ghi789 Update documentation
```

```
# Commands available:
# pick (p) = use commit as-is
# reword (r) = use commit, but edit message
# edit (e) = use commit, but stop for amending
# squash (s) = use commit, but meld into previous commit
# fixup (f) = like squash, but discard commit message
# exec (x) = run command (the rest of the line) using shell
# break (b) = stop here (continue rebase later with 'git rebase --continue')
# drop (d) = remove commit
# label (l) = label current HEAD with a name
# reset (t) = reset HEAD to a label
# merge (m) = create a merge commit using the original merge commit's message
```

Common Rebase Scenarios

Scenario 1: Squashing Commits

```
# Before: Multiple small commits
git log --oneline -4
# abc123 Fix typo
# def456 Update docs
# ghi789 Fix another typo
# jkl012 Add feature

# Interactive rebase to squash
git rebase -i HEAD~4

# Change to:
pick jkl012 Add feature
squash abc123 Fix typo
squash def456 Update docs
squash ghi789 Fix another typo

# Result: One clean commit with all changes
```

Scenario 2: Reordering Commits

```
# Reorder commits logically
git rebase -i HEAD~3

# Change from:
pick abc123 Add tests
pick def456 Add feature
pick ghi789 Fix feature bug

# To:
pick def456 Add feature
pick ghi789 Fix feature bug
pick abc123 Add tests
```

Scenario 3: Editing Commit Messages

```
# Fix commit messages
git rebase -i HEAD~2

# Change:
pick abc123 WIP stuff
pick def456 fix bug

# To:
reword abc123 WIP stuff
reword def456 fix bug

# Git will prompt for new messages:
# "Implement user authentication system"
# "Fix login validation bug"
```

Scenario 4: Splitting Commits

```
# Split a large commit
git rebase -i HEAD~1

# Change:
pick abc123 Add feature and fix bugs

# To:
edit abc123 Add feature and fix bugs

# When rebase stops:
git reset HEAD~1  # Unstage all changes
git add feature.js
git commit -m "Add new feature"
git add bugfix.js
git commit -m "Fix existing bugs"
git rebase --continue
```

Advanced Interactive Rebase

Using exec for Automated Tasks:

```
git rebase -i HEAD~3

# Add exec commands:
pick abc123 Add feature
exec npm test
pick def456 Fix bug
```

```
exec npm run lint
pick ghi789 Update docs
exec npm run build

# Rebase will run tests/linting after each commit
```

Fixup and Autosquash:

```
# Create fixup commit
git commit --fixup abc123

# Auto-rebase with fixup
git rebase -i --autosquash HEAD~5

# Git automatically arranges fixup commits
```

Handling Rebase Conflicts:

```
# When conflicts occur during rebase
git status  # Shows conflicted files

# Resolve conflicts, then:
git add resolved-file.js
git rebase --continue

# Skip problematic commit:
git rebase --skip

# Abort entire rebase:
git rebase --abort
```

Interactive Rebase Best Practices

```
# Always backup before major rebase
git branch backup-before-rebase
git rebase -i HEAD~10

# Use rebase only on private branches
# Never rebase commits that have been pushed to shared branches

# Configure editor for rebase
git config --global core.editor "code --wait"

# Set up rebase aliases
git config --global alias.ri 'rebase -i'
git config --global alias.rc 'rebase --continue'
git config --global alias.ra 'rebase --abort'
```

Cherry-picking Commits

What is Cherry-picking?

Cherry-picking allows you to apply specific commits from one branch to another without merging the entire branch.

Basic Cherry-pick

```
# Apply specific commit to current branch
git cherry-pick abc123

# Apply multiple commits
git cherry-pick abc123 def456 ghi789

# Apply range of commits
git cherry-pick abc123..ghi789

# Apply range excluding first commit
git cherry-pick abc123^..ghi789
```

Cherry-pick Options

```
# Cherry-pick without committing (stage only)
git cherry-pick --no-commit abc123

# Cherry-pick and edit commit message
git cherry-pick --edit abc123

# Cherry-pick with different author
git cherry-pick --signoff abc123

# Cherry-pick merge commit (specify parent)
git cherry-pick -m 1 merge-commit-hash
```

Real-World Cherry-pick Scenarios

Scenario 1: Hotfix to Multiple Branches

```
# Fix critical bug in main
git switch main
echo "security fix" > security.js
git add security.js
git commit -m "Fix critical security vulnerability"
```

```
# Apply same fix to release branch
git switch release-v1.0
git cherry-pick main  # Cherry-pick latest commit from main

# Apply to development branch
git switch develop
git cherry-pick main
```

Scenario 2: Selective Feature Porting

```
# Port specific features from experimental branch
git log experimental --oneline
# abc123 Add new API endpoint
# def456 Experimental UI changes
# ghi789 Add caching layer
# jkl012 More experimental stuff

# Cherry-pick only stable features
git switch main
git cherry-pick abc123  # API endpoint
git cherry-pick ghi789  # Caching layer
# Skip experimental UI changes
```

Scenario 3: Bug Fix Backporting

```
# Bug fixed in development, need in production
git switch develop
git log --oneline -5
# abc123 Fix memory leak in image processing
# def456 Add new feature
# ghi789 Update dependencies

# Backport only the bug fix
git switch production
git cherry-pick abc123
```

Handling Cherry-pick Conflicts

```
# When cherry-pick conflicts occur
git cherry-pick abc123
# CONFLICT (content): Merge conflict in file.js

# Resolve conflicts manually
# Edit conflicted files
git add file.js
```

```
# Continue cherry-pick
git cherry-pick --continue

# Or abort cherry-pick
git cherry-pick --abort
```

Advanced Cherry-pick Techniques

Cherry-pick with Strategy:

```
# Use merge strategy for cherry-pick
git cherry-pick -X theirs abc123
git cherry-pick -X ours abc123

# Ignore whitespace changes
git cherry-pick -X ignore-space-change abc123
```

Batch Cherry-picking:

```
# Cherry-pick multiple commits with script
for commit in abc123 def456 ghi789; do
    git cherry-pick $commit
    if [ $? -ne 0 ]; then
        echo "Conflict in $commit, resolve manually"
        break
    fi
done
```

Cherry-pick from Different Repository:

```
# Add remote repository
git remote add other-repo https://github.com/other/repo.git
git fetch other-repo

