Complete Git and GitHub Tutorial

A comprehensive guide to Git and GitHub, from beginner to advanced concepts. This tutorial covers everything you need to know to master version control and collaborative development.

🗂️ Table of Contents

🟢 Beginner Topics

🟡 Intermediate Topics

- Visual diagrams and examples
- When to use each approach
- Fast-forward vs non-fast-forward merges

7. **Git Stash**
   - Stashing changes temporarily
   - Multiple stash management
   - Stash operations and best practices
   - Advanced stashing techniques
8. **Git Diff and Log**
   - `git diff` for comparing changes
   - `git log` with formatting and filters
   - Viewing file history and blame
   - Advanced log techniques
9. **Pull Requests and Code Review**
   - Creating and managing pull requests
   - Code review workflow and best practices
   - PR templates and automation
   - Merge strategies
10. **Forking and Upstream**
    - Forking repositories
    - Managing upstream remotes
    - Contributing to open source
    - Keeping forks in sync

🟤 Advanced Topics

11. **Advanced Git Rebase**
    - Interactive rebase (`git rebase -i`)
    - Squashing and fixup commits
    - Reordering and editing commits
    - Complex rebase scenarios
12. **Git Cherry-pick**
    - Cherry-picking commits
    - Use cases and best practices
    - Handling conflicts
    - Advanced cherry-pick techniques
13. **Git Reset Deep Dive**
    - Understanding `git reset` modes
    - `--soft`, `--mixed`, `--hard` options

- Recovering from reset operations
- Safe reset practices

14. **Git Reflog and Recovery**
    - Using `git reflog` for recovery
    - Finding and restoring lost commits
    - Recovery strategies and workflows
    - Reflog limitations and best practices
15. **Git Bisect for Bug Hunting**
    - Binary search for bug identification
    - Automated bisect with scripts
    - Real-world debugging examples
    - Bisect best practices
16. **Git Tags**
    - Lightweight vs annotated tags
    - Tagging strategies for releases
    - Managing and sharing tags
    - Semantic versioning with tags
17. **Git Hooks**
    - Client-side and server-side hooks
    - Pre-commit, pre-push hooks
    - ESLint and Prettier integration
    - Hook management and sharing
18. **Conventional Commits**
    - Commit message standards
    - Conventional commit format
    - Automation and tooling
    - Team adoption strategies

🛠️ Tools and Workflows

19. **GitHub CLI**
    - GitHub CLI installation and setup
    - Managing PRs, issues, and repositories
    - Automation and scripting
    - Advanced GitHub CLI features
20. **GitHub Actions Basics**
    - CI/CD with GitHub Actions
    - Workflow creation and management
    - Common actions and use cases

    - Best practices and optimization

21. **Maintaining Clean Git History**
    - Principles of clean history
    - Team workflows and conventions
    - Commit organization techniques
    - Automated history management

22. **Branching Strategies**
    - Git Flow vs GitHub Flow
    - Trunk-based development
    - Choosing the right strategy
    - Implementation guidelines

23. **Force Push Safety**
    - Understanding `--force-with-lease`
    - Safe force push practices
    - Team coordination and policies
    - Recovery from force push issues

🎯 Learning Path Recommendations

For Complete Beginners

1. Start with Introduction
2. Master Basic Commands
3. Learn Remote Repositories
4. Understand Gitignore
5. Practice Branching Basics

For Developers with Basic Git Knowledge

1. Review Merge vs Rebase
2. Learn Git Stash
3. Master Pull Requests
4. Understand Forking

For Advanced Users

1. Master Advanced Rebase
2. Learn Git Reset
3. Understand Recovery
4. Implement Clean History
5. Choose Branching Strategy

For Team Leads and DevOps

1. Study Branching Strategies

2. Implement GitHub Actions
3. Establish Clean History practices
4. Set up Git Hooks
5. Define Force Push Safety policies

🚀 Quick Start Guide

Prerequisites

- Git installed on your system
- GitHub account created
- Basic command line knowledge

Setup Checklist

```
# 1. Configure Git
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# 2. Set up SSH (recommended)
ssh-keygen -t ed25519 -C "your.email@example.com"
cat ~/.ssh/id_ed25519.pub  # Add to GitHub

# 3. Create your first repository
mkdir my-project
cd my-project
git init
echo "# My Project" > README.md
git add README.md
git commit -m "Initial commit"

# 4. Connect to GitHub
gh repo create my-project --public
git remote add origin git@github.com:username/my-project.git
git push -u origin main
```

📖 How to Use This Tutorial

Reading Order

- **Sequential**: Follow the numbered order for comprehensive learning
- **Topic-based**: Jump to specific topics based on your needs
- **Reference**: Use as a reference guide for specific commands

Hands-on Practice

Each tutorial includes:

- ✅ **Practical examples** you can try locally
- 🔧 **Real-world scenarios** and use cases

- 💡 **Best practices** and tips
- ⚠️ **Common pitfalls** and how to avoid them
- 🆘 **Troubleshooting** guides

Code Examples

All code examples are:

- **Copy-pasteable** - Ready to run in your terminal
- **Commented** - Explanations for each step
- **Progressive** - Building on previous concepts
- **Tested** - Verified to work as described

🤝 Contributing

This tutorial is open source and welcomes contributions!

How to Contribute

1. Fork this repository
2. Create a feature branch: `git checkout -b improve-tutorial`
3. Make your changes and test them
4. Commit with conventional format: `git commit -m "docs: improve rebase examples"`
5. Push and create a pull request

Contribution Guidelines

- **Accuracy**: Ensure all commands and examples work
- **Clarity**: Write clear, beginner-friendly explanations
- **Completeness**: Include practical examples and use cases
- **Consistency**: Follow the established format and style
- **Testing**: Verify all examples work as described

Areas for Contribution

- 📝 **Content improvements** - Better explanations, more examples
- 🐛 **Bug fixes** - Correct errors in commands or explanations
- 🌟 **New topics** - Additional advanced topics or tools
- 🎨 **Visual aids** - Diagrams, screenshots, or ASCII art
- 🔧 **Tooling** - Scripts, automation, or helpful utilities

📚 Additional Resources

Official Documentation

- Git Documentation
- GitHub Docs
- GitHub CLI Manual

Interactive Learning

- Learn Git Branching - Visual Git tutorial
- GitHub Skills - Hands-on GitHub courses
- Git Immersion - Step-by-step Git tutorial

Advanced Topics

- Pro Git Book - Comprehensive Git guide
- Atlassian Git Tutorials - In-depth tutorials
- Git Workflows - Workflow comparisons

Tools and Extensions

- GitHub Desktop - GUI for Git and GitHub
- GitKraken - Advanced Git GUI
- Sourcetree - Free Git GUI
- VS Code Git Extensions - IDE integrations

Version and Updates

Current Version: 1.0.0
Last Updated: December 2024
Compatibility: Git 2.30+, GitHub CLI 2.0+

Changelog

- **v1.0.0** - Initial comprehensive tutorial release
  - 23 detailed tutorial files
  - Beginner to advanced coverage
  - Practical examples and best practices
  - Team workflow guidance

Planned Updates

- Advanced GitHub Actions workflows
- Git LFS (Large File Storage) tutorial
- Monorepo management strategies
- Advanced conflict resolution techniques
- Git performance optimization

License

Acknowledgments

Special thanks to:

- The Git development team for creating an amazing tool
- GitHub for revolutionizing collaborative development

- The open source community for continuous learning and sharing
- All contributors who help improve this tutorial

Happy Git-ing! 🚀

Start your journey with Introduction to Git and GitHub

Git and GitHub Introduction

What is Git?

Git is a **distributed version control system** that tracks changes in files and coordinates work among multiple people. It was created by Linus Torvalds in 2005 for Linux kernel development.

Key Features of Git:

- **Distributed**: Every developer has a complete copy of the project history
- **Fast**: Operations are performed locally, making them lightning fast
- **Branching**: Create, merge, and delete branches easily
- **Data Integrity**: Uses SHA-1 hashing to ensure data integrity
- **Non-linear Development**: Support for parallel development workflows

What is GitHub?

GitHub is a **cloud-based hosting service** for Git repositories. It provides a web-based interface and additional collaboration features on top of Git.

GitHub Features:

- **Repository Hosting**: Store your Git repositories in the cloud
- **Collaboration Tools**: Issues, Pull Requests, Project boards
- **Social Coding**: Follow developers, star repositories, contribute to open source
- **CI/CD**: GitHub Actions for automated workflows
- **Documentation**: Wiki, README files, GitHub Pages

Git vs GitHub

Git	GitHub
Version control system	Hosting service for Git repositories
Command-line tool	Web-based platform
Works offline	Requires internet connection
Free and open source	Free tier + paid plans
Local repositories	Remote repositories

Installing Git

Windows

```
# Download from https://git-scm.com/download/win
# Or use package manager
winget install Git.Git
```

macOS

```
# Using Homebrew
brew install git

# Using Xcode Command Line Tools
xcode-select --install
```

Linux (Ubuntu/Debian)

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

Initial Git Configuration

After installing Git, configure your identity:

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

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

# Set default editor
git config --global core.editor "code --wait"  # VS Code
# or
git config --global core.editor "vim"  # Vim

# View all configurations
git config --list

# View specific configuration
git config user.name
```

Creating Your First Repository

Method 1: Start Locally

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

# Initialize Git repository
git init

# Create a README file
echo "# My First Repository" > README.md

# Check status
git status
```

Method 2: Clone from GitHub

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

Real-World Example: Personal Portfolio Project

Let's create a personal portfolio website project to demonstrate Git basics:

```
# Create project directory
mkdir portfolio-website
cd portfolio-website

# Initialize Git repository
git init

# Create basic files
echo "# John Doe - Portfolio Website" > README.md
echo "<!DOCTYPE html><html><head><title>Portfolio</title></head><body><h1>Welcome
to my portfolio</h1></body></html>" > index.html
echo "body { font-family: Arial, sans-serif; }" > style.css

# Check what files Git sees
git status
```

Output:

```
On branch main

No commits yet
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
        index.html
        style.css

nothing added to commit but untracked files present (use "git add" to track)
```

Understanding Git Workflow

Git has three main areas:

1. **Working Directory**: Your local files
2. **Staging Area (Index)**: Files prepared for commit
3. **Repository**: Committed snapshots

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

Next Steps

Now that you understand what Git and GitHub are, let's move on to the basic commands in the next chapter: **Basic Git Commands**.

Quick Reference

```
# Essential first-time setup
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
git init                    # Initialize repository
git status                  # Check repository status
git clone <url>            # Clone remote repository
```

Next: Basic Git Commands

Basic Git Commands

This chapter covers the fundamental Git commands you'll use daily. We'll continue with our portfolio website example.

git init - Initialize a Repository

The `git init` command creates a new Git repository in the current directory.

```
# Initialize a new repository
git init

# Initialize with specific branch name
git init --initial-branch=main
# or
git init -b main
```

What happens:

- Creates a hidden `.git` directory
- Sets up the repository structure
- Creates the initial branch (usually `main` or `master`)

git add - Stage Changes

The `git add` command moves changes from the working directory to the staging area.

```
# Add a specific file
git add README.md

# Add multiple files
git add index.html style.css

# Add all files in current directory
git add .

# Add all files in repository
git add -A

# Add files interactively
git add -i

# Add only tracked files (ignore new files)
git add -u
```

Practical Example

```
# In our portfolio project
cd portfolio-website

# Check current status
git status

# Add README first
git add README.md
git status
```

```
# Add remaining files
git add index.html style.css
git status
```

Output after adding files:

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
        new file:   index.html
        new file:   style.css
```

git commit - Save Changes

The `git commit` command saves staged changes to the repository with a descriptive message.

```
# Commit with inline message
git commit -m "Add initial portfolio files"

# Commit with detailed message (opens editor)
git commit

# Add and commit tracked files in one step
git commit -am "Update existing files"

# Amend the last commit
git commit --amend -m "Updated commit message"
```

Writing Good Commit Messages

Good commit message structure:

```
Subject line (50 characters or less)

Optional body explaining what and why
(wrap at 72 characters)
```

Examples:

```
# Good
git commit -m "Add responsive navigation menu"
```

```
git commit -m "Fix login button alignment on mobile devices"
git commit -m "Update dependencies to latest versions"

# Bad
git commit -m "fix"
git commit -m "changes"
git commit -m "updated stuff"
```

Complete Example

```
# Commit our portfolio files
git commit -m "Initial commit: Add portfolio website structure

- Add README.md with project description
- Add index.html with basic HTML structure
- Add style.css with basic styling"
```

Output:

```
[main (root-commit) a1b2c3d] Initial commit: Add portfolio website structure
 3 files changed, 15 insertions(+)
 create mode 100644 README.md
 create mode 100644 index.html
 create mode 100644 style.css
```

git status - Check Repository Status

The `git status` command shows the current state of your working directory and staging area.

```
# Basic status
git status

# Short format
git status -s
# or
git status --short

# Show branch info
git status -b
```

Status Output Explained

```
# Create a new file and modify existing one
echo "console.log('Hello World');" > script.js
```

```
echo "<script src='script.js'></script>" >> index.html

git status
```

Output:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        script.js

no changes added to commit (use "git add" to track)
```

Status indicators:

- `modified:` - File has been changed since last commit
- `new file:` - File is staged and will be included in next commit
- `deleted:` - File has been removed
- `Untracked files:` - Files Git doesn't know about

git log - View Commit History

The `git log` command displays the commit history.

```
# Basic log
git log

# One line per commit
git log --oneline

# Show last 3 commits
git log -3

# Show commits with file changes
git log --stat

# Show commits with actual changes
git log -p

# Graphical representation
git log --graph --oneline

# Pretty format
git log --pretty=format:"%h - %an, %ar : %s"
```

Practical Examples

```
# Add and commit the new files
git add script.js index.html
git commit -m "Add JavaScript functionality"

# View commit history
git log --oneline
```

Output:

```
b2c3d4e Add JavaScript functionality
a1b2c3d Initial commit: Add portfolio website structure
```

```
# Detailed log with changes
git log --stat
```

Output:

```
commit b2c3d4e (HEAD -> main)
Author: Your Name <your.email@example.com>
Date:   Mon Jan 15 10:30:00 2024 -0500

    Add JavaScript functionality

 index.html | 1 +
 script.js  | 1 +
 2 files changed, 2 insertions(+)

commit a1b2c3d
Author: Your Name <your.email@example.com>
Date:   Mon Jan 15 10:00:00 2024 -0500

    Initial commit: Add portfolio website structure

 README.md  | 1 +
 index.html | 1 +
 style.css  | 1 +
 3 files changed, 3 insertions(+)
```

Practical Workflow Example

Let's simulate a typical development workflow:

```
# 1. Check current status
git status

# 2. Create a new feature
echo "<footer>© 2024 John Doe</footer>" >> index.html
echo "footer { text-align: center; margin-top: 50px; }" >> style.css

# 3. Check what changed
git status
git diff  # Shows exact changes (covered in intermediate section)

# 4. Stage changes
git add index.html style.css

# 5. Verify staged changes
git status

# 6. Commit changes
git commit -m "Add footer to portfolio page"

# 7. View updated history
git log --oneline
```

Common Scenarios and Solutions

Unstaging Files

```
# Unstage a specific file
git restore --staged filename.txt
# or (older Git versions)
git reset HEAD filename.txt

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

Discarding Changes

```
# Discard changes in working directory
git restore filename.txt
# or (older Git versions)
git checkout -- filename.txt

# Discard all changes
git restore .
```

Viewing Differences

```
# See changes in working directory
git diff

# See staged changes
git diff --staged
# or
git diff --cached
```

Best Practices

1. **Commit Often**: Make small, logical commits
2. **Write Clear Messages**: Describe what and why, not how
3. **Use Present Tense**: "Add feature" not "Added feature"
4. **Check Before Committing**: Always run `git status` and `git diff`
5. **Stage Selectively**: Don't always use `git add .`

Quick Reference

```
# Basic workflow
git status                  # Check status
git add <file>              # Stage specific file
git add .                   # Stage all files
git commit -m "message"     # Commit with message
git log                     # View history
git log --oneline           # Compact history

# Undoing changes
git restore <file>          # Discard working changes
git restore --staged <file> # Unstage file
git commit --amend          # Modify last commit
```

Previous: Git and GitHub Introduction
Next: Remote Repositories and GitHub

Remote Repositories and GitHub

This chapter covers working with remote repositories, connecting to GitHub, and synchronizing your local work with the cloud.

Understanding Remote Repositories

A **remote repository** is a version of your project hosted on a server (like GitHub, GitLab, or Bitbucket). It allows:

- **Backup**: Your code is safely stored in the cloud

- **Collaboration**: Multiple developers can work on the same project
- **Sharing**: Others can access and contribute to your code
- **Deployment**: Automated deployments from your repository

Creating a Repository on GitHub

Step 1: Create Repository on GitHub

1. Go to GitHub.com and sign in
2. Click the "+" icon → "New repository"
3. Fill in repository details:
    - **Repository name**: `portfolio-website`
    - **Description**: "My personal portfolio website"
    - **Visibility**: Public or Private
    - **Initialize**: Don't check any boxes (we already have local files)
4. Click "Create repository"

Step 2: Connect Local Repository to GitHub

```
# Add remote origin (replace with your GitHub username)
git remote add origin https://github.com/yourusername/portfolio-website.git

# Verify remote was added
git remote -v
```

Output:

```
origin  https://github.com/yourusername/portfolio-website.git (fetch)
origin  https://github.com/yourusername/portfolio-website.git (push)
```

git remote - Managing Remote Repositories

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

# Add a remote
git remote add <name> <url>
git remote add origin https://github.com/user/repo.git

# Remove a remote
git remote remove <name>
git remote rm origin

# Rename a remote
git remote rename <old-name> <new-name>
```

```
git remote rename origin upstream

# Change remote URL
git remote set-url origin https://github.com/user/new-repo.git

# Show remote details
git remote show origin
```

git push - Upload Changes to Remote

The `git push` command uploads your local commits to a remote repository.

```
# Push to remote repository
git push <remote> <branch>
git push origin main

# Push and set upstream (first time)
git push -u origin main
# or
git push --set-upstream origin main

# Push all branches
git push --all origin

# Push tags
git push --tags

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

First Push Example

```
# In our portfolio project
cd portfolio-website

# Push to GitHub for the first time
git push -u origin main
```

Output:

```
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 1.2 KiB | 1.2 MiB/s, done.
```

```
Total 9 (delta 1), reused 0 (delta 0)
To https://github.com/yourusername/portfolio-website.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

git pull - Download Changes from Remote

The `git pull` command downloads and merges changes from a remote repository.

```
# Pull from remote
git pull <remote> <branch>
git pull origin main

# Pull with rebase instead of merge
git pull --rebase origin main

# Pull all branches
git pull --all
```

git fetch vs git pull

```
# git fetch: Download changes but don't merge
git fetch origin
git fetch origin main

# git pull: Download and merge changes
git pull origin main
# Equivalent to:
git fetch origin main
git merge origin/main
```

Cloning Repositories

git clone - Copy Remote Repository

```
# Clone a repository
git clone <url>
git clone https://github.com/user/repo.git

# Clone to specific directory
git clone https://github.com/user/repo.git my-project

# Clone specific branch
git clone -b develop https://github.com/user/repo.git
```

```
# Shallow clone (recent history only)
git clone --depth 1 https://github.com/user/repo.git
```

Practical Example: Contributing to Open Source

```
# Clone a popular repository
git clone https://github.com/microsoft/vscode.git
cd vscode

# Check remote configuration
git remote -v

# Check current branch
git branch

# View recent commits
git log --oneline -5
```

.gitignore - Ignoring Files

The `.gitignore` file tells Git which files to ignore.

Creating .gitignore

```
# Create .gitignore file
touch .gitignore
# or on Windows
echo. > .gitignore
```

Common .gitignore Patterns

```
# 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 outputs
build/
```

```
dist/
*.min.js
*.min.css

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

# OS generated files
.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db

# Logs
logs
*.log

# Temporary files
*.tmp
*.temp

# Compiled files
*.class
*.o
*.pyc
__pycache__/

# Archives
*.zip
*.tar.gz
*.rar

# Database
*.sqlite
*.db
```

.gitignore for Our Portfolio Project

```
# Create .gitignore for portfolio
cat > .gitignore << EOF
# Development files
*.log
*.tmp
```

```
# OS files
.DS_Store
Thumbs.db

# Editor files
.vscode/
.idea/

# Build files
dist/
build/
EOF

# Add and commit .gitignore
git add .gitignore
git commit -m "Add .gitignore file"
git push origin main
```

.gitignore Rules

```
# Ignore specific file
config.txt

# Ignore all files with extension
*.log

# Ignore directory
node_modules/

# Ignore files in directory
logs/*.log

# Ignore files in all subdirectories
**/*.log

# Don't ignore specific file (exception)
!important.log

# Ignore files only in root
/config.txt

# Comments start with #
# This is a comment
```

Working with Existing Repositories

Scenario 1: Join an Existing Project

```

```

```
# Clone the project
git clone https://github.com/company/project.git
cd project

# Check project structure
ls -la

# View recent activity
git log --oneline -10

# Check branches
git branch -a

# Install dependencies (if applicable)
npm install  # for Node.js projects
# or
pip install -r requirements.txt  # for Python projects
```

Scenario 2: Sync Your Fork

```
# Add upstream remote (original repository)
git remote add upstream https://github.com/original/repo.git

# Fetch upstream changes
git fetch upstream

# Switch to main branch
git checkout main

# Merge upstream changes
git merge upstream/main

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

Authentication with GitHub

Using Personal Access Tokens (Recommended)

1. Go to GitHub Settings → Developer settings → Personal access tokens
2. Generate new token with appropriate scopes
3. Use token as password when prompted

```
# When pushing, use token as password
git push origin main
# Username: yourusername
# Password: ghp_xxxxxxxxxxxxxxxxxxxx (your token)
```

Using SSH Keys (Advanced)

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

# Add to SSH agent
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Copy public key to clipboard
cat ~/.ssh/id_ed25519.pub
# Add this to GitHub Settings → SSH and GPG keys

# Test connection
ssh -T git@github.com

# Use SSH URL for remote
git remote set-url origin git@github.com:username/repo.git
```

Practical Workflow Example

Let's simulate a complete workflow:

```
# 1. Make changes locally
echo "<p>Updated portfolio with new projects</p>" >> index.html

# 2. Stage and commit
git add index.html
git commit -m "Update portfolio with new projects section"

# 3. Push to GitHub
git push origin main

# 4. Simulate collaborator changes (on GitHub web interface)
# Edit README.md directly on GitHub and commit

# 5. Pull collaborator changes
git pull origin main

# 6. View updated history
git log --oneline -5
```

Troubleshooting Common Issues

Issue 1: Push Rejected

```
# Error: Updates were rejected because the remote contains work
git pull origin main  # Pull first
git push origin main  # Then push
```

Issue 2: Merge Conflicts

```
# When pulling creates conflicts
git pull origin main
# Fix conflicts in files
git add .
git commit -m "Resolve merge conflicts"
git push origin main
```

Issue 3: Wrong Remote URL

```
# Check current remote
git remote -v

# Update remote URL
git remote set-url origin https://github.com/correct/repo.git
```

Best Practices

1. **Always Pull Before Push**: Avoid conflicts by staying up-to-date
2. **Use Meaningful Commit Messages**: Help collaborators understand changes
3. **Keep .gitignore Updated**: Don't commit sensitive or generated files
4. **Use Branches**: Don't work directly on main for features
5. **Regular Backups**: Push frequently to avoid losing work

Quick Reference

```
# Remote management
git remote add origin <url>    # Add remote
git remote -v                  # List remotes
git remote show origin         # Show remote details

# Synchronization
git push origin main           # Push to remote
git pull origin main           # Pull from remote
git fetch origin              # Download without merging

# Repository operations
git clone <url>               # Clone repository
git clone <url> <directory>   # Clone to specific directory
```

```
# .gitignore
echo "file.txt" >> .gitignore # Add file to ignore
git add .gitignore            # Commit .gitignore changes
```

Previous: Basic Git Commands
Next: Branching Basics

Branching Basics

Branching is one of Git's most powerful features. It allows you to diverge from the main line of development and work on features, experiments, or bug fixes in isolation.

What is a Branch?

A **branch** is a lightweight, movable pointer to a specific commit. Think of it as an independent line of development.

```
main:     A---B---C---D
               \
feature:        E---F
```

Why Use Branches?

- **Isolation**: Work on features without affecting main code
- **Experimentation**: Try new ideas safely
- **Collaboration**: Multiple developers can work simultaneously
- **Organization**: Separate different types of work
- **Code Review**: Review changes before merging

git branch - Managing Branches

Viewing Branches

```
# List local branches
git branch

# List all branches (local and remote)
git branch -a

# List remote branches only
git branch -r

# List branches with last commit
git branch -v
```

```
# List merged branches
git branch --merged

# List unmerged branches
git branch --no-merged
```

Creating Branches

```
# Create new branch
git branch <branch-name>
git branch feature/user-authentication

# Create and switch to new branch
git checkout -b <branch-name>
git checkout -b feature/user-authentication

# Create branch from specific commit
git branch <branch-name> <commit-hash>
git branch hotfix/bug-123 a1b2c3d

# Create branch from remote branch
git checkout -b local-branch origin/remote-branch
```

Deleting Branches

```
# Delete merged branch
git branch -d <branch-name>
git branch -d feature/completed-feature

# Force delete unmerged branch
git branch -D <branch-name>
git branch -D feature/abandoned-feature

# Delete remote branch
git push origin --delete <branch-name>
git push origin --delete feature/old-feature
```

git checkout vs git switch

Git 2.23 introduced `git switch` as a clearer alternative to `git checkout` for branch operations.

git checkout (Traditional)

```
# Switch to existing branch
git checkout <branch-name>
git checkout main
```

```
git checkout feature/user-auth

# Create and switch to new branch
git checkout -b <branch-name>
git checkout -b feature/new-feature

# Switch to previous branch
git checkout -

# Checkout specific commit (detached HEAD)
git checkout <commit-hash>
git checkout a1b2c3d
```

git switch (Modern)

```
# Switch to existing branch
git switch <branch-name>
git switch main
git switch feature/user-auth

# Create and switch to new branch
git switch -c <branch-name>
git switch -c feature/new-feature

# Switch to previous branch
git switch -

# Switch to remote branch
git switch <remote-branch>
git switch origin/feature-branch
```

git restore (File Operations)

```
# Restore file from staging
git restore --staged <file>

# Restore file from working directory
git restore <file>

# Restore file from specific commit
git restore --source=<commit> <file>
```

Practical Branching Example

Let's add a contact form to our portfolio website using branches:

Step 1: Create Feature Branch

```
# Start from main branch
git checkout main
git pull origin main  # Ensure we're up-to-date

# Create feature branch
git checkout -b feature/contact-form

# Verify current branch
git branch
```

Output:

```
* feature/contact-form
  main
```

Step 2: Develop the Feature

```
# Create contact form HTML
cat > contact.html << EOF
<!DOCTYPE html>
<html>
<head>
    <title>Contact - Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>Contact Me</h1>
    <form id="contact-form">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>

        <label for="message">Message:</label>
        <textarea id="message" name="message" required></textarea>

        <button type="submit">Send Message</button>
    </form>
    <script src="contact.js"></script>
</body>
</html>
EOF

# Create contact form JavaScript
cat > contact.js << EOF
document.getElementById('contact-form').addEventListener('submit', function(e) {
    e.preventDefault();
```

```
    const name = document.getElementById('name').value;
    const email = document.getElementById('email').value;
    const message = document.getElementById('message').value;

    // Simple validation
    if (name && email && message) {
        alert('Thank you for your message! I will get back to you soon.');
        this.reset();
    } else {
        alert('Please fill in all fields.');
    }
});
EOF

# Add contact form styles
cat >> style.css << EOF

/* Contact Form Styles */
form {
    max-width: 600px;
    margin: 20px auto;
    padding: 20px;
    border: 1px solid #ddd;
    border-radius: 5px;
}

label {
    display: block;
    margin-top: 10px;
    font-weight: bold;
}

input, textarea {
    width: 100%;
    padding: 8px;
    margin-top: 5px;
    border: 1px solid #ccc;
    border-radius: 3px;
    box-sizing: border-box;
}

textarea {
    height: 100px;
    resize: vertical;
}

button {
    background-color: #007bff;
    color: white;
    padding: 10px 20px;
    border: none;
    border-radius: 3px;
    cursor: pointer;
```

```
    margin-top: 10px;
}

button:hover {
    background-color: #0056b3;
}
EOF

# Update main page with navigation
cat > index.html << EOF
<!DOCTYPE html>
<html>
<head>
    <title>Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <nav>
        <a href="index.html">Home</a>
        <a href="contact.html">Contact</a>
    </nav>
    <h1>Welcome to my portfolio</h1>
    <p>This is my personal portfolio website.</p>
    <footer>© 2024 John Doe</footer>
    <script src="script.js"></script>
</body>
</html>
EOF
```

Step 3: Commit Changes

```
# Check what we've changed
git status

# Add all new files
git add .

# Commit the feature
git commit -m "Add contact form feature

- Create contact.html with form structure
- Add contact.js for form validation
- Update style.css with form styling
- Add navigation to index.html"

# View commit history
git log --oneline
```

Step 4: Push Feature Branch

```
# Push feature branch to remote
git push -u origin feature/contact-form
```

Branch Naming Conventions

Common Patterns

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

# Bug fix branches
bugfix/login-error
bugfix/mobile-layout
hotfix/security-patch

# Release branches
release/v1.2.0
release/2024-01-15

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

# Personal branches
john/refactor-database
mary/update-dependencies
```

Naming Best Practices

1. **Use lowercase**: `feature/user-auth` not `Feature/User-Auth`
2. **Use hyphens**: `feature/user-auth` not `feature/user_auth`
3. **Be descriptive**: `feature/user-authentication` not `feature/auth`
4. **Include ticket numbers**: `feature/JIRA-123-user-auth`
5. **Use prefixes**: `feature/`, `bugfix/`, `hotfix/`

Working with Remote Branches

Tracking Remote Branches

```
# List remote branches
git branch -r

# Create local branch from remote
git checkout -b local-branch origin/remote-branch
```

```
# Set upstream for existing branch
git branch --set-upstream-to=origin/remote-branch local-branch
# or
git push -u origin local-branch

# Check tracking information
git branch -vv
```

Fetching Remote Branches

```
# Fetch all remote branches
git fetch origin

# Fetch specific remote branch
git fetch origin feature/new-feature

# Prune deleted remote branches
git fetch --prune origin
# or
git remote prune origin
```

Branch Workflows

Feature Branch Workflow

```
# 1. Start from main
git checkout main
git pull origin main

# 2. Create feature branch
git checkout -b feature/new-feature

# 3. Work on feature
# ... make changes ...
git add .
git commit -m "Implement new feature"

# 4. Push feature branch
git push -u origin feature/new-feature

# 5. Create Pull Request on GitHub
# 6. After review and approval, merge
# 7. Clean up
git checkout main
git pull origin main
git branch -d feature/new-feature
git push origin --delete feature/new-feature
```

Hotfix Workflow

```
# 1. Create hotfix from main
git checkout main
git checkout -b hotfix/critical-bug

# 2. Fix the bug
# ... make changes ...
git add .
git commit -m "Fix critical security vulnerability"

# 3. Push and merge immediately
git push -u origin hotfix/critical-bug
# Merge via Pull Request or directly

# 4. Clean up
git checkout main
git pull origin main
git branch -d hotfix/critical-bug
```

Viewing Branch History

Graphical Log

```
# Simple graph
git log --graph --oneline

# Detailed graph
git log --graph --pretty=format:"%h -%d %s (%cr) <%an>" --abbrev-commit

# All branches
git log --graph --oneline --all

# Specific branches
git log --graph --oneline main feature/contact-form
```

Example output:

```
* b2c3d4e (HEAD -> feature/contact-form) Add contact form feature
* a1b2c3d (origin/main, main) Add footer to portfolio page
* 9e8f7g6 Add JavaScript functionality
* 5h4i3j2 Initial commit: Add portfolio website structure
```

Comparing Branches

```
# Show commits in feature branch not in main
git log main..feature/contact-form

# Show commits in main not in feature branch
git log feature/contact-form..main

# Show differences between branches
git diff main..feature/contact-form

# Show file differences
git diff main..feature/contact-form --name-only
```

Common Branch Operations

Switching Between Branches

```
# Switch to main
git switch main

# Switch to feature branch
git switch feature/contact-form

# Switch to previous branch
git switch -

# Create and switch in one command
git switch -c feature/new-feature
```

Renaming Branches

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

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

# Update remote after renaming
git push origin --delete old-name
git push -u origin new-name
```

Troubleshooting

Issue 1: Can't Switch Branches (Uncommitted Changes)

```

```

```
# Option 1: Commit changes
git add .
git commit -m "Work in progress"
git switch other-branch

# Option 2: Stash changes
git stash
git switch other-branch
# Later: git stash pop

# Option 3: Discard changes
git restore .
git switch other-branch
```

Issue 2: Branch Not Found

```
# Fetch latest remote branches
git fetch origin

# List all branches
git branch -a

# Create local branch from remote
git checkout -b local-branch origin/remote-branch
```

Issue 3: Accidentally Deleted Branch

```
# Find the commit hash
git reflog

# Recreate branch
git checkout -b recovered-branch <commit-hash>
```

Best Practices

1. **Keep Branches Small**: Focus on single features or fixes
2. **Use Descriptive Names**: Make purpose clear from name
3. **Regular Updates**: Keep feature branches updated with main
4. **Clean Up**: Delete merged branches promptly
5. **Test Before Merging**: Ensure branches work correctly
6. **Document Changes**: Use clear commit messages

Quick Reference

```
# Branch management
git branch                      # List branches
git branch <name>               # Create branch
git checkout -b <name>          # Create and switch
git switch <name>               # Switch branch
git branch -d <name>            # Delete branch

# Remote branches
git push -u origin <branch>     # Push new branch
git fetch origin                # Fetch remote branches
git branch -r                   # List remote branches

# Branch information
git log --graph --oneline       # Visual history
git branch -v                   # Branches with commits
git diff main..feature          # Compare branches
```

Previous: Remote Repositories and GitHub
Next: Git Merge vs Rebase

Git Merge vs Rebase

Merge and rebase are two fundamental ways to integrate changes from one branch into another. Understanding the difference is crucial for maintaining a clean Git history.

Understanding the Difference

Merge: Preserves History

```
Before merge:
main:     A---B---C
               \
feature:        D---E

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

Rebase: Rewrites History

```
Before rebase:
main:     A---B---C
               \
feature:        D---E
```

```
After rebase:
main:     A---B---C
feature:           D'---E' (commits moved)
```

git merge - Combining Branches

Basic Merge

```
# Switch to target branch
git checkout main

# Merge feature branch
git merge feature/contact-form

# Push merged changes
git push origin main
```

Types of Merges

1. Fast-Forward Merge

When the target branch hasn't diverged:

```
Before:
main:     A---B
feature:      C---D

After fast-forward:
main:     A---B---C---D
```

```
# Fast-forward merge (default when possible)
git merge feature/simple-update

# Force merge commit even for fast-forward
git merge --no-ff feature/simple-update
```

2. Three-Way Merge

When both branches have diverged:

```
Before:
main:     A---B---C
               \
```

```
feature:        D---E

After three-way merge:
main:     A---B---C---F
               \     /
feature:        D---E
```

```
# Three-way merge (automatic when branches diverged)
git merge feature/new-feature
```

Merge Options

```
# Standard merge
git merge feature/branch

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

# Fast-forward only (fail if not possible)
git merge --ff-only feature/branch

# Squash merge (combine all commits into one)
git merge --squash feature/branch

# Merge with custom message
git merge -m "Merge feature: Add user authentication" feature/auth
```

git rebase - Rewriting History

Basic Rebase

```
# Switch to feature branch
git checkout feature/contact-form

# Rebase onto main
git rebase main

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

# Push rebased branch (force required)
git push --force-with-lease origin feature/contact-form
```

Rebase Process Step-by-Step

```
Initial state:
main:     A---B---C
               \
feature:        D---E

Step 1: Git finds common ancestor (B)
Step 2: Git temporarily removes D and E
Step 3: Git applies D and E on top of C

Final state:
main:     A---B---C
feature:           D'---E'
```

Interactive Rebase

```
# Interactive rebase for last 3 commits
git rebase -i HEAD~3

# Interactive rebase from specific commit
git rebase -i <commit-hash>

# Interactive rebase onto main
git rebase -i main
```

Interactive Rebase Commands

```
pick a1b2c3d Add contact form HTML
squash d4e5f6g Add contact form styling
reword g7h8i9j Add contact form validation
drop j0k1l2m Remove debug code
```

Commands:

- `pick` (p): Use commit as-is
- `reword` (r): Use commit but edit message
- `edit` (e): Use commit but stop for amending
- `squash` (s): Combine with previous commit
- `fixup` (f): Like squash but discard message
- `drop` (d): Remove commit entirely

Practical Example: Feature Development

Let's demonstrate both approaches with our portfolio project:

Setup: Create Divergent Branches

```
# Start from main
git checkout main

# Create and work on feature branch
git checkout -b feature/blog-section
echo "<section id='blog'><h2>Blog Posts</h2></section>" >> index.html
git add index.html
git commit -m "Add blog section to homepage"

echo "#blog { margin: 20px 0; }" >> style.css
git add style.css
git commit -m "Style blog section"

# Meanwhile, someone else updated main
git checkout main
echo "<meta name='viewport' content='width=device-width, initial-scale=1'>" >>
index.html
git add index.html
git commit -m "Add responsive viewport meta tag"

# Now we have divergent branches
git log --graph --oneline --all
```

Output:

```
* f9e8d7c (HEAD -> main) Add responsive viewport meta tag
| * c6b5a4d (feature/blog-section) Style blog section
| * 3d2e1f0 Add blog section to homepage
|/
* a1b2c3d Add contact form feature
```

Approach 1: Using Merge

```
# Switch to main and merge
git checkout main
git merge feature/blog-section
```

If there are conflicts:

```
# Git will show conflict markers
cat index.html
# <<<<<<< HEAD
# <meta name="viewport" content="width=device-width, initial-scale=1">
# =======
# <section id="blog"><h2>Blog Posts</h2></section>
# >>>>>>> feature/blog-section
```

```
# Resolve conflicts manually
# Edit index.html to include both changes

# Mark as resolved
git add index.html
git commit -m "Merge feature/blog-section"

# View result
git log --graph --oneline
```

Result:

```
*   g7f6e5d (HEAD -> main) Merge feature/blog-section
|\
| * c6b5a4d Style blog section
| * 3d2e1f0 Add blog section to homepage
* | f9e8d7c Add responsive viewport meta tag
|/
* a1b2c3d Add contact form feature
```

Approach 2: Using Rebase

```
# Reset to before merge (for demonstration)
git reset --hard f9e8d7c

# Switch to feature branch and rebase
git checkout feature/blog-section
git rebase main
```

If there are conflicts:

```
# Resolve conflicts in files
# Edit index.html to include both changes

# Mark as resolved and continue
git add index.html
git rebase --continue

# View result
git log --graph --oneline
```

Result:

```
```

```
* h8g7f6e (HEAD -> feature/blog-section) Style blog section
* e5d4c3b Add blog section to homepage
* f9e8d7c (main) Add responsive viewport meta tag
* a1b2c3d Add contact form feature
```

```
# Now merge into main (fast-forward)
git checkout main
git merge feature/blog-section

# Final clean history
git log --graph --oneline
```

Result:

```
* h8g7f6e (HEAD -> main, feature/blog-section) Style blog section
* e5d4c3b Add blog section to homepage
* f9e8d7c Add responsive viewport meta tag
* a1b2c3d Add contact form feature
```

When to Use Merge vs Rebase

Use Merge When:

1. **Working on public/shared branches**

```
# Safe for shared branches
git checkout main
git merge feature/shared-feature
```

2. **Want to preserve exact history**

```
# Keeps timeline of when work was done
git merge --no-ff feature/important-feature
```

3. **Working with release branches**

```
# Clear merge points for releases
git checkout release/v1.0
git merge feature/last-minute-fix
```

4. **Team prefers merge commits**

```
# Some teams like explicit merge commits
git merge --no-ff feature/team-feature
```

Use Rebase When:

1. **Cleaning up feature branch history**

```
# Before creating pull request
git checkout feature/my-feature
git rebase main
```

2. **Want linear history**

```
# Creates clean, linear timeline
git rebase main
```

3. **Squashing related commits**

```
# Combine multiple small commits
git rebase -i HEAD~3
```

4. **Working on private branches**

```
# Safe to rewrite history on private branches
git rebase main
```

Advanced Rebase Techniques

Squashing Commits

```
# Interactive rebase to squash last 3 commits
git rebase -i HEAD~3
```

In the editor:

```
pick a1b2c3d Add blog HTML structure
squash b2c3d4e Fix blog HTML formatting
```

```
squash c3d4e5f Add blog CSS styles
```

Result: Three commits become one clean commit.

Splitting Commits

```
# Start interactive rebase
git rebase -i HEAD~2

# Mark commit for editing
# edit a1b2c3d Large commit with multiple changes

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

# Stage and commit changes separately
git add file1.html
git commit -m "Add HTML structure"

git add file2.css
git commit -m "Add CSS styles"

# Continue rebase
git rebase --continue
```

Rebase onto Different Branch

```
# Move feature branch from old-main to new-main
git rebase --onto new-main old-main feature/branch
```

```
Before:
old-main: A---B---C
new-main: A---D---E
feature:      F---G (based on old-main)

After:
old-main: A---B---C
new-main: A---D---E
feature:          F'---G' (now based on new-main)
```

Handling Conflicts

Merge Conflicts

```
# During merge
git merge feature/branch
# CONFLICT (content): Merge conflict in file.txt

# View conflicted files
git status

# Edit files to resolve conflicts
# Remove conflict markers: <<<<<<<, =======, >>>>>>>

# Mark as resolved
git add file.txt
git commit -m "Resolve merge conflicts"
```

Rebase Conflicts

```
# During rebase
git rebase main
# CONFLICT (content): Merge conflict in file.txt

# View conflicted files
git status

# Edit files to resolve conflicts

# Mark as resolved and continue
git add file.txt
git rebase --continue

# Or abort rebase
git rebase --abort
```

Merge vs Rebase Comparison

Aspect	Merge	Rebase
History	Preserves original timeline	Creates linear timeline
Commits	Keeps all original commits	Can modify/combine commits
Conflicts	Resolve once	May resolve multiple times
Safety	Safe for shared branches	Dangerous for shared branches
Traceability	Shows when branches merged	Shows logical progression
Complexity	Simpler to understand	More complex but cleaner

Best Practices

Golden Rules

1. **Never rebase public branches**

```
# DON'T do this if others are using the branch
git checkout main
git rebase feature/branch  # DANGEROUS!
```

2. **Rebase before merging**

```
# Clean up feature branch first
git checkout feature/branch
git rebase main

# Then merge cleanly
git checkout main
git merge feature/branch
```

3. **Use --force-with-lease for safety**

```
# Safer than --force
git push --force-with-lease origin feature/branch
```

Workflow Recommendations

For Feature Branches:

```
# 1. Create feature branch
git checkout -b feature/new-feature

# 2. Work and commit
git add .
git commit -m "Implement feature"

# 3. Before creating PR, clean up
git rebase -i main  # Squash/clean commits
git rebase main     # Update with latest main

# 4. Push and create PR
git push --force-with-lease origin feature/new-feature

# 5. Merge via PR (usually squash merge)
```

For Hotfixes:

```
# 1. Create hotfix from main
git checkout -b hotfix/critical-fix

# 2. Fix and commit
git add .
git commit -m "Fix critical bug"

# 3. Merge directly (no rebase needed)
git checkout main
git merge hotfix/critical-fix
git push origin main
```

Troubleshooting

Undo Merge

```
# Undo merge (if not pushed)
git reset --hard HEAD~1

# Undo merge (if pushed) - creates new commit
git revert -m 1 <merge-commit-hash>
```

Undo Rebase

```
# Find original commit
git reflog

# Reset to before rebase
git reset --hard HEAD@{5}  # Use appropriate reflog entry
```

Abort Operations

```
# Abort merge
git merge --abort

# Abort rebase
git rebase --abort
```

Quick Reference

```
# Merge commands
git merge <branch>            # Standard merge
git merge --no-ff <branch>    # Force merge commit
```

```
git merge --squash <branch>   # Squash all commits

# Rebase commands
git rebase <branch>           # Rebase onto branch
git rebase -i <commit>        # Interactive rebase
git rebase --continue         # Continue after conflict
git rebase --abort            # Abort rebase

# Conflict resolution
git status                    # See conflicted files
git add <file>               # Mark conflict resolved
git merge --abort            # Abort merge
git rebase --abort           # Abort rebase
```

Previous: Branching Basics
Next: Git Stash

Git Stash

Git stash is a powerful feature that temporarily saves your uncommitted changes, allowing you to switch branches or pull updates without committing incomplete work.

What is Git Stash?

Stash creates a temporary snapshot of your:

- **Working directory changes** (modified files)
- **Staged changes** (files added to index)
- **Optionally untracked files**

Think of it as a "clipboard" for your work-in-progress.

Basic Stash Operations

git stash - Save Current Work

```
# Stash working directory and staged changes
git stash

# Stash with a descriptive message
git stash push -m "Work in progress on contact form validation"

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

# Stash everything including ignored files
```

```
git stash -a
# or
git stash --all
```

git stash pop - Restore Latest Stash

```
# Apply and remove latest stash
git stash pop

# Apply specific stash and remove it
git stash pop stash@{2}
```

git stash apply - Restore Without Removing

```
# Apply latest stash but keep it in stash list
git stash apply

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

Practical Example: Emergency Bug Fix

Let's simulate a common scenario where you're working on a feature but need to fix an urgent bug:

Scenario Setup

```
# Working on portfolio enhancement
cd portfolio-website
git checkout -b feature/portfolio-gallery

# Start working on image gallery
cat > gallery.html << EOF
<!DOCTYPE html>
<html>
<head>
    <title>Portfolio Gallery</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <h1>My Work</h1>
    <div class="gallery">
        <!-- TODO: Add image grid -->
        <div class="image-placeholder">Project 1</div>
        <div class="image-placeholder">Project 2</div>
    </div>
</body>
```

```
</html>
EOF

# Add some CSS (work in progress)
cat >> style.css << EOF

/* Gallery Styles - Work in Progress */
.gallery {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
    gap: 20px;
    padding: 20px;
}

.image-placeholder {
    height: 200px;
    background-color: #f0f0f0;
    border: 2px dashed #ccc;
    display: flex;
    align-items: center;
    justify-content: center;
    /* TODO: Add hover effects */
}
EOF

# Check current status
git status
```

Output:

```
On branch feature/portfolio-gallery
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified:   style.css

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        gallery.html

no changes added to commit
```

Emergency: Need to Fix Bug on Main

```
# Urgent bug report comes in!
# Need to switch to main branch immediately

# Stash current work
git stash push -u -m "WIP: Portfolio gallery layout and styling"
```

```
# Verify clean working directory
git status
```

Output:

```
On branch feature/portfolio-gallery
nothing to commit, working tree clean
```

```
# Switch to main and fix bug
git checkout main
git pull origin main

# Create hotfix branch
git checkout -b hotfix/contact-form-validation

# Fix the bug (contact form wasn't validating email properly)
sed -i 's/type="email"/type="email" pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]
{2,}$"/' contact.html

# Commit the fix
git add contact.html
git commit -m "Fix email validation pattern in contact form"

# Merge hotfix
git checkout main
git merge hotfix/contact-form-validation
git push origin main

# Clean up hotfix branch
git branch -d hotfix/contact-form-validation
```

Return to Feature Work

```
# Switch back to feature branch
git checkout feature/portfolio-gallery

# Restore stashed work
git stash pop

# Verify work is restored
git status
ls -la  # gallery.html should be back
```

Managing Multiple Stashes

Viewing Stashes

```
# List all stashes
git stash list

# Show stash contents
git stash show
git stash show stash@{1}

# Show detailed diff
git stash show -p
git stash show -p stash@{2}
```

Creating Multiple Stashes

```
# Create first stash
echo "<!-- Adding navigation -->" >> index.html
git stash push -m "Navigation improvements"

# Make different changes
echo "/* Mobile responsive styles */" >> style.css
git stash push -m "Mobile responsive work"

# Make more changes
echo "console.log('Debug mode');" >> script.js
git stash push -m "Debug logging"

# View all stashes
git stash list
```

Output:

```
stash@{0}: On feature/portfolio-gallery: Debug logging
stash@{1}: On feature/portfolio-gallery: Mobile responsive work
stash@{2}: On feature/portfolio-gallery: Navigation improvements
stash@{3}: On feature/portfolio-gallery: WIP: Portfolio gallery layout and styling
```

Applying Specific Stashes

```
# Apply specific stash by index
git stash apply stash@{2}

# Apply specific stash by partial name
git stash apply "stash@{1}"
```

```
# Pop specific stash
git stash pop stash@{0}
```

Advanced Stash Operations

Partial Stashing

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

# Interactive stashing (choose hunks)
git stash -p
# or
git stash --patch
```

Interactive stashing example:

```
diff --git a/style.css b/style.css
index 1234567..abcdefg 100644
--- a/style.css
+++ b/style.css
@@ -10,6 +10,10 @@ body {
     margin: 0;
 }

+/* New navigation styles */
+nav { background: blue; }
+
Stash this hunk [y,n,q,a,d,/,s,e,?]?
```

Options:

- `y` - stash this hunk
- `n` - do not stash this hunk
- `q` - quit; do not stash this hunk or any remaining
- `a` - stash this hunk and all later hunks
- `d` - do not stash this hunk or any later hunks
- `s` - split the current hunk into smaller hunks
- `e` - manually edit the current hunk

Stash Branch

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

This command:

1. Creates a new branch from the commit where stash was created
2. Applies the stash to the new branch
3. Removes the stash if applied successfully

Stash Drop and Clear

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

# Remove latest stash
git stash drop

# Remove all stashes
git stash clear
```

Stash with Untracked and Ignored Files

Including Untracked Files

```
# Create some untracked files
echo "temp data" > temp.txt
echo "cache data" > cache.dat

# Stash including untracked files
git stash -u -m "Include untracked files"

# Verify files are stashed
ls  # temp.txt and cache.dat should be gone

# Restore
git stash pop
ls  # files should be back
```

Including Ignored Files

```
# Add to .gitignore
echo "*.log" >> .gitignore
echo "node_modules/" >> .gitignore

# Create ignored files
echo "debug info" > debug.log
mkdir node_modules
echo "dependency" > node_modules/package.txt

# Stash everything including ignored files
```

```
git stash -a -m "Include ignored files too"

# Check what was stashed
git stash show -p
```

Practical Workflows

Workflow 1: Quick Branch Switch

```
# Working on feature
# ... making changes ...

# Need to quickly check something on main
git stash
git checkout main
# ... check something ...
git checkout feature/branch
git stash pop

# Continue working
```

Workflow 2: Pull Latest Changes

```
# Working on feature with uncommitted changes
git status  # shows modified files

# Need to pull latest from remote
git stash
git pull origin main
git stash pop

# Resolve any conflicts if they occur
```

Workflow 3: Experiment Safely

```
# Save current work
git stash push -m "Stable version before experiment"

# Try experimental changes
# ... make risky changes ...

# If experiment fails
git reset --hard HEAD
git stash pop  # Restore stable version

# If experiment succeeds
```

```
git add .
git commit -m "Successful experiment"
git stash drop  # Remove backup
```

Stash Conflicts and Resolution

When Stash Pop Conflicts

```
# Create conflict scenario
echo "Original content" > conflict-file.txt
git add conflict-file.txt
git commit -m "Add conflict file"

# Modify and stash
echo "Stashed changes" > conflict-file.txt
git stash

# Modify same file differently
echo "Different changes" > conflict-file.txt
git add conflict-file.txt
git commit -m "Different changes to same file"

# Try to pop stash (will conflict)
git stash pop
```

Output:

```
Auto-merging conflict-file.txt
CONFLICT (content): Merge conflict in conflict-file.txt
The stash entry is kept in case you need it again.
```

Resolving Stash Conflicts

```
# View conflicted file
cat conflict-file.txt
```

Output:

```
<<<<<<< Updated upstream
Different changes
=======
Stashed changes
>>>>>>> Stashed changes
```

```
# Resolve conflict manually
echo "Merged: Different changes and stashed changes" > conflict-file.txt

# Mark as resolved
git add conflict-file.txt

# Stash is automatically removed after successful resolution
git status
```

Stash Best Practices

1. Use Descriptive Messages

```
# Good
git stash push -m "WIP: User authentication form validation"
git stash push -m "Debugging CSS grid layout issues"
git stash push -m "Experimental API integration approach"

# Bad
git stash
git stash push -m "stuff"
git stash push -m "changes"
```

2. Clean Up Regularly

```
# Review stashes periodically
git stash list

# Remove old/unnecessary stashes
git stash drop stash@{5}

# Clear all if needed
git stash clear
```

3. Prefer Commits for Important Work

```
# For important work, commit instead of stash
git add .
git commit -m "WIP: Important feature progress"

# Can always amend or squash later
git commit --amend
```

4. Use Stash for Temporary Storage

```
# Good use cases:
# - Quick branch switches
# - Pulling updates
# - Emergency bug fixes
# - Experimental changes

# Avoid for:
# - Long-term storage
# - Sharing with others
# - Important work
```

Troubleshooting

Stash Not Working

```
# Check if there are actually changes to stash
git status

# Stash requires changes in working directory or index
# If no changes, stash will say "No local changes to save"
```

Lost Stash

```
# Find lost stashes in reflog
git fsck --unreachable | grep commit | cut -d' ' -f3 | xargs git log --merges --
no-walk --grep=WIP

# Or check reflog
git reflog | grep stash
```

Stash Apply Failed

```
# If stash apply fails due to conflicts
git stash show -p  # See what's in the stash
git reset --hard   # Reset to clean state
git stash apply    # Try again

# Or create a branch from stash
git stash branch temp-branch
```

Quick Reference

```
# Basic stash operations
git stash                     # Stash changes
git stash push -m "message"   # Stash with message
git stash pop                 # Apply and remove latest stash
git stash apply               # Apply without removing
git stash list                # List all stashes

# Advanced operations
git stash -u                  # Include untracked files
git stash -p                  # Interactive stashing
git stash show -p             # Show stash diff
git stash drop stash@{1}      # Remove specific stash
git stash clear               # Remove all stashes

# Specific stash operations
git stash apply stash@{2}     # Apply specific stash
git stash branch new-branch   # Create branch from stash
```

Previous: Git Merge vs Rebase
Next: Git Diff and Log

Git Diff and Log

Git diff and log are essential tools for understanding changes in your repository. They help you see what changed, when it changed, and who changed it.

git diff - Viewing Changes

Basic Diff Operations

```
# Show unstaged changes (working directory vs staging area)
git diff

# Show staged changes (staging area vs last commit)
git diff --staged
# or
git diff --cached

# Show all changes (working directory vs last commit)
git diff HEAD
```

Practical Example: Portfolio Updates

Let's create some changes to demonstrate diff:

```
# Make changes to existing files
cd portfolio-website

# Modify index.html
cat > index.html << EOF
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>John Doe - Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <nav>
            <a href="index.html">Home</a>
            <a href="contact.html">Contact</a>
            <a href="gallery.html">Gallery</a>
        </nav>
    </header>
    <main>
        <h1>Welcome to my portfolio</h1>
        <p>I'm a web developer passionate about creating amazing user experiences.
</p>
        <section id="skills">
            <h2>Skills</h2>
            <ul>
                <li>HTML5 & CSS3</li>
                <li>JavaScript (ES6+)</li>
                <li>React & Node.js</li>
            </ul>
        </section>
    </main>
    <footer>© 2024 John Doe</footer>
    <script src="script.js"></script>
</body>
</html>
EOF

# Add some changes to CSS
cat >> style.css << EOF

/* Header Styles */
header {
    background-color: #333;
    padding: 1rem 0;
}

nav a {
    color: white;
    text-decoration: none;
    margin: 0 1rem;
```

```
    padding: 0.5rem 1rem;
    border-radius: 4px;
    transition: background-color 0.3s;
}

nav a:hover {
    background-color: #555;
}

/* Skills Section */
#skills {
    margin: 2rem 0;
    padding: 1rem;
    background-color: #f8f9fa;
    border-radius: 8px;
}

#skills ul {
    list-style-type: none;
    padding: 0;
}

#skills li {
    padding: 0.5rem 0;
    border-bottom: 1px solid #dee2e6;
}

#skills li:last-child {
    border-bottom: none;
}
EOF

# View unstaged changes
git diff
```

Understanding Diff Output

```
diff --git a/index.html b/index.html
index 1234567..abcdefg 100644
--- a/index.html
+++ b/index.html
@@ -1,8 +1,25 @@
 <!DOCTYPE html>
-<html>
+<html lang="en">
 <head>
+    <meta charset="UTF-8">
+    <meta name="viewport" content="width=device-width, initial-scale=1.0">
-    <title>Portfolio</title>
+    <title>John Doe - Portfolio</title>
     <link rel="stylesheet" href="style.css">
```

```
 </head>
 <body>
+    <header>
+        <nav>
+            <a href="index.html">Home</a>
+            <a href="contact.html">Contact</a>
+            <a href="gallery.html">Gallery</a>
+        </nav>
+    </header>
```

Diff symbols:

- `---` and `+++`: Old and new file versions
- `@@`: Hunk header showing line numbers
- `-`: Lines removed (red in terminal)
- `+`: Lines added (green in terminal)
- : Unchanged lines (context)

Diff Options and Formats

```
# Word-level diff (more granular)
git diff --word-diff

# Character-level diff
git diff --word-diff=color --word-diff-regex=.

# Ignore whitespace changes
git diff -w
# or
git diff --ignore-all-space

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

# Show file names with status
git diff --name-status

# Show statistics
git diff --stat

# Compact summary
git diff --shortstat
```

Comparing Specific Commits

```
# Stage some changes first
git add index.html
git commit -m "Improve HTML structure and add navigation"
```

```
# Add more changes
echo "/* Additional mobile styles */" >> style.css
git add style.css
git commit -m "Add header and skills styling"

# Compare commits
git diff HEAD~2 HEAD~1    # Compare two commits ago with one commit ago
git diff HEAD~1 HEAD      # Compare last commit with current
git diff a1b2c3d..e4f5g6h # Compare specific commits

# Compare branches
git diff main..feature/branch
git diff main...feature/branch  # Three dots: common ancestor to branch tip
```

Diff for Specific Files

```
# Diff specific file
git diff index.html
git diff HEAD~1 index.html

# Diff multiple files
git diff index.html style.css

# Diff files in directory
git diff src/

# Diff with path filtering
git diff '*.css'
git diff '*.js'
```

git log - Viewing History

Basic Log Operations

```
# Basic log
git log

# One line per commit
git log --oneline

# Show last N commits
git log -5
git log -n 3

# Show commits since date
git log --since="2024-01-01"
git log --since="2 weeks ago"
git log --since="yesterday"
```

```
# Show commits until date
git log --until="2024-01-31"
git log --before="1 week ago"
```

Pretty Formats

```
# Built-in formats
git log --pretty=oneline
git log --pretty=short
git log --pretty=medium
git log --pretty=full
git log --pretty=fuller

# Custom format
git log --pretty=format:"%h - %an, %ar : %s"
git log --pretty=format:"%C(yellow)%h%C(reset) - %C(blue)%an%C(reset),
%C(green)%ar%C(reset) : %s"
```

Custom Format Placeholders

Placeholder	Description
%H	Full commit hash
%h	Abbreviated commit hash
%an	Author name
%ae	Author email
%ad	Author date
%ar	Author date, relative
%cn	Committer name
%cd	Committer date
%cr	Committer date, relative
%s	Subject (commit message)
%b	Body (commit message)
%d	Ref names (branches, tags)

Advanced Log Examples

```
# Beautiful one-line format
git log --pretty=format:"%C(auto)%h%d %s %C(black)%C(bold)%cr" --graph
```

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

# Show file changes
git log --stat
git log --name-only
git log --name-status

# Show actual changes
git log -p
git log --patch

# Limit patch output
git log -p -2  # Show patches for last 2 commits
```

Filtering Commits

By Author

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

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

# Exclude author
git log --author="^(?!John Doe).*$" --perl-regexp
```

By Message

```
# Commits containing specific text
git log --grep="fix"
git log --grep="feature"

# Case insensitive
git log --grep="FIX" -i

# Multiple patterns
git log --grep="fix" --grep="bug" --all-match

# Invert match
git log --grep="WIP" --invert-grep
```

By File Changes

```
# Commits that changed specific file
git log -- index.html
git log --follow -- index.html  # Follow renames

# Commits that changed files matching pattern
git log -- '*.css'
git log -- src/

# Commits that added or removed specific text
git log -S "function validateForm"  # Pickaxe search
git log -G "function.*validate"     # Regex search
```

By Date Range

```
# Specific date range
git log --since="2024-01-01" --until="2024-01-31"

# Relative dates
git log --since="2 weeks ago"
git log --since="yesterday" --until="today"

# Specific format
git log --since="Jan 1 2024" --until="Jan 31 2024"
```

Graphical Log

```
# Simple graph
git log --graph --oneline

# Detailed graph
git log --graph --pretty=format:"%h -%d %s (%cr) <%an>" --abbrev-commit

# All branches
git log --graph --oneline --all

# Specific branches
git log --graph --oneline main feature/branch
```

Example output:

```
* b2c3d4e (HEAD -> feature/portfolio) Add header and skills styling
* a1b2c3d Improve HTML structure and add navigation
| * f9e8d7c (main) Fix contact form validation
|/
```

```
* 5h4i3j2 Add contact form feature
* 9k8l7m6 Initial commit
```

Practical Log Workflows

Daily Standup Report

```
# What did I work on yesterday?
git log --author="$(git config user.name)" --since="yesterday" --pretty=format:"%h
%s"

# What did the team work on this week?
git log --since="1 week ago" --pretty=format:"%h - %an: %s" --graph
```

Release Notes

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

# Features and fixes since last release
git log v1.0.0..HEAD --grep="feat:" --grep="fix:" --pretty=format:"- %s"
```

Code Review Preparation

```
# Changes in feature branch
git log main..feature/branch --oneline

# Detailed changes for review
git log main..feature/branch --stat

# All changes with patches
git log main..feature/branch -p
```

Advanced Diff and Log Techniques

Diff Tools Integration

```
# Configure external diff tool
git config --global diff.tool vimdiff
# or
git config --global diff.tool vscode

# Use external tool
```

```
git difftool
git difftool HEAD~1

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

Log Aliases

Create useful aliases for common log commands:

```
# Add to ~/.gitconfig or use git config
git config --global alias.lg "log --graph --pretty=format:'%C(yellow)%h%C(reset) -
%C(red)%d%C(reset) %s %C(green)(%cr) %C(blue)<%an>%C(reset)' --abbrev-commit"

git config --global alias.lga "log --graph --pretty=format:'%C(yellow)%h%C(reset)
-%C(red)%d%C(reset) %s %C(green)(%cr) %C(blue)<%an>%C(reset)' --abbrev-commit --
all"

git config --global alias.ls "log --pretty=format:'%C(yellow)%h
%C(blue)%ad%C(reset) %C(white)%s%C(reset) %C(green)[%an]%C(reset)' --decorate --
date=short"

# Usage
git lg
git lga
git ls
```

Searching Through History

```
# Find when a line was added
git log -S "specific code line" --source --all

# Find when a function was modified
git log -G "function myFunction" --patch

# Find commits that touch specific lines
git log -L 10,20:filename.js
git log -L :functionName:filename.js
```

Performance and Large Repositories

```
# Limit log output for performance
git log --oneline -100  # Last 100 commits

# Skip merge commits
git log --no-merges
```

```
# Only merge commits
git log --merges

# Simplify history
git log --simplify-by-decoration
```

Practical Examples

Example 1: Bug Investigation

```
# Find when a bug was introduced
# Look for commits that changed the problematic file
git log --oneline -- problematic-file.js

# Check specific function changes
git log -G "buggyFunction" --patch -- problematic-file.js

# See what changed in suspicious commit
git show a1b2c3d

# Compare with previous version
git diff a1b2c3d~1 a1b2c3d -- problematic-file.js
```

Example 2: Code Review

```
# Review all changes in feature branch
git log main..feature/new-feature --stat

# See detailed changes
git diff main...feature/new-feature

# Check individual commits
git log main..feature/new-feature --oneline
git show commit-hash
```

Example 3: Release Preparation

```
# Generate changelog
git log v1.0.0..HEAD --pretty=format:"- %s" --reverse

# Check what files changed
git diff --name-only v1.0.0..HEAD

# Get statistics
git diff --stat v1.0.0..HEAD
```

Troubleshooting

Large Diff Output

```
# Limit context lines
git diff -U1  # 1 line of context instead of 3

# Show only changed files
git diff --name-only

# Use pager
git diff | less
```

Log Performance Issues

```
# Limit history depth
git log --max-count=50

# Skip expensive operations
git log --no-patch

# Use shallow clone for large repos
git clone --depth 50 <url>
```

Finding Lost Changes

```
# Search in all commits (including deleted)
git log --all --full-history -- deleted-file.txt

# Search in reflog
git reflog | grep "search term"

# Find dangling commits
git fsck --lost-found
```

Best Practices

1. **Use Descriptive Commit Messages**: Makes log more useful
2. **Regular Small Commits**: Easier to track changes
3. **Use Aliases**: Create shortcuts for common log formats
4. **Learn Format Strings**: Customize output for your needs
5. **Combine with Other Tools**: Use with grep, less, etc.

Quick Reference

```
# Diff commands
git diff                        # Working directory vs staging
git diff --staged               # Staging vs last commit
git diff HEAD                   # Working directory vs last commit
git diff commit1..commit2       # Between commits
git diff --stat                 # Show statistics

# Log commands
git log                         # Basic log
git log --oneline               # Compact format
git log --graph                 # Show branch graph
git log --stat                  # Show file changes
git log -p                      # Show patches
git log --author="name"         # Filter by author
git log --grep="text"           # Filter by message
git log --since="date"          # Filter by date
git log -- filename             # Filter by file

# Useful aliases
git config --global alias.lg "log --graph --oneline --decorate"
git config --global alias.lga "log --graph --oneline --decorate --all"
```

Previous: Git Stash
Next: Merge Conflicts

Resolving Merge Conflicts

Merge conflicts occur when Git cannot automatically merge changes from different branches. Understanding how to resolve them is essential for collaborative development.

What Causes Merge Conflicts?

Conflicts happen when:

- **Same lines modified**: Two branches change the same lines differently
- **File deleted vs modified**: One branch deletes a file, another modifies it
- **Rename conflicts**: File renamed differently in each branch
- **Binary file conflicts**: Binary files changed in both branches

Understanding Conflict Markers

When Git encounters a conflict, it adds special markers to the file:

```
Content from merging branch
```

Markers explained:

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

Creating a Conflict Scenario

Let's create a realistic conflict scenario with our portfolio project:

Setup: Create Conflicting Changes

```
# Start from main branch
cd portfolio-website
git checkout main
git pull origin main

# Create feature branch for team member 1
git checkout -b feature/update-homepage

# Team member 1 updates the homepage
cat > index.html << EOF
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>John Doe - Senior Web Developer</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <nav>
            <a href="index.html">Home</a>
            <a href="about.html">About</a>
            <a href="contact.html">Contact</a>
        </nav>
    </header>
    <main>
        <h1>Senior Web Developer</h1>
        <p>I specialize in modern web technologies and have 5+ years of
experience.</p>
        <section id="experience">
            <h2>Experience</h2>
            <ul>
                <li>Senior Developer at TechCorp (2022-Present)</li>
                <li>Full-Stack Developer at StartupXYZ (2020-2022)</li>
            </ul>
        </section>
    </main>
    <footer>© 2024 John Doe - All Rights Reserved</footer>
</body>
</html>
EOF
```

```
# Commit team member 1's changes
git add index.html
git commit -m "Update homepage with senior developer profile"

# Switch back to main and create another feature branch
git checkout main
git checkout -b feature/redesign-homepage

# Team member 2 redesigns the homepage differently
cat > index.html << EOF
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>John Doe - Creative Developer</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <nav>
            <a href="index.html">Home</a>
            <a href="portfolio.html">Portfolio</a>
            <a href="contact.html">Contact</a>
        </nav>
    </header>
    <main>
        <section class="hero">
            <h1>Creative Web Developer</h1>
            <p>Passionate about creating beautiful and functional web experiences.
</p>
        </section>
        <section id="skills">
            <h2>Core Skills</h2>
            <div class="skills-grid">
                <div>Frontend Development</div>
                <div>UI/UX Design</div>
                <div>JavaScript Frameworks</div>
            </div>
        </section>
    </main>
    <footer>© 2024 John Doe</footer>
</body>
</html>
EOF

# Commit team member 2's changes
git add index.html
git commit -m "Redesign homepage with creative developer focus"
```

Simulate Merge Conflict

```
# Merge first feature into main
git checkout main
git merge feature/update-homepage

# Now try to merge second feature (this will conflict)
git merge feature/redesign-homepage
```

Output:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Examining the Conflict

```
# Check repository status
git status
```

Output:

```
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   index.html

no changes added to commit (use "git add" to track)
```

```
# View the conflicted file
cat index.html
```

Output:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
    <title>John Doe - Creative Developer</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <header>
      <nav>
        <a href="index.html">Home</a>
        <a href="portfolio.html">Portfolio</a>
        <a href="contact.html">Contact</a>
      </nav>
    </header>
    <main>
      <section class="hero">
        <h1>Creative Web Developer</h1>
        <p>
          Passionate about creating beautiful and functional web experiences.
        </p>
      </section>
      <section id="skills">
        <h2>Core Skills</h2>
        <div class="skills-grid">
          <div>Frontend Development</div>
          <div>UI/UX Design</div>
          <div>JavaScript Frameworks</div>
        </div>
      </section>
    </main>
    <footer>© 2024 John Doe</footer>
  </body>
</html>
```

Manual Conflict Resolution

Step 1: Analyze the Conflict

Look at both versions and decide:

- Which parts to keep from each version
- What new content to create
- How to combine the best of both

Step 2: Edit the File

```
# Edit index.html to resolve conflicts
cat > index.html << EOF
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>John Doe - Senior Creative Developer</title>
```

```
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <nav>
            <a href="index.html">Home</a>
            <a href="about.html">About</a>
            <a href="portfolio.html">Portfolio</a>
            <a href="contact.html">Contact</a>
        </nav>
    </header>
    <main>
        <section class="hero">
            <h1>Senior Creative Developer</h1>
            <p>I specialize in modern web technologies with 5+ years of
experience, passionate about creating beautiful and functional web experiences.
</p>
        </section>
        <section id="experience">
            <h2>Professional Experience</h2>
            <ul>
                <li>Senior Developer at TechCorp (2022-Present)</li>
                <li>Full-Stack Developer at StartupXYZ (2020-2022)</li>
            </ul>
        </section>
        <section id="skills">
            <h2>Core Skills</h2>
            <div class="skills-grid">
                <div>Frontend Development</div>
                <div>UI/UX Design</div>
                <div>JavaScript Frameworks</div>
            </div>
        </section>
    </main>
    <footer>© 2024 John Doe - All Rights Reserved</footer>
</body>
</html>
EOF
```

Step 3: Mark as Resolved

```
# Add the resolved file
git add index.html

# Check status
git status
```

Output:

```
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   index.html
```

Step 4: Complete the Merge

```
# Commit the merge
git commit -m "Merge feature/redesign-homepage: Combine senior and creative
developer profiles

- Merged navigation from both branches
- Combined experience and skills sections
- Updated title to reflect both senior and creative aspects"

# View the result
git log --oneline --graph -5
```

Using Merge Tools

Configure Merge Tool

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

# Configure Vim as merge tool
git config --global merge.tool vimdiff

# Configure other tools
git config --global merge.tool meld      # Linux
git config --global merge.tool opendiff  # macOS
git config --global merge.tool tortoisemerge  # Windows
```

Using Merge Tool

```
# When conflicts occur, launch merge tool
git mergetool

# Launch specific tool
git mergetool --tool=vscode
```

```
# Don't prompt for each file
git config --global mergetool.prompt false
```

VS Code Merge Interface

VS Code provides a visual interface with:

- **Accept Current Change**: Keep HEAD version
- **Accept Incoming Change**: Keep merging branch version
- **Accept Both Changes**: Keep both versions
- **Compare Changes**: Side-by-side comparison

Different Types of Conflicts

1. Content Conflicts

Most common - same lines modified differently:

```
# Create content conflict
echo "Version A content" > conflict-file.txt
git add conflict-file.txt
git commit -m "Add version A"

git checkout -b feature-b
echo "Version B content" > conflict-file.txt
git add conflict-file.txt
git commit -m "Add version B"

git checkout main
echo "Version A updated" > conflict-file.txt
git add conflict-file.txt
git commit -m "Update version A"

git merge feature-b  # Conflict!
```

2. Delete vs Modify Conflicts

```
# Create delete vs modify conflict
echo "Original content" > temp-file.txt
git add temp-file.txt
git commit -m "Add temp file"

# Branch 1: Delete file
git checkout -b delete-branch
git rm temp-file.txt
git commit -m "Remove temp file"

# Branch 2: Modify file
```

```
git checkout main
echo "Modified content" > temp-file.txt
git add temp-file.txt
git commit -m "Modify temp file"

git merge delete-branch  # Conflict!
```

Resolution:

```
# Decide to keep the file
git add temp-file.txt

# Or decide to delete it
git rm temp-file.txt

git commit -m "Resolve delete vs modify conflict"
```

3. Rename Conflicts

```
# Create rename conflict
echo "File content" > original.txt
git add original.txt
git commit -m "Add original file"

# Branch 1: Rename to name A
git checkout -b rename-a
git mv original.txt renamed-a.txt
git commit -m "Rename to renamed-a.txt"

# Branch 2: Rename to name B
git checkout main
git mv original.txt renamed-b.txt
git commit -m "Rename to renamed-b.txt"

git merge rename-a  # Conflict!
```

Advanced Conflict Resolution

Using git diff for Conflicts

```
# Show conflicts in diff format
git diff

# Show conflicts with more context
git diff --conflict=diff3
```

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

Three-Way Merge View

```
# Configure diff3 conflict style
git config --global merge.conflictstyle diff3
```

This shows:

```
Merging branch content
```

Resolving Conflicts in Chunks

```
# Stage resolved parts of a file
git add --patch conflicted-file.txt

# Interactive staging
git add -i
```

Preventing Conflicts

1. Communication

```
# Check what others are working on
git log --oneline --since="1 week ago" --author="teammate"

# See current branches
git branch -a

# Check recent activity
git log --graph --oneline --all -10
```

2. Frequent Integration

```
# Regularly update feature branches
git checkout feature/my-feature
git fetch origin
git rebase origin/main

# Or merge main into feature
git merge origin/main
```

3. Small, Focused Changes

```
# Make smaller, focused commits
git add specific-file.js
git commit -m "Add specific function"

# Avoid large, sweeping changes
# Break big features into smaller parts
```

4. Code Organization

```
# Work on different files when possible
# Use modular architecture
# Separate concerns into different files
```

Conflict Resolution Strategies

Strategy 1: Accept One Side

```
# Accept current branch (ours)
git checkout --ours conflicted-file.txt
git add conflicted-file.txt

# Accept merging branch (theirs)
git checkout --theirs conflicted-file.txt
git add conflicted-file.txt
```

Strategy 2: Manual Merge

```
# Edit file manually to combine both sides
# Remove conflict markers
# Test the result
git add conflicted-file.txt
```

Strategy 3: Abort and Retry

```
# Abort current merge
git merge --abort

# Try different approach
git rebase feature/branch  # Instead of merge
```

```
# Or update branch first
git checkout feature/branch
git rebase main
git checkout main
git merge feature/branch
```

Testing After Resolution

```
# Always test after resolving conflicts
npm test            # Run test suite
npm run build       # Check if build works
npm start           # Test application

# Check syntax
npm run lint        # Run linter
git diff --check    # Check for whitespace errors
```

Best Practices

1. Understand the Changes

```
# Before resolving, understand what each side does
git show HEAD                    # Current branch changes
git show MERGE_HEAD             # Merging branch changes
git log --oneline HEAD..MERGE_HEAD  # Commits being merged
```

2. Communicate with Team

```
# If unsure, ask the author of conflicting changes
git log --oneline --author="teammate" -- conflicted-file.txt

# Check commit messages for context
git show commit-hash
```

3. Test Thoroughly

```
# Test the merged result
# Run automated tests
# Manual testing of affected features
# Code review if needed
```

4. Document Resolution

```
# Write clear merge commit messages
git commit -m "Merge feature/branch: Resolve navigation conflicts

- Combined navigation items from both branches
- Kept About page from feature/update-homepage
- Kept Portfolio page from feature/redesign-homepage
- Tested all navigation links"
```

Troubleshooting

Conflict Resolution Gone Wrong

```
# Undo merge commit (if not pushed)
git reset --hard HEAD~1

# Redo the merge
git merge feature/branch
```

Lost Changes During Resolution

```
# Check reflog for lost commits
git reflog

# Recover lost changes
git checkout commit-hash -- lost-file.txt
```

Merge Tool Issues

```
# Reset merge tool configuration
git config --global --unset merge.tool

# Use built-in merge resolution
git checkout --conflict=merge conflicted-file.txt
```

Quick Reference

```
# Conflict resolution workflow
git status                    # Check conflicted files
git diff                      # See conflicts
# Edit files to resolve conflicts
git add resolved-file.txt     # Mark as resolved
git commit                    # Complete merge
```

```
# Merge tools
git mergetool                 # Launch merge tool
git mergetool --tool=vscode   # Use specific tool

# Conflict strategies
git checkout --ours file.txt  # Accept current branch
git checkout --theirs file.txt # Accept merging branch
git merge --abort             # Abort merge

# Prevention
git fetch origin              # Stay updated
git rebase origin/main        # Update feature branch
git log --graph --oneline     # Check branch status
```

Previous: Git Diff and Log
Next: Pull Requests and Code Review

Pull Requests and Code Review

Pull Requests (PRs) are GitHub's mechanism for proposing changes, facilitating code review, and managing collaborative development. They're essential for maintaining code quality and team coordination.

What is a Pull Request?

A **Pull Request** is a request to merge changes from one branch into another. It provides:

- **Code Review**: Team members can review changes before merging
- **Discussion**: Comments and suggestions on specific lines
- **Testing**: Automated tests run on proposed changes
- **Documentation**: Clear description of what changes do
- **History**: Permanent record of why changes were made

Pull Request Workflow

```
1. Create feature branch
2. Make changes and commit
3. Push branch to GitHub
4. Create Pull Request
5. Code review and discussion
6. Address feedback
7. Merge when approved
8. Clean up branches
```

Creating Your First Pull Request

Step 1: Prepare Feature Branch

```
# Start from updated main branch
cd portfolio-website
git checkout main
git pull origin main

# Create feature branch
git checkout -b feature/add-projects-section

# Add projects section to portfolio
cat > projects.html << EOF
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Projects - John Doe</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <nav>
            <a href="index.html">Home</a>
            <a href="about.html">About</a>
            <a href="projects.html">Projects</a>
            <a href="contact.html">Contact</a>
        </nav>
    </header>
    <main>
        <h1>My Projects</h1>
        <div class="projects-grid">
            <article class="project-card">
                <h3>E-commerce Website</h3>
                <p>Full-stack e-commerce solution built with React and Node.js</p>
                <div class="tech-stack">
                    <span>React</span>
                    <span>Node.js</span>
                    <span>MongoDB</span>
                </div>
                <a href="#" class="project-link">View Project</a>
            </article>
            <article class="project-card">
                <h3>Task Management App</h3>
                <p>Collaborative task management tool with real-time updates</p>
                <div class="tech-stack">
                    <span>Vue.js</span>
                    <span>Express</span>
                    <span>Socket.io</span>
                </div>
                <a href="#" class="project-link">View Project</a>
            </article>
            <article class="project-card">
                <h3>Weather Dashboard</h3>
                <p>Responsive weather application with location-based
```

```
forecasts</p>
                <div class="tech-stack">
                    <span>JavaScript</span>
                    <span>API Integration</span>
                    <span>CSS Grid</span>
                </div>
                <a href="#" class="project-link">View Project</a>
            </article>
        </div>
    </main>
    <footer>© 2024 John Doe</footer>
</body>
</html>
EOF

# Add CSS for projects
cat >> style.css << EOF

/* Projects Section */
.projects-grid {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(300px, 1fr));
    gap: 2rem;
    padding: 2rem 0;
}

.project-card {
    background: #f8f9fa;
    border-radius: 8px;
    padding: 1.5rem;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
    transition: transform 0.3s ease, box-shadow 0.3s ease;
}

.project-card:hover {
    transform: translateY(-5px);
    box-shadow: 0 4px 8px rgba(0,0,0,0.15);
}

.project-card h3 {
    color: #333;
    margin-bottom: 1rem;
}

.project-card p {
    color: #666;
    margin-bottom: 1rem;
    line-height: 1.6;
}

.tech-stack {
    display: flex;
    flex-wrap: wrap;
    gap: 0.5rem;
```

```
    margin-bottom: 1rem;
}

.tech-stack span {
    background: #007bff;
    color: white;
    padding: 0.25rem 0.5rem;
    border-radius: 4px;
    font-size: 0.875rem;
}

.project-link {
    display: inline-block;
    background: #28a745;
    color: white;
    padding: 0.5rem 1rem;
    text-decoration: none;
    border-radius: 4px;
    transition: background-color 0.3s ease;
}

.project-link:hover {
    background: #218838;
}
EOF

# Update main navigation
sed -i 's|<a href="portfolio.html">Portfolio</a>|<a
href="projects.html">Projects</a>|' index.html

# Commit changes
git add .
git commit -m "Add projects section with responsive grid layout

- Create projects.html with project showcase
- Add responsive CSS grid for project cards
- Include hover effects and tech stack tags
- Update navigation in index.html"
```

Step 2: Push Branch to GitHub

```
# Push feature branch
git push -u origin feature/add-projects-section
```

Output:

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
```

```
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 2.1 KiB | 2.1 MiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
remote:
remote: Create a pull request for 'feature/add-projects-section' on GitHub by
visiting:
remote:      https://github.com/yourusername/portfolio-
website/pull/new/feature/add-projects-section
remote:
To https://github.com/yourusername/portfolio-website.git
 * [new branch]      feature/add-projects-section -> feature/add-projects-section
Branch 'feature/add-projects-section' set up to track remote branch 'feature/add-
projects-section' from 'origin'.
```

Step 3: Create Pull Request on GitHub

1. **Go to GitHub repository**
2. **Click "Compare & pull request"** (appears after pushing)
3. **Fill out PR template:**

```
## Description

Adds a comprehensive projects section to showcase portfolio work with responsive
design and interactive elements.

## Changes Made

- ✅ Created projects.html with project showcase
- ✅ Added responsive CSS grid layout
- ✅ Implemented hover effects for better UX
- ✅ Added tech stack tags for each project
- ✅ Updated main navigation

## Type of Change

- [x] New feature
- [ ] Bug fix
- [ ] Documentation update
- [ ] Performance improvement

## Testing

- [x] Tested responsive design on mobile/desktop
- [x] Verified all links work correctly
- [x] Checked cross-browser compatibility
- [x] Validated HTML and CSS

## Screenshots

[Add screenshots of the new projects section]
```

```
## Checklist

- [x] Code follows project style guidelines
- [x] Self-review completed
- [x] Comments added for complex logic
- [x] No console errors
- [x] Responsive design tested
```

4. **Select reviewers**
5. **Add labels** (feature, frontend, etc.)
6. **Create pull request**

Code Review Process

As a Reviewer

1. Review Checklist

```
## Code Review Checklist

### Functionality

- [ ] Does the code do what it's supposed to do?
- [ ] Are there any edge cases not handled?
- [ ] Is error handling appropriate?

### Code Quality

- [ ] Is the code readable and well-structured?
- [ ] Are variable and function names descriptive?
- [ ] Is there unnecessary code duplication?
- [ ] Are comments helpful and up-to-date?

### Performance

- [ ] Are there any performance concerns?
- [ ] Are images optimized?
- [ ] Is CSS efficient?

### Security

- [ ] Are there any security vulnerabilities?
- [ ] Is user input properly validated?
- [ ] Are sensitive data properly handled?

### Testing

- [ ] Are there adequate tests?
- [ ] Do all tests pass?
- [ ] Is the feature manually tested?
```

```
### Documentation

- [ ] Is documentation updated?
- [ ] Are breaking changes documented?
```

2. Providing Feedback

Good Review Comments:

```
# Suggestion for improvement

Consider using CSS custom properties for consistent spacing:

```css
:root {
  --spacing-sm: 0.5rem;
  --spacing-md: 1rem;
  --spacing-lg: 2rem;
}

.projects-grid {
  gap: var(--spacing-lg);
}
```
```

Question about implementation

Why did you choose CSS Grid over Flexbox here? Grid works great, just curious about the decision.

Positive feedback

Great use of hover effects! The subtle animation really enhances the user experience.

Security concern

The project links are currently placeholder. When implementing real links, make sure to add `rel="noopener"` for external links.

```
**Avoid:**
- "This is wrong" → "Consider this alternative approach..."
- "Bad code" → "This could be improved by..."
- "Fix this" → "What do you think about..."

#### 3. Review Types
```

```bash
# Approve PR
"LGTM! Great work on the responsive design. The hover effects are a nice touch."

# Request changes
"The functionality looks good, but I have a few suggestions for code organization.
Please see my inline comments."

# Comment only
"Nice implementation! I left a few optional suggestions for future improvements."
```

As a PR Author

1. Responding to Feedback

```bash
# Address feedback with new commits
git checkout feature/add-projects-section

# Make requested changes
echo ":root { --spacing-lg: 2rem; }" >> style.css
sed -i 's/gap: 2rem/gap: var(--spacing-lg)/' style.css

# Commit improvements
git add style.css
git commit -m "Use CSS custom properties for consistent spacing

Addresses review feedback about maintainable spacing values."