# Cherry-pick from other repository
git cherry-pick other-repo/main~3
```

Reset Strategies

Understanding Git Reset

Git reset moves the current branch pointer and optionally modifies the staging area and working directory.

Three Types of Reset

Soft Reset (--soft):

- Moves HEAD pointer only
- Keeps staging area and working directory unchanged
- Use case: Undo commits but keep changes staged

Mixed Reset (default):

- Moves HEAD pointer
- Resets staging area to match HEAD
- Keeps working directory unchanged
- Use case: Undo commits and unstage changes

Hard Reset (--hard):

- Moves HEAD pointer
- Resets staging area to match HEAD
- Resets working directory to match HEAD
- Use case: Completely undo commits and discard changes

Reset Commands and Examples

Soft Reset:

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

# Check status
git status
# Changes to be committed: (all files from undone commit)

# Recommit with different message
git commit -m "Better commit message"
```

Mixed Reset (Default):

```
# Undo last commit, unstage changes
git reset HEAD~1
# Same as: git reset --mixed HEAD~1

# Check status
git status
# Changes not staged for commit: (all files from undone commit)

# Selectively stage and recommit
git add important-file.js
git commit -m "Add important feature"
git add other-files.js
git commit -m "Add supporting changes"
```

Hard Reset:

```
# Completely undo last commit
git reset --hard HEAD~1

# Check status
git status
# nothing to commit, working tree clean

# WARNING: Changes are permanently lost!
```

Practical Reset Scenarios

Scenario 1: Fix Last Commit Message

```
# Wrong commit message
git commit -m "Fix bug"  # Too vague

# Fix it
git reset --soft HEAD~1
git commit -m "Fix memory leak in image processing module"
```

Scenario 2: Split Large Commit

```
# Large commit with multiple changes
git log --oneline -1
# abc123 Add feature, fix bugs, update docs

# Split it
git reset HEAD~1  # Mixed reset
git add feature.js
git commit -m "Add new user authentication feature"
git add bugfix.js
git commit -m "Fix login validation bug"
git add README.md
git commit -m "Update documentation"
```

Scenario 3: Discard Experimental Work

```
# Tried experimental approach, want to start over
git log --oneline -3
# abc123 Experimental approach attempt 3
# def456 Experimental approach attempt 2
# ghi789 Experimental approach attempt 1
```

```
# Go back to before experiments
git reset --hard HEAD~3

# Start fresh
echo "better approach" > feature.js
git add feature.js
git commit -m "Implement feature with proven approach"
```

Reset to Specific Commit

```
# Reset to specific commit hash
git reset --hard abc123

# Reset to tag
git reset --hard v1.0.0

# Reset to branch
git reset --hard origin/main

# Reset specific file to previous version
git checkout HEAD~2 -- file.js
```

Recovering from Reset

```
# Find lost commits with reflog
git reflog
# abc123 HEAD@{0}: reset: moving to HEAD~3
# def456 HEAD@{1}: commit: Lost commit

# Recover lost commit
git reset --hard def456

# Or create branch from lost commit
git branch recovered-work def456
```

Safe Reset Practices

```
# Always check what you're resetting
git log --oneline -5
git diff HEAD~2

# Create backup branch before major reset
git branch backup-before-reset
git reset --hard HEAD~5
```

```
# Use reset with caution on shared branches
# Prefer revert for public commits
git revert abc123  # Instead of reset
```

Recovering with Reflog

What is Git Reflog?

Reflog (reference log) records when the tips of branches and other references were updated in the local repository. It's your safety net for recovering "lost" commits.

Basic Reflog Commands

```
# Show reflog for current branch
git reflog

# Show reflog for specific branch
git reflog main

# Show reflog for all references
git reflog --all

# Show reflog with dates
git reflog --date=iso

# Show last 10 reflog entries
git reflog -10
```

Understanding Reflog Output

```
git reflog
# abc123 HEAD@{0}: commit: Add new feature
# def456 HEAD@{1}: reset: moving to HEAD~1
# ghi789 HEAD@{2}: commit: Fix bug
# jkl012 HEAD@{3}: checkout: moving from main to feature-branch

# Format: commit-hash HEAD@{index}: action: description
```

Recovery Scenarios

Scenario 1: Recover from Accidental Reset

```
# Accidentally reset too far
git reset --hard HEAD~5
```

```
git log --oneline  # Shows only old commits

# Find lost commits in reflog
git reflog
# abc123 HEAD@{0}: reset: moving to HEAD~5
# def456 HEAD@{1}: commit: Important feature  # This was lost!

# Recover the lost commit
git reset --hard def456
# or
git reset --hard HEAD@{1}
```

Scenario 2: Recover Deleted Branch

```
# Accidentally delete branch
git branch -D feature-branch
# error: branch 'feature-branch' not found.

# Find branch in reflog
git reflog --all | grep feature-branch
# abc123 refs/heads/feature-branch@{0}: commit: Last commit on feature

# Recreate branch
git branch feature-branch abc123
git switch feature-branch
```

Scenario 3: Recover from Failed Rebase

```
# Rebase went wrong
git rebase main
# Conflicts, confusion, abort
git rebase --abort

# But want to try again with different approach
git reflog
# Find state before rebase
# def456 HEAD@{5}: checkout: moving from main to feature-branch

# Reset to pre-rebase state
git reset --hard def456
```

Advanced Reflog Usage

Time-based Recovery:

```
# Go back to state 2 hours ago
git reset --hard HEAD@{2.hours.ago}
```

```
# Go back to yesterday
git reset --hard HEAD@{yesterday}

# Go back to specific date
git reset --hard HEAD@{2023-12-01}
```

Reflog for Specific Files:

```
# See when file was last modified
git log --follow -- important-file.js

# Recover specific file version
git show HEAD@{5}:important-file.js > recovered-file.js
```

Cleaning Reflog:

```
# Reflog entries expire automatically (default 90 days)
# Force cleanup
git reflog expire --expire=now --all
git gc --prune=now

# Set custom expiration
git config gc.reflogExpire "30 days"
```

Bug Hunting with Bisect

What is Git Bisect?

Git bisect uses binary search to find the commit that introduced a bug. It's incredibly efficient for large commit ranges.

Basic Bisect Workflow

```
# Start bisect session
git bisect start

# Mark current commit as bad (has the bug)
git bisect bad

# Mark a known good commit (before bug existed)
git bisect good v1.0.0
# or
git bisect good abc123
```

```
# Git checks out middle commit
# Test for bug, then mark as good or bad
git bisect good   # if no bug
# or
git bisect bad    # if bug exists

# Repeat until Git finds the culprit
# Git will output: "abc123 is the first bad commit"

# End bisect session
git bisect reset
```

Practical Bisect Example

Scenario: Performance Regression

```
# Performance was good in v1.0.0, bad now
git bisect start
git bisect bad HEAD
git bisect good v1.0.0

# Git: "Bisecting: 15 revisions left to test after this"
# Test performance at current checkout
npm run performance-test

# If performance is bad:
git bisect bad

# If performance is good:
git bisect good

# Continue until found:
# "commit def456 is the first bad commit"
# Author: John Doe
# Date: Mon Dec 1 10:00:00 2023
# Message: Optimize database queries

git bisect reset
git show def456  # Examine the problematic commit
```

Automated Bisect

Using Test Script:

```
# Create test script
cat > test-script.sh << 'EOF'
#!/bin/bash
npm test
if [ $? -eq 0 ]; then
```

```
    exit 0  # Good commit
else
    exit 1  # Bad commit
fi
EOF

chmod +x test-script.sh

# Run automated bisect
git bisect start
git bisect bad HEAD
git bisect good v1.0.0
git bisect run ./test-script.sh

# Git automatically finds the bad commit
```

Complex Test Script:

```
#!/bin/bash
# test-performance.sh

# Build the project
npm run build
if [ $? -ne 0 ]; then
    exit 125  # Skip this commit (build failed)
fi

# Run performance test
result=$(npm run perf-test 2>&1 | grep "Time:" | awk '{print $2}')
threshold=1000

if [ "$result" -lt "$threshold" ]; then
    exit 0  # Good performance
else
    exit 1  # Bad performance
fi
```

Bisect Best Practices

```
# Use descriptive good/bad markers
git bisect start --term-old=working --term-new=broken
git bisect broken
git bisect working v1.0.0

# Skip commits that can't be tested
git bisect skip  # Current commit can't be tested

# Visualize bisect progress
git bisect visualize
```

```
# or
gitk bisect/bad ^bisect/good

# Log bisect session
git bisect log > bisect-session.log

# Replay bisect session
git bisect replay bisect-session.log
```

Tagging Releases

Understanding Git Tags

Tags are references that point to specific commits, typically used to mark release points (v1.0, v2.0, etc.).

Two Types of Tags:

- **Lightweight**: Simple pointer to a commit
- **Annotated**: Full objects with metadata (recommended)

Creating Tags

Lightweight Tags:

```
# Create lightweight tag
git tag v1.0.0

# Tag specific commit
git tag v1.0.0 abc123

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

Annotated Tags (Recommended):

```
# Create annotated tag
git tag -a v1.0.0 -m "Release version 1.0.0"

# Tag with detailed message
git tag -a v1.0.0 -m "Major release with new features:
- User authentication
- Payment integration
- Performance improvements"

# Tag specific commit
git tag -a v1.0.0 abc123 -m "Release 1.0.0"
```

```
# Show tag information
git show v1.0.0
```

Working with Tags

Listing and Filtering:

```
# List all tags
git tag

# List tags with pattern
git tag -l "v1.*"
git tag -l "*beta*"

# List tags with commit info
git tag -n
git tag -n5  # Show 5 lines of annotation

# Show tags sorted by version
git tag --sort=version:refname

# Show latest tag
git describe --tags --abbrev=0
```

Checking Out Tags:

```
# Checkout specific tag (detached HEAD)
git checkout v1.0.0

# Create branch from tag
git checkout -b hotfix-v1.0.1 v1.0.0

# Switch back to branch
git switch main
```

Pushing and Sharing Tags

```
# Push specific tag
git push origin v1.0.0

# Push all tags
git push origin --tags

# Push only annotated tags
git push origin --follow-tags
```

```
# Delete local tag
git tag -d v1.0.0

# Delete remote tag
git push origin --delete v1.0.0
# or
git push origin :refs/tags/v1.0.0
```

Semantic Versioning with Tags

Version Format: MAJOR.MINOR.PATCH

```
# Major version (breaking changes)
git tag -a v2.0.0 -m "Major release with breaking changes"

# Minor version (new features, backward compatible)
git tag -a v1.1.0 -m "Add new features"

# Patch version (bug fixes)
git tag -a v1.0.1 -m "Fix critical bugs"

# Pre-release versions
git tag -a v1.1.0-beta.1 -m "Beta release for testing"
git tag -a v1.1.0-rc.1 -m "Release candidate"
```

Release Workflow with Tags

```
# Complete release process
# 1. Finish all features for release
git switch main
git pull origin main

# 2. Update version in package files
echo '"version": "1.1.0"' > package.json
git add package.json
git commit -m "Bump version to 1.1.0"

# 3. Create release tag
git tag -a v1.1.0 -m "Release 1.1.0

New features:
- User dashboard
- Email notifications

Bug fixes:
- Login timeout issue
- Memory leak in image processing"
```

```
# 4. Push everything
git push origin main
git push origin v1.1.0

# 5. Create GitHub release
gh release create v1.1.0 --title "Version 1.1.0" --notes-file CHANGELOG.md
```

Advanced Tagging

Signed Tags (GPG):

```
# Create signed tag
git tag -s v1.0.0 -m "Signed release 1.0.0"

# Verify signed tag
git tag -v v1.0.0

# Configure GPG signing
git config --global user.signingkey YOUR_GPG_KEY_ID
git config --global tag.gpgSign true
```

Tag Automation:

```
# Script for automated tagging
#!/bin/bash
# release.sh

VERSION=$1
if [ -z "$VERSION" ]; then
    echo "Usage: ./release.sh <version>"
    exit 1
fi

# Update version in files
sed -i "s/\"version\": \".*\"/\"version\": \"$VERSION\"/" package.json

# Commit version bump
git add package.json
git commit -m "Bump version to $VERSION"

# Create tag
git tag -a "v$VERSION" -m "Release version $VERSION"

# Push
git push origin main
git push origin "v$VERSION"

echo "Released version $VERSION"
```

Git Hooks

What are Git Hooks?

Git hooks are scripts that run automatically at certain points in the Git workflow. They help enforce policies, run tests, and automate tasks.

Types of Git Hooks

Client-side Hooks:

- `pre-commit`: Before commit is created
- `prepare-commit-msg`: Before commit message editor
- `commit-msg`: After commit message is entered
- `post-commit`: After commit is created
- `pre-push`: Before push to remote
- `pre-rebase`: Before rebase operation

Server-side Hooks:

- `pre-receive`: Before any refs are updated
- `update`: Before each ref is updated
- `post-receive`: After all refs are updated
- `post-update`: After all refs are updated (similar to post-receive)

Setting Up Hooks

Hook Location:

```
# Hooks are stored in .git/hooks/
ls .git/hooks/
# Shows sample hooks (*.sample files)

# Make hook executable
chmod +x .git/hooks/pre-commit
```

Pre-commit Hook Examples

Basic Syntax Check:

```
#!/bin/bash
# .git/hooks/pre-commit

echo "Running pre-commit checks..."