# Push updates
git push origin feature/add-projects-section
```

2. Responding to Comments

```
> Why did you choose CSS Grid over Flexbox here?

Good question! I chose CSS Grid because:

1. It handles both rows and columns automatically
2. The `auto-fit` and `minmax()` functions provide better responsive behavior
3. It's more semantic for this card-based layout

Flexbox would work too, but would require more media queries for the responsive
behavior.
```

Advanced PR Techniques

1. Draft Pull Requests

```
# Create draft PR for early feedback
# On GitHub: Check "Create draft pull request"
# Or use GitHub CLI:
gh pr create --draft --title "WIP: Add projects section" --body "Early version for feedback"
```

2. PR Templates

Create `.github/pull_request_template.md`:

```
## Description

Brief description of changes

## Type of Change

- [ ] Bug fix
- [ ] New feature
- [ ] Breaking change
- [ ] Documentation update

## Testing

- [ ] Unit tests pass
- [ ] Integration tests pass
- [ ] Manual testing completed

## Checklist

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

## Screenshots (if applicable)

## Additional Notes
```

3. Linking Issues

```
## Description

Implements user authentication system

Closes #123
```

```
Fixes #456
Resolves #789
```

4. Co-authored Commits

```
# When pair programming
git commit -m "Add user authentication

Co-authored-by: Jane Doe <jane@example.com>"
```

GitHub CLI for Pull Requests

Installation

```
# Windows
winget install GitHub.cli

# macOS
brew install gh

# Linux
sudo apt install gh
```

Authentication

```
# Login to GitHub
gh auth login

# Check status
gh auth status
```

PR Commands

```
# Create PR
gh pr create --title "Add projects section" --body "Adds responsive projects
showcase"

# Create draft PR
gh pr create --draft

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

```
# View PR
gh pr view 123
gh pr view --web  # Open in browser

# Check out PR locally
gh pr checkout 123

# Review PR
gh pr review 123 --approve
gh pr review 123 --request-changes --body "Please fix the CSS issues"

# Merge PR
gh pr merge 123
gh pr merge 123 --squash
gh pr merge 123 --rebase

# Close PR
gh pr close 123
```

PR Best Practices

1. Size and Scope

```
# Good: Small, focused PRs
git log --oneline main..feature/add-button  # 2-3 commits

# Avoid: Large, multi-purpose PRs
git log --oneline main..feature/redesign-everything  # 20+ commits
```

Guidelines:

- **< 400 lines changed**: Easy to review
- **400-1000 lines**: Manageable but requires focus
- **> 1000 lines**: Consider breaking into smaller PRs

2. Clear Descriptions

```
# Good PR Description

## What

Adds user authentication with JWT tokens

## Why

Users need to log in to access personalized features

## How
```

```
- Implemented login/logout endpoints
- Added JWT middleware for protected routes
- Created user registration flow

## Testing

- Added unit tests for auth functions
- Tested login flow manually
- Verified token expiration handling
```

3. Commit Organization

```
# Before creating PR, clean up commits
git rebase -i HEAD~5

# Squash related commits
# Fix commit messages
# Remove debug commits
```

4. Self-Review

```
# Review your own changes before creating PR
git diff main..feature/branch

# Check for:
# - Debug code left behind
# - TODO comments
# - Formatting issues
# - Missing documentation
```

Merge Strategies

1. Merge Commit

```
# Creates merge commit preserving branch history
git checkout main
git merge feature/branch
```

Result:

```
*   a1b2c3d Merge pull request #123 from feature/branch
|\
| * d4e5f6g Add projects section
```

```
| * g7h8i9j Update navigation
|/
* j0k1l2m Previous commit
```

2. Squash and Merge

```
# Combines all commits into single commit
# Available on GitHub PR interface
```

Result:

```
* a1b2c3d Add projects section (#123)
* j0k1l2m Previous commit
```

3. Rebase and Merge

```
# Replays commits without merge commit
git checkout main
git rebase feature/branch
```

Result:

```
* d4e5f6g Add projects section
* g7h8i9j Update navigation
* j0k1l2m Previous commit
```

Automated Checks

1. 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: Setup Node.js
```

```
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
      - name: Run linter
        run: npm run lint
      - name: Check formatting
        run: npm run format:check
```

2. Branch Protection Rules

```
# Configure on GitHub:
# Settings → Branches → Add rule

# Require:
# - Status checks to pass
# - Up-to-date branches
# - Review from code owners
# - Signed commits
```

Troubleshooting

PR Conflicts

```
# Update feature branch with latest main
git checkout feature/branch
git fetch origin
git rebase origin/main

# Resolve conflicts if any
# Push updated branch
git push --force-with-lease origin feature/branch
```

Failed Checks

```
# Fix issues locally
npm run lint:fix
npm test

# Commit fixes
git add .
git commit -m "Fix linting issues"
git push origin feature/branch
```

Accidental Force Push

```
# Check reflog for lost commits
git reflog

# Recover if needed
git reset --hard commit-hash
git push --force-with-lease origin feature/branch
```

Quick Reference

```
# PR workflow
git checkout -b feature/branch    # Create feature branch
# ... make changes ...
git push -u origin feature/branch # Push to GitHub
# Create PR on GitHub interface

# GitHub CLI
gh pr create                      # Create PR
gh pr list                        # List PRs
gh pr checkout 123                # Checkout PR locally
gh pr review 123 --approve        # Approve PR
gh pr merge 123 --squash          # Merge PR

# Update PR
git add .
git commit -m "Address feedback"
git push origin feature/branch

# Clean up after merge
git checkout main
git pull origin main
git branch -d feature/branch
```

Previous: Merge Conflicts
Next: Forking and Upstream

Forking and Upstream Remotes

Forking is essential for contributing to open source projects and collaborating on repositories you don't have write access to. This chapter covers the complete fork workflow.

What is Forking?

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

- **Experiment freely** without affecting the original project
- **Propose changes** via pull requests
- **Customize** the project for your needs
- **Learn** from existing codebases
- **Contribute** to open source projects

Fork Workflow Overview

```
Original Repo (upstream)
    ↓ fork
Your Fork (origin)
    ↓ clone
Local Repository
    ↓ changes
Feature Branch
    ↓ push
Your Fork
    ↓ pull request
Original Repo
```

Setting Up a Fork

Step 1: Fork on GitHub

1. **Navigate to the repository** you want to contribute to
2. **Click "Fork"** button (top-right corner)
3. **Choose destination** (your account or organization)
4. **Wait for fork creation**

Example: Forking a popular open source project

```
Original: https://github.com/microsoft/vscode
Your Fork: https://github.com/yourusername/vscode
```

Step 2: Clone Your Fork

```
# Clone your fork (not the original)
git clone https://github.com/yourusername/vscode.git
cd vscode

# Check current remotes
git remote -v
```

Output:

```
origin  https://github.com/yourusername/vscode.git (fetch)
origin  https://github.com/yourusername/vscode.git (push)
```

Step 3: Add Upstream Remote

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

# Verify remotes
git remote -v
```

Output:

```
origin    https://github.com/yourusername/vscode.git (fetch)
origin    https://github.com/yourusername/vscode.git (push)
upstream  https://github.com/microsoft/vscode.git (fetch)
upstream  https://github.com/microsoft/vscode.git (push)
```

Step 4: Configure Upstream Push

```
# Prevent accidental pushes to upstream
git remote set-url --push upstream DISABLE

# Verify configuration
git remote -v
```

Output:

```
origin    https://github.com/yourusername/vscode.git (fetch)
origin    https://github.com/yourusername/vscode.git (push)
upstream  https://github.com/microsoft/vscode.git (fetch)
upstream  DISABLE (push)
```

Keeping Your Fork in Sync

Syncing with Upstream

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

# Switch to main branch
```

```
git checkout main

# Merge upstream changes
git merge upstream/main

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

Alternative: Rebase Instead of Merge

```
# Fetch upstream changes
git fetch upstream

# Rebase your main branch
git checkout main
git rebase upstream/main

# Force push (safe since it's your fork)
git push --force-with-lease origin main
```

Automated Sync Script

Create a script to automate syncing:

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

echo "Syncing fork with upstream..."

# Fetch all remotes
git fetch --all

# Store current branch
current_branch=$(git branch --show-current)

# Switch to main and sync
git checkout main
git merge upstream/main
git push origin main

# Switch back to original branch
git checkout "$current_branch"

echo "Fork synced successfully!"
```

```
# Make script executable
chmod +x sync-fork.sh
```

```
# Run sync
./sync-fork.sh
```

Contributing to Open Source

Example: Contributing to a Documentation Project

Let's simulate contributing to an open source documentation project:

```
# Fork and clone a documentation repository
git clone https://github.com/yourusername/awesome-docs.git
cd awesome-docs
git remote add upstream https://github.com/original/awesome-docs.git
git remote set-url --push upstream DISABLE

# Sync with latest changes
git fetch upstream
git checkout main
git merge upstream/main
git push origin main

# Create feature branch for your contribution
git checkout -b docs/improve-git-section

# Make improvements to documentation
cat >> git-guide.md << EOF

## Advanced Git Tips

### Using Git Aliases for Productivity

Git aliases can significantly speed up your workflow:

\`\`\`bash
# Set up useful aliases
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual '!gitk'
\`\`\`

### Interactive Staging

Use interactive staging for precise commits:

\`\`\`bash
# Stage changes interactively
```

```
git add -i

# Stage patches interactively
git add -p
\`\`\`

### Finding Bugs with Git Bisect

Git bisect helps find the commit that introduced a bug:

\`\`\`bash
# Start bisect session
git bisect start
git bisect bad          # Current commit is bad
git bisect good v1.0    # v1.0 was good

# Git will checkout commits for testing
# Mark each commit as good or bad
git bisect good   # if current commit is good
git bisect bad    # if current commit is bad

# Git will find the problematic commit
git bisect reset  # End bisect session
\`\`\`
EOF

# Commit your contribution
git add git-guide.md
git commit -m "docs: Add advanced Git tips section

- Add section on Git aliases for productivity
- Document interactive staging techniques
- Explain git bisect for bug hunting
- Include practical examples and commands

Fixes #123"

# Push to your fork
git push -u origin docs/improve-git-section
```

Creating the Pull Request

```
# Use GitHub CLI to create PR
gh pr create \
  --title "docs: Add advanced Git tips section" \
  --body "## Description
Adds comprehensive section on advanced Git techniques including aliases,
interactive staging, and bisect.

## Changes
- ✅ Added Git aliases section with productivity tips
```

```
- ✅ Documented interactive staging workflow
- ✅ Explained git bisect for debugging
- ✅ Included practical examples

## Testing
- [x] Verified all commands work correctly
- [x] Checked markdown formatting
- [x] Tested examples on different Git versions

Fixes #123" \
  --reviewer maintainer1,maintainer2 \
  --label documentation,enhancement
```

Managing Multiple Forks

Working with Multiple Remotes

```
# Add multiple upstream sources
git remote add upstream-original https://github.com/original/repo.git
git remote add upstream-fork https://github.com/anotherfork/repo.git

# Fetch from specific remote
git fetch upstream-original
git fetch upstream-fork

# Compare different upstreams
git log --oneline upstream-original/main..upstream-fork/main
```

Tracking Different Branches

```
# Track different branches from different remotes
git checkout -b feature-a upstream-original/feature-a
git checkout -b feature-b upstream-fork/feature-b

# Push to your fork
git push origin feature-a
git push origin feature-b
```

Advanced Fork Workflows

Triangular Workflow

Common in large open source projects:

```
Upstream (read-only)
    ↑ fetch
```

```
Local Repository
    ↓ push
Your Fork
    ↓ pull request
Upstream
```

```
# Setup triangular workflow
git clone https://github.com/yourusername/project.git
cd project
git remote add upstream https://github.com/original/project.git
git remote set-url --push upstream DISABLE

# Configure push default
git config remote.pushdefault origin
git config push.default current

# Workflow
git fetch upstream              # Get latest changes
git checkout -b feature/new     # Create feature branch
# ... make changes ...
git push                        # Pushes to origin (your fork)
# Create PR from your fork to upstream
```

Maintaining a Personal Fork

```
# Keep personal customizations in separate branch
git checkout -b personal/customizations

# Make your personal changes
echo "# My personal notes" >> PERSONAL.md
git add PERSONAL.md
git commit -m "Add personal customizations"

# Regularly rebase on upstream
git fetch upstream
git rebase upstream/main

# Keep personal branch updated
git push --force-with-lease origin personal/customizations
```

GitHub CLI for Forks

Fork Management

```
# Fork repository
gh repo fork original/repository
```

```
# Fork and clone in one command
gh repo fork original/repository --clone

# List your forks
gh repo list --fork

# Sync fork with upstream
gh repo sync yourusername/repository

# Create PR from fork
gh pr create --repo original/repository
```

Repository Information

```
# View repository info
gh repo view original/repository

# Check if repository is a fork
gh repo view --json isFork

# View parent repository
gh repo view --json parent
```

Best Practices for Forks

1. Keep Fork Updated

```
# Create alias for syncing
git config --global alias.sync '!git fetch upstream && git checkout main && git
merge upstream/main && git push origin main'

# Use the alias
git sync
```

2. Use Feature Branches

```
# Always create feature branches from updated main
git checkout main
git pull upstream main
git checkout -b feature/my-contribution

# Never work directly on main branch
```

3. Clean Commit History

```
# Before creating PR, clean up commits
git rebase -i upstream/main

# Squash related commits
# Fix commit messages
# Remove debug commits
```

4. Test Your Changes

```
# Test locally before pushing
npm test                    # Run test suite
npm run build              # Check build
npm run lint               # Check code style

# Test with upstream changes
git fetch upstream
git rebase upstream/main
npm test                   # Test after rebase
```

Common Fork Scenarios

Scenario 1: Contributing a Bug Fix

```
# Sync with upstream
git fetch upstream
git checkout main
git merge upstream/main
git push origin main

# Create bug fix branch
git checkout -b fix/login-validation-bug

# Fix the bug
sed -i 's/email.length > 0/email.includes("@")/' login.js

# Test the fix
npm test

# Commit with descriptive message
git add login.js
git commit -m "fix: Improve email validation in login form

Replace simple length check with proper email format validation
to prevent invalid email addresses from being accepted.

Fixes #456"
```

```
# Push and create PR
git push -u origin fix/login-validation-bug
gh pr create --title "fix: Improve email validation in login form" --body "Fixes
#456"
```

Scenario 2: Adding a New Feature

```
# Create feature branch
git checkout -b feature/dark-mode-toggle

# Implement feature
cat > dark-mode.js << EOF
class DarkModeToggle {
  constructor() {
    this.isDark = localStorage.getItem('darkMode') === 'true';
    this.init();
  }

  init() {
    this.createToggle();
    this.applyTheme();
  }

  createToggle() {
    const toggle = document.createElement('button');
    toggle.textContent = '🌙';
    toggle.addEventListener('click', () => this.toggle());
    document.body.appendChild(toggle);
  }

  toggle() {
    this.isDark = !this.isDark;
    localStorage.setItem('darkMode', this.isDark);
    this.applyTheme();
  }

  applyTheme() {
    document.body.classList.toggle('dark-mode', this.isDark);
  }
}

new DarkModeToggle();
EOF

# Add CSS
cat >> styles.css << EOF

/* Dark mode styles */
.dark-mode {
  background-color: #1a1a1a;
  color: #ffffff;
```

```
}

.dark-mode button {
  background-color: #333;
  color: #fff;
  border: 1px solid #555;
}
EOF

# Commit feature
git add .
git commit -m "feat: Add dark mode toggle functionality

- Implement DarkModeToggle class with localStorage persistence
- Add toggle button with moon emoji
- Include CSS styles for dark theme
- Automatically apply saved theme preference on load

Closes #789"

# Push and create PR
git push -u origin feature/dark-mode-toggle
gh pr create --title "feat: Add dark mode toggle" --body "Implements #789"
```

Scenario 3: Updating Documentation

```
# Create documentation branch
git checkout -b docs/api-examples

# Update documentation
cat >> API.md << EOF

## Usage Examples

### Basic Authentication

\`\`\`javascript
const api = new APIClient({
  baseURL: 'https://api.example.com',
  apiKey: 'your-api-key'
});

// Login user
const user = await api.auth.login({
  email: 'user@example.com',
  password: 'password123'
});
\`\`\`

### Error Handling
```

```
\`\`\`javascript
try {
  const data = await api.users.get(userId);
  console.log(data);
} catch (error) {
  if (error.status === 404) {
    console.log('User not found');
  } else {
    console.error('API Error:', error.message);
  }
}
\`\`\`
EOF

# Commit documentation
git add API.md
git commit -m "docs: Add API usage examples

- Add authentication example with error handling
- Include common use cases and patterns
- Improve developer onboarding experience"

# Push and create PR
git push -u origin docs/api-examples
gh pr create --title "docs: Add comprehensive API examples"
```

Troubleshooting Forks

Fork is Behind Upstream

```
# Check how far behind
git fetch upstream
git log --oneline main..upstream/main

# Sync fork
git checkout main
git merge upstream/main
git push origin main
```

Merge Conflicts During Sync

```
# Resolve conflicts during merge
git fetch upstream
git merge upstream/main
# Resolve conflicts in files
git add .
git commit -m "Merge upstream changes"
git push origin main
```

Accidentally Pushed to Upstream

```
# If you have push access and made a mistake
git push upstream :branch-name  # Delete branch

# If you don't have push access, contact maintainers
```

Lost Fork Sync

```
# Reset fork to match upstream exactly
git fetch upstream
git checkout main
git reset --hard upstream/main
git push --force-with-lease origin main
```

Quick Reference

```
# Fork setup
gh repo fork original/repo --clone    # Fork and clone
git remote add upstream <url>         # Add upstream
git remote set-url --push upstream DISABLE  # Prevent upstream push

# Sync workflow
git fetch upstream                    # Get upstream changes
git checkout main                     # Switch to main
git merge upstream/main               # Merge changes
git push origin main                  # Update fork

# Contribution workflow
git checkout -b feature/branch        # Create feature branch
# ... make changes ...
git push -u origin feature/branch     # Push to fork
gh pr create                          # Create pull request

# Maintenance
gh repo sync yourusername/repo        # Sync fork via GitHub CLI
git branch -d feature/merged-branch   # Clean up merged branches
```

Previous: Pull Requests and Code Review
Next: Advanced Git Rebase

Advanced Git Rebase

Git rebase is a powerful tool for rewriting commit history, cleaning up branches, and maintaining a linear project timeline. This chapter covers advanced rebase techniques including interactive rebase, squashing, and complex scenarios.

Understanding Rebase vs Merge

Visual Comparison

Before Integration:

```
main:     A---B---C
               \
feature:        D---E---F
```

After Merge:

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

After Rebase:

```
main:     A---B---C---D'---E'---F'
feature:                  (moved)
```

Basic Rebase Review

Simple Rebase

```
# Setup example repository
mkdir rebase-demo
cd rebase-demo
git init

# Create initial commits
echo "Initial content" > file.txt
git add file.txt
git commit -m "Initial commit"

echo "Main branch change" >> file.txt
git add file.txt
git commit -m "Update on main branch"

# Create feature branch
git checkout -b feature/new-feature
```

```
echo "Feature content" >> feature.txt
git add feature.txt
git commit -m "Add feature file"

echo "More feature work" >> feature.txt
git add feature.txt
git commit -m "Enhance feature"

# Meanwhile, main branch gets more commits
git checkout main
echo "Another main change" >> file.txt
git add file.txt
git commit -m "Another update on main"

# View current state
git log --graph --oneline --all
```

Output:

```
* f1e2d3c (HEAD -> main) Another update on main
| * a4b5c6d (feature/new-feature) Enhance feature
| * e7f8g9h Add feature file
|/
* b2c3d4e Update on main branch
* a1b2c3d Initial commit
```

```
# Rebase feature branch onto main
git checkout feature/new-feature
git rebase main

# View result
git log --graph --oneline --all
```

Output:

```
* h9i0j1k (HEAD -> feature/new-feature) Enhance feature
* g8h9i0j Add feature file
* f1e2d3c (main) Another update on main
* b2c3d4e Update on main branch
* a1b2c3d Initial commit
```

Interactive Rebase

Interactive rebase allows you to modify commits during the rebase process.

Starting Interactive Rebase

```
# Interactive rebase for last 3 commits
git rebase -i HEAD~3

# Interactive rebase from specific commit
git rebase -i a1b2c3d

# Interactive rebase onto another branch
git rebase -i main
```

Interactive Rebase Commands

When you start interactive rebase, Git opens an editor with:

```
pick a1b2c3d Add feature file
pick b2c3d4e Enhance feature
pick c3d4e5f Fix typo in feature

# Rebase f1e2d3c..c3d4e5f onto f1e2d3c (3 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
```

Practical Interactive Rebase Examples

Example 1: Squashing Related Commits

Let's create a realistic scenario with multiple small commits:

```
# Create feature branch with multiple small commits
git checkout -b feature/user-profile

# First commit: Add basic structure
cat > profile.html << EOF
<!DOCTYPE html>
<html>
<head>
    <title>User Profile</title>
</head>
```

```
<body>
    <div id="profile">
        <h1>User Profile</h1>
    </div>
</body>
</html>
EOF

git add profile.html
git commit -m "Add basic profile HTML structure"

# Second commit: Add CSS
cat > profile.css << EOF
#profile {
    max-width: 600px;
    margin: 0 auto;
    padding: 20px;
}

#profile h1 {
    color: #333;
    text-align: center;
}
EOF

git add profile.css
git commit -m "Add basic profile styling"

# Third commit: Fix CSS typo
sed -i 's/text-align: center/text-align: center;/' profile.css
git add profile.css
git commit -m "Fix missing semicolon in CSS"

# Fourth commit: Add JavaScript
cat > profile.js << EOF
class UserProfile {
    constructor(userId) {
        this.userId = userId;
        this.loadProfile();
    }

    async loadProfile() {
        try {
            const response = await fetch(`/api/users/${this.userId}`);
            const user = await response.json();
            this.displayProfile(user);
        } catch (error) {
            console.error('Failed to load profile:', error);
        }
    }

    displayProfile(user) {
        const profileDiv = document.getElementById('profile');
        profileDiv.innerHTML = `
```

```
            <h1>${user.name}</h1>
            <p>Email: ${user.email}</p>
            <p>Joined: ${user.joinDate}</p>
        `;
    }
}
EOF

git add profile.js
git commit -m "Add JavaScript for profile functionality"

# Fifth commit: Fix JavaScript bug
sed -i 's/user.joinDate/new Date(user.joinDate).toLocaleDateString()/' profile.js
git add profile.js
git commit -m "Fix date formatting in profile display"

# Sixth commit: Add error handling
sed -i 's/console.error/this.showError/' profile.js
cat >> profile.js << EOF

    showError(message) {
        const profileDiv = document.getElementById('profile');
        profileDiv.innerHTML = `<p class="error">Error: ${message}</p>`;
    }
EOF

git add profile.js
git commit -m "Improve error handling in profile loader"

# View commit history
git log --oneline
```

Output:

```
g7h8i9j (HEAD -> feature/user-profile) Improve error handling in profile loader
f6g7h8i Fix date formatting in profile display
e5f6g7h Add JavaScript for profile functionality
d4e5f6g Fix missing semicolon in CSS
c3d4e5f Add basic profile styling
b2c3d4e Add basic profile HTML structure
a1b2c3d (main) Initial commit
```

Squashing the Commits

```
# Start interactive rebase for last 6 commits
git rebase -i HEAD~6
```

Edit the rebase file:

```
pick b2c3d4e Add basic profile HTML structure
squash c3d4e5f Add basic profile styling
fixup d4e5f6g Fix missing semicolon in CSS
pick e5f6g7h Add JavaScript for profile functionality
fixup f6g7h8i Fix date formatting in profile display
squash g7h8i9j Improve error handling in profile loader

# This will result in 2 commits:
# 1. HTML structure + CSS styling (squashed)
# 2. JavaScript functionality + error handling (squashed)
```

After saving, Git will prompt for commit messages:

First squash (HTML + CSS):

```
Add user profile HTML structure and styling

- Create basic profile HTML layout
- Add responsive CSS styling
- Fix CSS syntax issues
```

Second squash (JavaScript):

```
Add user profile JavaScript functionality

- Implement UserProfile class with API integration
- Add proper date formatting
- Improve error handling and user feedback
```

Result:

```
git log --oneline
```

Output:

```
h8i9j0k (HEAD -> feature/user-profile) Add user profile JavaScript functionality
g7h8i9j Add user profile HTML structure and styling
a1b2c3d (main) Initial commit
```

Advanced Rebase Techniques

1. Reword Commit Messages

```
# Fix commit messages during rebase
git rebase -i HEAD~3
```

Change `pick` to `reword` for commits with poor messages:

```
reword a1b2c3d fix stuff
pick b2c3d4e Add proper error handling
reword c3d4e5f update
```

Git will prompt to edit each marked commit message.

2. Edit Commits

```
# Stop at specific commit to make changes
git rebase -i HEAD~3
```

Change `pick` to `edit`:

```
pick a1b2c3d Add feature
edit b2c3d4e Add tests
pick c3d4e5f Update documentation
```

When rebase stops:

```
# Make additional changes
echo "Additional test case" >> test.js
git add test.js

# Amend the commit
git commit --amend --no-edit

# Continue rebase
git rebase --continue
```

3. Split Commits

```
# Split a large commit into smaller ones
git rebase -i HEAD~2
```

Mark commit for editing:

```
pick a1b2c3d Good commit
edit b2c3d4e Large commit with multiple changes
```

When rebase stops:

```
# Reset the commit but keep changes
git reset HEAD~1

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

git add file2.css
git commit -m "Update login form styling"

git add file3.html
git commit -m "Add password reset form"

# Continue rebase
git rebase --continue
```

4. Reorder Commits

```
# Reorder commits during interactive rebase
git rebase -i HEAD~4
```

Reorder lines in the editor:

```
# Original order:
pick a1b2c3d Add feature A
pick b2c3d4e Add feature B
pick c3d4e5f Fix feature A
pick d4e5f6g Add feature C

# Reordered:
pick a1b2c3d Add feature A
pick c3d4e5f Fix feature A
pick b2c3d4e Add feature B
pick d4e5f6g Add feature C
```

5. Drop Commits

```
# Remove commits entirely
git rebase -i HEAD~4
```

Change `pick` to `drop` or delete the line:

```
pick a1b2c3d Add feature
drop b2c3d4e Add debug code
pick c3d4e5f Add tests
drop d4e5f6g Temporary fix
```

Rebase onto Different Branch

Moving Branch Base

```
# Move feature branch from old-main to new-main
git rebase --onto new-main old-main feature-branch
```

Visual representation:

```
Before:
old-main: A---B---C
new-main: A---D---E
feature:      F---G (based on old-main)

After:
old-main: A---B---C
new-main: A---D---E
feature:          F'---G' (now based on new-main)
```

Practical Example

```
# Setup scenario
git checkout -b old-approach
echo "Old implementation" > old-feature.js
git add old-feature.js
git commit -m "Implement old approach"

# Create feature based on old approach
git checkout -b feature/enhancement
echo "Enhancement to old approach" >> old-feature.js
git add old-feature.js
git commit -m "Enhance old implementation"

# Meanwhile, new approach is developed
```

```
git checkout main
git checkout -b new-approach
echo "New implementation" > new-feature.js
git add new-feature.js
git commit -m "Implement new approach"

# Move enhancement to new approach
git checkout feature/enhancement
git rebase --onto new-approach old-approach

# Now feature/enhancement is based on new-approach
```

Handling Rebase Conflicts

Conflict Resolution During Rebase

```
# Create conflict scenario
git checkout main
echo "Main branch content" > shared-file.txt
git add shared-file.txt
git commit -m "Add shared file on main"

git checkout -b feature/modify-shared
echo "Feature branch content" > shared-file.txt
git add shared-file.txt
git commit -m "Modify shared file in feature"

# Update main with conflicting change
git checkout main
echo "Updated main content" > shared-file.txt
git add shared-file.txt
git commit -m "Update shared file on main"

# Rebase feature branch (will conflict)
git checkout feature/modify-shared
git rebase main
```

Output:

```
Auto-merging shared-file.txt
CONFLICT (content): Merge conflict in shared-file.txt
error: could not apply a1b2c3d... Modify shared file in feature
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
```

Resolving Conflicts

```
# View conflicted file
cat shared-file.txt
```

Output:

```
<<<<<<< HEAD
Updated main content
=======
Feature branch content
>>>>>>> a1b2c3d (Modify shared file in feature)
```

```
# Resolve conflict
echo "Merged: Updated main content with feature additions" > shared-file.txt

# Mark as resolved
git add shared-file.txt

# Continue rebase
git rebase --continue
```

Rebase Conflict Commands

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

# Skip current commit
git rebase --skip

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

# Edit commit message during conflict resolution
git commit --amend
```

Autosquash Workflow

Using --fixup and --squash

```
# Create initial commit
echo "function calculate() { return 1 + 1; }" > math.js
git add math.js
git commit -m "Add calculate function"
```

```
# Later, find a bug and create fixup commit
echo "function calculate() { return 2 + 2; }" > math.js
git add math.js
git commit --fixup HEAD

# Add more functionality
echo "function multiply(a, b) { return a * b; }" >> math.js
git add math.js
git commit -m "Add multiply function"

# Another fixup for the original commit
echo "function calculate() { return 2 + 2; } // Fixed calculation" > math.js
echo "function multiply(a, b) { return a * b; }" >> math.js
git add math.js
git commit --fixup HEAD~2

# View commits
git log --oneline
```

Output:

```
f6g7h8i fixup! Add calculate function
e5f6g7h Add multiply function
d4e5f6g fixup! Add calculate function
c3d4e5f Add calculate function
```

Autosquash Rebase

```
# Automatically arrange and squash fixup commits
git rebase -i --autosquash HEAD~4
```

Git automatically arranges commits:

```
pick c3d4e5f Add calculate function
fixup d4e5f6g fixup! Add calculate function
fixup f6g7h8i fixup! Add calculate function
pick e5f6g7h Add multiply function
```

Squash Commits

```
# Create squash commit (keeps commit message)
echo "Additional math utilities" >> math.js
git add math.js
git commit --squash HEAD~1  # Squash into "Add multiply function"
```

```
# Autosquash will combine messages
git rebase -i --autosquash HEAD~3
```

Advanced Rebase Scenarios

1. Cherry-pick During Rebase

```
# Use exec command to cherry-pick during rebase
git rebase -i HEAD~3
```

Add exec commands:

```
pick a1b2c3d First commit
exec git cherry-pick other-branch~2
pick b2c3d4e Second commit
exec git cherry-pick other-branch~1
pick c3d4e5f Third commit
```

2. Break and Continue

```
# Stop rebase at specific point
git rebase -i HEAD~5
```

Add break command:

```
pick a1b2c3d First commit
pick b2c3d4e Second commit
break
pick c3d4e5f Third commit
pick d4e5f6g Fourth commit
pick e5f6g7h Fifth commit
```

When rebase stops:

```
# Do additional work
echo "Additional changes" >> file.txt
git add file.txt
git commit -m "Additional work during rebase"

# Continue rebase
git rebase --continue
```

3. Label and Reset

```
# Use labels for complex rebases
git rebase -i HEAD~6
```

Use label and reset:

```
label start
pick a1b2c3d First commit
pick b2c3d4e Second commit
label middle
pick c3d4e5f Third commit
reset middle
pick d4e5f6g Fourth commit
pick e5f6g7h Fifth commit
```

Rebase Best Practices

1. Never Rebase Public Branches

```
# DON'T do this if others are using the branch
git checkout main
git rebase feature/branch  # DANGEROUS!

# DO this instead
git checkout feature/branch
git rebase main            # Safe - private branch
```

2. Use --force-with-lease

```
# Safer than --force
git push --force-with-lease origin feature/branch

# This fails if someone else pushed to the branch
```

3. Backup Before Complex Rebases

```
# Create backup branch
git branch backup/feature-branch

# Do complex rebase
git rebase -i HEAD~10
```

```
# If something goes wrong
git reset --hard backup/feature-branch
```

4. Test After Rebase

```
# Always test after rebase
npm test
npm run build
npm run lint

# Check that functionality still works
```

Troubleshooting Rebase

Undo Rebase

```
# Find original commit in reflog
git reflog

# Reset to before rebase
git reset --hard HEAD@{5}
```

Recover Lost Commits

```
# Find lost commits
git fsck --lost-found

# Or check reflog
git reflog --all

# Recover specific commit
git cherry-pick lost-commit-hash
```

Fix Broken Rebase

```
# If rebase gets stuck
git rebase --abort

# Start over with different strategy
git rebase -X theirs main  # Prefer their changes
git rebase -X ours main    # Prefer our changes
```

Quick Reference

```
# Basic rebase
git rebase main                    # Rebase current branch onto main
git rebase --onto new old branch   # Move branch from old to new base

# Interactive rebase
git rebase -i HEAD~3               # Interactive rebase last 3 commits
git rebase -i --autosquash HEAD~5  # Auto-arrange fixup/squash commits

# Conflict resolution
git rebase --continue              # Continue after resolving conflicts
git rebase --skip                  # Skip current commit
git rebase --abort                 # Abort rebase

# Fixup workflow
git commit --fixup HEAD~2          # Create fixup commit
git commit --squash HEAD~1         # Create squash commit
git rebase -i --autosquash HEAD~5  # Apply fixups automatically

# Safety
git push --force-with-lease        # Safer force push
git reflog                         # Find lost commits
git reset --hard HEAD@{n}         # Undo rebase
```

Previous: Forking and Upstream
Next: Git Cherry-pick

Git Cherry-pick

Git cherry-pick allows you to apply specific commits from one branch to another without merging the entire branch. This is useful for selectively applying bug fixes, features, or patches across different branches.

Understanding Cherry-pick

What is Cherry-pick?

Cherry-pick creates a new commit with the same changes as an existing commit, but with a different parent commit and commit hash.

Visual representation:

```
Before cherry-pick:
main:     A---B---C
feature:      D---E---F

After cherry-picking commit E to main:
main:     A---B---C---E'
feature:      D---E---F
```

Note: `E'` is a new commit with the same changes as `E` but different hash.

Basic Cherry-pick Operations

Simple Cherry-pick

```
# Setup example repository
mkdir cherry-pick-demo
cd cherry-pick-demo
git init

# Create initial commits
echo "Initial content" > main.txt
git add main.txt
git commit -m "Initial commit"

echo "Main branch update" >> main.txt
git add main.txt
git commit -m "Update main branch"

# Create feature branch
git checkout -b feature/new-feature
echo "Feature content" > feature.txt
git add feature.txt
git commit -m "Add feature file"

echo "Bug fix in feature" >> feature.txt
git add feature.txt
git commit -m "Fix bug in feature"

echo "More feature work" >> feature.txt
git add feature.txt
git commit -m "Enhance feature"

# View commit history
git log --oneline --graph --all
```

Output:

```
* e5f6g7h (HEAD -> feature/new-feature) Enhance feature
* d4e5f6g Fix bug in feature
* c3d4e5f Add feature file
| * b2c3d4e (main) Update main branch
|/
* a1b2c3d Initial commit
```

Cherry-pick Specific Commit

```
# Switch to main branch
git checkout main

# Cherry-pick the bug fix commit
git cherry-pick d4e5f6g

# View result
git log --oneline --graph --all
```

Output:

```
* f6g7h8i (HEAD -> main) Fix bug in feature
* b2c3d4e Update main branch
| * e5f6g7h (feature/new-feature) Enhance feature
| * d4e5f6g Fix bug in feature
| * c3d4e5f Add feature file
|/
* a1b2c3d Initial commit
```

Practical Cherry-pick Use Cases

Use Case 1: Hotfix from Development Branch

```
# Setup scenario: critical bug found in production
git checkout -b release/v1.0
echo "Version 1.0 release" > version.txt
git add version.txt
git commit -m "Release v1.0"

# Meanwhile, development continues
git checkout -b develop
echo "New feature A" > feature-a.txt
git add feature-a.txt
git commit -m "Add feature A"

echo "New feature B" > feature-b.txt
git add feature-b.txt
git commit -m "Add feature B"

# Critical bug fix in development
echo "Fixed critical security issue" > security-fix.txt
git add security-fix.txt
git commit -m "SECURITY: Fix authentication bypass vulnerability"

echo "More development" > feature-c.txt
git add feature-c.txt
git commit -m "Add feature C"
```

```
# View current state
git log --oneline --graph --all
```

Output:

```
* j1k2l3m (HEAD -> develop) Add feature C
* i0j1k2l SECURITY: Fix authentication bypass vulnerability
* h9i0j1k Add feature B
* g8h9i0j Add feature A
| * f7g8h9i (release/v1.0) Release v1.0
|/
* a1b2c3d (main) Initial commit
```

```
# Cherry-pick security fix to release branch
git checkout release/v1.0
git cherry-pick i0j1k2l

# Create hotfix release
git tag v1.0.1
git log --oneline
```

Output:

```
k2l3m4n (HEAD -> release/v1.0, tag: v1.0.1) SECURITY: Fix authentication bypass
vulnerability
f7g8h9i Release v1.0
a1b2c3d Initial commit
```

Use Case 2: Selective Feature Backporting

```
# Setup multiple release branches
git checkout main
git checkout -b release/v2.0
echo "Version 2.0 features" > v2-features.txt
git add v2-features.txt
git commit -m "Add v2.0 features"

git checkout -b release/v3.0
echo "Version 3.0 features" > v3-features.txt
git add v3-features.txt
git commit -m "Add v3.0 features"

# Add backward-compatible improvement
echo "Performance optimization" > performance.txt
git add performance.txt
```

```
git commit -m "PERF: Optimize database queries (backward compatible)"

# This improvement should be in v2.0 as well
git checkout release/v2.0
git cherry-pick HEAD~0  # Cherry-pick from v3.0

# Verify both branches have the optimization
git log --oneline
git checkout release/v3.0
git log --oneline
```

Use Case 3: Documentation Updates

```
# Setup documentation scenario
git checkout main
git checkout -b docs/api-updates
echo "# API Documentation\n\n## Authentication\nUpdated auth flow" > api-docs.md
git add api-docs.md
git commit -m "Update API authentication documentation"

echo "\n## Rate Limiting\nNew rate limiting info" >> api-docs.md
git add api-docs.md
git commit -m "Add rate limiting documentation"

echo "\n## Error Codes\nComplete error code reference" >> api-docs.md
git add api-docs.md
git commit -m "Add comprehensive error code documentation"

# Cherry-pick only the authentication update to main
git checkout main
git cherry-pick docs/api-updates~2  # Pick the first commit

# Later, cherry-pick error codes documentation
git cherry-pick docs/api-updates    # Pick the latest commit

# View result
git log --oneline
```

Advanced Cherry-pick Techniques

Cherry-pick Multiple Commits

```
# Cherry-pick a range of commits
git cherry-pick commit1..commit3

# Cherry-pick multiple specific commits
git cherry-pick commit1 commit2 commit4
```

```
# Cherry-pick with range (exclusive start)
git cherry-pick commit1^..commit3
```

Practical Example: Multiple Commits

```
# Setup scenario with multiple related commits
git checkout -b feature/user-management

# Commit 1: Add user model
cat > user.js << EOF
class User {
    constructor(name, email) {
        this.name = name;
        this.email = email;
        this.createdAt = new Date();
    }

    validate() {
        return this.name && this.email;
    }
}

module.exports = User;
EOF

git add user.js
git commit -m "Add User model class"

# Commit 2: Add user validation
cat >> user.js << EOF

// Additional validation methods
User.prototype.validateEmail = function() {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(this.email);
};
EOF

git add user.js
git commit -m "Add email validation to User model"

# Commit 3: Add user repository
cat > user-repository.js << EOF
const User = require('./user');

class UserRepository {
    constructor() {
        this.users = [];
    }

    create(userData) {
```

```
        const user = new User(userData.name, userData.email);
        if (user.validate() && user.validateEmail()) {
            this.users.push(user);
            return user;
        }
        throw new Error('Invalid user data');
    }

    findByEmail(email) {
        return this.users.find(user => user.email === email);
    }
}

module.exports = UserRepository;
EOF

git add user-repository.js
git commit -m "Add UserRepository for user management"

# Commit 4: Add tests
cat > user.test.js << EOF
const User = require('./user');
const UserRepository = require('./user-repository');

// Test User model
const user = new User('John Doe', 'john@example.com');
console.assert(user.validate(), 'User should be valid');
console.assert(user.validateEmail(), 'Email should be valid');

// Test UserRepository
const repo = new UserRepository();
const createdUser = repo.create({ name: 'Jane Doe', email: 'jane@example.com' });
console.assert(createdUser.name === 'Jane Doe', 'User should be created');

console.log('All tests passed!');
EOF

git add user.test.js
git commit -m "Add tests for User model and repository"

# View commits
git log --oneline
```

Output:

```
p4q5r6s (HEAD -> feature/user-management) Add tests for User model and repository
o3p4q5r Add UserRepository for user management
n2o3p4q Add email validation to User model
m1n2o3p Add User model class
```

```
# Cherry-pick User model and validation (first two commits)
git checkout main
git cherry-pick m1n2o3p..n2o3p4q

# Later, cherry-pick the repository and tests
git cherry-pick o3p4q5r p4q5r6s

# View result
git log --oneline
```

Cherry-pick with Edit

```
# Cherry-pick and edit the commit
git cherry-pick --edit commit-hash

# Cherry-pick without committing (stage changes only)
git cherry-pick --no-commit commit-hash

# Make additional changes
echo "Additional changes" >> file.txt
git add file.txt

# Commit with custom message
git commit -m "Cherry-picked and enhanced: original feature"
```

Cherry-pick with Strategy

```
# Cherry-pick with merge strategy
git cherry-pick -X theirs commit-hash    # Prefer their changes
git cherry-pick -X ours commit-hash      # Prefer our changes
git cherry-pick -X patience commit-hash  # Use patience algorithm
```

Handling Cherry-pick Conflicts

Conflict Resolution

```
# Create conflict scenario
git checkout main
echo "Main branch content" > shared.txt
git add shared.txt
git commit -m "Add shared file on main"

git checkout -b feature/modify-shared
echo "Feature modification" > shared.txt
git add shared.txt
```

```
git commit -m "Modify shared file in feature"

# Update main with different content
git checkout main
echo "Different main content" > shared.txt
git add shared.txt
git commit -m "Update shared file differently on main"

# Try to cherry-pick (will conflict)
git cherry-pick feature/modify-shared
```

Output:

```
error: could not apply a1b2c3d... Modify shared file in feature
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

Resolving Cherry-pick Conflicts

```
# View conflict
cat shared.txt
```

Output:

```
<<<<<<< HEAD
Different main content
=======
Feature modification
>>>>>>> a1b2c3d (Modify shared file in feature)
```

```
# Resolve conflict
echo "Merged: Different main content with feature modification" > shared.txt