# Check for syntax errors in JavaScript files
for file in $(git diff --cached --name-only --diff-filter=ACM | grep '\.js$'); do
```

```
    if [ -f "$file" ]; then
        node -c "$file"
        if [ $? -ne 0 ]; then
            echo "Syntax error in $file"
            exit 1
        fi
    fi
done

echo "Pre-commit checks passed!"
```

ESLint Integration:

```
#!/bin/bash
# .git/hooks/pre-commit

echo "Running ESLint..."

# Get list of staged JavaScript files
FILES=$(git diff --cached --name-only --diff-filter=ACM | grep '\.js$')

if [ -n "$FILES" ]; then
    # Run ESLint on staged files
    npx eslint $FILES
    if [ $? -ne 0 ]; then
        echo "ESLint failed. Please fix errors before committing."
        exit 1
    fi
fi

echo "ESLint passed!"
```

Prettier Code Formatting:

```
#!/bin/bash
# .git/hooks/pre-commit

echo "Running Prettier..."

# Get staged files
FILES=$(git diff --cached --name-only --diff-filter=ACM | grep -E '\.
(js|jsx|ts|tsx|css|md)$')

if [ -n "$FILES" ]; then
    # Format files with Prettier
    npx prettier --write $FILES

    # Add formatted files back to staging
    git add $FILES
```

```
    echo "Code formatted with Prettier"
fi
```

Commit Message Hook

Conventional Commits Validation:

```
#!/bin/bash
# .git/hooks/commit-msg

commit_regex='^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?: .{1,50}'

if ! grep -qE "$commit_regex" "$1"; then
    echo "Invalid commit message format!"
    echo "Format: type(scope): description"
    echo "Types: feat, fix, docs, style, refactor, test, chore"
    echo "Example: feat(auth): add user login functionality"
    exit 1
fi

echo "Commit message format is valid"
```

Pre-push Hook

Run Tests Before Push:

```
#!/bin/bash
# .git/hooks/pre-push

echo "Running tests before push..."

# Run test suite
npm test
if [ $? -ne 0 ]; then
    echo "Tests failed. Push aborted."
    exit 1
fi

# Run build to ensure it works
npm run build
if [ $? -ne 0 ]; then
    echo "Build failed. Push aborted."
    exit 1
fi

echo "All checks passed. Proceeding with push."
```

Advanced Hook Management

Shared Hooks with Husky:

```
# Install Husky
npm install --save-dev husky

# Initialize Husky
npx husky install

# Add to package.json
npm pkg set scripts.prepare="husky install"

# Add pre-commit hook
npx husky add .husky/pre-commit "npm run lint"
npx husky add .husky/pre-commit "npm test"

# Add commit-msg hook
npx husky add .husky/commit-msg 'npx commitlint --edit $1'
```

Hook Configuration:

```
# Skip hooks temporarily
git commit --no-verify -m "Emergency fix"
git push --no-verify

# Disable hooks globally
git config --global core.hooksPath /dev/null

# Re-enable hooks
git config --global --unset core.hooksPath
```

Complex Hook Example:

```
#!/bin/bash
# .git/hooks/pre-commit
# Comprehensive pre-commit hook

set -e  # Exit on any error

echo "🔍 Running pre-commit checks..."

# 1. Check for merge conflict markers
if git diff --cached | grep -E '^\+.*(<{7}|={7}|>{7})'; then
    echo "❌ Merge conflict markers found"
    exit 1
fi
```

```
# 2. Check for debugging statements
if git diff --cached | grep -E '^\+.*(console\.log|debugger|TODO|FIXME)'; then
    echo "⚠️  Debugging statements found. Remove before committing."
    exit 1
fi

# 3. Run linting
echo "🧹 Running ESLint..."
npx eslint $(git diff --cached --name-only --diff-filter=ACM | grep '\.js$')

# 4. Run tests
echo "🧪 Running tests..."
npm test

# 5. Check file sizes
echo "📏 Checking file sizes..."
for file in $(git diff --cached --name-only); do
    if [ -f "$file" ] && [ $(wc -c < "$file") -gt 1048576 ]; then
        echo "❌ File $file is larger than 1MB"
        exit 1
    fi
done

echo "✅ All pre-commit checks passed!"
```

Commit Message Best Practices

Why Good Commit Messages Matter

- **Documentation**: Explain what and why, not how
- **Debugging**: Help identify when bugs were introduced
- **Code Review**: Provide context for reviewers
- **Maintenance**: Help future developers understand changes

Conventional Commits Format

Structure:

```
type(scope): description

[optional body]

[optional footer(s)]
```

Types:

- `feat`: New feature
- `fix`: Bug fix

- `docs`: Documentation changes
- `style`: Code style changes (formatting, semicolons, etc.)
- `refactor`: Code refactoring
- `test`: Adding or updating tests
- `chore`: Maintenance tasks
- `perf`: Performance improvements
- `ci`: CI/CD changes
- `build`: Build system changes

Good Commit Message Examples

Basic Examples:

```
# Good
feat(auth): add user login functionality
fix(api): resolve timeout issue in user endpoint
docs(readme): update installation instructions
refactor(utils): simplify date formatting function

# Bad
fixed stuff
update
changes
wip
```

Detailed Examples:

```
# Feature with body
feat(payment): integrate Stripe payment processing

Add support for credit card payments using Stripe API.
Includes error handling for failed transactions and
webhook support for payment status updates.

Closes #123

# Bug fix with breaking change
fix(api): change user endpoint response format

BREAKING CHANGE: User endpoint now returns user object
instead of array. Update client code accordingly.

Before: GET /users/123 returned [{ id: 123, name: "John" }]
After: GET /users/123 returns { id: 123, name: "John" }

# Multiple changes
feat(dashboard): add user analytics

- Add user activity tracking
```

```
- Implement analytics dashboard
- Add export functionality for reports

Resolves #456, #789
```

Commit Message Templates

Set up template:

```
# Create template file
cat > ~/.gitmessage << 'EOF'
# Type(scope): Brief description (50 chars max)
#
# Longer explanation if needed (wrap at 72 chars)
# - What was changed and why
# - Any side effects or important notes
#
# Closes #issue-number
EOF

# Configure Git to use template
git config --global commit.template ~/.gitmessage
```

Team Template:

```
# .gitmessage in project root
# [TYPE](SCOPE): [DESCRIPTION]
#
# Types: feat, fix, docs, style, refactor, test, chore, perf
# Scope: component, module, or area affected
# Description: imperative mood, present tense
#
# Body (optional):
# - Explain what and why vs. how
# - Reference issues and pull requests
#
# Footer (optional):
# BREAKING CHANGE: describe breaking changes
# Closes #123
# Refs #456

# Configure for project
git config commit.template .gitmessage
```

Commit Message Rules

The Seven Rules:

1. **Separate subject from body** with a blank line
2. **Limit subject line** to 50 characters
3. **Capitalize** the subject line
4. **Don't end** subject line with a period
5. **Use imperative mood** in subject line
6. **Wrap body** at 72 characters
7. **Use body** to explain what and why vs. how

Examples of Imperative Mood:

```
# Good (imperative)
Add user authentication
Fix memory leak in image processor
Update documentation for API changes

# Bad (not imperative)
Added user authentication
Fixed memory leak
Updating documentation
```

Automated Commit Message Validation

Using commitlint:

```
# Install commitlint
npm install --save-dev @commitlint/cli @commitlint/config-conventional

# Create config file
echo "module.exports = {extends: ['@commitlint/config-conventional']}" >
commitlint.config.js

# Add to Husky
npx husky add .husky/commit-msg 'npx commitlint --edit $1'
```

Custom commitlint rules:

```
// commitlint.config.js
module.exports = {
  extends: ["@commitlint/config-conventional"],
  rules: {
    "type-enum": [
      2,
      "always",
      ["feat", "fix", "docs", "style", "refactor", "test", "chore", "perf"],
    ],
    "subject-max-length": [2, "always", 50],
    "body-max-line-length": [2, "always", 72],
```

```
    "scope-case": [2, "always", "lower-case"],
  },
};
```

🚀 PART 4: GITHUB MASTERY

GitHub CLI (gh)

Installation and Setup

Install GitHub CLI:

```
# Windows (using winget)
winget install --id GitHub.cli

# macOS (using Homebrew)
brew install gh

# Linux (Ubuntu/Debian)
curl -fsSL https://cli.github.com/packages/githubcli-archive-keyring.gpg | sudo dd
of=/usr/share/keyrings/githubcli-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/githubcli-archive-keyring.gpg]
https://cli.github.com/packages stable main" | sudo tee
/etc/apt/sources.list.d/github-cli.list > /dev/null
sudo apt update
sudo apt install gh
```

Authentication:

```
# Login to GitHub
gh auth login

# Choose authentication method:
# - GitHub.com
# - HTTPS/SSH
# - Login with web browser or token

# Check authentication status
gh auth status

# Switch between accounts
gh auth switch
```

Repository Management

Creating Repositories:

```
# Create new repository
gh repo create my-project
gh repo create my-project --public
gh repo create my-project --private

# Create with template
gh repo create my-project --template owner/template-repo

# Create and clone
gh repo create my-project --clone

# Create from current directory
gh repo create --source=. --public
```

Repository Operations:

```
# Clone repository
gh repo clone owner/repo
gh repo clone owner/repo custom-name

# Fork repository
gh repo fork owner/repo
gh repo fork owner/repo --clone

# View repository
gh repo view
gh repo view owner/repo
gh repo view owner/repo --web

# List repositories
gh repo list
gh repo list owner
gh repo list --limit 50
```

Issues Management

Creating Issues:

```
# Create issue interactively
gh issue create

# Create with title and body
gh issue create --title "Bug: Login fails" --body "Description of the bug"

# Create with template
gh issue create --template bug_report.md
```

```
# Create and assign
gh issue create --title "New feature" --assignee @me
gh issue create --title "Bug fix" --assignee username
```

Managing Issues:

```
# List issues
gh issue list
gh issue list --state open
gh issue list --state closed
gh issue list --assignee @me
gh issue list --label bug

# View issue
gh issue view 123
gh issue view 123 --web

# Edit issue
gh issue edit 123 --title "New title"
gh issue edit 123 --body "New description"
gh issue edit 123 --add-label "priority:high"

# Close/reopen issue
gh issue close 123
gh issue reopen 123

# Comment on issue
gh issue comment 123 --body "This is a comment"
```

Pull Requests

Creating Pull Requests:

```
# Create PR interactively
gh pr create

# Create with details
gh pr create --title "Add new feature" --body "Description of changes"

# Create draft PR
gh pr create --draft

# Create and assign reviewers
gh pr create --reviewer username1,username2
gh pr create --reviewer @team/reviewers

# Create with labels
gh pr create --label "enhancement,needs-review"
```

Managing Pull Requests:

```
# List PRs
gh pr list
gh pr list --state open
gh pr list --author @me
gh pr list --reviewer @me

# View PR
gh pr view 456
gh pr view 456 --web

# Checkout PR locally
gh pr checkout 456

# Review PR
gh pr review 456
gh pr review 456 --approve
gh pr review 456 --request-changes
gh pr review 456 --comment --body "Looks good!"

# Merge PR
gh pr merge 456
gh pr merge 456 --merge    # Create merge commit
gh pr merge 456 --squash   # Squash and merge
gh pr merge 456 --rebase   # Rebase and merge

# Close PR
gh pr close 456
```

Advanced GitHub CLI

Workflows and Actions:

```
# List workflow runs
gh run list
gh run list --workflow=ci.yml

# View workflow run
gh run view 123456
gh run view 123456 --log

# Re-run workflow
gh run rerun 123456

# Watch workflow run
gh run watch
```

Releases:

```
# Create release
gh release create v1.0.0
gh release create v1.0.0 --title "Version 1.0.0" --notes "Release notes"

# Upload assets
gh release create v1.0.0 dist/*.zip

# List releases
gh release list

# View release
gh release view v1.0.0

# Download release assets
gh release download v1.0.0
```

Gists:

```
# Create gist
gh gist create file.js
gh gist create file.js --desc "Useful function"
gh gist create file.js --public

# List gists
gh gist list

# View gist
gh gist view abc123

# Edit gist
gh gist edit abc123

# Clone gist
gh gist clone abc123
```

GitHub CLI Configuration

```
# Set default editor
gh config set editor vim
gh config set editor "code --wait"

# Set default protocol
gh config set git_protocol ssh
gh config set git_protocol https

# View configuration
```

```
gh config list

# Set aliases
gh alias set pv 'pr view'
gh alias set co 'pr checkout'
gh alias set prs 'pr list --author @me'
```

GitHub Actions

Understanding GitHub Actions

Core Concepts:

- **Workflow**: Automated process defined in YAML
- **Event**: Trigger that starts a workflow
- **Job**: Set of steps that execute on the same runner
- **Step**: Individual task within a job
- **Action**: Reusable unit of code
- **Runner**: Server that runs workflows

Basic Workflow Structure

```
# .github/workflows/ci.yml
name: CI

# Events that trigger the workflow
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

# Jobs to run
jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: "18"
          cache: "npm"

      - name: Install dependencies
        run: npm ci
```

```
      - name: Run tests
        run: npm test

      - name: Run linting
        run: npm run lint
```

Common Workflow Patterns

Node.js CI/CD:

```
name: Node.js CI/CD

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [16, 18, 20]

    steps:
      - uses: actions/checkout@v4

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
          cache: "npm"

      - run: npm ci
      - run: npm run build --if-present
      - run: npm test

      - name: Upload coverage reports
        uses: codecov/codecov-action@v3
        if: matrix.node-version == '18'

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v4
```

```
      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: "18"
          cache: "npm"

      - run: npm ci
      - run: npm run build

      - name: Deploy to production
        run: |
          echo "Deploying to production..."
          # Add your deployment commands here
```

Docker Build and Push:

```
name: Docker Build and Push

on:
  push:
    branches: [main]
    tags: ["v*"]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Log in to Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
```

```
            type=ref,event=branch
            type=ref,event=pr
            type=semver,pattern={{version}}
            type=semver,pattern={{major}}.{{minor}}

      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

Advanced GitHub Actions

Conditional Workflows:

```
name: Conditional Deployment

on:
  push:
    branches: [main, staging, develop]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Deploy to staging
        if: github.ref == 'refs/heads/staging'
        run: echo "Deploying to staging"

      - name: Deploy to production
        if: github.ref == 'refs/heads/main'
        run: echo "Deploying to production"

      - name: Deploy to development
        if: github.ref == 'refs/heads/develop'
        run: echo "Deploying to development"
```

Matrix Builds:

```
name: Cross-platform Testing

on: [push, pull_request]

jobs:
```

```yaml
  test:
    runs-on: ${{ matrix.os }}

    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [16, 18, 20]
        include:
          - os: ubuntu-latest
            node-version: 20
            coverage: true
        exclude:
          - os: windows-latest
            node-version: 16

    steps:
      - uses: actions/checkout@v4

      - name: Setup Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}

      - run: npm ci
      - run: npm test

      - name: Upload coverage
        if: matrix.coverage
        uses: codecov/codecov-action@v3
```

Custom Actions:

```yaml
# .github/actions/setup-project/action.yml
name: "Setup Project"
description: "Setup Node.js project with caching"

inputs:
  node-version:
    description: "Node.js version"
    required: true
    default: "18"
  cache-dependency-path:
    description: "Path to package-lock.json"
    required: false
    default: "package-lock.json"

runs:
  using: "composite"
  steps:
    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
```

```
        node-version: ${{ inputs.node-version }}
        cache: "npm"
        cache-dependency-path: ${{ inputs.cache-dependency-path }}

    - name: Install dependencies
      run: npm ci
      shell: bash
```

Using Custom Action:

```
name: Use Custom Action

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Setup project
        uses: ./.github/actions/setup-project
        with:
          node-version: "18"

      - run: npm test
```

Secrets and Environment Variables

```
name: Deploy with Secrets

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production

    env:
      NODE_ENV: production
      API_URL: https://api.example.com

    steps:
      - uses: actions/checkout@v4

      - name: Deploy
```

```
        env:
          DEPLOY_TOKEN: ${{ secrets.DEPLOY_TOKEN }}
          DATABASE_URL: ${{ secrets.DATABASE_URL }}
        run: |
          echo "Deploying with token: ${DEPLOY_TOKEN:0:8}..."
          # Deployment commands here
```

Managing Secrets:

```
# Using GitHub CLI
gh secret set DEPLOY_TOKEN --body "your-secret-token"
gh secret set DATABASE_URL --body "postgresql://..."

# List secrets
gh secret list

# Delete secret
gh secret delete DEPLOY_TOKEN
```

🎯 PART 5: ADVANCED WORKFLOWS

Maintaining Clean Git History

Why Clean History Matters

- **Easier debugging**: Clear commit progression
- **Better code reviews**: Logical change grouping
- **Simplified maintenance**: Easy to understand evolution
- **Professional appearance**: Shows attention to detail

Interactive Rebase for History Cleanup

Basic Interactive Rebase:

```
# Rebase last 5 commits
git rebase -i HEAD~5

# Rebase from specific commit
git rebase -i abc123

# Rebase from branch point
git rebase -i main
```

Rebase Commands:

```
# In the interactive editor:
pick abc123 Add user authentication
squash def456 Fix typo in auth
reword ghi789 Implement user dashboard
drop jkl012 Debug commit - remove this
edit mno345 Add payment integration

# Commands:
# pick (p) = use commit
# reword (r) = use commit, but edit message
# edit (e) = use commit, but stop for amending
# squash (s) = use commit, but meld into previous commit
# fixup (f) = like squash, but discard commit message
# drop (d) = remove commit
```

Squashing Related Commits:

```
# Before rebase:
abc123 Add user model
def456 Fix user model validation
ghi789 Add user model tests
jkl012 Fix typo in user model

# After squashing:
abc123 Add user model with validation and tests

# Interactive rebase:
pick abc123 Add user model
squash def456 Fix user model validation
squash ghi789 Add user model tests
squash jkl012 Fix typo in user model
```

Fixup Commits Workflow

Creating Fixup Commits:

```
# Make a small fix to previous commit
git add .
git commit --fixup abc123

# Automatically squash fixup commits
git rebase -i --autosquash HEAD~10
```

Automated Fixup Workflow:

```
# Configure autosquash by default
git config --global rebase.autosquash true

# Create fixup commit
echo "Fixed typo" >> file.js
git add file.js
git commit --fixup HEAD~2

# Rebase will automatically arrange commits
git rebase -i HEAD~5
```

Splitting Commits

```
# Start interactive rebase
git rebase -i HEAD~3

# Mark commit to split with 'edit'
edit abc123 Large commit with multiple changes

# When rebase stops, reset the commit
git reset HEAD~1

# Stage and commit changes separately
git add file1.js
git commit -m "Add user authentication"

git add file2.js
git commit -m "Add user validation"

git add file3.js
git commit -m "Add user tests"

# Continue rebase
git rebase --continue
```

Rewriting Commit Messages

```
# Change last commit message
git commit --amend -m "New commit message"

# Change multiple commit messages
git rebase -i HEAD~5
# Mark commits with 'reword'

# Batch update commit messages
git filter-branch --msg-filter '
    if [ "$GIT_COMMIT" = "abc123" ]; then
        echo "New message for specific commit"
```

```
    else
        cat
    fi
' HEAD~10..HEAD
```

Branching Strategies

Git Flow

Branch Types:

- **main**: Production-ready code
- **develop**: Integration branch for features
- **feature/***: New features
- **release/***: Release preparation
- **hotfix/***: Critical fixes for production