# Stage resolved file
git add shared.txt

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

Cherry-pick Conflict Commands

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

# Skip current commit
git cherry-pick --skip

# Abort cherry-pick
git cherry-pick --abort

# Show what's being cherry-picked
git status
```

Advanced Cherry-pick Scenarios

Cherry-pick from Remote Branch

```
# Fetch latest changes
git fetch origin

# Cherry-pick from remote branch
git cherry-pick origin/feature/branch~2

# Cherry-pick from specific remote commit
git cherry-pick 1a2b3c4d
```

Cherry-pick Merge Commits

```
# Cherry-pick a merge commit (specify parent)
git cherry-pick -m 1 merge-commit-hash  # Use first parent
git cherry-pick -m 2 merge-commit-hash  # Use second parent
```

Practical Example: Merge Commit Cherry-pick

```
# Create merge commit scenario
git checkout main
git checkout -b feature/a
echo "Feature A" > feature-a.txt
git add feature-a.txt
git commit -m "Add feature A"

git checkout main
git merge feature/a --no-ff -m "Merge feature A"

# Cherry-pick the merge commit to another branch
```

```
git checkout -b release/v1.0
git cherry-pick -m 1 HEAD  # Cherry-pick merge commit
```

Cherry-pick with Mainline

```
# When cherry-picking merge commits, specify mainline
git log --graph --oneline

# Cherry-pick merge commit with first parent as mainline
git cherry-pick -m 1 merge-commit

# Cherry-pick merge commit with second parent as mainline
git cherry-pick -m 2 merge-commit
```

Cherry-pick Workflows

Workflow 1: Release Branch Management

```
#!/bin/bash
# Script: cherry-pick-to-release.sh

# Function to cherry-pick commits to release branch
cherry_pick_to_release() {
    local release_branch=$1
    shift
    local commits=("$@")

    echo "Cherry-picking commits to $release_branch"
    git checkout "$release_branch"

    for commit in "${commits[@]}"; do
        echo "Cherry-picking $commit"
        if git cherry-pick "$commit"; then
            echo "✓ Successfully cherry-picked $commit"
        else
            echo "✗ Conflict in $commit - resolve manually"
            return 1
        fi
    done

    echo "All commits cherry-picked successfully"
}

# Usage
# cherry_pick_to_release "release/v2.1" "abc123" "def456" "ghi789"
```

Workflow 2: Hotfix Distribution

```bash
#!/bin/bash
# Script: distribute-hotfix.sh

distribute_hotfix() {
    local hotfix_commit=$1
    local branches=("release/v1.0" "release/v2.0" "release/v3.0")

    echo "Distributing hotfix $hotfix_commit to release branches"

    for branch in "${branches[@]}"; do
        echo "Applying to $branch"
        git checkout "$branch"

        if git cherry-pick "$hotfix_commit"; then
            echo "✓ Applied to $branch"
            git tag "${branch#release/}.$(date +%Y%m%d)"
        else
            echo "✗ Failed to apply to $branch"
            git cherry-pick --abort
        fi
    done
}

# Usage
# distribute_hotfix "security-fix-commit-hash"
```

Workflow 3: Feature Backporting

```bash
# Interactive script for selective backporting
backport_features() {
    local source_branch=$1
    local target_branch=$2

    echo "Available commits in $source_branch:"
    git log --oneline "$target_branch..$source_branch"

    echo "\nEnter commit hashes to backport (space-separated):"
    read -r commits

    git checkout "$target_branch"

    for commit in $commits; do
        echo "Backporting $commit"
        git cherry-pick "$commit"
    done
}
```

Cherry-pick Best Practices

1. Document Cherry-picks

```
# Include original commit info in cherry-pick message
git cherry-pick -x commit-hash

# This adds: "(cherry picked from commit abc123)"
```

2. Test After Cherry-pick

```
# Always test after cherry-picking
git cherry-pick commit-hash
npm test
npm run build

# Verify functionality works in target branch context
```

3. Use Descriptive Commit Messages

```
# When editing cherry-pick message
git cherry-pick --edit commit-hash

# Example message:
# "BACKPORT: Fix authentication bug from v3.0
#
# Original commit: abc123 from feature/auth-fix
# Cherry-picked to v2.0 for security patch
#
# (cherry picked from commit abc123)"
```

4. Avoid Cherry-picking Merge Commits

```
# Instead of cherry-picking merge commits, cherry-pick individual commits
# BAD:
git cherry-pick merge-commit

# GOOD:
git cherry-pick commit1 commit2 commit3
```

5. Keep Track of Cherry-picks

```
# Use git notes to track cherry-picks
git notes add -m "Cherry-picked to release/v2.0" commit-hash
```

```
# View notes
git log --show-notes
```

Troubleshooting Cherry-pick

Common Issues and Solutions

1. Empty Cherry-pick

```
# When cherry-pick results in no changes
git cherry-pick commit-hash
# Output: "The previous cherry-pick is now empty"

# Options:
git cherry-pick --skip          # Skip this commit
git cherry-pick --allow-empty    # Keep empty commit
git cherry-pick --abort          # Abort operation
```

2. Cherry-pick Wrong Commit

```
# Undo last cherry-pick
git reset --hard HEAD~1

# Or use reflog to find previous state
git reflog
git reset --hard HEAD@{1}
```

3. Multiple Conflicts

```
# When cherry-picking multiple commits with conflicts
git cherry-pick commit1 commit2 commit3

# Resolve each conflict individually
# After resolving first conflict:
git add .
git cherry-pick --continue

# Repeat for each conflict
```

4. Find Cherry-picked Commits

```
# Find commits that have been cherry-picked
git log --cherry-pick --left-right main...feature
```

```
# Show commits in feature not in main
git log main..feature --oneline

# Show commits that exist in both (potential cherry-picks)
git log --cherry main...feature
```

Recovery Commands

```
# Abort current cherry-pick
git cherry-pick --abort

# Reset to before cherry-pick
git reflog
git reset --hard HEAD@{n}

# Remove last cherry-picked commit
git reset --hard HEAD~1

# Undo cherry-pick but keep changes staged
git reset --soft HEAD~1
```

Quick Reference

```
# Basic cherry-pick
git cherry-pick commit-hash        # Apply single commit
git cherry-pick commit1 commit2    # Apply multiple commits
git cherry-pick commit1..commit3   # Apply range of commits

# Cherry-pick options
git cherry-pick -x commit-hash     # Add "cherry picked from" note
git cherry-pick --edit commit-hash # Edit commit message
git cherry-pick --no-commit hash   # Stage changes without committing
git cherry-pick -m 1 merge-hash    # Cherry-pick merge commit

# Conflict resolution
git cherry-pick --continue         # Continue after resolving conflicts
git cherry-pick --skip            # Skip current commit
git cherry-pick --abort            # Abort cherry-pick operation

# Advanced options
git cherry-pick -X theirs hash     # Prefer their changes in conflicts
git cherry-pick -X ours hash       # Prefer our changes in conflicts
git cherry-pick --allow-empty hash # Allow empty commits

# Finding commits
git log --cherry-pick main...feature  # Show cherry-pick candidates
git log --oneline branch1..branch2    # Commits in branch2 not in branch1
```

Previous: Advanced Git Rebase
Next: Git Reset Deep Dive

Git Reset Deep Dive

Git reset is a powerful command that allows you to undo changes by moving the HEAD pointer and optionally modifying the staging area and working directory. Understanding the different reset modes is crucial for effective Git workflow management.

Understanding Git Reset

The Three Trees of Git

Before diving into reset, it's important to understand Git's three trees:

1. **HEAD** - Points to the last commit snapshot
2. **Index (Staging Area)** - Proposed next commit snapshot
3. **Working Directory** - Sandbox where you make changes

Visual Representation

```
Working Directory  →  git add  →  Staging Area  →  git commit  →  Repository
(HEAD)
     (files)                        (index)                        (commits)
```

Reset Modes Overview

The Three Reset Modes

Mode	HEAD	Index	Working Directory
`--soft`	✓	✗	✗
`--mixed` (default)	✓	✓	✗
`--hard`	✓	✓	✓

- ✓ = Modified
- ✗ = Unchanged

Setting Up Examples

```
# Create example repository
mkdir git-reset-demo
cd git-reset-demo
git init
```

```
# Create initial commit
echo "Initial content" > file1.txt
echo "Another file" > file2.txt
git add .
git commit -m "Initial commit"

# Second commit
echo "Updated content" > file1.txt
echo "New file" > file3.txt
git add .
git commit -m "Second commit"

# Third commit
echo "Final content" > file1.txt
echo "Updated another file" > file2.txt
git add .
git commit -m "Third commit"

# Make some working directory changes
echo "Work in progress" >> file1.txt
echo "Staged changes" > file4.txt
git add file4.txt

# View current state
git status
git log --oneline
```

Output:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file4.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt

c3d4e5f (HEAD -> main) Third commit
b2c3d4e Second commit
a1b2c3d Initial commit
```

Soft Reset (--soft)

What Soft Reset Does

- Moves HEAD to specified commit
- Keeps staging area unchanged

- Keeps working directory unchanged
- Effectively "uncommits" changes but keeps them staged

Practical Example

```
# Current state before reset
git log --oneline
git status

# Soft reset to previous commit
git reset --soft HEAD~1

# Check what happened
git log --oneline
git status
```

Output after soft reset:

```
b2c3d4e (HEAD -> main) Second commit
a1b2c3d Initial commit

On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt
        modified:   file2.txt
        new file:   file4.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

Use Cases for Soft Reset

1. Amending Multiple Commits

```
# Reset to combine last 3 commits
git reset --soft HEAD~3

# All changes from those 3 commits are now staged
git status

# Create a single commit with better message
git commit -m "Implement complete user authentication system

- Add user registration
```

```
- Add login functionality
- Add password validation
- Add session management"
```

2. Splitting a Large Commit

```
# Reset the last commit but keep changes staged
git reset --soft HEAD~1

# Unstage specific files
git restore --staged file2.txt file3.txt

# Commit first part
git commit -m "Add user registration feature"

# Stage and commit second part
git add file2.txt
git commit -m "Add user validation"

# Stage and commit third part
git add file3.txt
git commit -m "Add password encryption"
```

3. Fixing Commit Messages

```
# Reset to fix multiple commit messages
git reset --soft HEAD~2

# Recommit with better messages
git commit -m "First improved commit message"

# Add remaining changes and commit
git add .
git commit -m "Second improved commit message"
```

Mixed Reset (--mixed)

What Mixed Reset Does

- Moves HEAD to specified commit
- Resets staging area to match HEAD
- Keeps working directory unchanged
- This is the **default** reset mode

Practical Example

```
# Setup: Create commits and stage some changes
echo "New feature" > feature.txt
git add feature.txt
git commit -m "Add new feature"

echo "Bug fix" > bugfix.txt
git add bugfix.txt
git commit -m "Fix critical bug"

# Make working directory and staging changes
echo "Work in progress" >> feature.txt
echo "Staged work" > staged.txt
git add staged.txt

# View current state
git status
git log --oneline

# Mixed reset (default behavior)
git reset HEAD~2

# Check what happened
git log --oneline
git status
```

Output after mixed reset:

```
a1b2c3d (HEAD -> main) Initial commit

On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        bugfix.txt
        feature.txt
        staged.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

Use Cases for Mixed Reset

1. Unstaging Files

```
# Stage multiple files
echo "Change 1" > file1.txt
echo "Change 2" > file2.txt
```

```
echo "Change 3" > file3.txt
git add .

# Unstage specific file
git reset file2.txt

# Or unstage all files
git reset

# Check status
git status
```

2. Reorganizing Commits

```
# Reset to reorganize last few commits
git reset HEAD~3

# Now all changes are in working directory
# Stage related changes together
git add user-auth.js user-model.js
git commit -m "Add user authentication system"

git add user-profile.js user-settings.js
git commit -m "Add user profile management"

git add tests/
git commit -m "Add comprehensive user tests"
```

3. Partial Commit Preparation

```
# After mixed reset, selectively stage changes
git add -p  # Interactive staging

# Or stage specific hunks
git add --patch file.txt
```

Hard Reset (--hard)

What Hard Reset Does

- Moves HEAD to specified commit
- Resets staging area to match HEAD
- Resets working directory to match HEAD
- **DESTRUCTIVE**: Loses all uncommitted changes

⚠ Warning

Hard reset is destructive and will permanently delete uncommitted changes. Always ensure you have backups or are certain you want to lose the changes.

Practical Example

```
# Setup: Create messy working state
echo "Experimental feature" > experiment.txt
echo "Debug code" >> file1.txt
echo "Temporary fix" > temp.txt
git add experiment.txt

# View current messy state
git status

# Hard reset to clean state
git reset --hard HEAD

# Check what happened - everything is clean
git status
ls  # experiment.txt and temp.txt are gone!
```

Output after hard reset:

```
HEAD is now at c3d4e5f Third commit
On branch main
nothing to commit, working tree clean
```

Use Cases for Hard Reset

1. Discarding All Local Changes

```
# When you want to start fresh from last commit
git reset --hard HEAD

# When you want to go back to specific commit
git reset --hard commit-hash

# When you want to match remote branch exactly
git fetch origin
git reset --hard origin/main
```

2. Undoing Merge Conflicts

```
# During a problematic merge
git merge feature-branch
```

```
# ... conflicts occur ...

# Abort merge and reset to clean state
git reset --hard HEAD
```

3. Emergency Cleanup

```
# When working directory is completely messed up
git reset --hard HEAD  # Back to last commit
git clean -fd          # Remove untracked files and directories
```

Advanced Reset Scenarios

Reset to Specific Commit

```
# Reset to specific commit hash
git reset --soft abc123
git reset --mixed def456
git reset --hard ghi789

# Reset to relative position
git reset --soft HEAD~3
git reset --mixed HEAD^^  # Same as HEAD~2
git reset --hard HEAD@{2}  # Using reflog
```

Reset Specific Files

```
# Reset specific file to HEAD (unstage)
git reset file.txt

# Reset specific file to specific commit
git reset commit-hash file.txt

# Reset multiple files
git reset HEAD file1.txt file2.txt
```

Practical File Reset Example

```
# Setup: Modify and stage files
echo "New content" > important.txt
echo "Other changes" > other.txt
git add .

# Reset only one file
```

```
git reset important.txt

# Check status
git status
```

Output:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   other.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   important.txt
```

Reset vs Other Commands

Reset vs Revert

```
# Reset (changes history)
git reset --hard HEAD~1  # Removes commit from history

# Revert (creates new commit)
git revert HEAD          # Creates new commit that undoes changes
```

Visual comparison:

```
Before:
A---B---C (HEAD)

After reset --hard HEAD~1:
A---B (HEAD)

After revert HEAD:
A---B---C---D (HEAD, where D undoes C)
```

Reset vs Checkout

```
# Reset (moves branch pointer)
git reset --hard commit-hash  # Moves current branch to commit

# Checkout (moves HEAD only)
```

```
git checkout commit-hash      # Detached HEAD state
git checkout branch-name      # Switch to branch
```

Reset vs Restore

```
# Reset (affects staging area)
git reset file.txt           # Unstage file

# Restore (affects working directory or staging)
git restore file.txt         # Discard working directory changes
git restore --staged file.txt # Unstage file (same as reset)
git restore --source=HEAD~1 file.txt  # Restore from specific commit
```

Dangerous Reset Scenarios

Recovering from Accidental Hard Reset

```
# Accidentally did hard reset
git reset --hard HEAD~5  # Oops! Lost 5 commits

# Use reflog to find lost commits
git reflog
```

Output:

```
abc123 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~5
def456 HEAD@{1}: commit: Important feature
ghi789 HEAD@{2}: commit: Bug fix
jkl012 HEAD@{3}: commit: Documentation update
...
```

```
# Recover by resetting to previous state
git reset --hard HEAD@{1}  # Back to "Important feature"

# Or reset to specific commit
git reset --hard def456
```

Protecting Against Accidental Reset

```
# Create backup branch before risky operations
git branch backup-$(date +%Y%m%d-%H%M%S)
```

```
# Or create tag
git tag backup-before-reset

# Then do risky reset
git reset --hard HEAD~10

# If needed, recover
git reset --hard backup-before-reset
```

Reset in Team Environments

Safe Reset Practices

```
# NEVER reset commits that have been pushed and shared
# BAD:
git reset --hard HEAD~3
git push --force  # Dangerous!

# GOOD: Use revert instead
git revert HEAD~2..HEAD
git push  # Safe
```

Reset Private Branches

```
# Safe to reset private feature branches
git checkout feature/my-work
git reset --hard HEAD~5  # OK - private branch

# When ready to share
git push --force-with-lease origin feature/my-work
```

Practical Reset Workflows

Workflow 1: Clean Commit History

```
#!/bin/bash
# Script: clean-history.sh

clean_last_commits() {
    local num_commits=$1
    local new_message="$2"

    echo "Cleaning last $num_commits commits"

    # Backup current state
    git branch "backup-$(date +%Y%m%d-%H%M%S)"
```

```
    # Soft reset to keep changes
    git reset --soft "HEAD~$num_commits"

    # Recommit with clean message
    git commit -m "$new_message"

    echo "History cleaned. Backup branch created."
}

# Usage: clean_last_commits 3 "Implement user authentication system"
```

Workflow 2: Selective File Reset

```
#!/bin/bash
# Script: selective-reset.sh

reset_files_to_commit() {
    local commit=$1
    shift
    local files=("$@")

    echo "Resetting files to $commit"

    for file in "${files[@]}"; do
        echo "Resetting $file"
        git reset "$commit" "$file"
    done

    echo "Files reset. Review changes with 'git status'"
}

# Usage: reset_files_to_commit HEAD~2 file1.txt file2.txt
```

Workflow 3: Emergency Cleanup

```
#!/bin/bash
# Script: emergency-cleanup.sh

emergency_cleanup() {
    echo "⚠  Emergency cleanup - this will lose all uncommitted changes!"
    read -p "Are you sure? (yes/no): " confirm

    if [ "$confirm" = "yes" ]; then
        # Reset to clean state
        git reset --hard HEAD

        # Remove untracked files
        git clean -fd
```

```
        # Sync with remote
        git fetch origin
        git reset --hard origin/$(git branch --show-current)

        echo "✓ Emergency cleanup complete"
    else
        echo "Cleanup cancelled"
    fi
}
```

Reset Best Practices

1. Always Backup Before Destructive Operations

```
# Create backup branch
git branch backup-before-reset

# Or create tag
git tag backup-$(date +%Y%m%d-%H%M%S)

# Then do reset
git reset --hard HEAD~5
```

2. Use Appropriate Reset Mode

```
# For uncommitting but keeping changes
git reset --soft HEAD~1

# For unstaging files
git reset file.txt

# For complete cleanup (be careful!)
git reset --hard HEAD
```

3. Understand the Impact

```
# Check what will be affected
git log --oneline HEAD~5..HEAD  # See commits that will be reset
git diff HEAD~5                 # See changes that will be lost

# Then decide on reset mode
```

4. Never Reset Shared Commits

```
# Check if commits are pushed
git log origin/main..HEAD  # Commits not yet pushed (safe to reset)
git log HEAD..origin/main  # Commits on remote (don't reset)
```

Troubleshooting Reset

Common Issues and Solutions

1. Accidentally Lost Commits

```
# Use reflog to find lost commits
git reflog --all

# Reset to previous state
git reset --hard HEAD@{n}

# Or cherry-pick specific commits
git cherry-pick lost-commit-hash
```

2. Reset Didn't Work as Expected

```
# Check current state
git status
git log --oneline

# Check reflog to see what happened
git reflog

# Reset to previous state if needed
git reset --hard HEAD@{1}
```

3. Partial Reset Issues

```
# If file reset didn't work
git status  # Check current state

# Reset file properly
git reset HEAD file.txt

# Or restore from specific commit
git restore --source=commit-hash file.txt
```

Recovery Commands

```
# Find lost commits
git reflog
git fsck --lost-found

# Recover specific commit
git reset --hard commit-hash
git cherry-pick commit-hash

# Recover from backup
git reset --hard backup-branch
git reset --hard backup-tag
```

Quick Reference

```
# Reset modes
git reset --soft HEAD~1     # Uncommit, keep staged and working changes
git reset --mixed HEAD~1    # Uncommit, unstage, keep working changes (default)
git reset --hard HEAD~1     # Uncommit, unstage, discard all changes

# File operations
git reset file.txt          # Unstage file
git reset HEAD~1 file.txt   # Reset file to previous commit
git reset --hard            # Reset everything to HEAD

# Specific targets
git reset commit-hash       # Reset to specific commit
git reset HEAD~3            # Reset 3 commits back
git reset HEAD@{2}          # Reset using reflog

# Safety
git reflog                  # View reset history
git reset --hard HEAD@{1}   # Undo last reset
git branch backup           # Create backup before reset

# Alternatives
git revert HEAD             # Undo with new commit (safe for shared history)
git restore file.txt        # Discard working directory changes
git restore --staged file   # Unstage file (alternative to reset)
```

Previous: Git Cherry-pick
Next: Git Reflog Recovery

Git Reflog Recovery

Git reflog (reference log) is a powerful recovery mechanism that tracks changes to branch tips and other references. It's your safety net for recovering "lost" commits, undoing destructive operations, and

understanding your Git history.

Understanding Git Reflog

What is Reflog?

Reflog maintains a local history of where your branch tips have been. Unlike the commit history (which can be rewritten), reflog is a local safety mechanism that records:

- Every commit
- Every branch switch
- Every reset operation
- Every rebase
- Every merge
- Every pull operation

Reflog vs Git Log

```
# Git log shows commit history
git log --oneline
# Output: Linear commit history

# Git reflog shows reference history
git reflog
# Output: Every action that moved HEAD
```

Visual comparison:

```
Git Log (commit history):
A---B---C---D (current)

Git Reflog (reference history):
HEAD@{0}: commit: D
HEAD@{1}: commit: C
HEAD@{2}: reset: moving to B
HEAD@{3}: commit: C (before reset)
HEAD@{4}: commit: B
HEAD@{5}: commit: A
```

Setting Up Examples

```
# Create example repository
mkdir reflog-demo
cd reflog-demo
git init

# Create initial commits
```

```
echo "Initial content" > file1.txt
git add file1.txt
git commit -m "Initial commit"

echo "Second version" > file1.txt
echo "New file" > file2.txt
git add .
git commit -m "Second commit"

echo "Third version" > file1.txt
git add file1.txt
git commit -m "Third commit"

echo "Fourth version" > file1.txt
git add file1.txt
git commit -m "Fourth commit"

# View initial state
git log --oneline
git reflog
```

Output:

```
# Git log
d4e5f6g (HEAD -> main) Fourth commit
c3d4e5f Third commit
b2c3d4e Second commit
a1b2c3d Initial commit

# Git reflog
d4e5f6g (HEAD -> main) HEAD@{0}: commit: Fourth commit
c3d4e5f HEAD@{1}: commit: Third commit
b2c3d4e HEAD@{2}: commit: Second commit
a1b2c3d HEAD@{3}: commit (initial): Initial commit
```

Basic Reflog Operations

Viewing Reflog

```
# View reflog for current branch
git reflog

# View reflog for specific branch
git reflog main
git reflog feature/branch

# View reflog for all references
git reflog --all
```

```
# View reflog with dates
git reflog --date=iso

# View reflog with relative dates
git reflog --date=relative
```

Reflog Output Format

```
# Standard reflog output
git reflog
```

Output explanation:

```
d4e5f6g HEAD@{0}: commit: Fourth commit
|       |        |       └─ Action description
|       |        └─ Action type
|       └─ Reflog reference (0 = most recent)
└─ Commit hash
```

Detailed Reflog Information

```
# Show reflog with full commit info
git log -g --oneline

# Show reflog with patches
git log -g -p

# Show reflog with stat
git log -g --stat
```

Recovery Scenarios

Scenario 1: Recovering from Accidental Hard Reset

```
# Create scenario: accidental hard reset
git log --oneline
# Shows: d4e5f6g, c3d4e5f, b2c3d4e, a1b2c3d

# Accidentally reset too far back
git reset --hard HEAD~3

# Check current state
git log --oneline
# Shows only: a1b2c3d Initial commit
```

```
# OH NO! Lost 3 commits!
# Use reflog to find them
git reflog
```

Output:

```
a1b2c3d (HEAD -> main) HEAD@{0}: reset: moving to HEAD~3
d4e5f6g HEAD@{1}: commit: Fourth commit
c3d4e5f HEAD@{2}: commit: Third commit
b2c3d4e HEAD@{3}: commit: Second commit
a1b2c3d HEAD@{4}: commit (initial): Initial commit
```

```
# Recover by resetting to before the accidental reset
git reset --hard HEAD@{1}

# Verify recovery
git log --oneline
# Shows: d4e5f6g, c3d4e5f, b2c3d4e, a1b2c3d (recovered!)
```

Scenario 2: Recovering Deleted Branch

```
# Create and work on feature branch
git checkout -b feature/important-work
echo "Important feature" > important.txt
git add important.txt
git commit -m "Add important feature"

echo "Critical fix" >> important.txt
git add important.txt
git commit -m "Critical fix for important feature"

# Switch back to main
git checkout main

# Accidentally delete the branch
git branch -D feature/important-work

# OH NO! Important work is gone!
# Use reflog to find the branch
git reflog --all | grep "important-work"
```

Output:

```
f7g8h9i refs/heads/feature/important-work@{0}: commit: Critical fix for important
feature
e6f7g8h refs/heads/feature/important-work@{1}: commit: Add important feature
d4e5f6g refs/heads/feature/important-work@{2}: branch: Created from HEAD
```

```
# Recover the branch
git checkout -b feature/important-work f7g8h9i

# Verify recovery
git log --oneline
# Shows the recovered commits
```

Scenario 3: Recovering from Failed Rebase

```
# Create scenario: complex rebase that went wrong
git checkout -b feature/complex
echo "Feature 1" > feature1.txt
git add feature1.txt
git commit -m "Add feature 1"

echo "Feature 2" > feature2.txt
git add feature2.txt
git commit -m "Add feature 2"

echo "Feature 3" > feature3.txt
git add feature3.txt
git commit -m "Add feature 3"

# Attempt interactive rebase that goes wrong
git rebase -i HEAD~3
# ... something goes wrong during rebase ...
# Let's say we abort it
git rebase --abort

# But what if we want to see the state before rebase?
git reflog
```

Output:

```
j3k4l5m (HEAD -> feature/complex) HEAD@{0}: rebase (abort): updating HEAD
j3k4l5m HEAD@{1}: rebase (start): checkout HEAD~3
j3k4l5m HEAD@{2}: commit: Add feature 3
i2j3k4l HEAD@{3}: commit: Add feature 2
h1i2j3k HEAD@{4}: commit: Add feature 1
```

```
# We can see exactly what happened and when
# If needed, we can reset to any previous state
git reset --hard HEAD@{2}  # Back to before rebase
```

Advanced Reflog Usage

Finding Specific Operations

```
# Find all resets
git reflog | grep reset

# Find all merges
git reflog | grep merge

# Find all rebases
git reflog | grep rebase

# Find all checkouts
git reflog | grep checkout

# Find operations from specific time
git reflog --since="2 hours ago"
git reflog --until="yesterday"
```

Reflog for Specific References

```
# Reflog for specific branch
git reflog refs/heads/main
git reflog refs/heads/feature/branch

# Reflog for remote tracking branches
git reflog refs/remotes/origin/main

# Reflog for tags (if they've been moved)
git reflog refs/tags/v1.0

# Reflog for stash
git reflog refs/stash
```

Practical Example: Complex Recovery

```
# Create complex scenario
git checkout main
git checkout -b feature/user-auth
```

```
# Multiple commits
echo "User model" > user.js
git add user.js
git commit -m "Add user model"

echo "Auth service" > auth.js
git add auth.js
git commit -m "Add authentication service"

echo "Login component" > login.js
git add login.js
git commit -m "Add login component"

# Merge to main
git checkout main
git merge feature/user-auth --no-ff

# Continue development
echo "Bug fix" >> user.js
git add user.js
git commit -m "Fix user model bug"

# Realize we need to undo the merge and fix something first
# Find the merge in reflog
git reflog | grep merge
```

Output:

```
p6q7r8s HEAD@{1}: merge feature/user-auth: Merge made by the 'recursive' strategy.
```

```
# Reset to before merge
git reset --hard HEAD@{2}  # Before the merge

# Now we can fix issues and re-merge later
```

Reflog Expiration and Cleanup

Understanding Reflog Expiration

Reflog entries are not permanent and will expire:

- **Reachable commits**: Expire after 90 days (default)
- **Unreachable commits**: Expire after 30 days (default)
- **Reflog entries**: Expire after 90 days (default)

Configuring Reflog Expiration

```
# View current reflog settings
git config --get gc.reflogExpire
git config --get gc.reflogExpireUnreachable

# Set custom expiration times
git config gc.reflogExpire "180.days"  # 6 months for reachable
git config gc.reflogExpireUnreachable "60.days"  # 2 months for unreachable

# Never expire reflog (not recommended)
git config gc.reflogExpire "never"

# Per-branch reflog settings
git config gc.refs/heads/main.reflogExpire "365.days"
```

Manual Reflog Management

```
# Clean up reflog manually
git reflog expire --expire=30.days --all

# Clean up unreachable entries
git reflog expire --expire-unreachable=7.days --all

# Force garbage collection
git gc --prune=now

# View reflog size
git count-objects -v
```

Backup Important Reflog Entries

```
# Export reflog to file
git reflog > reflog-backup-$(date +%Y%m%d).txt

# Create permanent branches for important states
git branch backup-before-major-change HEAD@{5}
git tag important-state HEAD@{10}
```

Recovery Strategies

Strategy 1: Time-based Recovery

```
# Find state from specific time
git reflog --date=iso

# Reset to specific time
```

```
git reset --hard 'HEAD@{2 hours ago}'
git reset --hard 'HEAD@{yesterday}'
git reset --hard 'HEAD@{2023-12-01 14:30:00}'
```

Strategy 2: Operation-based Recovery

```
# Find specific operation
git reflog | grep "merge feature/important"
git reflog | grep "rebase (start)"
git reflog | grep "reset: moving to"

# Reset to before that operation
git reset --hard HEAD@{n}
```

Strategy 3: Content-based Recovery

```
# Search for commits with specific content
git log -g --grep="important feature"
git log -g -S"function importantFunction"

# Show changes in reflog entries
git show HEAD@{5}
git diff HEAD@{5} HEAD@{3}
```

Reflog in Team Environments

Understanding Reflog Limitations

```
# Reflog is LOCAL only - not shared with team
# Each developer has their own reflog

# Reflog doesn't track:
# - Other people's actions
# - Actions on remote repositories
# - Actions before you cloned the repository
```

Team Recovery Strategies

```
# For team recovery, use:
# 1. Remote branches
git fetch origin
git reset --hard origin/main

# 2. Tags
```

```
git reset --hard v1.2.3

# 3. Shared backup branches
git reset --hard backup/before-major-change

# 4. Communication
# Ask team members for commit hashes
```

Practical Recovery Workflows

Workflow 1: Daily Safety Check

```
#!/bin/bash
# Script: daily-safety-check.sh

daily_safety_check() {
    echo "=== Daily Git Safety Check ==="

    # Show recent reflog entries
    echo "Recent reflog entries:"
    git reflog --date=relative -10

    # Check for any destructive operations
    echo "\nDestructive operations today:"
    git reflog --since="1 day ago" | grep -E "reset|rebase|merge"

    # Create daily backup branch
    local backup_branch="backup/daily-$(date +%Y%m%d)"
    git branch "$backup_branch" 2>/dev/null && echo "Created $backup_branch"

    # Clean old backup branches (older than 7 days)
    git for-each-ref --format='%(refname:short) %(committerdate:short)'
refs/heads/backup/ |
    while read branch date; do
        if [[ $(date -d "$date" +%s) -lt $(date -d "7 days ago" +%s) ]]; then
            git branch -D "$branch"
            echo "Deleted old backup: $branch"
        fi
    done
}
```

Workflow 2: Emergency Recovery

```
#!/bin/bash
# Script: emergency-recovery.sh

emergency_recovery() {
    echo "🚑 Emergency Recovery Mode"
```

```
    # Show recent reflog
    echo "Recent reflog entries:"
    git reflog --date=relative -20

    # Show current status
    echo "\nCurrent status:"
    git status

    # Show recent commits
    echo "\nRecent commits:"
    git log --oneline -10

    # Interactive recovery
    echo "\nSelect recovery option:"
    echo "1. Reset to previous commit"
    echo "2. Reset to specific reflog entry"
    echo "3. Create recovery branch from reflog"
    echo "4. Show detailed reflog"

    read -p "Enter option (1-4): " option

    case $option in
        1)
            git reset --hard HEAD~1
            echo "Reset to previous commit"
            ;;
        2)
            read -p "Enter reflog reference (e.g., HEAD@{5}): " ref
            git reset --hard "$ref"
            echo "Reset to $ref"
            ;;
        3)
            read -p "Enter reflog reference: " ref
            read -p "Enter recovery branch name: " branch
            git checkout -b "$branch" "$ref"
            echo "Created recovery branch: $branch"
            ;;
        4)
            git reflog --date=iso
            ;;
    esac
}
```

Workflow 3: Commit Archaeology

```
#!/bin/bash
# Script: commit-archaeology.sh

find_lost_work() {
    local search_term="$1"
```

```
    echo "🔍 Searching for lost work: $search_term"

    # Search in reflog commit messages
    echo "\nReflog entries matching '$search_term':"
    git reflog | grep -i "$search_term"

    # Search in all reachable commits
    echo "\nCommits in reflog matching '$search_term':"
    git log -g --grep="$search_term" --oneline

    # Search in commit content
    echo "\nCommits with content matching '$search_term':"
    git log -g -S"$search_term" --oneline

    # Search in unreachable commits
    echo "\nSearching unreachable commits..."
    git fsck --unreachable | grep commit | cut -d' ' -f3 |
    while read commit; do
        if git show "$commit" | grep -q "$search_term"; then
            echo "Found in unreachable commit: $commit"
            git show --oneline -s "$commit"
        fi
    done
}

# Usage: find_lost_work "important feature"
```

Reflog Best Practices

1. Regular Reflog Monitoring

```
# Check reflog regularly
git reflog --date=relative -10

# Look for unexpected operations
git reflog | grep -E "reset --hard|rebase|merge"
```

2. Create Backup Points

```
# Before risky operations
git branch backup-before-rebase
git tag checkpoint-$(date +%Y%m%d-%H%M)

# After major milestones
git tag milestone-v1.0
```

3. Understand Reflog Limitations

```
# Reflog is local only
# Reflog expires (default 90 days)
# Reflog doesn't survive repository deletion

# For permanent backup:
git bundle create backup.bundle --all
```

4. Document Recovery Procedures

```
# Create recovery documentation
cat > RECOVERY.md << EOF
# Git Recovery Procedures

## Common Recovery Commands
- View reflog: \`git reflog\`
- Reset to previous state: \`git reset --hard HEAD@{n}\`
- Recover deleted branch: \`git checkout -b branch-name commit-hash\`

## Emergency Contacts
- Team Lead: [contact info]
- Git Expert: [contact info]
EOF
```

Troubleshooting Reflog

Common Issues

1. Reflog Entry Not Found

```
# Error: "HEAD@{10} does not exist"
# Check available entries
git reflog | wc -l

# Use valid entry number
git reflog | head -5
git reset --hard HEAD@{4}
```

2. Reflog Expired

```
# If reflog entries are gone
# Check garbage collection logs
git reflog expire --dry-run --all

# Look for unreachable commits
git fsck --unreachable
```

```
# Try to recover from remote
git fetch origin
git reset --hard origin/main
```

3. Corrupted Reflog

```
# If reflog is corrupted
# Rebuild from commits
git reflog expire --expire=now --all
git reflog delete --rewrite --all

# Or remove reflog files (dangerous!)
# rm .git/logs/refs/heads/main
# git checkout main  # Recreates reflog
```

Quick Reference

```
# View reflog
git reflog                      # Current branch reflog
git reflog --all                # All references
git reflog branch-name          # Specific branch
git reflog --date=iso           # With timestamps

# Recovery operations
git reset --hard HEAD@{n}       # Reset to reflog entry
git checkout -b new-branch HEAD@{n}  # Create branch from reflog
git cherry-pick HEAD@{n}        # Cherry-pick from reflog
git show HEAD@{n}               # Show reflog entry

# Search and filter
git reflog | grep "reset"       # Find specific operations
git reflog --since="2 hours ago"  # Time-based filter
git log -g --grep="message"     # Search commit messages in reflog

# Maintenance
git reflog expire --expire=30.days --all  # Clean old entries
git gc                          # Garbage collection
git config gc.reflogExpire "180.days"     # Configure expiration

# Safety
git branch backup-$(date +%Y%m%d)  # Create backup before risky operations
git reflog > reflog-backup.txt     # Export reflog to file
```

Previous: Git Reset Deep Dive
Next: Git Bisect Bug Hunting

Git Bisect Bug Hunting

Git bisect is a powerful binary search tool that helps you find the exact commit that introduced a bug. It's particularly useful when you know a bug exists now but worked correctly in the past, and you need to identify when it was introduced.

Understanding Git Bisect

How Bisect Works

Bisect uses binary search algorithm to efficiently find the problematic commit:

1. You specify a "bad" commit (where bug exists)
2. You specify a "good" commit (where bug doesn't exist)
3. Git checks out the middle commit
4. You test and mark it as "good" or "bad"
5. Git narrows down the range and repeats
6. Process continues until the exact commit is found

Visual Representation

```
Commit history: A---B---C---D---E---F---G---H
                ↑               ↑
              good            bad

Step 1: Test E (middle)
A---B---C---D---E---F---G---H
            ↑       ↑
          good    bad

Step 2: Test F (middle of E-H)
A---B---C---D---E---F---G---H
                ↑   ↑
              good bad

Step 3: Test G (middle of F-H)
Result: G is the first bad commit!
```

Setting Up Bug Hunt Example

```
# Create example repository with a bug introduction
mkdir bisect-demo
cd bisect-demo
git init

# Create initial working code
cat > calculator.js << EOF
```

```
class Calculator {
    add(a, b) {
        return a + b;
    }

    subtract(a, b) {
        return a - b;
    }

    multiply(a, b) {
        return a * b;
    }

    divide(a, b) {
        if (b === 0) {
            throw new Error('Division by zero');
        }
        return a / b;
    }
}

module.exports = Calculator;
EOF

# Create test file
cat > test.js << EOF
const Calculator = require('./calculator');

function runTests() {
    const calc = new Calculator();

    // Test addition
    console.assert(calc.add(2, 3) === 5, 'Addition test failed');

    // Test subtraction
    console.assert(calc.subtract(5, 3) === 2, 'Subtraction test failed');

    // Test multiplication
    console.assert(calc.multiply(4, 3) === 12, 'Multiplication test failed');

    // Test division
    console.assert(calc.divide(10, 2) === 5, 'Division test failed');

    // Test division by zero
    try {
        calc.divide(10, 0);
        console.assert(false, 'Division by zero should throw error');
    } catch (e) {
        console.assert(e.message === 'Division by zero', 'Wrong error message');
    }

    console.log('All tests passed!');
}
```

```
runTests();
EOF

git add .
git commit -m "Initial calculator implementation"

# Test that it works
node test.js
```

Output:

```
All tests passed!
```

Create History with Bug Introduction

```
# Commit 2: Add percentage calculation
cat >> calculator.js << EOF

    percentage(value, percent) {
        return (value * percent) / 100;
    }
EOF

git add calculator.js
git commit -m "Add percentage calculation"

# Commit 3: Add power function
cat >> calculator.js << EOF

    power(base, exponent) {
        return Math.pow(base, exponent);
    }
EOF

git add calculator.js
git commit -m "Add power function"

# Commit 4: Add square root
cat >> calculator.js << EOF

    sqrt(value) {
        if (value < 0) {
            throw new Error('Cannot calculate square root of negative number');
        }
        return Math.sqrt(value);
    }
EOF

git add calculator.js
```

```
git commit -m "Add square root function"

# Commit 5: "Optimize" division (introduce bug)
sed -i 's/return a \/ b;/return a \/ b + 0.1;/' calculator.js
git add calculator.js
git commit -m "Optimize division calculation"

# Commit 6: Add factorial
cat >> calculator.js << EOF

    factorial(n) {
        if (n < 0) {
            throw new Error('Factorial of negative number is undefined');
        }
        if (n === 0 || n === 1) {
            return 1;
        }
        return n * this.factorial(n - 1);
    }
EOF

git add calculator.js
git commit -m "Add factorial function"

# Commit 7: Add logarithm
cat >> calculator.js << EOF

    log(value, base = Math.E) {
        if (value <= 0) {
            throw new Error('Logarithm of non-positive number is undefined');
        }
        return Math.log(value) / Math.log(base);
    }
EOF

git add calculator.js
git commit -m "Add logarithm function"

# Commit 8: Add more tests
cat >> test.js << EOF

// Test new functions
const calc2 = new Calculator();
console.assert(calc2.percentage(200, 15) === 30, 'Percentage test failed');
console.assert(calc2.power(2, 3) === 8, 'Power test failed');
console.assert(calc2.sqrt(16) === 4, 'Square root test failed');
console.log('Extended tests passed!');
EOF

git add test.js
git commit -m "Add tests for new functions"

# View commit history
git log --oneline
```

Output:

```
h8i9j0k (HEAD -> main) Add tests for new functions
g7h8i9j Add logarithm function
f6g7h8i Add factorial function
e5f6g7h Optimize division calculation  ← Bug introduced here!
d4e5f6g Add square root function
c3d4e5f Add power function
b2c3d4e Add percentage calculation
a1b2c3d Initial calculator implementation
```

```
# Test current state (should fail)
node test.js
```

Output:

```
Assertion failed: Division test failed
```

Basic Bisect Workflow

Starting Bisect Session

```
# Start bisect session
git bisect start

# Mark current commit as bad (bug exists)
git bisect bad

# Mark a known good commit (first commit)
git bisect good a1b2c3d

# Git will checkout a middle commit
```

Output:

```
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[d4e5f6g] Add square root function
```

Testing and Marking Commits

```
# Test the current commit
node test.js
```

Output:

```
All tests passed!
```

```
# Mark as good since tests pass
git bisect good
```

Output:

```
Bisecting: 1 revision left to test after this (roughly 1 step)
[f6g7h8i] Add factorial function
```

```
# Test this commit
node test.js
```

Output:

```
Assertion failed: Division test failed
```

```
# Mark as bad since tests fail
git bisect bad
```

Output:

```
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[e5f6g7h] Optimize division calculation
```

```
# Test this commit
node test.js
```

Output:

```
Assertion failed: Division test failed
```

```
# Mark as bad
git bisect bad
```

Output:

```
e5f6g7h is the first bad commit
commit e5f6g7h
Author: Your Name <your.email@example.com>
Date:   [timestamp]

    Optimize division calculation

 calculator.js | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Finishing Bisect Session

```
# End bisect session and return to original HEAD
git bisect reset

# View the problematic commit
git show e5f6g7h
```

Output:

```
commit e5f6g7h
Author: Your Name <your.email@example.com>
Date:   [timestamp]

    Optimize division calculation

diff --git a/calculator.js b/calculator.js
index abc123..def456 100644
--- a/calculator.js
+++ b/calculator.js
@@ -15,7 +15,7 @@ class Calculator {
         if (b === 0) {
             throw new Error('Division by zero');
         }
-        return a / b;
+        return a / b + 0.1;  ← Here's the bug!
```

```
    }
}
```

Advanced Bisect Techniques

Automated Bisect with Scripts

```
# Create test script for automated bisect
cat > bisect-test.sh << 'EOF'
#!/bin/bash

# Run the test
node test.js > /dev/null 2>&1

# Return exit code (0 = good, 1 = bad)
if [ $? -eq 0 ]; then
    echo "Tests passed - marking as good"
    exit 0
else
    echo "Tests failed - marking as bad"
    exit 1
fi
EOF

chmod +x bisect-test.sh

# Start automated bisect
git bisect start
git bisect bad HEAD
git bisect good a1b2c3d

# Run automated bisect
git bisect run ./bisect-test.sh
```

Output:

```
running ./bisect-test.sh
Tests passed - marking as good
Bisecting: 1 revision left to test after this (roughly 1 step)
[f6g7h8i] Add factorial function
running ./bisect-test.sh
Tests failed - marking as bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[e5f6g7h] Optimize division calculation
running ./bisect-test.sh
Tests failed - marking as bad
e5f6g7h is the first bad commit
```

Complex Test Scripts

```
# Create more sophisticated test script
cat > advanced-bisect-test.sh << 'EOF'
#!/bin/bash

set -e  # Exit on any error

echo "Testing commit: $(git rev-parse --short HEAD)"

# Check if required files exist
if [ ! -f "calculator.js" ] || [ ! -f "test.js" ]; then
    echo "Required files missing - skipping"
    exit 125  # Skip this commit
fi

# Check if Node.js syntax is valid
node -c calculator.js || {
    echo "Syntax error in calculator.js - skipping"
    exit 125
}

node -c test.js || {
    echo "Syntax error in test.js - skipping"
    exit 125
}

# Run tests with timeout
timeout 10s node test.js > test-output.log 2>&1
test_result=$?

if [ $test_result -eq 0 ]; then
    echo "✓ All tests passed"
    exit 0  # Good commit
elif [ $test_result -eq 124 ]; then
    echo "⏰ Tests timed out - skipping"
    exit 125  # Skip commit
else
    echo "✗ Tests failed"
    cat test-output.log
    exit 1  # Bad commit
fi
EOF

chmod +x advanced-bisect-test.sh
```

Bisect with Build Systems

```
# Example for projects with build steps
cat > build-and-test.sh << 'EOF'
```

```
#!/bin/bash

echo "Building and testing commit: $(git rev-parse --short HEAD)"

# Install dependencies (if package.json exists)
if [ -f "package.json" ]; then
    npm install --silent || {
        echo "npm install failed - skipping"
        exit 125
    }
fi

# Build project
if [ -f "Makefile" ]; then
    make clean && make || {
        echo "Build failed - skipping"
        exit 125
    }
elif [ -f "package.json" ] && grep -q '"build"' package.json; then
    npm run build || {
        echo "Build failed - skipping"
        exit 125
    }
fi

# Run tests
if [ -f "package.json" ] && grep -q '"test"' package.json; then
    npm test
elif [ -f "test.js" ]; then
    node test.js
else
    echo "No tests found - skipping"
    exit 125
fi
EOF

chmod +x build-and-test.sh
```

Real-world Bisect Examples

Example 1: Performance Regression

```
# Create performance test scenario
cat > performance-test.js << EOF
const Calculator = require('./calculator');

function performanceTest() {
    const calc = new Calculator();
    const start = Date.now();

    // Perform many calculations
```

```
    for (let i = 0; i < 100000; i++) {
        calc.add(i, i + 1);
        calc.multiply(i, 2);
        calc.divide(i + 1, 2);
    }

    const duration = Date.now() - start;
    console.log(\`Performance test completed in \${duration}ms\`);

    // Fail if too slow (arbitrary threshold)
    if (duration > 1000) {
        console.error('Performance regression detected!');
        process.exit(1);
    }

    console.log('Performance test passed');
}

performanceTest();
EOF

# Create bisect script for performance
cat > perf-bisect.sh << 'EOF'
#!/bin/bash
node performance-test.js
EOF

chmod +x perf-bisect.sh
```

Example 2: Integration Test Failure

```
# Create integration test
cat > integration-test.js << EOF
const Calculator = require('./calculator');

function integrationTest() {
    const calc = new Calculator();

    // Complex calculation that might reveal bugs
    const result = calc.divide(
        calc.multiply(
            calc.add(10, 5),
            calc.subtract(20, 5)
        ),
        calc.power(2, 3)
    );

    const expected = (15 * 15) / 8; // 28.125

    console.log(\`Result: \${result}, Expected: \${expected}\`);
```

```
    if (Math.abs(result - expected) > 0.01) {
        console.error('Integration test failed!');
        process.exit(1);
    }

    console.log('Integration test passed');
}

integrationTest();
EOF

# Run integration test
node integration-test.js
```

Example 3: Cross-platform Compatibility

```
# Create platform-specific test
cat > platform-test.sh << 'EOF'
#!/bin/bash

echo "Testing on platform: $(uname -s)"

# Test platform-specific functionality
if [[ "$OSTYPE" == "msys" || "$OSTYPE" == "win32" ]]; then
    # Windows-specific tests
    echo "Running Windows-specific tests"
    # Add Windows-specific test logic
elif [[ "$OSTYPE" == "darwin"* ]]; then
    # macOS-specific tests
    echo "Running macOS-specific tests"
    # Add macOS-specific test logic
else
    # Linux-specific tests
    echo "Running Linux-specific tests"
    # Add Linux-specific test logic
fi

# Run common tests
node test.js
EOF

chmod +x platform-test.sh
```

Bisect Best Practices

1. Prepare Good Test Cases

```
# Create comprehensive test that clearly identifies the bug
cat > comprehensive-test.js << EOF
```

```
const Calculator = require('./calculator');

function runComprehensiveTests() {
    const calc = new Calculator();
    const tests = [
        // Basic operations
        { method: 'add', args: [2, 3], expected: 5 },
        { method: 'subtract', args: [5, 3], expected: 2 },
        { method: 'multiply', args: [4, 3], expected: 12 },
        { method: 'divide', args: [10, 2], expected: 5 },

        // Edge cases
        { method: 'divide', args: [1, 3], expected: 0.3333333333333333 },
        { method: 'add', args: [0, 0], expected: 0 },
        { method: 'multiply', args: [-2, 3], expected: -6 },
    ];

    for (const test of tests) {
        const result = calc[test.method](...test.args);
        if (Math.abs(result - test.expected) > 0.0001) {
            console.error(\`Test failed: \${test.method}(\${test.args.join(',
')})\`);
            console.error(\`Expected: \${test.expected}, Got: \${result}\`);
            process.exit(1);
        }
    }

    console.log('All comprehensive tests passed');
}

runComprehensiveTests();
EOF
```

2. Handle Edge Cases in Bisect Scripts

```
# Robust bisect script
cat > robust-bisect.sh << 'EOF'
#!/bin/bash

set -e

# Function to log with timestamp
log() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] $1"
}

log "Testing commit: $(git rev-parse --short HEAD)"

# Check if this is a merge commit (might want to skip)
if git rev-parse --verify HEAD^2 >/dev/null 2>&1; then
    log "Merge commit detected - skipping"
```

```
    exit 125
fi

# Check for required files
required_files=("calculator.js" "test.js")
for file in "${required_files[@]}"; do
    if [ ! -f "$file" ]; then
        log "Required file $file missing - skipping"
        exit 125
    fi
done

# Check syntax
for file in *.js; do
    if ! node -c "$file" 2>/dev/null; then
        log "Syntax error in $file - skipping"
        exit 125
    fi
done

# Run tests with proper error handling
if timeout 30s node comprehensive-test.js; then
    log "Tests passed - good commit"
    exit 0
else
    exit_code=$?
    if [ $exit_code -eq 124 ]; then
        log "Tests timed out - skipping"
        exit 125
    else
        log "Tests failed - bad commit"
        exit 1
    fi
fi
EOF

chmod +x robust-bisect.sh
```

3. Document Bisect Process

```
# Create bisect documentation
cat > BISECT_GUIDE.md << 'EOF'
# Bisect Guide for This Project

## Quick Start

```bash
# Start bisect
git bisect start
git bisect bad HEAD
git bisect good v1.0.0  # Known good version
```
```

```
# Run automated bisect
git bisect run ./robust-bisect.sh
```

Manual Testing

If automated bisect fails, test manually:

1. Run: `node comprehensive-test.js`
2. If tests pass: `git bisect good`
3. If tests fail: `git bisect bad`
4. If unsure/broken: `git bisect skip`

Common Issues

- **Syntax errors**: Use `git bisect skip`
- **Missing dependencies**: Check if commit predates dependency
- **Build failures**: May need to skip or fix build process

Exit Codes

- 0: Good commit (tests pass)
- 1: Bad commit (tests fail)
- 125: Skip commit (can't test) EOF

````
## Troubleshooting Bisect

### Common Issues and Solutions

#### 1. Skipping Untestable Commits

```bash
# When a commit can't be tested
git bisect skip

# Skip multiple commits
git bisect skip commit1 commit2 commit3

# Skip a range
git bisect skip commit1..commit5
```
````

2. Bisect Got Confused

```
# Reset and start over
git bisect reset
git bisect start
```

```
# Be more careful with good/bad marking
git bisect bad HEAD
git bisect good known-good-commit
```

3. Multiple Bugs

```
# If there are multiple bugs, bisect one at a time
# Create specific test for each bug

# Test for bug A only
cat > test-bug-a.sh << 'EOF'
#!/bin/bash
# Test only for specific bug A
node test-bug-a.js
EOF

# Run bisect for bug A
git bisect run ./test-bug-a.sh
```

4. Non-linear History

```
# For complex merge histories
git bisect start --first-parent

# This follows only the first parent of merges
```

Recovery from Bisect Problems

```
# View bisect log
git bisect log

# Replay bisect with modifications
git bisect replay bisect-log-file

# Visualize bisect progress
git bisect visualize
git bisect view  # Same as visualize
```

Advanced Bisect Workflows

Workflow 1: Continuous Integration Bisect

```
#!/bin/bash
# CI-friendly bisect script

ci_bisect() {
    local good_commit=$1
    local bad_commit=$2
    local test_command=$3

    echo "Starting CI bisect from $good_commit to $bad_commit"

    git bisect start "$bad_commit" "$good_commit"

    # Create CI test script
    cat > ci-test.sh << EOF
#!/bin/bash
set -e

# Setup CI environment
export NODE_ENV=test
export CI=true

# Install dependencies
npm ci --silent

# Run build
npm run build

# Run tests
$test_command
EOF

    chmod +x ci-test.sh

    # Run automated bisect
    git bisect run ./ci-test.sh

    # Cleanup
    rm ci-test.sh
    git bisect reset
}

# Usage: ci_bisect v1.0.0 HEAD "npm test"
```

Workflow 2: Performance Bisect

```
#!/bin/bash
# Performance regression bisect

performance_bisect() {
    local baseline_commit=$1
```

```
    local regression_commit=$2
    local performance_threshold=$3

    cat > perf-test.sh << EOF
#!/bin/bash

# Build project
npm run build > /dev/null 2>&1 || exit 125

# Run performance test
start_time=\$(date +%s%N)
npm run perf-test > /dev/null 2>&1
end_time=\$(date +%s%N)

# Calculate duration in milliseconds
duration=\$(( (end_time - start_time) / 1000000 ))

echo "Performance test took: \${duration}ms"

if [ \$duration -gt $performance_threshold ]; then
    echo "Performance regression detected"
    exit 1
else
    echo "Performance acceptable"
    exit 0
fi
EOF

    chmod +x perf-test.sh

    git bisect start "$regression_commit" "$baseline_commit"
    git bisect run ./perf-test.sh

    rm perf-test.sh
    git bisect reset
}
```

Quick Reference

```
# Start bisect
git bisect start                # Begin bisect session
git bisect bad [commit]         # Mark commit as bad (default: HEAD)
git bisect good [commit]        # Mark commit as good
git bisect skip [commit]        # Skip untestable commit

# Automated bisect
git bisect run <script>         # Run script automatically
git bisect run make test        # Example with make
git bisect run npm test         # Example with npm

# Bisect control
```

```
git bisect reset [commit]       # End bisect, return to commit (default: original
HEAD)
git bisect log                  # Show bisect log
git bisect replay <file>        # Replay bisect from log file

# Visualization
git bisect visualize            # Show commits being bisected
git bisect view                 # Same as visualize

# Advanced options
git bisect start --first-parent # Follow only first parent in merges
git bisect start --no-checkout  # Don't checkout commits (for worktrees)

# Script exit codes
# 0: Good commit
# 1-124, 126-255: Bad commit
# 125: Skip commit (untestable)
```

Previous: Git Reflog Recovery
Next: Git Tags

Git Tags

Git tags are references that point to specific commits, typically used to mark release points, important milestones, or significant versions in your project history. Tags provide a way to create permanent bookmarks in your repository.

Understanding Git Tags

What are Tags?

Tags are immutable references to specific commits that:

- Mark important points in history (releases, milestones)
- Provide human-readable names for commits
- Can include additional metadata (annotated tags)
- Are typically used for versioning (v1.0, v2.1.3, etc.)

Tags vs Branches

Feature	Tags	Branches
Mutability	Immutable	Mutable
Purpose	Mark specific points	Track ongoing work
Movement	Stay at one commit	Move with new commits
Typical Use	Releases, milestones	Feature development

Types of Tags

Lightweight Tags

Lightweight tags are simple pointers to commits:

- Just a name pointing to a commit
- No additional metadata
- Stored as a simple reference
- Quick and easy to create

Annotated Tags

Annotated tags are full objects in Git database:

- Contain tagger name, email, and date
- Include a tagging message
- Can be signed with GPG
- Recommended for releases

Setting Up Examples

```
# Create example repository
mkdir git-tags-demo
cd git-tags-demo
git init

# Create initial project structure
cat > package.json << EOF
{
  "name": "awesome-calculator",
  "version": "0.1.0",
  "description": "An awesome calculator library",
  "main": "index.js",
  "scripts": {
    "test": "node test.js"
  },
  "author": "Your Name",
  "license": "MIT"
}
EOF

cat > index.js << EOF
class Calculator {
    add(a, b) {
        return a + b;
    }

    subtract(a, b) {
        return a - b;
    }
```

```
}

module.exports = Calculator;
EOF

cat > README.md << EOF
# Awesome Calculator

A simple calculator library for Node.js.

## Installation

\`\`\`bash
npm install awesome-calculator
\`\`\`

## Usage

\`\`\`javascript
const Calculator = require('awesome-calculator');
const calc = new Calculator();
console.log(calc.add(2, 3)); // 5
\`\`\`
EOF

git add .
git commit -m "Initial project setup"

# Add basic functionality
cat >> index.js << EOF

    multiply(a, b) {
        return a * b;
    }

    divide(a, b) {
        if (b === 0) {
            throw new Error('Division by zero');
        }
        return a / b;
    }
EOF

# Update version
sed -i 's/"version": "0.1.0"/"version": "1.0.0"/' package.json

git add .
git commit -m "Add multiply and divide functions - ready for v1.0.0"

# View commit history
git log --oneline
```

Output:

```
b2c3d4e (HEAD -> main) Add multiply and divide functions - ready for v1.0.0
a1b2c3d Initial project setup
```

Creating Tags

Lightweight Tags

```
# Create lightweight tag for current commit
git tag v1.0.0

# Create lightweight tag for specific commit
git tag v0.1.0 a1b2c3d

# View tags
git tag
```

Output:

```
v0.1.0
v1.0.0
```

Annotated Tags

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

Features:
- Basic arithmetic operations
- Error handling for division by zero
- Clean API design"

# Create annotated tag with editor
git tag -a v1.0.1 -m "Patch release v1.0.1"

# View tag information
git show v1.0.0-annotated
```

Output:

```
tag v1.0.0-annotated
Tagger: Your Name <your.email@example.com>
Date:   [timestamp]
```

```
Release version 1.0.0

Features:
- Basic arithmetic operations
- Error handling for division by zero
- Clean API design

commit b2c3d4e...
Author: Your Name <your.email@example.com>
Date:   [timestamp]

    Add multiply and divide functions - ready for v1.0.0

[commit details...]
```

Signed Tags

```
# Create GPG-signed tag (requires GPG setup)
git tag -s v1.0.0-signed -m "Signed release v1.0.0"

# Verify signed tag
git tag -v v1.0.0-signed

# Create signed tag with specific key
git tag -u key-id v1.0.0-signed -m "Signed with specific key"
```

Working with Tags

Listing Tags

```
# 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 -n3  # Show 3 lines of annotation

# List tags sorted by version
git tag --sort=version:refname
git tag --sort=-version:refname  # Reverse order
```

Viewing Tag Information

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

# Show only tag object (for annotated tags)
git cat-file -p v1.0.0-annotated

# Show commit that tag points to
git rev-list -n 1 v1.0.0

# Show tag type
git cat-file -t v1.0.0          # commit (lightweight)
git cat-file -t v1.0.0-annotated # tag (annotated)
```

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

# View current state
git status
```

Output:

```
HEAD detached at v1.0.0
nothing to commit, working tree clean
```

Practical Tagging Workflows

Semantic Versioning Workflow

```
# Create development commits
echo "// Added input validation" >> index.js
git add index.js
git commit -m "Add input validation"

echo "// Performance improvements" >> index.js
git add index.js
git commit -m "Improve performance"

# Update version for minor release
sed -i 's/"version": "1.0.0"/"version": "1.1.0"/' package.json
git add package.json
git commit -m "Bump version to 1.1.0"
```

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

New Features:
- Input validation for all operations
- Performance improvements

Breaking Changes:
- None

Bug Fixes:
- None"

# Create patch release
echo "// Bug fix" >> index.js
git add index.js
git commit -m "Fix edge case in division"

sed -i 's/"version": "1.1.0"/"version": "1.1.1"/' package.json
git add package.json
git commit -m "Bump version to 1.1.1"

git tag -a v1.1.1 -m "Release v1.1.1

Bug Fixes:
- Fix edge case in division operation"

# View version history
git tag --sort=version:refname
```

Output:

```
v0.1.0
v1.0.0
v1.0.0-annotated
v1.1.0
v1.1.1
```

Release Branch Workflow

```
# Create release branch
git checkout -b release/v2.0.0

# Prepare release
cat > CHANGELOG.md << EOF
# Changelog

## [2.0.0] - $(date +%Y-%m-%d)
```

```
### Added
- New advanced mathematical operations
- Comprehensive error handling
- TypeScript definitions

### Changed
- API restructured for better usability
- Improved performance

### Breaking Changes
- Constructor now requires configuration object
- Method signatures changed for consistency
EOF

# Update version
sed -i 's/"version": "1.1.1"/"version": "2.0.0"/' package.json

git add .
git commit -m "Prepare release v2.0.0"

# Create release candidate tag
git tag -a v2.0.0-rc.1 -m "Release candidate v2.0.0-rc.1"

# After testing, create final release
git tag -a v2.0.0 -m "Release v2.0.0

Major release with breaking changes.
See CHANGELOG.md for details."

# Merge back to main
git checkout main
git merge release/v2.0.0 --no-ff
```

Hotfix Workflow

```
# Create hotfix from production tag
git checkout -b hotfix/v2.0.1 v2.0.0

# Fix critical bug
echo "// Critical security fix" >> index.js
git add index.js
git commit -m "SECURITY: Fix critical vulnerability"

# Update version
sed -i 's/"version": "2.0.0"/"version": "2.0.1"/' package.json
git add package.json
git commit -m "Bump version to 2.0.1"

# Create hotfix tag
git tag -a v2.0.1 -m "Hotfix v2.0.1
```

```
Security Fix:
- Fix critical vulnerability in input processing

This is a security release. All users should upgrade immediately."

# Merge hotfix to main and develop
git checkout main
git merge hotfix/v2.0.1 --no-ff

# Clean up hotfix branch
git branch -d hotfix/v2.0.1
```

Advanced Tag Operations

Moving and Deleting Tags

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

# Recreate tag at different commit
git tag -a v1.0.0-corrected HEAD~2 -m "Corrected release tag"

# Force move existing tag
git tag -f v1.0.0 HEAD~1

# Delete remote tag
git push origin --delete v1.0.0-annotated

# Push corrected tag
git push origin v1.0.0-corrected
```

Tag Ranges and Comparisons

```
# Show commits between tags
git log v1.0.0..v1.1.0 --oneline

# Show changes between tags
git diff v1.0.0..v1.1.0

# Show files changed between tags
git diff --name-only v1.0.0..v1.1.0

# Show commits since last tag
git log $(git describe --tags --abbrev=0)..HEAD --oneline

# Count commits since tag
git rev-list --count v1.0.0..HEAD
```

Finding Tags

```
# Find tag containing specific commit
git tag --contains commit-hash

# Find most recent tag
git describe --tags
git describe --tags --abbrev=0  # Just tag name

# Find tag with pattern
git tag -l "v1.*" --sort=-version:refname | head -1

# Find tags by date
git for-each-ref --format='%(refname:short) %(taggerdate)' refs/tags
```

Working with Remote Tags

Pushing 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

# Push tags and commits together
git push origin main --tags
```

Fetching Tags

```
# Fetch all tags
git fetch origin --tags

# Fetch specific tag
git fetch origin refs/tags/v1.0.0:refs/tags/v1.0.0

# Fetch and prune deleted tags
git fetch origin --prune --prune-tags
```

Syncing Tags

```
# Script to sync tags with remote
cat > sync-tags.sh << 'EOF'
#!/bin/bash

echo "Syncing tags with remote..."

# Fetch all remote tags
git fetch origin --tags

# Remove local tags that don't exist on remote
git tag -l | while read tag; do
    if ! git ls-remote --tags origin | grep -q "refs/tags/$tag"; then
        echo "Removing local tag: $tag"
        git tag -d "$tag"
    fi
done

# Fetch any new tags
git fetch origin --prune --prune-tags

echo "Tag sync complete"
EOF

chmod +x sync-tags.sh
```

Tag Automation

Automated Tagging Script

```
# Create automated release script
cat > create-release.sh << 'EOF'
#!/bin/bash

set -e

# Configuration
VERSION_FILE="package.json"
CHANGELOG_FILE="CHANGELOG.md"

# Functions
get_current_version() {
    grep '"version":' "$VERSION_FILE" | sed 's/.*"version": "\([^"]*\)".*/\1/'
}

update_version() {
    local new_version=$1
    sed -i "s/\"version\": \"[^\"]*\"/\"version\": \"$new_version\"/"
"$VERSION_FILE"
}
```

```
update_changelog() {
    local version=$1
    local date=$(date +%Y-%m-%d)

    # Create temporary file with new entry
    cat > temp_changelog << EOF
# Changelog

## [$version] - $date

### Added
-

### Changed
-

### Fixed
-

EOF

    # Append existing changelog (skip first line)
    tail -n +2 "$CHANGELOG_FILE" >> temp_changelog
    mv temp_changelog "$CHANGELOG_FILE"
}

create_release() {
    local version=$1
    local message="$2"

    echo "Creating release $version"

    # Update version
    update_version "$version"

    # Update changelog
    update_changelog "$version"

    # Commit changes
    git add "$VERSION_FILE" "$CHANGELOG_FILE"
    git commit -m "Prepare release $version"

    # Create annotated tag
    git tag -a "v$version" -m "$message"

    echo "Release $version created successfully"
    echo "Don't forget to push: git push origin main --tags"
}

# Main script
if [ $# -lt 2 ]; then
    echo "Usage: $0 <version> <release_message>"
    echo "Example: $0 1.2.0 'Release with new features'"
    exit 1
```

```
fi

VERSION=$1
MESSAGE="$2"

# Validate version format
if ! echo "$VERSION" | grep -qE '^[0-9]+\.[0-9]+\.[0-9]+$'; then
    echo "Error: Version must be in format X.Y.Z"
    exit 1
fi

# Check if tag already exists
if git tag -l | grep -q "^v$VERSION$"; then
    echo "Error: Tag v$VERSION already exists"
    exit 1
fi

# Create release
create_release "$VERSION" "$MESSAGE"
EOF

chmod +x create-release.sh

# Usage example
# ./create-release.sh 1.2.0 "Release with new features and bug fixes"
```

Git Hooks for Tagging

```
# Create pre-push hook to validate tags
cat > .git/hooks/pre-push << 'EOF'
#!/bin/bash

# Validate tag format before pushing
while read local_ref local_sha remote_ref remote_sha; do
    if [[ "$remote_ref" == refs/tags/* ]]; then
        tag_name=$(basename "$remote_ref")

        # Check semantic versioning format
        if ! echo "$tag_name" | grep -qE '^v[0-9]+\.[0-9]+\.[0-9]+(-[a-zA-Z0-
9.-]+)?$'; then
            echo "Error: Tag '$tag_name' does not follow semantic versioning"
            echo "Expected format: vX.Y.Z or vX.Y.Z-suffix"
            exit 1
        fi

        # Check if tag is annotated
        if [ "$(git cat-file -t "$tag_name")" != "tag" ]; then
            echo "Warning: Tag '$tag_name' is lightweight. Consider using
annotated tags for releases."
        fi
```

```
        echo "✓ Tag '$tag_name' validation passed"
    fi
done
EOF

chmod +x .git/hooks/pre-push
```

Tag Best Practices

1. Use Semantic Versioning

```
# Follow semantic versioning (semver.org)
# MAJOR.MINOR.PATCH

# Examples:
v1.0.0      # Initial release
v1.0.1      # Patch release (bug fixes)
v1.1.0      # Minor release (new features, backward compatible)
v2.0.0      # Major release (breaking changes)

# Pre-release versions
v1.0.0-alpha.1
v1.0.0-beta.2
v1.0.0-rc.1
```

2. Use Annotated Tags for Releases

```
# Always use annotated tags for releases
git tag -a v1.0.0 -m "Release v1.0.0

Features:
- Feature A
- Feature B

Bug Fixes:
- Fix issue #123
- Fix issue #456"

# Include release notes
git tag -a v1.1.0 -F release-notes.txt
```

3. Consistent Tagging Strategy

```
# Establish team conventions
# - Tag format: v{major}.{minor}.{patch}
# - Tag message format: standardized template
```

```
# - Who can create tags: maintainers only
# - When to tag: after successful CI/CD

# Example tag message template
cat > tag-template.txt << EOF
Release v{VERSION}

## New Features
-

## Bug Fixes
-

## Breaking Changes
-

## Migration Guide
-
EOF
```

4. Protect Important Tags

```
# Use signed tags for security
git tag -s v1.0.0 -m "Signed release v1.0.0"

# Backup important tags
git tag > important-tags-backup.txt

# Document tag policies
cat > TAG_POLICY.md << EOF
# Tag Policy

## Tag Creation
- Only maintainers can create release tags
- All release tags must be annotated
- Tag names must follow semantic versioning
- Tags must include comprehensive release notes

## Tag Protection
- Release tags should never be deleted
- If tag needs correction, create new tag
- Use signed tags for security-critical releases
EOF
```

Troubleshooting Tags

Common Issues

1. Tag Already Exists

```
# Error: tag 'v1.0.0' already exists
# Solution: Use different name or force update
git tag -f v1.0.0 HEAD

# Or delete and recreate
git tag -d v1.0.0
git tag -a v1.0.0 -m "Corrected release"
```

2. Tag Points to Wrong Commit

```
# Move tag to correct commit
git tag -f v1.0.0 correct-commit-hash

# If already pushed, delete remote and push corrected
git push origin --delete v1.0.0
git push origin v1.0.0
```

3. Missing Tags After Clone

```
# Fetch all tags
git fetch origin --tags

# Or fetch specific tag
git fetch origin refs/tags/v1.0.0:refs/tags/v1.0.0
```

4. Tag Conflicts

```
# When local and remote tags differ
git fetch origin
git tag -l | while read tag; do
    local_commit=$(git rev-list -n 1 "$tag")
    remote_commit=$(git rev-list -n 1 "origin/$tag" 2>/dev/null || echo "missing")

    if [ "$local_commit" != "$remote_commit" ]; then
        echo "Tag conflict: $tag"
        echo "  Local:  $local_commit"
        echo "  Remote: $remote_commit"
    fi
done
```

Quick Reference

```
# Create tags
git tag <tagname>                    # Lightweight tag
git tag -a <tagname> -m "message"    # Annotated tag
git tag -s <tagname> -m "message"    # Signed tag
git tag <tagname> <commit>           # Tag specific commit

# List tags
git tag                              # List all tags
git tag -l "pattern"                 # List with pattern
git tag -n                           # List with messages
git tag --sort=version:refname       # Sort by version

# View tags
git show <tagname>                   # Show tag details
git describe --tags                  # Most recent tag
git tag --contains <commit>          # Tags containing commit

# Delete tags
git tag -d <tagname>                 # Delete local tag
git push origin --delete <tagname>   # Delete remote tag

# Push tags
git push origin <tagname>            # Push specific tag
git push origin --tags               # Push all tags
git push origin --follow-tags        # Push annotated tags

# Checkout tags
git checkout <tagname>               # Checkout tag (detached HEAD)
git checkout -b <branch> <tagname>   # Create branch from tag
```

Previous: Git Bisect Bug Hunting
Next: Git Hooks

Git Hooks

Git hooks are scripts that Git executes automatically at certain points in the Git workflow. They allow you to automate tasks, enforce policies, and integrate with external tools to maintain code quality and consistency.

Understanding Git Hooks

What are Git Hooks?

Git hooks are executable scripts that:

- Run automatically at specific Git events
- Can prevent actions if they exit with non-zero status
- Are stored in `.git/hooks/` directory
- Can be written in any scripting language

- Are not version controlled by default

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 completed
- **pre-rebase**: Before rebase operation
- **post-checkout**: After checkout
- **post-merge**: After merge
- **pre-push**: Before push to remote
- **post-rewrite**: After commands that rewrite commits

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 Hook Examples

```
# Create example project
mkdir git-hooks-demo
cd git-hooks-demo
git init

# Create package.json for Node.js project
cat > package.json << EOF
{
  "name": "git-hooks-demo",
  "version": "1.0.0",
  "description": "Demo project for Git hooks",
  "main": "index.js",
  "scripts": {
    "test": "node test.js",
    "lint": "eslint .",
    "format": "prettier --write .",
    "lint:fix": "eslint . --fix"
  },
  "devDependencies": {
    "eslint": "^8.0.0",
    "prettier": "^2.0.0"
  }
}
EOF
```

```
# Create sample JavaScript files
cat > index.js << EOF
const Calculator = require('./calculator');

function main() {
    const calc = new Calculator();
    console.log('2 + 3 =', calc.add(2, 3));
    console.log('10 - 4 =', calc.subtract(10, 4));
}

if (require.main === module) {
    main();
}

module.exports = { main };
EOF

cat > calculator.js << EOF
class Calculator {
    add(a, b) {
        return a + b;
    }

    subtract(a, b) {
        return a - b;
    }

    multiply(a, b) {
        return a * b;
    }

    divide(a, b) {
        if (b === 0) {
            throw new Error('Division by zero');
        }
        return a / b;
    }
}

module.exports = Calculator;
EOF

# Create test file
cat > test.js << EOF
const Calculator = require('./calculator');

function runTests() {
    const calc = new Calculator();
    let passed = 0;
    let failed = 0;

    function test(description, actual, expected) {
        if (actual === expected) {
            console.log(`✓ ${description}`);
```

```
            passed++;
        } else {
            console.log(`✗ ${description}: expected ${expected}, got ${actual}`);
            failed++;
        }
    }

    // Run tests
    test('Addition: 2 + 3', calc.add(2, 3), 5);
    test('Subtraction: 10 - 4', calc.subtract(10, 4), 6);
    test('Multiplication: 3 * 4', calc.multiply(3, 4), 12);
    test('Division: 15 / 3', calc.divide(15, 3), 5);

    console.log(`\nTests completed: ${passed} passed, ${failed} failed`);

    if (failed > 0) {
        process.exit(1);
    }
}

runTests();
EOF

# Create ESLint configuration
cat > .eslintrc.js << EOF
module.exports = {
    env: {
        node: true,
        es2021: true,
    },
    extends: ['eslint:recommended'],
    parserOptions: {
        ecmaVersion: 12,
        sourceType: 'module',
    },
    rules: {
        'no-console': 'off',
        'no-unused-vars': 'error',
        'no-undef': 'error',
        'semi': ['error', 'always'],
        'quotes': ['error', 'single'],
        'indent': ['error', 4],
    },
};
EOF

# Create Prettier configuration
cat > .prettierrc << EOF
{
  "semi": true,
  "trailingComma": "es5",
  "singleQuote": true,
  "printWidth": 80,
  "tabWidth": 4
```

```
}
EOF

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

Pre-commit Hook

Basic Pre-commit Hook

```
# Create pre-commit hook
cat > .git/hooks/pre-commit << 'EOF'
#!/bin/bash

echo "Running pre-commit checks..."

# Check if this is the initial commit
if git rev-parse --verify HEAD >/dev/null 2>&1
then
    against=HEAD
else
    # Initial commit: diff against an empty tree object
    against=$(git hash-object -t tree /dev/null)
fi

# Get list of staged files
staged_files=$(git diff --cached --name-only --diff-filter=ACM)

if [ -z "$staged_files" ]; then
    echo "No staged files to check"
    exit 0
fi

echo "Checking staged files: $staged_files"

# Check for syntax errors in JavaScript files
for file in $staged_files; do
    if [[ "$file" =~ \.(js|jsx)$ ]]; then
        echo "Checking syntax: $file"
        node -c "$file"
        if [ $? -ne 0 ]; then
            echo "❌ Syntax error in $file"
            exit 1
        fi
    fi
done

echo "✅ Pre-commit checks passed"
EOF

chmod +x .git/hooks/pre-commit
```

Advanced Pre-commit Hook with ESLint and Prettier

```
# Create comprehensive pre-commit hook
cat > .git/hooks/pre-commit << 'EOF'
#!/bin/bash

set -e

echo "🔍 Running pre-commit checks..."

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

# Function to print colored output
print_status() {
    local color=$1
    local message=$2
    echo -e "${color}${message}${NC}"
}

# Get staged files
staged_files=$(git diff --cached --name-only --diff-filter=ACM)
js_files=$(echo "$staged_files" | grep -E '\.(js|jsx|ts|tsx)$' || true)
json_files=$(echo "$staged_files" | grep -E '\.(json)$' || true)

if [ -z "$staged_files" ]; then
    print_status $YELLOW "No staged files to check"
    exit 0
fi

print_status $YELLOW "Staged files: $(echo $staged_files | wc -w)"

# Check 1: Prevent commits to main branch
current_branch=$(git symbolic-ref --short HEAD)
if [ "$current_branch" = "main" ] || [ "$current_branch" = "master" ]; then
    print_status $RED "❌ Direct commits to $current_branch branch are not
allowed"
    print_status $YELLOW "Please create a feature branch:"
    print_status $YELLOW "  git checkout -b feature/your-feature-name"
    exit 1
fi

# Check 2: Prevent large files
max_file_size=1048576  # 1MB in bytes
for file in $staged_files; do
    if [ -f "$file" ]; then
        file_size=$(stat -c%s "$file" 2>/dev/null || stat -f%z "$file" 2>/dev/null
|| echo 0)
```

```
        if [ "$file_size" -gt "$max_file_size" ]; then
            print_status $RED "❌ File $file is too large ($(($file_size /
1024))KB > 1MB)"
            exit 1
        fi
    fi
done

# Check 3: Prevent sensitive information
print_status $YELLOW "🔒 Checking for sensitive information..."
sensitive_patterns=(
    "password\s*=\s*['\"][^'\"]+['\"]"  # password = "..."
    "api[_-]?key\s*=\s*['\"][^'\"]+['\"]"  # api_key = "..."
    "secret\s*=\s*['\"][^'\"]+['\"]"  # secret = "..."
    "token\s*=\s*['\"][^'\"]+['\"]"  # token = "..."
    "-----BEGIN [A-Z ]+-----"  # Private keys
)

for file in $staged_files; do
    if [ -f "$file" ]; then
        for pattern in "${sensitive_patterns[@]}"; do
            if grep -iE "$pattern" "$file" >/dev/null; then
                print_status $RED "❌ Potential sensitive information found in
$file"
                print_status $RED "   Pattern: $pattern"
                exit 1
            fi
        done
    fi
done

# Check 4: JSON syntax validation
if [ -n "$json_files" ]; then
    print_status $YELLOW "📋 Validating JSON files..."
    for file in $json_files; do
        if [ -f "$file" ]; then
            if ! python -m json.tool "$file" >/dev/null 2>&1; then
                print_status $RED "❌ Invalid JSON syntax in $file"
                exit 1
            fi
        fi
    done
    print_status $GREEN "✅ JSON validation passed"
fi

# Check 5: JavaScript/TypeScript linting
if [ -n "$js_files" ]; then
    print_status $YELLOW "🔍 Running ESLint..."

    # Check if ESLint is available
    if command -v npx >/dev/null 2>&1 && [ -f "package.json" ]; then
        # Run ESLint on staged files
        echo "$js_files" | xargs npx eslint
        if [ $? -ne 0 ]; then
```

```
            print_status $RED "❌ ESLint found issues"
            print_status $YELLOW "💡 Try running: npm run lint:fix"
            exit 1
        fi
        print_status $GREEN "✅ ESLint passed"
    else
        print_status $YELLOW "⚠️  ESLint not available, skipping lint check"
    fi

    # Check 6: Prettier formatting
    print_status $YELLOW "🎨 Checking Prettier formatting..."
    if command -v npx >/dev/null 2>&1 && [ -f "package.json" ]; then
        # Check if files are formatted
        echo "$js_files" | xargs npx prettier --check
        if [ $? -ne 0 ]; then
            print_status $RED "❌ Code formatting issues found"
            print_status $YELLOW "💡 Try running: npm run format"
            exit 1
        fi
        print_status $GREEN "✅ Prettier formatting passed"
    else
        print_status $YELLOW "⚠️  Prettier not available, skipping format check"
    fi
fi

# Check 7: Run tests
print_status $YELLOW "🧪 Running tests..."
if [ -f "package.json" ] && grep -q '"test"' package.json; then
    npm test
    if [ $? -ne 0 ]; then
        print_status $RED "❌ Tests failed"
        exit 1
    fi
    print_status $GREEN "✅ Tests passed"
elif [ -f "test.js" ]; then
    node test.js
    if [ $? -ne 0 ]; then
        print_status $RED "❌ Tests failed"
        exit 1
    fi
    print_status $GREEN "✅ Tests passed"
else
    print_status $YELLOW "⚠️  No tests found, skipping test check"
fi

# Check 8: Commit message preparation
print_status $YELLOW "📝 Preparing commit message validation..."

print_status $GREEN "🎉 All pre-commit checks passed!"
EOF

chmod +x .git/hooks/pre-commit
```

Testing Pre-commit Hook

```
# Test the pre-commit hook
echo "console.log('test');" > test-file.js
git add test-file.js

# This should trigger the pre-commit hook
git commit -m "Test commit"

# Clean up
git reset --soft HEAD~1
rm test-file.js
```

Commit Message Hook

Commit Message Validation

```
# Create commit-msg hook for conventional commits
cat > .git/hooks/commit-msg << 'EOF'
#!/bin/bash

# Conventional Commits validation
# Format: type(scope): description
# Example: feat(auth): add login functionality

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

commit_message=$(cat "$1")

# Skip merge commits
if echo "$commit_message" | grep -qE '^Merge (branch|pull request)'; then
    exit 0
fi

# Skip revert commits
if echo "$commit_message" | grep -qE '^Revert '; then
    exit 0
fi

# Validate commit message format
if ! echo "$commit_message" | grep -qE "$commit_regex"; then
    echo "❌ Invalid commit message format!"
    echo ""
    echo "Commit message must follow Conventional Commits format:"
    echo "  type(scope): description"
    echo ""
    echo "Types: feat, fix, docs, style, refactor, test, chore, perf, ci, build,
revert"
    echo "Scope: optional, e.g., (auth), (ui), (api)"
```

```
    echo "Description: 1-50 characters"
    echo ""
    echo "Examples:"
    echo "  feat(auth): add login functionality"
    echo "  fix: resolve memory leak in calculator"
    echo "  docs: update README with installation steps"
    echo "  test(calculator): add unit tests for division"
    echo ""
    echo "Your message: $commit_message"
    exit 1
fi

# Check for imperative mood
first_word=$(echo "$commit_message" | sed 's/^[^:]*: *//' | cut -d' ' -f1)
if echo "$first_word" | grep -qE '(ed|ing)$'; then
    echo "⚠️  Warning: Use imperative mood in commit messages"
    echo "   Instead of '$first_word', use the base form of the verb"
    echo "   Example: 'add' instead of 'added', 'fix' instead of 'fixing'"
fi

echo "✅ Commit message format is valid"
EOF

chmod +x .git/hooks/commit-msg
```

Advanced Commit Message Hook

```
# Create comprehensive commit-msg hook
cat > .git/hooks/commit-msg << 'EOF'
#!/bin/bash

set -e

# Colors
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m'

print_status() {
    local color=$1
    local message=$2
    echo -e "${color}${message}${NC}"
}

commit_message_file="$1"
commit_message=$(cat "$commit_message_file")

# Skip certain types of commits
if echo "$commit_message" | grep -qE '^(Merge|Revert|Initial commit)'; then
    print_status $GREEN "✅ Special commit type, skipping validation"
```

```
    exit 0
fi

# Extract first line (subject)
subject=$(echo "$commit_message" | head -n1)
body=$(echo "$commit_message" | tail -n +3)

print_status $YELLOW "📝 Validating commit message..."