Git Flow Workflow:

```
# Initialize Git Flow
git flow init

# Start new feature
git flow feature start user-authentication
# Creates and switches to feature/user-authentication

# Work on feature
git add .
git commit -m "Add login functionality"
git commit -m "Add password validation"

# Finish feature
git flow feature finish user-authentication
# Merges into develop and deletes feature branch

# Start release
git flow release start 1.0.0
# Creates release/1.0.0 from develop

# Prepare release (bug fixes, version bumps)
git add .
git commit -m "Bump version to 1.0.0"
git commit -m "Fix release bugs"

# Finish release
git flow release finish 1.0.0
# Merges into main and develop, creates tag

# Hotfix for production
git flow hotfix start critical-security-fix
```

```
# Creates hotfix/critical-security-fix from main

# Fix the issue
git add .
git commit -m "Fix security vulnerability"

# Finish hotfix
git flow hotfix finish critical-security-fix
# Merges into main and develop
```

GitHub Flow (Simplified)

Workflow:

1. Create feature branch from main
2. Make changes and commit
3. Open Pull Request
4. Review and discuss
5. Merge to main
6. Deploy

```
# Create feature branch
git checkout main
git pull origin main
git checkout -b feature/new-dashboard

# Make changes
git add .
git commit -m "Add dashboard layout"
git commit -m "Add dashboard widgets"

# Push and create PR
git push origin feature/new-dashboard
gh pr create --title "Add new dashboard" --body "Implements user dashboard with
widgets"

# After review and approval
gh pr merge --squash

# Clean up
git checkout main
git pull origin main
git branch -d feature/new-dashboard
```

Trunk-Based Development

Principles:

- Short-lived feature branches (< 2 days)

- Frequent integration to main
- Feature flags for incomplete features
- Continuous deployment

```
# Short-lived feature branch
git checkout main
git pull origin main
git checkout -b quick-fix

# Make small, focused changes
git add .
git commit -m "Fix button alignment"

# Push and merge quickly
git push origin quick-fix
gh pr create --title "Fix button alignment"
gh pr merge --squash

# Immediate cleanup
git checkout main
git pull origin main
git branch -d quick-fix
```

Feature Flags Example:

```
// Use feature flags for incomplete features
if (featureFlags.newDashboard) {
  return <NewDashboard />;
} else {
  return <OldDashboard />;
}
```

Safe Force Push Techniques

Understanding Force Push Dangers

Never Force Push When:

- Working on shared branches (main, develop)
- Others have pulled your branch
- Uncertain about the impact

Safe to Force Push When:

- Working on personal feature branches
- No one else has the branch
- Cleaning up before merge

Force Push with Lease

```
# Safer alternative to git push --force
git push --force-with-lease

# Even safer - specify expected commit
git push --force-with-lease=origin/feature-branch:abc123

# Check what would be overwritten
git push --force-with-lease --dry-run
```

How Force-with-Lease Works:

```
# Scenario: Someone else pushed to your branch
# Your local branch
abc123 (HEAD) Your latest commit
def456 Previous commit

# Remote branch (someone else pushed)
ghi789 (origin/feature) Someone else's commit
def456 Previous commit

# Force push fails safely
git push --force-with-lease
# Error: Updates were rejected because the remote contains work
# that you do not have locally.

# Fetch and rebase first
git fetch origin
git rebase origin/feature-branch
# Then force push safely
git push --force-with-lease
```

Safe Force Push Workflow

```
# 1. Communicate with team
echo "About to force push to feature/user-auth. Please don't push to this branch."

# 2. Fetch latest changes
git fetch origin

# 3. Check if anyone else has pushed
git log --oneline origin/feature/user-auth..HEAD
git log --oneline HEAD..origin/feature/user-auth

# 4. If safe, force push with lease
git push --force-with-lease origin feature/user-auth
```

```
# 5. Notify team
echo "Force push complete. Safe to work on feature/user-auth again."
```

Alternative to Force Push

Create New Branch:

```
# Instead of force pushing, create new branch
git checkout -b feature/user-auth-v2
git push origin feature/user-auth-v2

# Update PR to point to new branch
gh pr edit --base feature/user-auth-v2

# Delete old branch after merge
git push origin --delete feature/user-auth
```

Revert and Redo:

```
# Revert problematic commits
git revert abc123..def456
git push origin feature-branch

# Then apply correct changes
git add .
git commit -m "Correct implementation"
git push origin feature-branch
```

🛠️ PART 6: TROUBLESHOOTING & BEST PRACTICES

Common Git Problems and Solutions

"I committed to the wrong branch"

Solution 1: Move commits to correct branch

```
# Currently on wrong-branch with new commits
git log --oneline -3
# abc123 (HEAD) New feature commit
# def456 Another commit
# ghi789 Base commit

# Create correct branch from current state
git branch correct-branch
```

```
# Reset wrong branch to before the commits
git reset --hard ghi789

# Switch to correct branch
git checkout correct-branch
```

Solution 2: Cherry-pick commits

```
# Switch to correct branch
git checkout correct-branch

# Cherry-pick the commits
git cherry-pick abc123
git cherry-pick def456

# Remove commits from wrong branch
git checkout wrong-branch
git reset --hard ghi789
```

"I need to undo the last commit"

Keep changes (soft reset):

```
git reset --soft HEAD~1
# Commit is undone, changes remain staged
```

Keep changes unstaged (mixed reset):

```
git reset HEAD~1
# or
git reset --mixed HEAD~1
# Commit is undone, changes remain in working directory
```

Discard everything (hard reset):

```
git reset --hard HEAD~1
# Commit and changes are completely removed
```

"I accidentally deleted a file"

Restore from last commit:

```
git checkout HEAD -- filename.js
# or
git restore filename.js
```

Restore from specific commit:

```
git checkout abc123 -- filename.js
# or
git restore --source=abc123 filename.js
```

Find when file was deleted:

```
git log --oneline --follow -- filename.js
git log --diff-filter=D --summary | grep filename.js
```

"My working directory is messy"

Clean untracked files:

```
# See what would be removed
git clean -n

# Remove untracked files
git clean -f

# Remove untracked files and directories
git clean -fd

# Remove ignored files too
git clean -fX

# Interactive clean
git clean -i
```

Reset all changes:

```
# Discard all changes in working directory
git checkout .
# or
git restore .

# Reset staged changes
git reset
```

```
# Complete reset
git reset --hard HEAD
git clean -fd
```

"I have merge conflicts"

Understanding conflict markers:

```
<<<<<<< HEAD
// Your changes
function login(username, password) {
    return authenticate(username, password);
}
=======
// Incoming changes
function login(user, pass) {
    return auth.validate(user, pass);
}
>>>>>>> feature-branch
```