# Validation 1: Conventional Commits format
conventional_regex='^(feat|fix|docs|style|refactor|test|chore|perf|ci|build|revert
)(\(.+\))?!?: .{1,50}'
if ! echo "$subject" | grep -qE "$conventional_regex"; then
    print_status $RED "❌ Invalid commit message format!"
    echo ""
    echo "Format: type(scope): description"
    echo "Types: feat, fix, docs, style, refactor, test, chore, perf, ci, build,
revert"
    echo "Scope: optional, e.g., (auth), (ui), (api)"
    echo "Breaking change: add ! after type/scope"
    echo ""
    echo "Examples:"
    echo "  feat(auth): add OAuth2 integration"
    echo "  fix!: resolve breaking API change"
    echo "  docs: update installation guide"
    echo ""
    echo "Your subject: $subject"
    exit 1
fi

# Validation 2: Subject line length
subject_length=${#subject}
if [ $subject_length -gt 72 ]; then
    print_status $RED "❌ Subject line too long ($subject_length > 72
characters)"
    exit 1
fi

# Validation 3: Subject line should not end with period
if echo "$subject" | grep -q '\.$'; then
    print_status $RED "❌ Subject line should not end with a period"
    exit 1
fi

# Validation 4: Body line length (if body exists)
if [ -n "$body" ]; then
    while IFS= read -r line; do
        if [ ${#line} -gt 72 ]; then
            print_status $YELLOW "⚠️  Body line exceeds 72 characters: ${#line}"
            print_status $YELLOW "   Consider wrapping: $line"
        fi
    done <<< "$body"
fi
```

```
# Validation 5: Check for issue references
if echo "$commit_message" | grep -qE '#[0-9]+'; then
    print_status $GREEN "✅ Issue reference found"
fi

# Validation 6: Suggest breaking change format
if echo "$subject" | grep -iE '(break|breaking)' && ! echo "$subject" | grep -q
'!'; then
    print_status $YELLOW "⚠️  Possible breaking change detected"
    print_status $YELLOW "   Consider using ! in type: feat!: or fix!:"
fi

# Validation 7: Check for co-authors
if echo "$commit_message" | grep -qE '^Co-authored-by:'; then
    print_status $GREEN "✅ Co-author information found"
fi

print_status $GREEN "✅ Commit message validation passed"

# Optional: Add commit statistics
files_changed=$(git diff --cached --name-only | wc -l)
lines_added=$(git diff --cached --numstat | awk '{add += $1} END {print add+0}')
lines_deleted=$(git diff --cached --numstat | awk '{del += $2} END {print del+0}')

print_status $YELLOW "📊 Commit stats: $files_changed files,
+$lines_added/-$lines_deleted lines"
EOF

chmod +x .git/hooks/commit-msg
```

Pre-push Hook

Basic Pre-push Hook

```
# Create pre-push hook
cat > .git/hooks/pre-push << 'EOF'
#!/bin/bash

echo "🚀 Running pre-push checks..."

remote="$1"
url="$2"

echo "Pushing to: $remote ($url)"

# Get current branch
current_branch=$(git symbolic-ref --short HEAD)

# Prevent push to main/master from local branches
if [ "$current_branch" = "main" ] || [ "$current_branch" = "master" ]; then
    echo "❌ Direct push to $current_branch is not allowed"
```

```
    echo "Please use pull requests for main branch changes"
    exit 1
fi

# Run tests before push
echo "🧪 Running tests before push..."
if [ -f "package.json" ] && grep -q '"test"' package.json; then
    npm test
    if [ $? -ne 0 ]; then
        echo "❌ Tests failed, push aborted"
        exit 1
    fi
fi

echo "✅ Pre-push checks passed"
EOF

chmod +x .git/hooks/pre-push
```

Advanced Pre-push Hook

```
# Create comprehensive pre-push hook
cat > .git/hooks/pre-push << 'EOF'
#!/bin/bash

set -e

# Colors
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
NC='\033[0m'

print_status() {
    local color=$1
    local message=$2
    echo -e "${color}${message}${NC}"
}

remote="$1"
url="$2"

print_status $BLUE "🚀 Running pre-push checks..."
print_status $YELLOW "Remote: $remote ($url)"

# Get current branch and commits being pushed
current_branch=$(git symbolic-ref --short HEAD)
print_status $YELLOW "Branch: $current_branch"

# Read stdin to get the range of commits being pushed
```

```
while read local_ref local_sha remote_ref remote_sha; do
    if [ "$local_sha" = "0000000000000000000000000000000000000000" ]; then
        # Branch is being deleted
        print_status $YELLOW "🗑️  Deleting branch: $(basename "$remote_ref")"
        continue
    fi

    if [ "$remote_sha" = "0000000000000000000000000000000000000000" ]; then
        # New branch
        range="$local_sha"
        print_status $YELLOW "🆕 New branch: $(basename "$remote_ref")"
    else
        # Existing branch
        range="$remote_sha..$local_sha"
        print_status $YELLOW "📤 Updating branch: $(basename "$remote_ref")"
    fi

    # Check 1: Prevent direct push to protected branches
    branch_name=$(basename "$remote_ref")
    if [ "$branch_name" = "main" ] || [ "$branch_name" = "master" ] || [
"$branch_name" = "develop" ]; then
        print_status $RED "❌ Direct push to protected branch '$branch_name' is
not allowed"
        print_status $YELLOW "Please use pull requests for protected branch
changes"
        exit 1
    fi

    # Check 2: Validate commit messages in the push
    print_status $YELLOW "📝 Validating commit messages..."

conventional_regex='^(feat|fix|docs|style|refactor|test|chore|perf|ci|build|revert
)(\(.+\))?!?: .{1,50}'

    invalid_commits=$(git rev-list "$range" | while read commit; do
        message=$(git log --format=%s -n 1 "$commit")
        if ! echo "$message" | grep -qE "$conventional_regex" &&
           ! echo "$message" | grep -qE '^(Merge|Revert|Initial commit)'; then
            echo "$commit: $message"
        fi
    done)

    if [ -n "$invalid_commits" ]; then
        print_status $RED "❌ Invalid commit messages found:"
        echo "$invalid_commits"
        print_status $YELLOW "Please fix commit messages or use interactive
rebase"
        exit 1
    fi

    # Check 3: Prevent force push (unless explicitly allowed)
    if git merge-base --is-ancestor "$local_sha" "$remote_sha" 2>/dev/null; then
        print_status $YELLOW "⚠️  Force push detected"
        if [ "$ALLOW_FORCE_PUSH" != "true" ]; then
```

```
            print_status $RED "❌ Force push is not allowed"
            print_status $YELLOW "Use 'git push --force-with-lease' if you're
sure"
            print_status $YELLOW "Or set ALLOW_FORCE_PUSH=true environment
variable"
            exit 1
        fi
    fi

    # Check 4: Ensure no merge commits in feature branches
    if [ "$branch_name" != "main" ] && [ "$branch_name" != "master" ] && [
"$branch_name" != "develop" ]; then
        merge_commits=$(git rev-list --merges "$range")
        if [ -n "$merge_commits" ]; then
            print_status $YELLOW "⚠️  Merge commits found in feature branch"
            print_status $YELLOW "Consider rebasing to maintain linear history"
        fi
    fi

    # Check 5: Large file detection
    print_status $YELLOW "📁 Checking for large files..."
    large_files=$(git rev-list "$range" | while read commit; do
        git ls-tree -r -l "$commit" | awk '$4 > 1048576 {print $4, $5}' # Files >
1MB
    done | sort -u)

    if [ -n "$large_files" ]; then
        print_status $YELLOW "⚠️  Large files detected:"
        echo "$large_files"
        print_status $YELLOW "Consider using Git LFS for large files"
    fi
done

# Check 6: Run comprehensive tests
print_status $YELLOW "🧪 Running test suite..."
if [ -f "package.json" ]; then
    if grep -q '"lint"' package.json; then
        print_status $YELLOW "🔍 Running linter..."
        npm run lint
        if [ $? -ne 0 ]; then
            print_status $RED "❌ Linting failed"
            exit 1
        fi
    fi

    if grep -q '"test"' package.json; then
        print_status $YELLOW "🧪 Running tests..."
        npm test
        if [ $? -ne 0 ]; then
            print_status $RED "❌ Tests failed"
            exit 1
        fi
    fi
```

```
    if grep -q '"build"' package.json; then
        print_status $YELLOW "🔨 Running build..."
        npm run build
        if [ $? -ne 0 ]; then
            print_status $RED "❌ Build failed"
            exit 1
        fi
    fi
fi

# Check 7: Security scan (if tools available)
if command -v npm >/dev/null 2>&1 && [ -f "package.json" ]; then
    print_status $YELLOW "🔒 Running security audit..."
    npm audit --audit-level=high
    if [ $? -ne 0 ]; then
        print_status $YELLOW "⚠️  Security vulnerabilities found"
        print_status $YELLOW "Consider running 'npm audit fix'"
    fi
fi

# Check 8: Branch protection rules
if [ -f ".github/branch-protection.json" ]; then
    print_status $YELLOW "🛡️  Checking branch protection rules..."
    # Custom branch protection logic here
fi

print_status $GREEN "🎉 All pre-push checks passed!"
print_status $BLUE "🚀 Proceeding with push..."
EOF

chmod +x .git/hooks/pre-push
```

Post-commit Hook

Notification and Automation

```
# Create post-commit hook for notifications
cat > .git/hooks/post-commit << 'EOF'
#!/bin/bash

# Get commit information
commit_hash=$(git rev-parse HEAD)
commit_message=$(git log -1 --pretty=%B)
author=$(git log -1 --pretty=%an)
email=$(git log -1 --pretty=%ae)
date=$(git log -1 --pretty=%ad --date=iso)
branch=$(git symbolic-ref --short HEAD)

# Count changes
files_changed=$(git diff-tree --no-commit-id --name-only -r HEAD | wc -l)
lines_added=$(git show --numstat HEAD | awk '{add += $1} END {print add+0}')
```

```
lines_deleted=$(git show --numstat HEAD | awk '{del += $2} END {print del+0}')

echo "📝 Commit completed successfully!"
echo "   Hash: $commit_hash"
echo "   Author: $author"
echo "   Branch: $branch"
echo "   Files: $files_changed changed"
echo "   Lines: +$lines_added/-$lines_deleted"
echo "   Message: $commit_message"

# Optional: Send notification to team chat
if [ -n "$SLACK_WEBHOOK_URL" ]; then
    curl -X POST -H 'Content-type: application/json' \
        --data "{
            \"text\": \"New commit by $author on
$branch\\n$commit_message\\n$commit_hash\"
        }" \
        "$SLACK_WEBHOOK_URL"
fi

# Optional: Update issue tracker
if echo "$commit_message" | grep -qE '#[0-9]+'; then
    issue_number=$(echo "$commit_message" | grep -oE '#[0-9]+' | head -1 | tr -d
'#')
    echo "🔗 Referenced issue: #$issue_number"
    # Add issue tracker integration here
fi

# Optional: Trigger CI/CD
if [ "$branch" = "main" ] || [ "$branch" = "develop" ]; then
    echo "🚀 Triggering CI/CD pipeline..."
    # Add CI/CD trigger logic here
fi
EOF

chmod +x .git/hooks/post-commit
```

Hook Management and Sharing

Sharing Hooks with Team

```
# Create hooks directory in repository
mkdir -p .githooks

# Move hooks to shared directory
cp .git/hooks/pre-commit .githooks/
cp .git/hooks/commit-msg .githooks/
cp .git/hooks/pre-push .githooks/

# Create installation script
cat > .githooks/install.sh << 'EOF'
```

```
#!/bin/bash

echo "Installing Git hooks..."

# Get the directory where this script is located
HOOKS_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
GIT_HOOKS_DIR="$(git rev-parse --git-dir)/hooks"

# Install each hook
for hook in "$HOOKS_DIR"/*; do
    hook_name=$(basename "$hook")

    # Skip the install script itself
    if [ "$hook_name" = "install.sh" ]; then
        continue
    fi

    echo "Installing $hook_name..."
    cp "$hook" "$GIT_HOOKS_DIR/$hook_name"
    chmod +x "$GIT_HOOKS_DIR/$hook_name"
done

echo "✅ Git hooks installed successfully!"
echo "To update hooks, run this script again."
EOF

chmod +x .githooks/install.sh

# Create README for hooks
cat > .githooks/README.md << 'EOF'
# Git Hooks

This directory contains shared Git hooks for the project.

## Installation

Run the installation script to set up hooks:

```bash
./.githooks/install.sh
```

Available Hooks

- **pre-commit**: Runs linting, formatting, and tests before commit
- **commit-msg**: Validates commit message format (Conventional Commits)
- **pre-push**: Runs comprehensive checks before push
- **post-commit**: Sends notifications and updates trackers

Bypassing Hooks

In emergency situations, you can bypass hooks:
```

```
# Skip pre-commit hook
git commit --no-verify -m "Emergency fix"

# Skip pre-push hook
git push --no-verify
```

Note: Use bypass sparingly and only in emergencies. EOF

Add hooks to version control

git add .githooks/ git commit -m "feat: add shared Git hooks for code quality"

````
### Hook Configuration

```bash
# Create hook configuration file
cat > .githooks/config.json << 'EOF'
{
  "hooks": {
    "pre-commit": {
      "enabled": true,
      "checks": {
        "syntax": true,
        "linting": true,
        "formatting": true,
        "tests": true,
        "security": true
      },
      "maxFileSize": "1MB",
      "allowedBranches": ["feature/*", "bugfix/*", "hotfix/*"]
    },
    "commit-msg": {
      "enabled": true,
      "format": "conventional",
      "maxLength": 72,
      "requireIssueReference": false
    },
    "pre-push": {
      "enabled": true,
      "protectedBranches": ["main", "master", "develop"],
      "allowForceWithLease": true,
      "runTests": true,
      "runBuild": true
    }
  },
  "notifications": {
    "slack": {
      "enabled": false,
````
````

```
      "webhook": ""
    },
    "email": {
      "enabled": false,
      "recipients": []
    }
  }
}
EOF
```

Hook Best Practices

1. Keep Hooks Fast

```
# Optimize hook performance
cat > .githooks/performance-tips.md << 'EOF'
# Hook Performance Tips

## Pre-commit Optimization
- Only check staged files, not entire repository
- Use parallel processing for multiple files
- Cache results when possible
- Skip checks for certain file types

## Example: Parallel linting
```bash
# Instead of:
for file in $js_files; do
    eslint "$file"
done

# Use:
echo "$js_files" | xargs -P 4 eslint
```

Pre-push Optimization

- Only run full test suite, not individual tests
- Use test result caching
- Skip redundant checks if CI will run them EOF

```
### 2. Provide Clear Error Messages

```bash
# Example of good error messaging
cat > .githooks/error-example.sh << 'EOF'
#!/bin/bash
```

```
# BAD: Unclear error
if ! eslint .; then
    echo "ESLint failed"
    exit 1
fi

# GOOD: Clear, actionable error
if ! eslint .; then
    echo "❌ ESLint found code quality issues"
    echo ""
    echo "To fix automatically:"
    echo "  npm run lint:fix"
    echo ""
    echo "To see detailed errors:"
    echo "  npm run lint"
    echo ""
    echo "To bypass this check (not recommended):"
    echo "  git commit --no-verify"
    exit 1
fi
EOF
```

3. Make Hooks Configurable

```
# Create configurable hook
cat > .githooks/configurable-pre-commit << 'EOF'
#!/bin/bash

# Load configuration
CONFIG_FILE=".githooks/config.json"
if [ -f "$CONFIG_FILE" ]; then
    # Parse JSON config (requires jq)
    LINT_ENABLED=$(jq -r '.hooks."pre-commit".checks.linting' "$CONFIG_FILE")
    TEST_ENABLED=$(jq -r '.hooks."pre-commit".checks.tests' "$CONFIG_FILE")
else
    # Default values
    LINT_ENABLED="true"
    TEST_ENABLED="true"
fi

# Conditional checks
if [ "$LINT_ENABLED" = "true" ]; then
    echo "Running linter..."
    npm run lint
fi

if [ "$TEST_ENABLED" = "true" ]; then
    echo "Running tests..."
    npm test
fi
EOF
```

4. Document Hook Behavior

```
# Create comprehensive documentation
cat > .githooks/HOOKS_GUIDE.md << 'EOF'
# Git Hooks Guide

## Overview

This project uses Git hooks to maintain code quality and consistency.

## Hook Descriptions

### pre-commit
**When**: Before each commit is created
**Purpose**: Ensure code quality and prevent common issues
**Checks**:
- Syntax validation for JavaScript/TypeScript files
- ESLint code quality checks
- Prettier code formatting
- Unit tests execution
- File size limits (max 1MB)
- Sensitive information detection
- JSON syntax validation

**Bypass**: `git commit --no-verify`

### commit-msg
**When**: After commit message is entered
**Purpose**: Enforce consistent commit message format
**Format**: Conventional Commits (type(scope): description)
**Examples**:
- `feat(auth): add OAuth2 integration`
- `fix: resolve memory leak in calculator`
- `docs: update API documentation`

**Bypass**: `git commit --no-verify`

### pre-push
**When**: Before pushing to remote repository
**Purpose**: Final quality gate before sharing code
**Checks**:
- Prevent direct push to protected branches
- Validate all commit messages in push
- Run full test suite
- Build verification
- Security audit
- Large file detection

**Bypass**: `git push --no-verify`

## Troubleshooting
```

```
### Common Issues

1. **Hook not executing**
   - Check if hook file is executable: `chmod +x .git/hooks/hook-name`
   - Verify hook file exists in `.git/hooks/`

2. **Permission denied**
   - Make hook executable: `chmod +x .git/hooks/hook-name`

3. **Hook fails unexpectedly**
   - Check hook logs
   - Run hook manually: `.git/hooks/hook-name`
   - Verify dependencies are installed

### Getting Help

- Check hook output for specific error messages
- Review this documentation
- Ask team members for assistance
- Use `--no-verify` flag only in emergencies
EOF
```

Quick Reference

```
# Hook locations
.git/hooks/                    # Local hooks (not version controlled)
.githooks/                     # Shared hooks (version controlled)

# Common hooks
pre-commit                     # Before commit creation
commit-msg                     # Validate commit message
pre-push                       # Before push to remote
post-commit                    # After commit creation
post-checkout                  # After checkout
post-merge                     # After merge

# Hook management
chmod +x .git/hooks/hook-name  # Make hook executable
git config core.hooksPath .githooks  # Use custom hooks directory

# Bypass hooks (use sparingly)
git commit --no-verify         # Skip pre-commit and commit-msg
git push --no-verify           # Skip pre-push

# Hook installation
./.githooks/install.sh         # Install shared hooks
cp .githooks/* .git/hooks/     # Manual installation

# Testing hooks
```

```
.git/hooks/pre-commit            # Run hook manually
echo "test" | .git/hooks/commit-msg /dev/stdin  # Test commit-msg
```

Previous: Git Tags
Next: Conventional Commits

Conventional Commits

Conventional Commits is a specification for adding human and machine-readable meaning to commit messages. It provides an easy set of rules for creating an explicit commit history, making it easier to write automated tools on top of.

What are Conventional Commits?

Format Structure

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Basic Examples

```
# Simple feature addition
feat: add user authentication

# Bug fix with scope
fix(auth): resolve login redirect issue

# Breaking change
feat!: change API response format

# With body and footer
feat(api): add user profile endpoint

Implement GET /api/users/:id endpoint to retrieve user profile information.
Includes validation for user ID and proper error handling.

Closes #123
```

Commit Types

Primary Types

Type	Description	Example
`feat`	New feature	`feat: add shopping cart`
`fix`	Bug fix	`fix: resolve payment processing error`
`docs`	Documentation	`docs: update API documentation`
`style`	Code style changes	`style: fix indentation in components`
`refactor`	Code refactoring	`refactor: extract validation logic`
`test`	Adding/updating tests	`test: add unit tests for calculator`
`chore`	Maintenance tasks	`chore: update dependencies`

Additional Types

Type	Description	Example
`perf`	Performance improvements	`perf: optimize database queries`
`ci`	CI/CD changes	`ci: add automated testing workflow`
`build`	Build system changes	`build: update webpack configuration`
`revert`	Revert previous commit	`revert: undo feature X implementation`

Practical Examples

Setting Up a Project

```
# Create example project
mkdir conventional-commits-demo
cd conventional-commits-demo
git init

# Initial commit
echo "# Conventional Commits Demo" > README.md
git add README.md
git commit -m "chore: initial project setup"

# Add package.json
cat > package.json << EOF
{
  "name": "conventional-commits-demo",
  "version": "1.0.0",
  "description": "Demo project for conventional commits",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "node test.js",
    "lint": "eslint ."
  },
```

```
  "keywords": ["demo", "conventional-commits"],
  "author": "Your Name",
  "license": "MIT"
}
EOF

git add package.json
git commit -m "feat: add package.json with project configuration"
```

Feature Development Examples

```
# Add main application file
cat > index.js << EOF
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;

app.get('/', (req, res) => {
  res.json({ message: 'Hello, World!' });
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});

module.exports = app;
EOF

git add index.js
git commit -m "feat(server): add basic Express.js server

Implement basic HTTP server with Express.js framework.
Includes health check endpoint and configurable port.

Closes #1"

# Add user routes
mkdir routes
cat > routes/users.js << EOF
const express = require('express');
const router = express.Router();

// Mock user data
const users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];

// GET /users
router.get('/', (req, res) => {
  res.json(users);
```

```
});

// GET /users/:id
router.get('/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }
  res.json(user);
});

module.exports = router;
EOF

git add routes/
git commit -m "feat(api): add user management endpoints

Implement REST API endpoints for user operations:
- GET /users - list all users
- GET /users/:id - get user by ID

Includes proper error handling for non-existent users."

# Update main server to use routes
cat > index.js << EOF
const express = require('express');
const userRoutes = require('./routes/users');
const app = express();
const port = process.env.PORT || 3000;

app.use(express.json());
app.use('/api/users', userRoutes);

app.get('/', (req, res) => {
  res.json({
    message: 'API Server',
    version: '1.0.0',
    endpoints: {
      users: '/api/users'
    }
  });
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});

module.exports = app;
EOF

git add index.js
git commit -m "feat(server): integrate user routes with main application
```

```
Connect user management routes to the main Express application.
Update root endpoint to provide API documentation."
```

Bug Fix Examples

```
# Fix a bug in user route
cat > routes/users.js << EOF
const express = require('express');
const router = express.Router();

// Mock user data
const users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];

// GET /users
router.get('/', (req, res) => {
  res.json(users);
});

// GET /users/:id
router.get('/:id', (req, res) => {
  const userId = parseInt(req.params.id);

  // Fix: Add validation for invalid ID
  if (isNaN(userId) || userId < 1) {
    return res.status(400).json({ error: 'Invalid user ID' });
  }

  const user = users.find(u => u.id === userId);
  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }
  res.json(user);
});

module.exports = router;
EOF

git add routes/users.js
git commit -m "fix(api): add validation for user ID parameter

Resolve issue where invalid user IDs (non-numeric, negative)
were not properly validated, causing unexpected behavior.

Fixes #15"
```

Documentation Examples

```
# Add API documentation
cat > API.md << EOF
# API Documentation

## Base URL
```

http://localhost:3000

```
## Endpoints

### GET /
Returns API information and available endpoints.

### GET /api/users
Returns list of all users.

**Response:**
```json
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com"
  }
]
```

GET /api/users/:id

Returns specific user by ID.

Parameters:

- `id` (number): User ID

Response:

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com"
}
```

Error Responses:

- `400`: Invalid user ID
- `404`: User not found EOF
```

git add API.md git commit -m "docs: add comprehensive API documentation

Include endpoint descriptions, request/response examples, and error handling documentation for the user API."

Update README

cat > README.md << EOF

Conventional Commits Demo

A demonstration project showcasing conventional commit practices with a simple Express.js API.

Features

- RESTful user API
- Express.js server
- Conventional commit history
- Comprehensive documentation

Getting Started

1. Install dependencies:

```
npm install
```

2. Start the server:

```
npm start
```

3. Access the API at http://localhost:3000

API Documentation

See API.md for detailed endpoint documentation.

Development

This project follows Conventional Commits specification for commit messages. EOF

git add README.md git commit -m "docs(readme): enhance project documentation

Add comprehensive setup instructions, feature list, and development guidelines to improve project onboarding."

````
### Testing Examples

```bash
# Add test file
cat > test.js << EOF
const request = require('supertest');
const app = require('./index');

function runTests() {
  console.log('🧪 Running API tests...');

  // Note: This is a simplified test example
  // In real projects, use proper testing frameworks like Jest or Mocha

  const tests = [
    {
      name: 'GET / should return API info',
      test: async () => {
        // Simplified test - in real scenario use supertest
        console.log('✓ Root endpoint test passed');
        return true;
      }
    },
    {
      name: 'GET /api/users should return users array',
      test: async () => {
        console.log('✓ Users endpoint test passed');
        return true;
      }
    }
  ];

  let passed = 0;
  let failed = 0;

  tests.forEach(async ({ name, test }) => {
    try {
      await test();
      console.log(`✓ ${name}`);
      passed++;
    } catch (error) {
      console.log(`✗ ${name}: ${error.message}`);
      failed++;
    }
  });

  console.log(`\nTest Results: ${passed} passed, ${failed} failed`);

  if (failed > 0) {
    process.exit(1);
  }
}
````
````

```
if (require.main === module) {
  runTests();
}

module.exports = { runTests };
EOF

git add test.js
git commit -m "test(api): add basic API endpoint tests

Implement test suite for API endpoints including:
- Root endpoint validation
- User endpoints testing
- Test result reporting

Sets foundation for comprehensive test coverage."
```

Refactoring Examples

```
# Extract configuration
mkdir config
cat > config/server.js << EOF
module.exports = {
  port: process.env.PORT || 3000,
  env: process.env.NODE_ENV || 'development',
  api: {
    version: '1.0.0',
    prefix: '/api'
  }
};
EOF

# Update main server file
cat > index.js << EOF
const express = require('express');
const config = require('./config/server');
const userRoutes = require('./routes/users');
const app = express();

app.use(express.json());
app.use(`${config.api.prefix}/users`, userRoutes);

app.get('/', (req, res) => {
  res.json({
    message: 'API Server',
    version: config.api.version,
    environment: config.env,
    endpoints: {
      users: `${config.api.prefix}/users`
    }
```

```
  });
});

app.listen(config.port, () => {
  console.log(`Server running on port ${config.port} in ${config.env} mode`);
});

module.exports = app;
EOF

git add config/ index.js
git commit -m "refactor(config): extract server configuration

Move server configuration to dedicated config module
for better maintainability and environment management.

Improves code organization and makes configuration
easier to modify across different environments."
```

Performance Examples

```
# Add caching middleware
cat > middleware/cache.js << EOF
const cache = new Map();

function cacheMiddleware(duration = 300000) { // 5 minutes default
  return (req, res, next) => {
    const key = req.originalUrl;
    const cached = cache.get(key);

    if (cached && Date.now() - cached.timestamp < duration) {
      console.log(`Cache hit for ${key}`);
      return res.json(cached.data);
    }

    // Override res.json to cache the response
    const originalJson = res.json;
    res.json = function(data) {
      cache.set(key, {
        data: data,
        timestamp: Date.now()
      });
      console.log(`Cache set for ${key}`);
      return originalJson.call(this, data);
    };

    next();
  };
}

module.exports = cacheMiddleware;
```

```
EOF

# Update user routes to use caching
cat > routes/users.js << EOF
const express = require('express');
const cacheMiddleware = require('../middleware/cache');
const router = express.Router();

// Mock user data
const users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];

// GET /users with caching
router.get('/', cacheMiddleware(60000), (req, res) => {
  res.json(users);
});

// GET /users/:id
router.get('/:id', (req, res) => {
  const userId = parseInt(req.params.id);

  if (isNaN(userId) || userId < 1) {
    return res.status(400).json({ error: 'Invalid user ID' });
  }

  const user = users.find(u => u.id === userId);
  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }
  res.json(user);
});

module.exports = router;
EOF

git add middleware/ routes/users.js
git commit -m "perf(api): add response caching middleware

Implement in-memory caching for user list endpoint
to reduce response time for frequently accessed data.

Improves API performance by caching responses for 1 minute,
reducing unnecessary data processing on repeated requests."
```

Breaking Changes

```
# Make a breaking change to API response format
cat > routes/users.js << EOF
const express = require('express');
```

```
const cacheMiddleware = require('../middleware/cache');
const router = express.Router();

// Mock user data
const users = [
  { id: 1, name: 'John Doe', email: 'john@example.com', createdAt: '2024-01-01' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com', createdAt: '2024-01-02'
}
];

// GET /users with new response format
router.get('/', cacheMiddleware(60000), (req, res) => {
  res.json({
    success: true,
    data: users,
    meta: {
      total: users.length,
      version: '2.0.0'
    }
  });
});

// GET /users/:id with new response format
router.get('/:id', (req, res) => {
  const userId = parseInt(req.params.id);

  if (isNaN(userId) || userId < 1) {
    return res.status(400).json({
      success: false,
      error: 'Invalid user ID'
    });
  }

  const user = users.find(u => u.id === userId);
  if (!user) {
    return res.status(404).json({
      success: false,
      error: 'User not found'
    });
  }

  res.json({
    success: true,
    data: user
  });
});

module.exports = router;
EOF

git add routes/users.js
git commit -m "feat!: change API response format to include metadata

BREAKING CHANGE: API responses now include success flag and metadata.
```

````
Before:
```json
[{"id": 1, "name": "John"}]
````

After:

```
{
  "success": true,
  "data": [{ "id": 1, "name": "John" }],
  "meta": { "total": 1, "version": "2.0.0" }
}
```

This change improves API consistency and provides additional metadata for client applications.

Migration guide available in MIGRATION.md"

````
## Advanced Conventional Commits

### Scopes

Scopes provide additional context about the change:

```bash
# Component-based scopes
feat(auth): add OAuth2 integration
fix(payment): resolve credit card validation
test(user): add profile update tests

# Layer-based scopes
feat(api): add new endpoint
fix(ui): resolve button alignment
perf(db): optimize query performance

# Feature-based scopes
feat(shopping-cart): add item quantity controls
fix(checkout): resolve payment processing error
````

Multiple Scopes

```
# Multiple related scopes
feat(auth,api): add JWT token validation
fix(ui,ux): improve form validation feedback
refactor(db,api): optimize user data queries
```

Body and Footer

```
# Detailed commit with body and footer
git commit -m "feat(api): add user profile image upload

Implement multipart form data handling for user profile images.
Includes image validation, resizing, and secure storage.

Supported formats: JPEG, PNG, WebP
Maximum file size: 5MB
Automatic resizing to 300x300 pixels

Closes #45
Reviewed-by: @johndoe
Tested-by: @janedoe"
```

Co-authored Commits

```
# Pair programming commit
git commit -m "feat(search): implement full-text search functionality

Add Elasticsearch integration for advanced search capabilities.
Includes fuzzy matching, filters, and result highlighting.

Co-authored-by: Jane Doe <jane@example.com>
Co-authored-by: Bob Smith <bob@example.com>"
```

Automation and Tooling

Commitizen Setup

```
# Install commitizen globally
npm install -g commitizen
npm install -g cz-conventional-changelog

# Configure commitizen
echo '{ "path": "cz-conventional-changelog" }' > ~/.czrc

# Or configure per project
npm install --save-dev commitizen cz-conventional-changelog
echo '{ "path": "./node_modules/cz-conventional-changelog" }' > .czrc

# Add to package.json
cat > package.json << EOF
{
  "name": "conventional-commits-demo",
  "version": "1.0.0",
```

```
  "scripts": {
    "commit": "cz"
  },
  "config": {
    "commitizen": {
      "path": "./node_modules/cz-conventional-changelog"
    }
  },
  "devDependencies": {
    "commitizen": "^4.3.0",
    "cz-conventional-changelog": "^3.3.0"
  }
}
EOF

# Use commitizen for commits
npm run commit
# or
git cz
```

Commit Linting

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

# Create commitlint config
cat > .commitlintrc.js << EOF
module.exports = {
  extends: ['@commitlint/config-conventional'],
  rules: {
    'type-enum': [
      2,
      'always',
      [
        'feat',
        'fix',
        'docs',
        'style',
        'refactor',
        'test',
        'chore',
        'perf',
        'ci',
        'build',
        'revert'
      ]
    ],
    'subject-case': [2, 'always', 'lower-case'],
    'subject-empty': [2, 'never'],
    'subject-full-stop': [2, 'never', '.'],
    'header-max-length': [2, 'always', 72]
```

```
  }
};
EOF

# Add to package.json scripts
"scripts": {
  "commitlint": "commitlint --from HEAD~1 --to HEAD --verbose"
}

# Test commitlint
echo "invalid commit message" | npx commitlint
```

Husky Integration

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

# Initialize husky
npx husky install

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

# Add prepare-commit-msg hook for commitizen
npx husky add .husky/prepare-commit-msg 'exec < /dev/tty && node_modules/.bin/cz -
-hook || true'

# Update package.json
"scripts": {
  "prepare": "husky install"
}
```

Semantic Release

```
# Install semantic-release
npm install --save-dev semantic-release

# Create release config
cat > .releaserc.json << EOF
{
  "branches": ["main"],
  "plugins": [
    "@semantic-release/commit-analyzer",
    "@semantic-release/release-notes-generator",
    "@semantic-release/changelog",
    "@semantic-release/npm",
    "@semantic-release/github"
  ]
}
```

```
EOF

# Add to package.json
"scripts": {
  "semantic-release": "semantic-release"
}
```

Best Practices

1. Commit Message Guidelines

```
# DO: Clear, concise, imperative mood
feat: add user authentication
fix: resolve memory leak in parser
docs: update API documentation

# DON'T: Unclear, past tense, too verbose
Added some stuff
Fixed a bug
Updated documentation and also refactored some code and fixed tests
```

2. Scope Usage

```
# DO: Consistent, meaningful scopes
feat(auth): add login functionality
feat(auth): add logout functionality
fix(auth): resolve token expiration

# DON'T: Inconsistent or meaningless scopes
feat(stuff): add things
fix(misc): fix issue
feat(component1): add feature
```

3. Breaking Changes

```
# DO: Clear breaking change indication
feat!: change API response format
feat(api)!: remove deprecated endpoints

# With detailed explanation
feat!: change user authentication method

BREAKING CHANGE: Replace session-based auth with JWT tokens.
Clients must now include Authorization header with Bearer token.

Migration:
```

```
- Update client to use JWT tokens
- Remove session cookie handling
- Add Authorization header to requests
```

4. Commit Frequency

```
# DO: Logical, atomic commits
feat: add user model
feat: add user controller
feat: add user routes
test: add user API tests

# DON'T: Massive commits or micro-commits
feat: add entire user management system
fix: typo
fix: another typo
fix: one more typo
```

Integration with Development Workflow

Feature Branch Workflow

```
# Start feature branch
git checkout -b feat/user-authentication

# Make atomic commits
git commit -m "feat(auth): add user model with validation"
git commit -m "feat(auth): implement password hashing"
git commit -m "feat(auth): add login endpoint"
git commit -m "test(auth): add authentication tests"
git commit -m "docs(auth): add authentication API docs"

# Squash if needed before merge
git rebase -i main
```

Release Workflow

```
# Prepare release branch
git checkout -b release/v2.0.0

# Update version and changelog
git commit -m "chore(release): bump version to 2.0.0"

# Merge to main
git checkout main
git merge release/v2.0.0
```

```
git tag v2.0.0

# Deploy
git commit -m "ci: deploy version 2.0.0 to production"
```

Troubleshooting

Common Issues

1. **Commitlint fails on merge commits**

```
# Configure commitlint to ignore merge commits
echo "extends: ['@commitlint/config-conventional']
rules: {
'subject-case': [0]
}" > .commitlintrc.yml
```

2. **Commitizen not working**

```
# Check configuration
cat .czrc

# Reinstall if needed
npm uninstall -g commitizen
npm install -g commitizen
```

3. **Semantic release not creating releases**

```
# Check commit format
git log --oneline

# Verify configuration
cat .releaserc.json
```

Quick Reference

```
# Commit types
feat     # New feature
fix      # Bug fix
docs     # Documentation
style    # Code style (formatting, etc.)
refactor # Code refactoring
test     # Adding/updating tests
chore    # Maintenance tasks
```

```
perf     # Performance improvements
ci       # CI/CD changes
build    # Build system changes
revert   # Revert previous commit

# Format
type(scope): description

# Breaking changes
type!: description
type(scope)!: description

# Tools
npx commitizen                    # Interactive commit
npx commitlint --from HEAD~1      # Lint last commit
npx semantic-release              # Create release

# Examples
feat: add shopping cart
fix(auth): resolve login issue
docs: update README
feat!: change API response format
```

Previous: Git Hooks
Next: GitHub CLI

GitHub CLI (gh)

GitHub CLI is a command-line tool that brings GitHub functionality to your terminal. It allows you to manage repositories, pull requests, issues, and other GitHub features without leaving the command line.

Installation

Windows

```
# Using winget
winget install --id GitHub.cli

# Using Chocolatey
choco install gh

# Using Scoop
scoop install gh

# Download from GitHub releases
# Visit: https://github.com/cli/cli/releases
```

macOS

```
# Using Homebrew
brew install gh

# Using MacPorts
sudo port 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

# CentOS/RHEL/Fedora
sudo dnf install gh

# Arch Linux
sudo pacman -S github-cli
```

Authentication

Initial Setup

```
# Authenticate with GitHub
gh auth login

# Choose authentication method:
# 1. GitHub.com
# 2. GitHub Enterprise Server
# 3. Login with a token
# 4. Login with web browser

# Check authentication status
gh auth status

# View current user
gh api user
```

Token Authentication

```
# Create personal access token at:
# https://github.com/settings/tokens

# Login with token
gh auth login --with-token < token.txt

# Or set environment variable
export GITHUB_TOKEN="your_token_here"

# Refresh token
gh auth refresh

# Logout
gh auth logout
```

Repository Management

Creating Repositories

```
# Create a new repository
gh repo create my-new-repo

# Create with options
gh repo create my-new-repo \
  --description "My awesome project" \
  --public \
  --clone \
  --gitignore Node \
  --license MIT

# Create from template
gh repo create my-new-repo \
  --template owner/template-repo \
  --clone

# Create private repository
gh repo create my-private-repo --private

# Create repository in organization
gh repo create myorg/team-project --public
```

Repository Operations

```
# Clone repository
gh repo clone owner/repo-name
gh repo clone https://github.com/owner/repo-name
```

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

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

# List repositories
gh repo list
gh repo list owner
gh repo list --limit 50
gh repo list --language javascript
gh repo list --public

# Delete repository (be careful!)
gh repo delete owner/repo-name
```

Repository Settings

```
# Edit repository settings
gh repo edit
gh repo edit --description "Updated description"
gh repo edit --homepage "https://example.com"
gh repo edit --visibility private
gh repo edit --enable-issues=false
gh repo edit --enable-wiki=false

# Archive repository
gh repo archive owner/repo-name

# Set default branch
gh repo edit --default-branch main
```

Pull Request Management

Creating Pull Requests

```
# Create pull request from current branch
gh pr create

# Create with title and body
gh pr create --title "Add new feature" --body "This PR adds..."

# Create draft pull request
gh pr create --draft

# Create with reviewers and assignees
```

```
gh pr create \
  --title "Fix bug in authentication" \
  --body "Resolves issue with login timeout" \
  --reviewer @johndoe,@janedoe \
  --assignee @myself \
  --label bug,priority-high

# Create from template
gh pr create --template .github/pull_request_template.md

# Create and open in browser
gh pr create --web
```

Pull Request Operations

```
# List pull requests
gh pr list
gh pr list --state open
gh pr list --state closed
gh pr list --author @me
gh pr list --assignee johndoe
gh pr list --label bug

# View pull request details
gh pr view 123
gh pr view 123 --web
gh pr view 123 --comments

# Check out pull request locally
gh pr checkout 123
gh pr checkout https://github.com/owner/repo/pull/123

# Review pull request
gh pr review 123
gh pr review 123 --approve
gh pr review 123 --request-changes
gh pr review 123 --comment "Looks good!"

# Merge pull request
gh pr merge 123
gh pr merge 123 --merge      # Create merge commit
gh pr merge 123 --squash     # Squash and merge
gh pr merge 123 --rebase     # Rebase and merge
gh pr merge 123 --delete-branch  # Delete branch after merge

# Close pull request
gh pr close 123

# Reopen pull request
gh pr reopen 123
```

Pull Request Status

```
# Check PR status
gh pr status

# View PR checks
gh pr checks 123
gh pr checks 123 --watch  # Watch checks in real-time

# View PR diff
gh pr diff 123
gh pr diff 123 --name-only

# Ready draft PR
gh pr ready 123

# Convert to draft
gh pr ready 123 --undo
```

Issue Management

Creating Issues

```
# Create new issue
gh issue create

# Create with title and body
gh issue create \
  --title "Bug: Login form not working" \
  --body "Steps to reproduce: 1. Go to login page..."

# Create with labels and assignees
gh issue create \
  --title "Feature request: Dark mode" \
  --body "Add dark mode support" \
  --label enhancement,ui \
  --assignee johndoe

# Create from template
gh issue create --template .github/ISSUE_TEMPLATE/bug_report.md

# Create and open in browser
gh issue create --web
```

Issue Operations

```

```

```
# List issues
gh issue list
gh issue list --state open
gh issue list --state closed
gh issue list --author @me
gh issue list --assignee johndoe
gh issue list --label bug
gh issue list --milestone "v1.0"

# View issue details
gh issue view 456
gh issue view 456 --web
gh issue view 456 --comments

# Edit issue
gh issue edit 456
gh issue edit 456 --title "Updated title"
gh issue edit 456 --body "Updated description"
gh issue edit 456 --add-label priority-high
gh issue edit 456 --remove-label low-priority

# Close issue
gh issue close 456
gh issue close 456 --comment "Fixed in PR #123"

# Reopen issue
gh issue reopen 456

# Pin/unpin issue
gh issue pin 456
gh issue unpin 456

# Transfer issue
gh issue transfer 456 owner/other-repo
```

Issue Comments

```
# Add comment to issue
gh issue comment 456 --body "Thanks for reporting this!"

# Edit comment
gh issue comment 456 --edit-last

# View issue comments
gh issue view 456 --comments
```

Workflow and Actions

GitHub Actions

```
# List workflow runs
gh run list
gh run list --workflow=ci.yml
gh run list --status=failure
gh run list --branch=main

# View workflow run details
gh run view 123456789
gh run view 123456789 --web
gh run view 123456789 --log

# Download run artifacts
gh run download 123456789
gh run download 123456789 --name artifact-name

# Re-run workflow
gh run rerun 123456789
gh run rerun 123456789 --failed-jobs

# Cancel workflow run
gh run cancel 123456789

# Watch workflow run
gh run watch 123456789
```

Workflow Management

```
# List workflows
gh workflow list

# View workflow details
gh workflow view ci.yml
gh workflow view ci.yml --web

# Run workflow manually
gh workflow run ci.yml
gh workflow run ci.yml --ref feature-branch

# Enable/disable workflow
gh workflow enable ci.yml
gh workflow disable ci.yml
```

Release Management

Creating Releases

```

```

```
# Create release
gh release create v1.0.0

# Create with title and notes
gh release create v1.0.0 \
  --title "Version 1.0.0" \
  --notes "First stable release"

# Create from notes file
gh release create v1.0.0 \
  --title "Version 1.0.0" \
  --notes-file CHANGELOG.md

# Create draft release
gh release create v1.0.0 --draft

# Create pre-release
gh release create v1.0.0-beta --prerelease

# Create with assets
gh release create v1.0.0 \
  --title "Version 1.0.0" \
  --notes "Release notes" \
  dist/*.zip dist/*.tar.gz
```

Release Operations

```
# List releases
gh release list
gh release list --limit 10

# View release details
gh release view v1.0.0
gh release view v1.0.0 --web

# Download release assets
gh release download v1.0.0
gh release download v1.0.0 --pattern "*.zip"
gh release download v1.0.0 --archive=zip

# Upload assets to existing release
gh release upload v1.0.0 dist/app.zip
gh release upload v1.0.0 dist/*.zip

# Edit release
gh release edit v1.0.0
gh release edit v1.0.0 --title "Updated title"
gh release edit v1.0.0 --notes "Updated notes"

# Delete release
```

```
gh release delete v1.0.0
gh release delete v1.0.0 --yes  # Skip confirmation
```

Advanced Features

GitHub API Access

```
# Make API requests
gh api repos/:owner/:repo
gh api user
gh api orgs/:org/repos

# POST request
gh api repos/:owner/:repo/issues \
  --method POST \
  --field title="New issue" \
  --field body="Issue description"

# Paginate through results
gh api repos/:owner/:repo/issues --paginate

# Use JQ for JSON processing
gh api repos/:owner/:repo | jq '.name'
gh api user/repos | jq '.[].name'
```

Gist Management

```
# Create gist
gh gist create file.txt
gh gist create file1.txt file2.txt
gh gist create --desc "My gist" --public file.txt

# List gists
gh gist list
gh gist list --public
gh gist list --secret

# View gist
gh gist view abc123
gh gist view abc123 --web

# Edit gist
gh gist edit abc123

# Clone gist
gh gist clone abc123

# Delete gist
gh gist delete abc123
```

SSH Key Management

```
# List SSH keys
gh ssh-key list

# Add SSH key
gh ssh-key add ~/.ssh/id_rsa.pub
gh ssh-key add ~/.ssh/id_rsa.pub --title "My laptop key"

# Delete SSH key
gh ssh-key delete key-id
```

Practical Workflows

Feature Development Workflow

```
# 1. Create feature branch and work on it
git checkout -b feature/new-dashboard
# ... make changes ...
git add .
git commit -m "feat: add new dashboard component"
git push -u origin feature/new-dashboard

# 2. Create pull request
gh pr create \
  --title "Add new dashboard component" \
  --body "Implements new dashboard with charts and metrics" \
  --reviewer @teamlead \
  --label enhancement

# 3. Check PR status
gh pr status
gh pr checks

# 4. Address review comments
# ... make changes ...
git add .
git commit -m "fix: address review comments"
git push

# 5. Merge when approved
gh pr merge --squash --delete-branch
```

Bug Fix Workflow

```
# 1. Create issue for bug
gh issue create \
  --title "Bug: Login form validation error" \
  --body "Login form shows error even with valid credentials" \
  --label bug,priority-high

# 2. Create hotfix branch
git checkout -b hotfix/login-validation
# ... fix the bug ...
git add .
git commit -m "fix: resolve login form validation issue"
git push -u origin hotfix/login-validation

# 3. Create PR that references the issue
gh pr create \
  --title "Fix login form validation" \
  --body "Fixes #123 - resolves validation error in login form" \
  --label bug

# 4. Merge and close issue
gh pr merge --squash
gh issue close 123 --comment "Fixed in PR #124"
```

Release Workflow

```
# 1. Create release branch
git checkout -b release/v1.2.0

# 2. Update version and changelog
echo "1.2.0" > VERSION
echo "## v1.2.0\n- New dashboard\n- Bug fixes" >> CHANGELOG.md
git add VERSION CHANGELOG.md
git commit -m "chore: bump version to 1.2.0"
git push -u origin release/v1.2.0

# 3. Create release PR
gh pr create \
  --title "Release v1.2.0" \
  --body "Release version 1.2.0 with new features and bug fixes" \
  --base main

# 4. Merge release PR
gh pr merge --merge

# 5. Create GitHub release
git checkout main
git pull
git tag v1.2.0
git push origin v1.2.0
```

```
gh release create v1.2.0 \
  --title "Version 1.2.0" \
  --notes-file CHANGELOG.md \
  dist/*.zip
```

Code Review Workflow

```
# 1. List PRs to review
gh pr list --review-requested=@me

# 2. Check out PR for testing
gh pr checkout 456
npm test
npm run build

# 3. Review the changes
gh pr view 456
gh pr diff 456

# 4. Leave review
gh pr review 456 \
  --approve \
  --body "LGTM! Great work on the implementation."

# Or request changes
gh pr review 456 \
  --request-changes \
  --body "Please address the following issues: ..."
```

Configuration and Customization

Configuration

```
# View configuration
gh config list

# Set configuration values
gh config set editor vim
gh config set git_protocol ssh
gh config set prompt enabled

# Set default repository
gh config set default_repo owner/repo-name

# Configuration file location
# Windows: %APPDATA%\GitHub CLI\config.yml
# macOS/Linux: ~/.config/gh/config.yml
```

Aliases

```
# Create aliases
gh alias set pv 'pr view'
gh alias set co 'pr checkout'
gh alias set prs 'pr list --state=open --author=@me'

# List aliases
gh alias list

# Delete alias
gh alias delete pv

# Example useful aliases
gh alias set bugs 'issue list --label=bug'
gh alias set features 'issue list --label=enhancement'
gh alias set myissues 'issue list --assignee=@me'
gh alias set myprs 'pr list --author=@me'
```

Extensions

```
# List available extensions
gh extension list

# Install extension
gh extension install owner/gh-extension-name

# Popular extensions
gh extension install dlvhdr/gh-dash      # Dashboard
gh extension install vilmibm/gh-screensaver  # Fun screensaver
gh extension install mislav/gh-branch    # Branch utilities

# Use extension
gh dash  # If gh-dash is installed

# Update extensions
gh extension upgrade --all

# Remove extension
gh extension remove owner/gh-extension-name
```

Scripting and Automation

Bash Scripts with GitHub CLI

```
#!/bin/bash
# create-feature-branch.sh
```

```
set -e

if [ $# -eq 0 ]; then
    echo "Usage: $0 <feature-name>"
    exit 1
fi

FEATURE_NAME="$1"
BRANCH_NAME="feature/$FEATURE_NAME"

echo "Creating feature branch: $BRANCH_NAME"

# Create and checkout branch
git checkout -b "$BRANCH_NAME"

# Push branch to remote
git push -u origin "$BRANCH_NAME"

echo "Feature branch created successfully!"
echo "Start working on your feature and create a PR when ready:"
echo "  gh pr create --title 'Add $FEATURE_NAME' --body 'Description of the
feature'"
```

```
#!/bin/bash
# auto-merge-dependabot.sh

set -e

echo "Checking for Dependabot PRs..."

# Get Dependabot PRs
DEPENDABOT_PRS=$(gh pr list --author "app/dependabot" --json
number,title,statusCheckRollup --jq '.[] | select(.statusCheckRollup[].state ==
"SUCCESS") | .number')

if [ -z "$DEPENDABOT_PRS" ]; then
    echo "No Dependabot PRs with passing checks found."
    exit 0
fi

echo "Found Dependabot PRs with passing checks:"
echo "$DEPENDABOT_PRS"

for pr in $DEPENDABOT_PRS; do
    echo "Auto-merging PR #$pr"
    gh pr merge "$pr" --squash --delete-branch
    echo "Merged PR #$pr"
done

echo "All eligible Dependabot PRs have been merged."
```

PowerShell Scripts

```powershell
# create-release.ps1
param(
    [Parameter(Mandatory=$true)]
    [string]$Version,

    [string]$Title = "Version $Version",

    [string]$NotesFile = "CHANGELOG.md"
)

$ErrorActionPreference = "Stop"

Write-Host "Creating release $Version" -ForegroundColor Green

# Create and push tag
git tag $Version
git push origin $Version

# Create GitHub release
if (Test-Path $NotesFile) {
    gh release create $Version --title $Title --notes-file $NotesFile
} else {
    gh release create $Version --title $Title --generate-notes
}

Write-Host "Release $Version created successfully!" -ForegroundColor Green
```

JSON Processing with jq

```
# Get repository statistics
gh api repos/:owner/:repo | jq '{
  name: .name,
  stars: .stargazers_count,
  forks: .forks_count,
  issues: .open_issues_count,
  language: .language
}'

# List all open PRs with details
gh pr list --json number,title,author,createdAt | jq '.[] | {
  number: .number,
  title: .title,
  author: .author.login,
  created: .createdAt
}'

# Get workflow run summary
```

```
gh run list --json status,conclusion,workflowName | jq 'group_by(.workflowName) |
map({
  workflow: .[0].workflowName,
  total: length,
  success: map(select(.conclusion == "success")) | length,
  failure: map(select(.conclusion == "failure")) | length
})'
```

Best Practices

1. Use Meaningful Commit Messages

```
# Good commit messages work well with GitHub CLI
git commit -m "feat: add user authentication system"
git commit -m "fix: resolve memory leak in data processing"
git commit -m "docs: update API documentation"

# These create better PR titles when using:
gh pr create  # Uses last commit message as default title
```

2. Template Usage

```
# Create PR template
mkdir -p .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 pass locally
- [ ] Added new tests

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

# Template will be used automatically with:
gh pr create
```

3. Automation Scripts

```
# Create daily standup script
cat > daily-standup.sh << 'EOF'
#!/bin/bash

echo "📊 Daily Standup Report - $(date)"
echo "================================"

echo "\n🔄 My Open PRs:"
gh pr list --author @me --state open

echo "\n👀 PRs Awaiting My Review:"
gh pr list --review-requested @me

echo "\n🐛 My Open Issues:"
gh issue list --assignee @me --state open

echo "\n📈 Recent Activity:"
gh api user/events --paginate | jq -r '.[] | select(.created_at > "'$(date -d '1
day ago' -I)'T00:00:00Z") | "\(.type): \(.repo.name) - \(.created_at)"' | head -10
EOF

chmod +x daily-standup.sh
```

Troubleshooting

Common Issues

1. **Authentication Problems**

```
# Check auth status
gh auth status

# Re-authenticate
gh auth logout
gh auth login

# Check token permissions
gh api user
```

2. **API Rate Limiting**

```
# Check rate limit status
gh api rate_limit

# Use authenticated requests (higher limits)
gh auth login
```

3. **Repository Not Found**

```
# Check if you're in a git repository
git remote -v

# Specify repository explicitly
gh pr list --repo owner/repo-name
```

4. **Permission Denied**

```
# Check repository permissions
gh api repos/:owner/:repo

# Ensure you have necessary permissions for the operation
```

Quick Reference

```
# Authentication
gh auth login                    # Login to GitHub
gh auth status                   # Check auth status
gh auth logout                   # Logout

# Repository
gh repo create name              # Create repository
gh repo clone owner/repo         # Clone repository
gh repo fork owner/repo          # Fork repository
gh repo view                     # View repository info

# Pull Requests
gh pr create                     # Create PR
gh pr list                       # List PRs
gh pr view 123                   # View PR details
gh pr checkout 123               # Checkout PR
gh pr merge 123                  # Merge PR
gh pr review 123                 # Review PR

# Issues
gh issue create                  # Create issue
gh issue list                    # List issues
gh issue view 456                # View issue
gh issue close 456               # Close issue

# Workflows
gh run list                      # List workflow runs
gh run view 789                  # View run details
gh workflow run ci.yml           # Trigger workflow
```

```
# Releases
gh release create v1.0.0         # Create release
gh release list                  # List releases
gh release download v1.0.0       # Download release

# API
gh api user                      # Make API request
gh api repos/:owner/:repo        # Repository API

# Configuration
gh config set editor vim         # Set configuration
gh alias set co 'pr checkout'    # Create alias
gh extension install owner/ext   # Install extension
```

Previous: Conventional Commits
Next: GitHub Actions Basics

GitHub Actions Basics

GitHub Actions is a powerful CI/CD platform that allows you to automate your software development workflows directly in your GitHub repository. You can build, test, and deploy your code right from GitHub.

Understanding GitHub Actions

Key Concepts

- **Workflow**: Automated process defined by a YAML file
- **Event**: Specific activity that triggers a workflow
- **Job**: Set of steps that execute on the same runner
- **Step**: Individual task that can run commands or actions
- **Action**: Reusable unit of code
- **Runner**: Server that runs your workflows

Workflow Structure

```
name: Workflow Name
on: [events]
jobs:
  job-name:
    runs-on: runner-type
    steps:
      - name: Step name
        uses: action@version
      - name: Another step
        run: command
```

Setting Up Your First Workflow

Basic Project Setup

```
# Create example project
mkdir github-actions-demo
cd github-actions-demo
git init

# Create package.json
cat > package.json << EOF
{
  "name": "github-actions-demo",
  "version": "1.0.0",
  "description": "Demo project for GitHub Actions",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "node test.js",
    "lint": "eslint .",
    "format": "prettier --write .",
    "build": "echo 'Building application...'"
  },
  "devDependencies": {
    "eslint": "^8.0.0",
    "prettier": "^2.0.0"
  }
}
EOF

# Create main application
cat > index.js << EOF
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'application/json' });
  res.end(JSON.stringify({
    message: 'Hello from GitHub Actions!',
    timestamp: new Date().toISOString()
  }));
});

const port = process.env.PORT || 3000;
server.listen(port, () => {
  console.log(`Server running on port ${port}`);
});

module.exports = server;
EOF

# Create test file
cat > test.js << EOF
```

```
const http = require('http');
const server = require('./index');

function runTests() {
  console.log('🧪 Running tests...');

  let passed = 0;
  let failed = 0;

  function test(name, condition) {
    if (condition) {
      console.log(`✅ ${name}`);
      passed++;
    } else {
      console.log(`❌ ${name}`);
      failed++;
    }
  }

  // Basic tests
  test('Server module exports', typeof server === 'object');
  test('Environment variables', process.env.NODE_ENV !== undefined || true);
  test('Package.json exists', require('./package.json').name === 'github-actions-
demo');

  console.log(`\nTest Results: ${passed} passed, ${failed} failed`);

  if (failed > 0) {
    process.exit(1);
  }

  console.log('✅ All tests passed!');
}

if (require.main === module) {
  runTests();
}

module.exports = { runTests };
EOF

# Create ESLint config
cat > .eslintrc.js << EOF
module.exports = {
  env: {
    node: true,
    es2021: true,
  },
  extends: ['eslint:recommended'],
  parserOptions: {
    ecmaVersion: 12,
  },
  rules: {
    'no-console': 'off',
```

```
    'no-unused-vars': 'error',
  },
};
EOF

# Create Prettier config
cat > .prettierrc << EOF
{
  "semi": true,
  "trailingComma": "es5",
  "singleQuote": true,
  "printWidth": 80
}
EOF

# Initial commit
git add .
git commit -m "feat: initial project setup"
```

Creating Workflows Directory

```
# Create GitHub Actions directory
mkdir -p .github/workflows

# This is where all workflow files will be stored
# Files must have .yml or .yaml extension
```

Basic CI Workflow

Simple CI Pipeline

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

# Trigger events
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

# Jobs
jobs:
  test:
    name: Test Application
    runs-on: ubuntu-latest

    steps:
      # Checkout code
```

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

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

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

      # Run linting
      - name: Run ESLint
        run: npm run lint

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

      # Check formatting
      - name: Check Prettier formatting
        run: npx prettier --check .
```

Multi-Job Workflow

```
# .github/workflows/comprehensive-ci.yml
name: Comprehensive CI

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

jobs:
  # Linting job
  lint:
    name: Code Linting
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

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

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

      - name: Run ESLint
        run: npm run lint

      - name: Check Prettier
        run: npx prettier --check .

  # Testing job
  test:
    name: Run Tests
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [16, 18, 20]

    steps:
      - uses: actions/checkout@v4

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

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test

  # Build job
  build:
    name: Build Application
    runs-on: ubuntu-latest
    needs: [lint, test]

    steps:
      - 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: Build application
        run: npm run build
```

```
      - name: Upload build artifacts
        uses: actions/upload-artifact@v4
        with:
          name: build-files
          path: dist/
          retention-days: 30
```

Workflow Triggers

Event Types

```
# Push events
on:
  push:
    branches: [ main, develop ]
    tags: [ 'v*' ]
    paths:
      - 'src/**'
      - '!docs/**'

# Pull request events
on:
  pull_request:
    branches: [ main ]
    types: [opened, synchronize, reopened]

# Scheduled events (cron)
on:
  schedule:
    - cron: '0 2 * * 1-5'  # 2 AM, Monday to Friday

# Manual trigger
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy'
        required: true
        default: 'staging'
        type: choice
        options:
          - staging
          - production

# Multiple events
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

```
  release:
    types: [published]
```

Conditional Execution

```
# .github/workflows/conditional.yml
name: Conditional Workflow

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      # Only run on main branch
      - name: Deploy to staging
        if: github.ref == 'refs/heads/main'
        run: echo "Deploying to staging"

      # Only run on pull requests
      - name: Comment on PR
        if: github.event_name == 'pull_request'
        run: echo "This is a pull request"

      # Only run if specific files changed
      - name: Check for changes
        uses: dorny/paths-filter@v2
        id: changes
        with:
          filters: |
            src:
              - 'src/**'
            docs:
              - 'docs/**'

      - name: Run tests
        if: steps.changes.outputs.src == 'true'
        run: npm test
```

Environment Variables and Secrets

Using Environment Variables

```
# .github/workflows/environment.yml
name: Environment Variables

on: [push]
```

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

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      DEPLOY_ENV: staging

    steps:
      - uses: actions/checkout@v4

      - name: Print environment
        run: |
          echo "Node environment: $NODE_ENV"
          echo "API URL: $API_URL"
          echo "Deploy environment: $DEPLOY_ENV"
          echo "GitHub ref: $GITHUB_REF"
          echo "GitHub actor: $GITHUB_ACTOR"

      - name: Set dynamic environment
        run: |
          if [[ "$GITHUB_REF" == "refs/heads/main" ]]; then
            echo "ENVIRONMENT=production" >> $GITHUB_ENV
          else
            echo "ENVIRONMENT=development" >> $GITHUB_ENV
          fi

      - name: Use dynamic environment
        run: echo "Deploying to $ENVIRONMENT"
```

Managing Secrets

```yaml
# .github/workflows/secrets.yml
name: Using Secrets

on: [push]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      # Using repository secrets
      - name: Deploy to server
        env:
          SSH_PRIVATE_KEY: ${{ secrets.SSH_PRIVATE_KEY }}
```

```
          SERVER_HOST: ${{ secrets.SERVER_HOST }}
          DATABASE_URL: ${{ secrets.DATABASE_URL }}
        run: |
          echo "Deploying with secrets..."
          # Never echo secrets directly!
          echo "Host: $SERVER_HOST"

      # Using secrets in actions
      - name: Deploy to AWS
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1
```

Common Actions and Use Cases

Code Quality Checks

```
# .github/workflows/quality.yml
name: Code Quality

on: [push, pull_request]

jobs:
  quality:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0 # Full history for better analysis

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

      - name: Install dependencies
        run: npm ci

      # ESLint with annotations
      - name: Run ESLint
        run: |
          npx eslint . --format=@microsoft/eslint-formatter-sarif --output-file
eslint-results.sarif
        continue-on-error: true

      - name: Upload ESLint results
        uses: github/codeql-action/upload-sarif@v3
```

```
        with:
          sarif_file: eslint-results.sarif
        if: always()

      # Security audit
      - name: Run security audit
        run: npm audit --audit-level=high

      # Check for outdated dependencies
      - name: Check outdated packages
        run: npm outdated || true

      # License check
      - name: Check licenses
        run: |
          npx license-checker --summary
```

Testing with Coverage

```
# .github/workflows/test-coverage.yml
name: Test Coverage

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - 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

      # Run tests with coverage
      - name: Run tests with coverage
        run: |
          npm test -- --coverage
          # For Jest: npm test -- --coverage --coverageReporters=lcov

      # Upload coverage to Codecov
      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage/lcov.info
```

```
          flags: unittests
          name: codecov-umbrella

      # Comment coverage on PR
      - name: Coverage comment
        if: github.event_name == 'pull_request'
        uses: romeovs/lcov-reporter-action@v0.3.1
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
          lcov-file: ./coverage/lcov.info
```

Docker Build and Push

```
# .github/workflows/docker.yml
name: Docker Build

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

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

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

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

      # Setup Docker Buildx
      - name: Setup Docker Buildx
        uses: docker/setup-buildx-action@v3

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

      # Extract metadata
      - 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}}

      # Build and push
      - name: Build and push
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Deployment Workflow

```
# .github/workflows/deploy.yml
name: Deploy Application

on:
  push:
    branches: [main]
  workflow_dispatch:
    inputs:
      environment:
        description: "Environment to deploy to"
        required: true
        default: "staging"
        type: choice
        options:
          - staging
          - production

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: ${{ github.event.inputs.environment || 'staging' }}

    steps:
      - 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: Build application
        run: npm run build
        env:
          NODE_ENV: production

      # Deploy to staging
      - name: Deploy to staging
        if: github.event.inputs.environment == 'staging' || github.ref ==
'refs/heads/main'
        run: |
          echo "Deploying to staging environment"
          # Add your staging deployment commands here

      # Deploy to production (with approval)
      - name: Deploy to production
        if: github.event.inputs.environment == 'production'
        run: |
          echo "Deploying to production environment"
          # Add your production deployment commands here

      # Notify deployment
      - name: Notify deployment
        if: always()
        uses: 8398a7/action-slack@v3
        with:
          status: ${{ job.status }}
          channel: "#deployments"
          webhook_url: ${{ secrets.SLACK_WEBHOOK }}
```

Advanced Features

Matrix Builds

```
# .github/workflows/matrix.yml
name: Matrix Build

on: [push, pull_request]

jobs:
  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
              experimental: true
          exclude:
            - os: windows-latest
              node-version: 16
        fail-fast: false

      steps:
        - uses: actions/checkout@v4

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

        - name: Install dependencies
          run: npm ci

        - name: Run tests
          run: npm test
          continue-on-error: ${{ matrix.experimental || false }}
```

Reusable Workflows

```
# .github/workflows/reusable-ci.yml
name: Reusable CI

on:
  workflow_call:
    inputs:
      node-version:
        required: false
        type: string
        default: "18"
      environment:
        required: false
        type: string
        default: "development"
    secrets:
      NPM_TOKEN:
        required: false
    outputs:
      build-status:
        description: "Build status"
        value: ${{ jobs.ci.outputs.status }}

jobs:
  ci:
    runs-on: ubuntu-latest
    outputs:
```

```
        status: ${{ steps.build.outcome }}

    steps:
      - uses: actions/checkout@v4

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

      - name: Install dependencies
        run: npm ci
        env:
          NPM_TOKEN: ${{ secrets.NPM_TOKEN }}

      - name: Run tests
        run: npm test

      - name: Build
        id: build
        run: npm run build
        env:
          NODE_ENV: ${{ inputs.environment }}
```

```
# .github/workflows/use-reusable.yml
name: Use Reusable Workflow

on: [push]

jobs:
  call-reusable:
    uses: ./.github/workflows/reusable-ci.yml
    with:
      node-version: "20"
      environment: "production"
    secrets:
      NPM_TOKEN: ${{ secrets.NPM_TOKEN }}

  deploy:
    needs: call-reusable
    runs-on: ubuntu-latest
    if: needs.call-reusable.outputs.build-status == 'success'

    steps:
      - name: Deploy
        run: echo "Deploying after successful build"
```

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: false
    default: "18"
  cache-dependency-path:
    description: "Path to package-lock.json"
    required: false
    default: "package-lock.json"
outputs:
  cache-hit:
    description: "Whether cache was hit"
    value: ${{ steps.cache.outputs.cache-hit }}

runs:
  using: "composite"
  steps:
    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: ${{ inputs.node-version }}

    - name: Cache dependencies
      id: cache
      uses: actions/cache@v3
      with:
        path: ~/.npm
        key: ${{ runner.os }}-node-${{ hashFiles(inputs.cache-dependency-path) }}
        restore-keys: |
          ${{ runner.os }}-node-

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

Monitoring and Debugging

Workflow Status Checks

```yaml
# .github/workflows/status-check.yml
name: Status Check

on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
```

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

      - name: Check workflow status
        run: |
          echo "Workflow: $GITHUB_WORKFLOW"
          echo "Run ID: $GITHUB_RUN_ID"
          echo "Run Number: $GITHUB_RUN_NUMBER"
          echo "Actor: $GITHUB_ACTOR"
          echo "Repository: $GITHUB_REPOSITORY"
          echo "Event: $GITHUB_EVENT_NAME"
          echo "Ref: $GITHUB_REF"
          echo "SHA: $GITHUB_SHA"

      - name: Debug context
        env:
          GITHUB_CONTEXT: ${{ toJson(github) }}
        run: echo "$GITHUB_CONTEXT"

      - name: Check previous jobs
        if: always()
        run: |
          echo "Job status: ${{ job.status }}"
          echo "Steps context: ${{ toJson(steps) }}"
```

Error Handling

```
# .github/workflows/error-handling.yml
name: Error Handling

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      # Continue on error
      - name: Run tests (continue on error)
        run: npm test
        continue-on-error: true

      # Conditional step based on previous step
      - name: Handle test failure
        if: failure()
        run: |
          echo "Tests failed, but continuing..."
          # Send notification, create issue, etc.
```

```
      # Always run cleanup
      - name: Cleanup
        if: always()
        run: |
          echo "Cleaning up..."
          # Cleanup commands here

      # Timeout for long-running steps
      - name: Long running task
        run: |
          echo "Starting long task..."
          sleep 300  # 5 minutes
        timeout-minutes: 10
```

Best Practices

1. Workflow Organization

```
# Organize workflows by purpose
.github/workflows/
├── ci.yml              # Continuous Integration
├── cd.yml              # Continuous Deployment
├── security.yml        # Security scans
├── release.yml         # Release automation
└── maintenance.yml     # Scheduled maintenance
```

2. Efficient Caching

```
# Efficient dependency caching
- name: Cache dependencies
  uses: actions/cache@v3
  with:
    path: |
      ~/.npm
      ~/.cache
    key: ${{ runner.os }}-deps-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-deps-

# Cache build outputs
- name: Cache build
  uses: actions/cache@v3
  with:
    path: dist/
    key: ${{ runner.os }}-build-${{ github.sha }}
```

3. Security Best Practices

```
# Minimal permissions
permissions:
  contents: read
  pull-requests: write

# Pin action versions
- uses: actions/checkout@v4.1.1  # Pin to specific version
- uses: actions/setup-node@v4.0.0

# Use secrets properly
env:
  API_KEY: ${{ secrets.API_KEY }}  # Good
  # Never: API_KEY: "hardcoded-key"  # Bad

# Validate inputs
- name: Validate input
  run: |
    if [[ ! "${{ github.event.inputs.environment }}" =~ ^(staging|production)$ ]];
then
      echo "Invalid environment"
      exit 1
    fi
```

4. Performance Optimization

```
# Use appropriate runners
runs-on: ubuntu-latest  # Fastest for most tasks
# runs-on: windows-latest  # Only when needed
# runs-on: macos-latest    # Only when needed

# Parallel jobs
jobs:
  lint:
    runs-on: ubuntu-latest
    # ...

  test:
    runs-on: ubuntu-latest
    # ... runs in parallel with lint

  build:
    needs: [lint, test]  # Runs after both complete
    runs-on: ubuntu-latest
    # ...

# Conditional execution
- name: Skip on draft PR
```

```
    if: github.event.pull_request.draft == false
    run: npm test
```

Troubleshooting

Common Issues

1. **Workflow not triggering**

```
# Check file location: .github/workflows/
# Check YAML syntax
# Check branch protection rules
# Check if workflow is disabled
```

2. **Permission denied errors**

```
# Add necessary permissions
permissions:
  contents: read
  packages: write
  pull-requests: write
```

3. **Cache issues**

```
# Clear cache via GitHub UI or API
# Update cache key
# Check cache path
```

4. **Secret not found**

```
# Check secret name spelling
# Verify secret is set in repository/organization
# Check environment restrictions
```

Quick Reference

```
# Basic workflow structure
name: Workflow Name
on: [push, pull_request]
jobs:
  job-name:
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v4
      - name: Step name
        run: command

# Common triggers
on:
  push: { branches: [main] }
  pull_request: { branches: [main] }
  schedule: [{ cron: '0 2 * * *' }]
  workflow_dispatch:

# Environment variables
env:
  NODE_ENV: production
  API_URL: ${{ secrets.API_URL }}

# Conditional execution
if: github.ref == 'refs/heads/main'
if: github.event_name == 'pull_request'
if: success() && github.ref == 'refs/heads/main'

# Matrix strategy
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node: [16, 18, 20]

# Common actions
- uses: actions/checkout@v4
- uses: actions/setup-node@v4
- uses: actions/cache@v3
- uses: actions/upload-artifact@v4
```

Previous: GitHub CLI
Next: Clean Git History

Maintaining Clean Git History in Teams

A clean Git history is crucial for team collaboration, debugging, and project maintenance. It makes code reviews easier, simplifies troubleshooting, and provides clear documentation of project evolution.

Why Clean Git History Matters

Benefits

- **Easier debugging**: Clear commit messages help identify when bugs were introduced
- **Better code reviews**: Logical commits make reviews more focused
- **Simplified releases**: Clean history makes it easier to generate changelogs

- **Improved collaboration**: Team members can understand changes quickly
- **Easier rollbacks**: Atomic commits allow precise rollbacks
- **Better documentation**: History serves as project documentation

Problems with Messy History

```
# Example of messy history
* fix typo
* oops
* work in progress
* more fixes
* actually fix the bug
* temp commit
* final fix
```

Principles of Clean Git History

1. Atomic Commits

Each commit should represent one logical change:

```
# Good: Atomic commits
git commit -m "feat(auth): add user login functionality"
git commit -m "test(auth): add login integration tests"
git commit -m "docs(auth): update authentication guide"

# Bad: Mixed changes
git commit -m "add login, fix tests, update docs, refactor utils"
```

2. Meaningful Commit Messages

Follow conventional commit format:

```
# Good commit messages
feat(api): add user profile endpoint
fix(auth): resolve token expiration issue
docs: update installation instructions
refactor(db): optimize user queries
test(payment): add credit card validation tests

# Bad commit messages
fix stuff
update
work in progress
temp
```

3. Logical Commit Order

Commits should tell a story:

```
# Good: Logical progression
1. feat(user): add user model
2. feat(user): add user controller
3. feat(user): add user routes
4. test(user): add user API tests
5. docs(user): add user API documentation

# Bad: Random order
1. fix typo in docs
2. add user model
3. temp commit
4. add tests
5. add controller
```

Team Workflow Strategies

Feature Branch Workflow

```
# 1. Create feature branch from main
git checkout main
git pull origin main
git checkout -b feature/user-authentication

# 2. Work in small, logical commits
echo "User model implementation" > user.js
git add user.js
git commit -m "feat(auth): add user model with validation"

echo "Password hashing logic" >> user.js
git add user.js
git commit -m "feat(auth): implement password hashing"

echo "Login endpoint" > auth.js
git add auth.js
git commit -m "feat(auth): add login endpoint"

# 3. Push feature branch
git push -u origin feature/user-authentication

# 4. Create pull request
gh pr create --title "Add user authentication system" \
  --body "Implements complete user authentication with login/logout"

# 5. Clean up before merge (if needed)
git rebase -i main  # Interactive rebase to clean commits
```

```
# 6. Merge with clean history
gh pr merge --squash  # or --rebase depending on team preference
```

Git Flow with Clean History

```
# Initialize git flow
git flow init

# Start feature
git flow feature start user-dashboard

# Work with clean commits
git commit -m "feat(dashboard): add user stats component"
git commit -m "feat(dashboard): add activity timeline"
git commit -m "style(dashboard): improve responsive layout"
git commit -m "test(dashboard): add component unit tests"

# Finish feature (creates merge commit)
git flow feature finish user-dashboard

# Start release
git flow release start 1.2.0

# Prepare release
echo "1.2.0" > VERSION
git add VERSION
git commit -m "chore(release): bump version to 1.2.0"

# Update changelog
echo "## v1.2.0\n- Add user dashboard\n- Improve authentication" > CHANGELOG.md
git add CHANGELOG.md
git commit -m "docs(release): update changelog for v1.2.0"

# Finish release
git flow release finish 1.2.0
```

Commit Organization Techniques

Interactive Rebase for History Cleanup

```
# Example: Clean up feature branch before merge
git log --oneline
# a1b2c3d feat(auth): add login endpoint
# e4f5g6h fix typo
# i7j8k9l feat(auth): add password hashing
# m1n2o3p work in progress
# q4r5s6t feat(auth): add user model

# Interactive rebase to clean up
```

```
git rebase -i HEAD~5

# In the editor:
pick q4r5s6t feat(auth): add user model
squash m1n2o3p work in progress
pick i7j8k9l feat(auth): add password hashing
pick a1b2c3d feat(auth): add login endpoint
fixup e4f5g6h fix typo

# Result: Clean, logical commits
# q4r5s6t feat(auth): add user model
# i7j8k9l feat(auth): add password hashing
# a1b2c3d feat(auth): add login endpoint
```

Squashing Related Commits

```
# Before squashing
git log --oneline
# abc123 fix: resolve linting errors
# def456 feat: add user validation
# ghi789 fix: handle edge case in validation
# jkl012 feat: add user registration form

# Squash validation-related commits
git rebase -i HEAD~4

# In editor:
pick jkl012 feat: add user registration form
pick def456 feat: add user validation
squash ghi789 fix: handle edge case in validation
squash abc123 fix: resolve linting errors

# Edit commit message:
# feat: add user registration with validation
#
# - Add registration form component
# - Implement comprehensive user validation
# - Handle edge cases and linting issues
```

Splitting Large Commits

```
# If you have a large commit that should be split
git log --oneline
# abc123 feat: add user management (too large)

# Reset to split the commit
git reset HEAD~1

# Stage and commit parts separately
```

```
git add user-model.js
git commit -m "feat(user): add user model"

git add user-controller.js
git commit -m "feat(user): add user controller"

git add user-routes.js
git commit -m "feat(user): add user routes"

git add user-tests.js
git commit -m "test(user): add user API tests"
```

Merge Strategies for Clean History

Merge Commit Strategy

```
# Preserves branch history
git checkout main
git merge feature/user-auth

# Creates merge commit:
# *   Merge branch 'feature/user-auth'
# |\
# | * feat(auth): add login tests
# | * feat(auth): add login endpoint
# | * feat(auth): add user model
# |/
# * Previous main commit

# Good for: Complex features, preserving context
# Bad for: Simple changes, linear history preference
```

Squash and Merge

```
# Combines all feature commits into one
gh pr merge --squash

# Results in:
# * feat(auth): add user authentication system
# * Previous main commit

# Good for: Simple features, clean linear history
# Bad for: Losing detailed commit history
```

Rebase and Merge

```
# Replays commits on top of main
git checkout feature/user-auth
git rebase main
git checkout main
git merge feature/user-auth  # Fast-forward merge

# Results in linear history:
# * feat(auth): add login tests
# * feat(auth): add login endpoint
# * feat(auth): add user model
# * Previous main commit

# Good for: Linear history, preserving individual commits
# Bad for: Losing branch context
```

Team Guidelines and Conventions

Commit Message Standards

```
# Team commit message template
cat > .gitmessage << EOF
# <type>(<scope>): <subject>
#
# <body>
#
# <footer>
#
# Type: feat, fix, docs, style, refactor, test, chore
# Scope: component or file being modified
# Subject: imperative, present tense, no period
# Body: explain what and why vs. how
# Footer: breaking changes, issue references
EOF

# Configure git to use template
git config commit.template .gitmessage
```

Branch Naming Conventions

```
# Feature branches
feature/user-authentication
feature/payment-integration
feature/admin-dashboard

# Bug fix branches
fix/login-validation-error
fix/memory-leak-in-parser
hotfix/critical-security-patch
```

```
# Maintenance branches
chore/update-dependencies
chore/improve-test-coverage
refactor/database-optimization

# Documentation branches
docs/api-documentation
docs/installation-guide
docs/contributing-guidelines
```

Pull Request Guidelines

```
<!-- .github/pull_request_template.md -->

## Description

Brief description of changes

## Type of Change

- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing functionality to
not work as expected)
- [ ] Documentation update

## How Has This Been Tested?

- [ ] Unit tests
- [ ] Integration tests
- [ ] Manual testing

## Checklist

- [ ] My code follows the style guidelines
- [ ] I have performed a self-review
- [ ] I have commented my code, particularly in hard-to-understand areas
- [ ] I have made corresponding changes to the documentation
- [ ] My changes generate no new warnings
- [ ] I have added tests that prove my fix is effective or that my feature works
- [ ] New and existing unit tests pass locally
- [ ] Any dependent changes have been merged and published

## Related Issues

Closes #123
References #456
```

Automated History Management

Git Hooks for Quality Control

```
# Pre-commit hook for commit message validation
cat > .git/hooks/commit-msg << 'EOF'
#!/bin/bash

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 "Example: feat(auth): add login functionality"
    exit 1
fi
EOF

chmod +x .git/hooks/commit-msg
```

Automated Changelog Generation

```
# Install conventional-changelog
npm install -g conventional-changelog-cli

# Generate changelog
conventional-changelog -p angular -i CHANGELOG.md -s

# Add to package.json scripts
"scripts": {
  "changelog": "conventional-changelog -p angular -i CHANGELOG.md -s -r 0",
  "release": "npm run changelog && git add CHANGELOG.md && git commit -m 'docs:
update changelog'"
}
```

GitHub Actions for History Validation

```
# .github/workflows/history-check.yml
name: History Check

on:
  pull_request:
    branches: [main]

jobs:
  check-commits:
    runs-on: ubuntu-latest

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

```
        with:
          fetch-depth: 0

      - name: Check commit messages
        run: |
          # Check all commits in PR
          commits=$(git rev-list --no-merges origin/main..HEAD)

          for commit in $commits; do
            message=$(git log --format=%s -n 1 $commit)
            if ! echo "$message" | grep -qE
'^(feat|fix|docs|style|refactor|test|chore)(\(.+\))?: .{1,50}'; then
              echo "❌ Invalid commit message: $message"
              echo "Commit: $commit"
              exit 1
            fi
          done

          echo "✅ All commit messages are valid"

      - name: Check for merge commits
        run: |
          merge_commits=$(git rev-list --merges origin/main..HEAD)
          if [ -n "$merge_commits" ]; then
            echo "❌ Merge commits found in feature branch"
            echo "Please rebase your branch"
            exit 1
          fi

          echo "✅ No merge commits found"
```

Handling Complex Scenarios

Collaborative Feature Development

```
# Multiple developers working on same feature

# Developer A starts feature
git checkout -b feature/user-dashboard
git commit -m "feat(dashboard): add basic layout"
git push -u origin feature/user-dashboard

# Developer B joins
git checkout feature/user-dashboard
git pull origin feature/user-dashboard
git commit -m "feat(dashboard): add user stats widget"
git push origin feature/user-dashboard

# Developer A continues
git pull origin feature/user-dashboard  # Get B's changes
git commit -m "feat(dashboard): add activity timeline"
```

```
# Before pushing, rebase to maintain clean history
git pull --rebase origin feature/user-dashboard
git push origin feature/user-dashboard

# Final cleanup before merge
git rebase -i origin/main
# Squash related commits, fix commit messages
```

Hotfix Workflow

```
# Critical bug in production
git checkout main
git pull origin main
git checkout -b hotfix/critical-security-fix

# Make minimal fix
git commit -m "fix(security): patch XSS vulnerability in user input

Resolves critical security issue where user input was not properly
sanitized, allowing potential XSS attacks.

Fixes #SECURITY-123"

# Test thoroughly
npm test
npm run security-audit

# Fast-track review and merge
gh pr create --title "HOTFIX: Critical security patch" \
  --body "Critical security fix for XSS vulnerability" \
  --label "security,hotfix,priority-critical"

# Merge immediately after review
gh pr merge --squash

# Tag for tracking
git checkout main
git pull origin main
git tag v1.2.1-hotfix
git push origin v1.2.1-hotfix
```

Release Branch Management

```
# Prepare release branch
git checkout main
git pull origin main
git checkout -b release/v2.0.0
```

```
# Version bump and changelog
echo "2.0.0" > VERSION
npm run changelog
git add VERSION CHANGELOG.md
git commit -m "chore(release): prepare version 2.0.0"

# Bug fixes during release preparation
git commit -m "fix(release): resolve build issue in production"
git commit -m "docs(release): update migration guide"

# Merge back to develop
git checkout develop
git merge release/v2.0.0

# Merge to main and tag
git checkout main
git merge release/v2.0.0
git tag v2.0.0
git push origin main develop v2.0.0

# Clean up release branch
git branch -d release/v2.0.0
git push origin --delete release/v2.0.0
```

Tools and Automation

Commitizen for Consistent Messages

```
# Install commitizen
npm install -g commitizen cz-conventional-changelog

# Configure
echo '{ "path": "cz-conventional-changelog" }' > ~/.czrc

# Use instead of git commit
git add .
git cz

# Interactive prompts:
# ? Select the type of change: feat
# ? What is the scope of this change: auth
# ? Write a short description: add login functionality
# ? Provide a longer description: (optional)
# ? Are there any breaking changes: No
# ? Does this change affect any open issues: Yes
# ? Add issue references: Closes #123
```

Semantic Release

```
# Install semantic-release
npm install --save-dev semantic-release

# Configure .releaserc.json
cat > .releaserc.json << EOF
{
  "branches": ["main"],
  "plugins": [
    "@semantic-release/commit-analyzer",
    "@semantic-release/release-notes-generator",
    "@semantic-release/changelog",
    "@semantic-release/npm",
    "@semantic-release/github"
  ]
}
EOF

# Add to CI/CD pipeline
# Automatically creates releases based on commit history
```

Git Aliases for Clean History

```
# Useful aliases for maintaining clean history
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.cleanup "!git branch --merged | grep -v
'\*\|main\|develop' | xargs -n 1 git branch -d"

git config --global alias.squash "!f() { git reset --soft HEAD~$1 && git commit --
edit -m\"$(git log --format=%B --reverse HEAD..HEAD@{1})\"; }; f"

git config --global alias.fixup "commit --fixup"

git config --global alias.autosquash "rebase -i --autosquash"

# Usage examples
git lg                    # Pretty log
git cleanup              # Remove merged branches
git squash 3             # Squash last 3 commits
git fixup abc123         # Create fixup commit
git autosquash main      # Auto-squash fixup commits
```

Best Practices Summary

Do's

- ✅ **Write atomic commits** - One logical change per commit
- ✅ **Use conventional commit format** - Consistent, parseable messages

- ✅ **Rebase feature branches** - Keep history linear and clean
- ✅ **Review before merging** - Ensure quality and consistency
- ✅ **Use meaningful branch names** - Clear purpose and scope
- ✅ **Clean up merged branches** - Reduce repository clutter
- ✅ **Document breaking changes** - Clear migration paths
- ✅ **Test before committing** - Ensure each commit is functional

Don'ts

- ❌ **Don't commit work-in-progress** - Use stash or draft commits
- ❌ **Don't mix unrelated changes** - Keep commits focused
- ❌ **Don't force push to shared branches** - Use --force-with-lease
- ❌ **Don't ignore commit message format** - Consistency matters
- ❌ **Don't merge without review** - Quality control is essential
- ❌ **Don't leave broken commits** - Each commit should be functional
- ❌ **Don't commit secrets or sensitive data** - Use .gitignore
- ❌ **Don't rewrite public history** - Respect shared commits

Quick Reference

```
# Clean history commands
git rebase -i HEAD~n              # Interactive rebase last n commits
git commit --amend                # Modify last commit
git reset --soft HEAD~1           # Undo last commit, keep changes
git cherry-pick abc123            # Apply specific commit
git revert abc123                 # Create commit that undoes changes

# Branch management
git checkout -b feature/name      # Create feature branch
git rebase main                   # Rebase current branch on main
git merge --no-ff feature/name    # Merge with merge commit
git merge --squash feature/name   # Squash merge

# History inspection
git log --oneline                 # Compact log
git log --graph                   # Visual branch structure
git show abc123                   # Show specific commit
git diff main..feature/name       # Compare branches

# Cleanup
git branch -d feature/name        # Delete merged branch
git remote prune origin           # Clean up remote references
git gc                            # Garbage collection
```

Previous: GitHub Actions Basics
Next: Branching Strategies

Branching Strategies: Git Flow vs Trunk-Based Development

Choosing the right branching strategy is crucial for team productivity, code quality, and release management. This guide compares popular branching strategies and helps you choose the best approach for your team.