Resolving conflicts:

```
# 1. Edit files to resolve conflicts
# Remove conflict markers and choose/combine changes

# 2. Stage resolved files
git add resolved-file.js

# 3. Continue merge/rebase
git commit  # for merge
git rebase --continue  # for rebase
```

Conflict resolution tools:

```
# Use configured merge tool
git mergetool

# Use specific tool
git mergetool --tool=vimdiff
git mergetool --tool=code

# Configure default merge tool
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'
```

"I pushed sensitive information"

Remove from last commit:

```
# Remove file and amend commit
git rm --cached sensitive-file.txt
echo "sensitive-file.txt" >> .gitignore
git add .gitignore
git commit --amend --no-edit

# Force push (if safe)
git push --force-with-lease
```

Remove from history:

```
# Use git filter-branch (for older Git)
git filter-branch --force --index-filter \
    'git rm --cached --ignore-unmatch sensitive-file.txt' \
    --prune-empty --tag-name-filter cat -- --all

# Use git filter-repo (recommended)
pip install git-filter-repo
git filter-repo --path sensitive-file.txt --invert-paths

# Force push all branches and tags
git push origin --force --all
git push origin --force --tags
```

Invalidate exposed secrets:

```
# Immediately rotate/revoke:
# - API keys
# - Passwords
# - Tokens
# - Certificates

# Update all systems using the compromised credentials
```

Git Configuration Best Practices

Essential Global Configuration

```
# User information
git config --global user.name "Your Name"
```

```
git config --global user.email "your.email@example.com"

# Default branch name
git config --global init.defaultBranch main

# Editor
git config --global core.editor "code --wait"
# or
git config --global core.editor vim

# Line ending handling
git config --global core.autocrlf input  # Linux/Mac
git config --global core.autocrlf true   # Windows

# Push behavior
git config --global push.default simple
git config --global push.autoSetupRemote true

# Pull behavior
git config --global pull.rebase true

# Merge behavior
git config --global merge.ff false
git config --global merge.conflictstyle diff3

# Rebase behavior
git config --global rebase.autosquash true
git config --global rebase.autoStash true

# Color output
git config --global color.ui auto

# Credential helper
git config --global credential.helper store  # Linux/Mac
git config --global credential.helper manager-core  # Windows
```

Useful Aliases

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

# Enhanced log
git config --global alias.lg "log --oneline --graph --decorate --all"
git config --global alias.lga "log --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --
all"

# Useful shortcuts
```

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

# Advanced aliases
git config --global alias.find "!git ls-files | grep -i"
git config --global alias.grep "grep --break --heading --line-number"
git config --global alias.amend "commit --amend --no-edit"
git config --global alias.pushf "push --force-with-lease"
git config --global alias.root "rev-parse --show-toplevel"
```

Project-Specific Configuration

```
# Different email for work projects
cd work-project
git config user.email "work.email@company.com"

# Different signing key
git config user.signingkey WORK_GPG_KEY_ID

# Project-specific hooks
git config core.hooksPath .githooks

# Custom merge driver
git config merge.ours.driver true
```

Performance Optimization

Repository Maintenance

```
# Garbage collection
git gc
git gc --aggressive  # More thorough, slower

# Prune unreachable objects
git prune
git prune --expire="2 weeks ago"

# Clean up remote tracking branches
git remote prune origin

# Repack repository
git repack -ad

# Check repository integrity
git fsck
git fsck --full
```

Large Repository Optimization

Shallow Clone:

```
# Clone with limited history
git clone --depth 1 https://github.com/user/repo.git
git clone --depth 50 https://github.com/user/repo.git

# Deepen shallow repository
git fetch --unshallow
git fetch --depth=100
```

Partial Clone:

```
# Clone without large files
git clone --filter=blob:limit=1m https://github.com/user/repo.git

# Clone specific paths only
git clone --filter=tree:0 https://github.com/user/repo.git
git sparse-checkout init --cone
git sparse-checkout set src/ docs/
```

Git LFS for Large Files:

```
# Install Git LFS
git lfs install

# Track large files
git lfs track "*.psd"
git lfs track "*.zip"
git lfs track "videos/*"

# Add .gitattributes
git add .gitattributes
git commit -m "Add LFS tracking"

# Normal workflow
git add large-file.psd
git commit -m "Add design file"
git push origin main
```

🎓 CONCLUSION

Quick Reference Cheat Sheet

Essential Commands

```
# Repository setup
git init
git clone <url>
git remote add origin <url>

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

# Branching
git branch <name>
git checkout <branch>
git merge <branch>
git branch -d <branch>

# Status and history
git status
git log --oneline
git diff
git show <commit>

# Undo operations
git reset --soft HEAD~1
git reset HEAD~1
git reset --hard HEAD~1
git revert <commit>
```

Advanced Commands

```
# Interactive rebase
git rebase -i HEAD~5

# Stashing
git stash
git stash pop
git stash list

# Cherry-pick
git cherry-pick <commit>

# Reflog
git reflog
git reset --hard HEAD@{2}

# Bisect
git bisect start
```

```
git bisect bad
git bisect good <commit>

# Tags
git tag -a v1.0.0 -m "Release 1.0.0"
git push origin v1.0.0
```

Learning Path Summary

Beginner (Weeks 1-2):

1. Basic Git concepts and commands
2. Repository creation and cloning
3. Add, commit, push, pull workflow
4. Basic branching and merging
5. GitHub account setup and first repository

Intermediate (Weeks 3-4):

1. Advanced branching strategies
2. Merge conflict resolution
3. Stashing and cherry-picking
4. Pull requests and code review
5. Git hooks and automation

Advanced (Weeks 5-6):

1. Interactive rebase and history rewriting
2. Advanced Git commands (bisect, reflog)
3. GitHub Actions and CI/CD
4. Repository maintenance and optimization
5. Team workflows and best practices

Next Steps

1. **Practice Daily**: Use Git for all your projects
2. **Contribute to Open Source**: Practice collaboration
3. **Learn Team Workflows**: Understand different branching strategies
4. **Automate with Actions**: Set up CI/CD pipelines
5. **Teach Others**: Best way to solidify knowledge

Additional Resources

- **Official Documentation**: git-scm.com
- **GitHub Docs**: docs.github.com
- **Interactive Learning**: learngitbranching.js.org
- **Git Cheat Sheet**: education.github.com/git-cheat-sheet-education.pdf
- **Pro Git Book**: git-scm.com/book

Happy Git-ing! 🚀

Remember: Git is a powerful tool, but with great power comes great responsibility. Always double-check before force pushing, and when in doubt, create a backup branch!