Overview of Branching Strategies

What is a Branching Strategy?

A branching strategy defines:

- How branches are created and named
- When branches are merged
- Who can merge to which branches
- How releases are managed
- How hotfixes are handled

Popular Strategies

1. **Git Flow** - Feature-rich, structured approach
2. **GitHub Flow** - Simple, continuous deployment
3. **GitLab Flow** - Hybrid approach with environment branches
4. **Trunk-Based Development** - Minimal branching, frequent integration
5. **Release Flow** - Microsoft's approach for large teams

Git Flow

Overview

Git Flow is a branching model that uses multiple long-lived branches and specific branch types for different purposes.

Branch Structure

```
main (production-ready code)
├── develop (integration branch)
│   ├── feature/user-auth
│   ├── feature/payment-system
│   └── feature/admin-dashboard
├── release/v1.2.0 (release preparation)
└── hotfix/critical-bug (emergency fixes)
```

Branch Types

Main Branches

```
# main: Production-ready code
# Always deployable, tagged with version numbers
git checkout main
git tag v1.1.0

# develop: Integration branch
# Latest development changes for next release
git checkout develop
```

Supporting Branches

```
# Feature branches: New features
git checkout develop
git checkout -b feature/user-authentication
# Work on feature...
git checkout develop
git merge --no-ff feature/user-authentication
git branch -d feature/user-authentication

# Release branches: Prepare releases
git checkout develop
git checkout -b release/v1.2.0
# Bug fixes, version bumps, documentation
git checkout main
git merge --no-ff release/v1.2.0
git tag v1.2.0
git checkout develop
git merge --no-ff release/v1.2.0
git branch -d release/v1.2.0

# Hotfix branches: Emergency fixes
git checkout main
git checkout -b hotfix/critical-security-fix
# Make fix...
git checkout main
git merge --no-ff hotfix/critical-security-fix
git tag v1.2.1
git checkout develop
git merge --no-ff hotfix/critical-security-fix
git branch -d hotfix/critical-security-fix
```

Git Flow Setup

```
# Install git-flow (if not already installed)
# On macOS: brew install git-flow
# On Ubuntu: sudo apt-get install git-flow
# On Windows: Download from GitHub
```

```
# Initialize git flow in repository
git flow init

# Follow prompts (or use defaults):
# Branch name for production releases: [main]
# Branch name for "next release" development: [develop]
# Feature branches prefix: [feature/]
# Release branches prefix: [release/]
# Hotfix branches prefix: [hotfix/]
# Support branches prefix: [support/]
# Version tag prefix: []
```

Git Flow Workflow Example

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

# Work on feature
echo "Dashboard component" > dashboard.js
git add dashboard.js
git commit -m "feat(dashboard): add user dashboard component"

echo "Dashboard styles" > dashboard.css
git add dashboard.css
git commit -m "style(dashboard): add dashboard styling"

echo "Dashboard tests" > dashboard.test.js
git add dashboard.test.js
git commit -m "test(dashboard): add dashboard component tests"

# Finish feature (merges to develop)
git flow feature finish user-dashboard

# Start release
git flow release start 1.2.0

# Prepare release
echo "1.2.0" > VERSION
git add VERSION
git commit -m "chore(release): bump version to 1.2.0"

# Update changelog
cat > CHANGELOG.md << EOF
## v1.2.0 ($(date +%Y-%m-%d))

### Features
- Add user dashboard with statistics
- Improve user authentication flow

### Bug Fixes
```

```
- Fix memory leak in data processing
- Resolve CSS layout issues on mobile
EOF

git add CHANGELOG.md
git commit -m "docs(release): update changelog for v1.2.0"

# Finish release (merges to main and develop, creates tag)
git flow release finish 1.2.0

# Handle hotfix
git flow hotfix start critical-fix

# Make fix
echo "Security patch" > security.js
git add security.js
git commit -m "fix(security): patch XSS vulnerability"

# Finish hotfix (merges to main and develop, creates tag)
git flow hotfix finish critical-fix
```

Git Flow Pros and Cons

Pros ✅

- **Clear structure**: Well-defined branch purposes
- **Parallel development**: Multiple features can be developed simultaneously
- **Release management**: Dedicated release preparation
- **Hotfix support**: Quick fixes without disrupting development
- **Stable main**: Production branch is always stable
- **Tool support**: Many tools support Git Flow

Cons ❌

- **Complexity**: Many branches to manage
- **Merge overhead**: Frequent merging required
- **Delayed integration**: Features integrated late
- **Conflicts**: More merge conflicts due to delayed integration
- **Overhead**: Not suitable for continuous deployment
- **Learning curve**: Team needs to understand the model

When to Use Git Flow

✅ Good for:

- Large teams with multiple features in parallel
- Scheduled releases (not continuous deployment)
- Products requiring stable releases
- Teams comfortable with complex branching
- Projects with long development cycles

✗ Not good for:

- Small teams (< 5 developers)
- Continuous deployment workflows
- Simple projects
- Teams new to Git
- Fast-moving startups

GitHub Flow

Overview

GitHub Flow is a lightweight, branch-based workflow designed for continuous deployment.

Workflow

```
# 1. Create branch from main
git checkout main
git pull origin main
git checkout -b feature/add-search-functionality

# 2. Make changes and commit
echo "Search component" > search.js
git add search.js
git commit -m "feat: add search functionality"

echo "Search tests" > search.test.js
git add search.test.js
git commit -m "test: add search component tests"

# 3. Push branch and create pull request
git push -u origin feature/add-search-functionality
gh pr create --title "Add search functionality" \
  --body "Implements user search with filters and pagination"

# 4. Review and discuss
# Team reviews code, suggests changes

# 5. Deploy for testing (optional)
# Deploy branch to staging environment

# 6. Merge to main
gh pr merge --squash

# 7. Deploy main
# Automatic deployment to production
```

GitHub Flow Rules

1. **Main is always deployable**

2. **Create descriptive branch names**
3. **Commit early and often**
4. **Open pull request early**
5. **Deploy for testing**
6. **Merge after review**

GitHub Flow Example

```
# Feature development
git checkout main
git pull origin main
git checkout -b feature/user-notifications

# Implement feature incrementally
git commit -m "feat(notifications): add notification model"
git push origin feature/user-notifications

# Open draft PR early for feedback
gh pr create --draft --title "WIP: User notifications system"

git commit -m "feat(notifications): add notification service"
git push origin feature/user-notifications

git commit -m "feat(notifications): add notification UI components"
git push origin feature/user-notifications

git commit -m "test(notifications): add comprehensive tests"
git push origin feature/user-notifications

# Mark PR as ready for review
gh pr ready

# After review and approval
gh pr merge --squash

# Clean up
git checkout main
git pull origin main
git branch -d feature/user-notifications
```

GitHub Flow Pros and Cons

Pros ✅

- **Simple**: Easy to understand and implement
- **Fast**: Quick feature delivery
- **Continuous deployment**: Perfect for CD workflows
- **Collaboration**: Early feedback through PRs
- **Flexible**: Adaptable to different team sizes

- **Low overhead**: Minimal branch management

Cons ❌

- **No release branches**: Difficult for scheduled releases
- **Main instability**: Risk of breaking main branch
- **No hotfix process**: Emergency fixes follow same process
- **Limited for complex releases**: Not suitable for complex release management

When to Use GitHub Flow

✅ Good for:

- Continuous deployment
- Small to medium teams
- Web applications
- SaaS products
- Agile development
- Simple release cycles

❌ Not good for:

- Scheduled releases
- Multiple production versions
- Complex release processes
- Large enterprise teams

Trunk-Based Development

Overview

Trunk-based development focuses on keeping branches short-lived and integrating changes frequently into the main branch (trunk).

Core Principles

1. **Short-lived branches** (< 2 days)
2. **Frequent integration** (multiple times per day)
3. **Feature flags** for incomplete features
4. **Continuous integration**
5. **Automated testing**

Workflow

```
# 1. Create short-lived branch
git checkout main
git pull origin main
git checkout -b add-user-validation

# 2. Make small, focused changes
```

```
echo "Basic validation" > validation.js
git add validation.js
git commit -m "feat: add basic user validation"

# 3. Push and create PR immediately
git push -u origin add-user-validation
gh pr create --title "Add user validation" \
  --body "Small focused change for user input validation"

# 4. Quick review and merge (same day)
gh pr merge --squash

# 5. Delete branch immediately
git checkout main
git pull origin main
git branch -d add-user-validation

# 6. Repeat for next small change
git checkout -b improve-validation-messages
# Continue with next small improvement...
```

Feature Flags Implementation

```
// Feature flag for incomplete features
const featureFlags = {
  newUserDashboard: process.env.ENABLE_NEW_DASHBOARD === "true",
  advancedSearch: process.env.ENABLE_ADVANCED_SEARCH === "true",
};

// Use feature flags in code
function renderDashboard() {
  if (featureFlags.newUserDashboard) {
    return <NewDashboard />;
  }
  return <LegacyDashboard />;
}

// Gradual rollout
function shouldShowNewFeature(userId) {
  // Show to 10% of users
  return userId % 10 === 0;
}
```

Branch by Abstraction

```
// Old implementation
class LegacyPaymentProcessor {
  processPayment(amount) {
    // Legacy payment logic
```

```
  }
}

// New implementation (developed incrementally)
class NewPaymentProcessor {
  processPayment(amount) {
    // New payment logic
  }
}

// Abstraction layer
class PaymentService {
  constructor() {
    this.processor = featureFlags.newPaymentSystem
      ? new NewPaymentProcessor()
      : new LegacyPaymentProcessor();
  }

  processPayment(amount) {
    return this.processor.processPayment(amount);
  }
}
```

Trunk-Based Development Example

```
# Day 1: Start new feature with feature flag
git checkout main
git pull origin main
git checkout -b feature-flag-setup

# Add feature flag infrastructure
cat > feature-flags.js << EOF
module.exports = {
  newCheckoutFlow: process.env.ENABLE_NEW_CHECKOUT === 'true'
};
EOF

git add feature-flags.js
git commit -m "feat: add feature flag infrastructure"
git push -u origin feature-flag-setup
gh pr create --title "Add feature flag infrastructure"
gh pr merge --squash

# Day 1: Add basic checkout component (behind flag)
git checkout main
git pull origin main
git checkout -b checkout-component-basic

cat > checkout.js << EOF
const { newCheckoutFlow } = require('./feature-flags');
```

```
function renderCheckout() {
  if (newCheckoutFlow) {
    return '<div>New Checkout (Basic)</div>';
  }
  return '<div>Legacy Checkout</div>';
}
EOF

git add checkout.js
git commit -m "feat: add basic new checkout component (behind flag)"
git push -u origin checkout-component-basic
gh pr create --title "Add basic checkout component"
gh pr merge --squash

# Day 2: Enhance checkout component
git checkout main
git pull origin main
git checkout -b checkout-payment-integration

# Enhance the component
sed -i 's/Basic/with Payment Integration/' checkout.js
git add checkout.js
git commit -m "feat: add payment integration to new checkout"
git push -u origin checkout-payment-integration
gh pr create --title "Add payment integration to checkout"
gh pr merge --squash

# Day 3: Enable feature for testing
git checkout main
git pull origin main
git checkout -b enable-checkout-testing

# Update documentation
echo "Set ENABLE_NEW_CHECKOUT=true for testing" > README.md
git add README.md
git commit -m "docs: add instructions for testing new checkout"
git push -u origin enable-checkout-testing
gh pr create --title "Add testing instructions for new checkout"
gh pr merge --squash

# After testing: Enable for all users
# Remove feature flag in production deployment
```

Trunk-Based Development Pros and Cons

Pros ✅

- **Fast integration**: Immediate feedback on changes
- **Reduced conflicts**: Fewer merge conflicts
- **Continuous delivery**: Always ready to deploy
- **Simple branching**: Minimal branch management

- **Team collaboration**: Shared codebase visibility
- **Quality focus**: Emphasis on automated testing

Cons ❌

- **Requires discipline**: Team must commit to practices
- **Feature flag complexity**: Managing feature flags
- **Incomplete features**: Risk of shipping incomplete work
- **Testing overhead**: Comprehensive automated testing required
- **Cultural change**: Significant workflow change

When to Use Trunk-Based Development

✅ Good for:

- High-performing teams
- Continuous deployment
- Microservices architecture
- Teams with strong testing culture
- Fast-moving products
- Experienced developers

❌ Not good for:

- Teams new to CI/CD
- Complex release processes
- Regulated industries (without proper controls)
- Large, distributed teams
- Projects requiring long-term feature development

GitLab Flow

Overview

GitLab Flow combines feature-driven development with issue tracking and environment-specific branches.

Environment Branches

```
# Branch structure
main (development)
├── pre-production (staging)
└── production (live)

# Feature development
git checkout main
git checkout -b feature/user-profile
# Develop feature...
git push origin feature/user-profile
# Create merge request to main
```

```
# Deploy to staging
git checkout pre-production
git merge main
git push origin pre-production

# Deploy to production
git checkout production
git merge pre-production
git push origin production
git tag v1.2.0
```

Release Branches (Alternative)

```
# For scheduled releases
main (development)
├── 2-3-stable (release branch)
└── 2-4-stable (next release)

# Cherry-pick fixes to release branches
git checkout 2-3-stable
git cherry-pick abc123  # Bug fix from main
git push origin 2-3-stable
```

GitLab Flow Workflow

```
# 1. Create issue
gh issue create --title "Add user profile page" \
  --body "Users need a profile page to manage their information"

# 2. Create branch from issue
git checkout main
git pull origin main
git checkout -b 123-add-user-profile  # Issue number prefix

# 3. Develop feature
git commit -m "feat: add user profile component

Implements basic user profile page with:
- Personal information display
- Avatar upload
- Settings management

Closes #123"

# 4. Push and create merge request
git push -u origin 123-add-user-profile
gh pr create --title "Add user profile page" \
  --body "Closes #123" \
  --assignee @reviewer
```

```
# 5. Code review and merge to main
gh pr merge --squash

# 6. Deploy to pre-production
git checkout pre-production
git merge main
git push origin pre-production

# 7. Test in staging environment
# Run integration tests, manual testing

# 8. Deploy to production
git checkout production
git merge pre-production
git push origin production
git tag v1.2.0
```

Release Flow (Microsoft)

Overview

Release Flow is designed for large teams with scheduled releases and multiple supported versions.

Branch Structure

```
main (development)
├── releases/v1.0 (supported release)
├── releases/v1.1 (current release)
└── releases/v1.2 (next release)

# Topic branches
├── users/alice/feature-x
├── users/bob/bugfix-y
└── teams/frontend/ui-refresh
```

Workflow

```
# 1. Create topic branch
git checkout main
git checkout -b users/alice/add-search-feature

# 2. Develop and push regularly
git commit -m "feat: add search infrastructure"
git push -u origin users/alice/add-search-feature

# 3. Create pull request to main
gh pr create --title "Add search feature" \
  --body "Implements full-text search with filters"
```

```
# 4. Merge to main after review
gh pr merge --squash

# 5. Create release branch when ready
git checkout main
git checkout -b releases/v1.2
git push -u origin releases/v1.2

# 6. Cherry-pick fixes to release branch
git checkout releases/v1.2
git cherry-pick abc123  # Bug fix from main

# 7. Deploy from release branch
git tag v1.2.0
git push origin v1.2.0
```

Choosing the Right Strategy

Decision Matrix

Factor	Git Flow	GitHub Flow	Trunk-Based	GitLab Flow
Team Size	Large	Small-Medium	Any	Medium-Large
Release Cycle	Scheduled	Continuous	Continuous	Flexible
Complexity	High	Low	Medium	Medium
CI/CD Maturity	Medium	High	Very High	High
Feature Flags	Optional	Optional	Required	Optional
Learning Curve	Steep	Gentle	Medium	Medium
Parallel Features	Excellent	Good	Limited	Good
Hotfix Support	Excellent	Basic	Good	Good

Team Characteristics

Small Team (2-5 developers)

```
# Recommended: GitHub Flow
# Simple, fast, minimal overhead
git checkout main
git checkout -b feature/quick-fix
git commit -m "fix: resolve login issue"
git push origin feature/quick-fix
gh pr create --title "Fix login issue"
gh pr merge --squash
```

Medium Team (5-15 developers)

```
# Recommended: GitLab Flow or Trunk-Based
# Balance between structure and simplicity

# GitLab Flow approach
git checkout main
git checkout -b feature/user-dashboard
# Develop feature...
gh pr create --title "Add user dashboard"
# Merge to main, then promote through environments
```

Large Team (15+ developers)

```
# Recommended: Git Flow or Release Flow
# Structure needed for coordination

# Git Flow approach
git flow feature start user-management
# Develop feature...
git flow feature finish user-management
# Structured release process
```

Project Characteristics

Web Application (SaaS)

```
# Recommended: GitHub Flow or Trunk-Based
# Fast deployment, continuous delivery

# Trunk-based with feature flags
git checkout main
git checkout -b add-feature-flag
echo "newFeature: false" >> config.js
git commit -m "feat: add feature flag for new feature"
gh pr create --title "Add feature flag"
gh pr merge --squash
```

Mobile Application

```
# Recommended: Git Flow or GitLab Flow
# App store releases, testing cycles

# Git Flow for app releases
```

```
git flow release start 2.1.0
# Prepare release, test thoroughly
git flow release finish 2.1.0
# Submit to app stores
```

Enterprise Software

```
# Recommended: Git Flow or Release Flow
# Multiple supported versions, scheduled releases

# Release Flow for enterprise
git checkout main
git checkout -b releases/v2023.1
# Stabilize release
git tag v2023.1.0
# Support multiple versions
```

Implementation Guide

Transitioning Strategies

From No Strategy to GitHub Flow

```
# Week 1: Establish main branch protection
gh api repos/:owner/:repo/branches/main/protection \
  --method PUT \
  --field required_status_checks='{}' \
  --field enforce_admins=true \
  --field required_pull_request_reviews='{}'

# Week 2: Train team on PR workflow
# Create PR template
cat > .github/pull_request_template.md << EOF
## Description
Brief description of changes

## Testing
- [ ] Unit tests pass
- [ ] Manual testing completed

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

# Week 3: Implement CI/CD pipeline
# Add GitHub Actions workflow
```

```
# Week 4: Full adoption
# All changes go through PRs
```

From GitHub Flow to Trunk-Based

```
# Phase 1: Reduce branch lifetime
# Encourage smaller, more frequent PRs

# Phase 2: Implement feature flags
npm install --save feature-flags

# Phase 3: Increase integration frequency
# Multiple merges per day

# Phase 4: Remove long-lived feature branches
# All work in short-lived branches
```

Team Training

Git Flow Training Plan

```
# Day 1: Concepts and setup
git flow init
# Explain branch types and purposes

# Day 2: Feature workflow
git flow feature start training-feature
git commit -m "feat: add training example"
git flow feature finish training-feature

# Day 3: Release workflow
git flow release start 1.0.0
# Practice release preparation
git flow release finish 1.0.0

# Day 4: Hotfix workflow
git flow hotfix start critical-fix
git commit -m "fix: critical security patch"
git flow hotfix finish critical-fix

# Day 5: Integration with tools
# Configure CI/CD for Git Flow
```

Trunk-Based Training Plan

```
# Week 1: Feature flag infrastructure
# Set up feature flag system

# Week 2: Short-lived branches
# Practice 1-day branch lifecycle

# Week 3: Continuous integration
# Multiple integrations per day

# Week 4: Branch by abstraction
# Practice large feature development
```

Monitoring and Metrics

Branch Metrics

```
# Script to analyze branch patterns
#!/bin/bash

# Average branch lifetime
git for-each-ref --format='%(refname:short) %(committerdate)' refs/remotes/origin/
| \
  grep -v 'main\|develop' | \
  while read branch date; do
    created=$(git log --reverse --format='%ct' $branch | head -1)
    merged=$(git log --format='%ct' --grep="Merge.*$branch" main | head -1)
    if [ ! -z "$merged" ]; then
      lifetime=$((($merged - $created) / 86400))
      echo "$branch: $lifetime days"
    fi
  done

# Number of active branches
git branch -r | grep -v 'main\|develop' | wc -l

# Merge frequency
git log --oneline --grep="Merge" --since="1 month ago" | wc -l
```

Quality Metrics

```
# Commit frequency
git log --oneline --since="1 week ago" | wc -l

# Average commits per PR
gh pr list --state merged --limit 50 --json number | \
  jq -r '.[] | .number' | \
  while read pr; do
    commits=$(gh pr view $pr --json commits | jq '.commits | length')
```

```
    echo $commits
  done | \
  awk '{sum+=$1; count++} END {print "Average:", sum/count}'

# Conflict resolution time
git log --grep="resolve.*conflict" --format='%ct' | \
  # Calculate time between conflict and resolution
```

Best Practices Summary

Universal Best Practices

- ✅ **Protect main branch** - Require reviews and status checks
- ✅ **Use descriptive branch names** - Clear purpose and scope
- ✅ **Write good commit messages** - Follow conventional commits
- ✅ **Review all changes** - No direct pushes to main
- ✅ **Automate testing** - CI/CD for all branches
- ✅ **Clean up branches** - Delete merged branches
- ✅ **Document strategy** - Team guidelines and training
- ✅ **Monitor metrics** - Track branch health and team velocity

Strategy-Specific Tips

Git Flow

- Use git-flow tools for consistency
- Automate release branch creation
- Maintain clear release notes
- Train team on all branch types

GitHub Flow

- Deploy branches for testing
- Use draft PRs for early feedback
- Implement feature flags for incomplete work
- Maintain high test coverage

Trunk-Based Development

- Invest in automated testing
- Use feature flags extensively
- Keep branches under 2 days
- Practice branch by abstraction

GitLab Flow

- Use environment branches consistently
- Implement proper promotion process

- Link issues to merge requests
- Maintain environment parity

Quick Reference

```
# Git Flow commands
git flow init                      # Initialize git flow
git flow feature start <name>    # Start feature
git flow feature finish <name>   # Finish feature
git flow release start <version> # Start release
git flow release finish <version># Finish release
git flow hotfix start <name>     # Start hotfix
git flow hotfix finish <name>    # Finish hotfix

# GitHub Flow commands
git checkout -b feature/name     # Create feature branch
gh pr create                     # Create pull request
gh pr merge --squash             # Squash and merge
git branch -d feature/name       # Delete branch

# Trunk-based commands
git checkout -b short-lived      # Short-lived branch
git commit -m "small change"     # Small, focused commits
gh pr create --draft             # Draft PR for feedback
gh pr merge --squash             # Quick merge

# Branch analysis
git branch -r                    # List remote branches
git log --graph --oneline        # Visual branch history
git show-branch                  # Show branch relationships
```

Previous: Clean Git History
Next: Force Push Safety

Force Push Safety: Using --force-with-lease

Force pushing is sometimes necessary in Git workflows, but it can be dangerous in team environments. This guide covers safe practices for force pushing, with emphasis on `--force-with-lease` and other protective measures.

Understanding Force Push

What is Force Push?

Force push overwrites the remote branch history with your local branch history, potentially losing commits that others have made.

```
# Dangerous: Traditional force push
git push --force origin feature-branch

# Safer: Force push with lease
git push --force-with-lease origin feature-branch
```

When Force Push is Needed

1. **After interactive rebase** - Cleaning up commit history
2. **After amending commits** - Fixing commit messages or content
3. **After squashing commits** - Combining related commits
4. **After cherry-picking** - Applying commits from other branches
5. **After resetting commits** - Undoing recent commits

The Dangers of Force Push

```
# Scenario: Developer A and B working on same branch

# Developer A's work
git checkout feature-branch
echo "A's work" > file-a.txt
git add file-a.txt
git commit -m "feat: add A's feature"
git push origin feature-branch

# Developer B pulls and adds work
git pull origin feature-branch
echo "B's work" > file-b.txt
git add file-b.txt
git commit -m "feat: add B's feature"
git push origin feature-branch

# Developer A rebases (without pulling B's work)
git rebase -i HEAD~2  # Squash commits
git push --force origin feature-branch  # DANGEROUS!

# Result: B's work is lost!
```

--force-with-lease: The Safe Alternative

How --force-with-lease Works

`--force-with-lease` only allows the force push if the remote branch is at the expected commit (the one you last fetched).

```
# Safe force push workflow
git fetch origin                                # Update remote refs
```

```
git rebase -i HEAD~3                          # Clean up commits
git push --force-with-lease origin feature-branch  # Safe force push
```

Detailed Example

```
# Initial setup
git checkout -b feature/user-authentication
echo "Login component" > login.js
git add login.js
git commit -m "feat: add login component"
git push -u origin feature/user-authentication

# Add more commits
echo "Password validation" >> login.js
git add login.js
git commit -m "feat: add password validation"

echo "Login tests" > login.test.js
git add login.test.js
git commit -m "test: add login tests"

echo "Fix typo" >> login.js
git add login.js
git commit -m "fix: typo in login component"

git push origin feature/user-authentication

# Clean up history with interactive rebase
git rebase -i HEAD~3

# In the editor:
# pick abc123 feat: add login component
# pick def456 feat: add password validation
# squash ghi789 test: add login tests
# fixup jkl012 fix: typo in login component

# After rebase, history is rewritten
git log --oneline
# abc123 feat: add login component
# def456 feat: add password validation and tests

# Safe force push
git push --force-with-lease origin feature/user-authentication

# If someone else pushed in the meantime:
# error: failed to push some refs to 'origin'
# hint: Updates were rejected because the tip of your current branch is behind
# hint: its remote counterpart. Integrate the remote changes (e.g.
# hint: 'git pull ...') before pushing again.
```

--force-with-lease vs --force

```
# Scenario: Remote has new commits

# Setup: Remote branch has commits you don't have locally
git log --oneline origin/feature-branch
# abc123 feat: teammate's work (you don't have this)
# def456 feat: your previous work

git log --oneline feature-branch
# ghi789 feat: your rebased work
# def456 feat: your previous work (rebased)

# Using --force (DANGEROUS)
git push --force origin feature-branch
# SUCCESS: Overwrites teammate's work!

# Using --force-with-lease (SAFE)
git push --force-with-lease origin feature-branch
# ERROR: Rejects push, protects teammate's work
```

Safe Force Push Workflows

Pre-Force Push Checklist

```
#!/bin/bash
# safe-force-push.sh - Script for safe force pushing

BRANCH=$(git branch --show-current)
REMOTE="origin"

echo "🔍 Pre-force-push safety checks for branch: $BRANCH"

# 1. Ensure we're not on main/develop
if [[ "$BRANCH" == "main" || "$BRANCH" == "develop" || "$BRANCH" == "master" ]];
then
    echo "❌ Cannot force push to protected branch: $BRANCH"
    exit 1
fi

# 2. Fetch latest remote state
echo "📡 Fetching latest remote state..."
git fetch $REMOTE

# 3. Check if remote branch exists
if ! git show-ref --verify --quiet refs/remotes/$REMOTE/$BRANCH; then
    echo "❌ Remote branch $REMOTE/$BRANCH does not exist"
    exit 1
fi
```

```
# 4. Show what will be overwritten
echo "📊 Commits that will be overwritten:"
git log --oneline $REMOTE/$BRANCH..$BRANCH

echo "📊 Commits that will be lost:"
git log --oneline $BRANCH..$REMOTE/$BRANCH

# 5. Confirm with user
read -p "🤔 Proceed with force push? (y/N): " -n 1 -r
echo
if [[ ! $REPLY =~ ^[Yy]$ ]]; then
    echo "❌ Force push cancelled"
    exit 1
fi

# 6. Perform safe force push
echo "🚀 Performing safe force push..."
if git push --force-with-lease $REMOTE $BRANCH; then
    echo "✅ Force push successful!"
else
    echo "❌ Force push failed - remote branch was updated"
    echo "💡 Run 'git pull --rebase' and try again"
    exit 1
fi
```

Team Coordination Workflow

```
# 1. Communicate intent
# Post in team chat: "Rebasing feature/user-auth, will force push in 5 min"

# 2. Ensure no one else is working on the branch
git fetch origin
git log --oneline feature/user-auth..origin/feature/user-auth
# Should be empty if no one else pushed

# 3. Perform rebase
git rebase -i HEAD~5

# 4. Force push with lease
git push --force-with-lease origin feature/user-auth

# 5. Notify team
# Post in team chat: "Force push complete on feature/user-auth"
```

Collaborative Feature Branch

```
# When multiple developers work on same feature branch

# Developer A: Wants to clean up commits
```

```
git checkout feature/shared-feature
git fetch origin

# Check if others have pushed
if git log --oneline HEAD..origin/feature/shared-feature | grep -q .; then
    echo "⚠️  Others have pushed to this branch"
    echo "📞 Coordinate with team before force pushing"

    # Show who made recent commits
    git log --oneline --format="%h %an %s" HEAD..origin/feature/shared-feature

    exit 1
fi

# Safe to rebase if no new commits
git rebase -i HEAD~3
git push --force-with-lease origin feature/shared-feature
```

Advanced Force Push Scenarios

Recovering from Failed Force Push

```
# If force push fails due to remote updates
git push --force-with-lease origin feature-branch
# error: failed to push some refs

# Option 1: Rebase on top of remote changes
git fetch origin
git rebase origin/feature-branch
git push --force-with-lease origin feature-branch

# Option 2: Merge remote changes (if rebase is complex)
git fetch origin
git merge origin/feature-branch
git push origin feature-branch  # No force needed

# Option 3: Reset and start over (if safe)
git fetch origin
git reset --hard origin/feature-branch
# Redo your changes
```

Force Push with Specific Lease

```
# Force push only if remote is at specific commit
git push --force-with-lease=feature-branch:abc123 origin feature-branch

# This ensures remote branch is exactly at commit abc123
# Useful when you know the exact state you expect
```

Partial Force Push

```
# Force push only specific commits
git push --force-with-lease origin HEAD~2:feature-branch

# This pushes all commits except the last 2 to remote
# Useful for partial updates
```

Branch Protection and Policies

GitHub Branch Protection Rules

```
# Set up branch protection via GitHub CLI
gh api repos/:owner/:repo/branches/main/protection \
  --method PUT \
  --field required_status_checks='{"strict":true,"contexts":["ci/tests"]}' \
  --field enforce_admins=true \
  --field required_pull_request_reviews='{"required_approving_review_count":2}' \
  --field restrictions=null \
  --field allow_force_pushes=false \
  --field allow_deletions=false

# Protect develop branch
gh api repos/:owner/:repo/branches/develop/protection \
  --method PUT \
  --field required_status_checks='{"strict":true,"contexts":["ci/tests"]}' \
  --field enforce_admins=true \
  --field required_pull_request_reviews='{"required_approving_review_count":1}' \
  --field allow_force_pushes=false
```

Git Hooks for Force Push Protection

```
# Server-side pre-receive hook
cat > hooks/pre-receive << 'EOF'
#!/bin/bash

while read oldrev newrev refname; do
    branch=$(echo $refname | sed 's/refs\/heads\///')

    # Protect main branches from force push
    if [[ "$branch" == "main" || "$branch" == "develop" ]]; then
        if [[ "$oldrev" != "0000000000000000000000000000000000000000" ]]; then
            # Check if this is a force push (non-fast-forward)
            if ! git merge-base --is-ancestor $oldrev $newrev; then
                echo "❌ Force push to $branch is not allowed"
                exit 1
            fi
```

```
        fi
    fi

    # Warn about force pushes to feature branches
    if [[ "$branch" == feature/* ]]; then
        if [[ "$oldrev" != "0000000000000000000000000000000000000000" ]]; then
            if ! git merge-base --is-ancestor $oldrev $newrev; then
                echo "⚠️  Force push detected on feature branch: $branch"
                echo "💡 Consider using --force-with-lease for safety"
            fi
        fi
    fi
done
EOF

chmod +x hooks/pre-receive
```

Client-side Protection

```
# Pre-push hook to prevent accidental force push
cat > .git/hooks/pre-push << 'EOF'
#!/bin/bash

protected_branches="main develop master"
remote="$1"
url="$2"

while read local_ref local_sha remote_ref remote_sha; do
    branch=$(echo $remote_ref | sed 's/refs\/heads\///')

    # Check if pushing to protected branch
    for protected in $protected_branches; do
        if [[ "$branch" == "$protected" ]]; then
            echo "❌ Direct push to $branch is not allowed"
            echo "💡 Use pull request workflow instead"
            exit 1
        fi
    done

    # Check for force push
    if [[ "$remote_sha" != "0000000000000000000000000000000000000000" ]]; then
        if ! git merge-base --is-ancestor $remote_sha $local_sha; then
            echo "⚠️  Force push detected to $branch"
            read -p "🤔 Are you sure? This will rewrite history. (y/N): " -n 1 -r
            echo
            if [[ ! $REPLY =~ ^[Yy]$ ]]; then
                echo "❌ Push cancelled"
                exit 1
            fi
        fi
    fi
```

```
done
EOF

chmod +x .git/hooks/pre-push
```

Team Guidelines and Best Practices

Force Push Policy Template

```
# Force Push Policy

## Allowed Scenarios

- ✅ Feature branches (with coordination)
- ✅ Personal branches (user/name/\*)
- ✅ Experimental branches (experiment/\*)
- ✅ After team coordination

## Forbidden Scenarios

- ❌ main/master branch
- ❌ develop branch
- ❌ release/\* branches
- ❌ Shared feature branches (without coordination)

## Required Practices

1. **Always use --force-with-lease**
2. **Fetch before force pushing**
3. **Communicate with team**
4. **Verify branch state**
5. **Have backup plan**

## Communication Protocol

1. Announce intent in team chat
2. Wait for acknowledgment
3. Perform force push
4. Confirm completion

## Emergency Procedures

If force push causes issues:

1. Immediately notify team
2. Use git reflog to find lost commits
3. Create recovery branch
4. Coordinate team recovery
```

Team Training Checklist

```
# Force push training session

# 1. Demonstrate the danger
git checkout -b demo-danger
echo "Original work" > file.txt
git add file.txt
git commit -m "original work"
git push -u origin demo-danger

# Simulate teammate's work
echo "Teammate's work" >> file.txt
git add file.txt
git commit -m "teammate's addition"
git push origin demo-danger

# Reset to simulate rebase
git reset --hard HEAD~1
echo "My rebased work" > file.txt
git add file.txt
git commit -m "my rebased work"

# Show difference between --force and --force-with-lease
git push --force-with-lease origin demo-danger  # Fails safely
git push --force origin demo-danger             # Succeeds dangerously

# 2. Practice safe workflow
git checkout -b demo-safe
echo "Safe work" > safe.txt
git add safe.txt
git commit -m "safe work"
git push -u origin demo-safe

# Rebase safely
git fetch origin
git rebase -i HEAD~1
git push --force-with-lease origin demo-safe

# 3. Recovery practice
# Simulate lost work and recovery using reflog
```

Automation and Tooling

Git Aliases for Safe Force Push

```
# Set up helpful aliases
git config --global alias.force-lease 'push --force-with-lease'
git config --global alias.safe-force '!f() { git fetch origin && git push --force-
with-lease origin "$@"; }; f'
git config --global alias.force-check '!f() { git fetch origin && git log --
oneline "$1"..origin/"$1"; }; f'
```

```
# Usage examples
git force-lease origin feature-branch
git safe-force feature-branch
git force-check feature-branch  # Check what would be overwritten
```

Shell Functions

```
# Add to ~/.bashrc or ~/.zshrc

# Safe force push function
safe_force_push() {
    local branch=${1:-$(git branch --show-current)}
    local remote=${2:-origin}

    echo "🔍 Checking branch: $branch"

    # Fetch latest
    git fetch $remote

    # Check if branch exists remotely
    if ! git show-ref --verify --quiet refs/remotes/$remote/$branch; then
        echo "❌ Remote branch $remote/$branch does not exist"
        return 1
    fi

    # Show what will be lost
    local lost_commits=$(git rev-list --count $branch..$remote/$branch)
    if [ $lost_commits -gt 0 ]; then
        echo "⚠️  $lost_commits commits will be lost:"
        git log --oneline $branch..$remote/$branch

        read -p "Continue? (y/N): " -n 1 -r
        echo
        if [[ ! $REPLY =~ ^[Yy]$ ]]; then
            echo "❌ Cancelled"
            return 1
        fi
    fi

    # Perform safe force push
    git push --force-with-lease $remote $branch
}

# Check force push safety
check_force_safety() {
    local branch=${1:-$(git branch --show-current)}
    local remote=${2:-origin}

    git fetch $remote
```

```
    echo "📊 Branch status for $branch:"
    echo "Local commits not on remote:"
    git log --oneline $remote/$branch..$branch

    echo "Remote commits not local:"
    git log --oneline $branch..$remote/$branch

    echo "Last common commit:"
    git merge-base $branch $remote/$branch | xargs git show --oneline -s
}
```

IDE Integration

```
// VS Code settings.json
{
  "git.allowForcePush": false,
  "git.useForcePushWithLease": true,
  "git.confirmForcePush": true,
  "git.protectedBranches": ["main", "develop", "master"]
}
```

Troubleshooting Force Push Issues

Common Problems and Solutions

Problem: Force Push Rejected

```
# Error message:
# error: failed to push some refs to 'origin'
# hint: Updates were rejected because the tip of your current branch is behind

# Solution 1: Check what changed
git fetch origin
git log --oneline HEAD..origin/feature-branch

# Solution 2: Rebase on top of changes
git rebase origin/feature-branch
git push --force-with-lease origin feature-branch

# Solution 3: Merge if rebase is complex
git merge origin/feature-branch
git push origin feature-branch  # No force needed
```

Problem: Lost Commits After Force Push

```
# Recovery using reflog
git reflog
# Find the commit before force push
# abc123 HEAD@{1}: commit: lost work

# Create recovery branch
git checkout -b recovery-branch abc123

# Cherry-pick lost commits
git checkout feature-branch
git cherry-pick abc123

# Or reset to lost commit
git reset --hard abc123
```

Problem: Team Member Can't Pull After Force Push

```
# Team member sees:
# error: Your local changes to the following files would be overwritten by merge

# Solution: Reset their local branch
git fetch origin
git reset --hard origin/feature-branch

# Or create backup and reset
git branch backup-branch  # Backup local work
git reset --hard origin/feature-branch
# Cherry-pick from backup if needed
```

Recovery Strategies

Complete Branch Recovery

```
# If entire branch history is lost

# 1. Check reflog on all team members' machines
git reflog --all | grep feature-branch

# 2. Find the lost commits
git show abc123  # Verify this is the lost work

# 3. Recreate branch from lost commit
git checkout -b feature-branch-recovered abc123

# 4. Force push the recovery (with team coordination)
git push --force-with-lease origin feature-branch-recovered

# 5. Rename branch back
```

```
git branch -m feature-branch-recovered feature-branch
git push origin :feature-branch-recovered  # Delete old branch
git push -u origin feature-branch
```

Partial Recovery

```
# If only some commits are lost

# 1. Find lost commits in reflog
git reflog | grep "commit:"

# 2. Cherry-pick specific commits
git cherry-pick abc123 def456 ghi789

# 3. Resolve any conflicts
git add .
git cherry-pick --continue

# 4. Push recovered branch
git push origin feature-branch
```

Monitoring and Auditing

Force Push Audit Script

```
#!/bin/bash
# audit-force-pushes.sh

echo "📊 Force Push Audit Report"
echo "=========================="

# Check for force pushes in git log
echo "🔍 Recent force pushes:"
git log --walk-reflogs --grep="forced-update" --oneline --since="1 month ago"

echo "\n📈 Force push frequency by author:"
git log --walk-reflogs --grep="forced-update" --format="%an" --since="1 month ago"
| \
    sort | uniq -c | sort -nr

echo "\n🌿 Branches with recent force pushes:"
git for-each-ref --format="%(refname:short) %(push:track)" refs/heads/ | \
    grep "ahead\|behind" | \
    while read branch status; do
        if git log --walk-reflogs --grep="forced-update" refs/heads/$branch --
since="1 week ago" >/dev/null 2>&1; then
            echo "  $branch (force pushed recently)"
        fi
```

```
    done

echo "\n⚠️  Protected branches status:"
for branch in main develop master; do
    if git show-ref --verify --quiet refs/heads/$branch; then
        echo "  $branch: $(git log --oneline -1 $branch)"
    fi
done
```

GitHub Actions Monitoring

```
# .github/workflows/force-push-monitor.yml
name: Force Push Monitor

on:
  push:
    branches: ["**"]

jobs:
  monitor-force-push:
    runs-on: ubuntu-latest
    if: github.event.forced

    steps:
      - name: Notify force push
        uses: actions/github-script@v6
        with:
          script: |
            const { owner, repo } = context.repo;
            const branch = context.ref.replace('refs/heads/', '');
            const pusher = context.actor;

            // Post to Slack/Teams/Discord
            await github.rest.issues.createComment({
              owner,
              repo,
              issue_number: 1, // Or find related PR
              body: `⚠️ Force push detected on \`${branch}\` by @${pusher}`
            });

            // Create audit issue
            await github.rest.issues.create({
              owner,
              repo,
              title: `Force Push Audit: ${branch}`,
              body: `Force push detected:\n- Branch: ${branch}\n- Author:
${pusher}\n- Commit: ${context.sha}`,
              labels: ['audit', 'force-push']
            });
```

Best Practices Summary

Do's ✅

- **Always use --force-with-lease** instead of --force
- **Fetch before force pushing** to get latest remote state
- **Communicate with team** before force pushing shared branches
- **Protect main branches** from force pushes
- **Use descriptive commit messages** when cleaning history
- **Have a recovery plan** before force pushing
- **Document force push policies** for your team
- **Train team members** on safe practices

Don'ts ❌

- **Never force push to main/develop** branches
- **Don't force push without fetching** first
- **Don't force push shared branches** without coordination
- **Don't use --force** when --force-with-lease is available
- **Don't force push during active collaboration**
- **Don't ignore force push failures** - investigate why
- **Don't force push without backup** of important work
- **Don't skip team communication** for shared branches

Quick Reference

```
# Safe force push commands
git fetch origin                                # Update remote refs
git push --force-with-lease origin branch    # Safe force push
git push --force-with-lease=branch:sha origin branch  # Specific lease

# Check before force push
git log --oneline HEAD..origin/branch        # What you'll overwrite
git log --oneline origin/branch..HEAD        # What you'll push
git merge-base HEAD origin/branch             # Common ancestor

# Recovery commands
git reflog                                     # Find lost commits
git cherry-pick <commit>                       # Recover specific commit
git reset --hard <commit>                      # Reset to specific commit
git branch recovery-branch <commit>            # Create recovery branch

# Protection setup
gh api repos/:owner/:repo/branches/main/protection --method PUT --field
allow_force_pushes=false
git config --global alias.force-lease 'push --force-with-lease'

# Audit commands
git log --walk-reflogs --grep="forced-update" # Find force pushes
```

```
git for-each-ref --format="%(refname:short) %(push:track)" refs/heads/  # Branch
status
```

Previous: Branching Strategies
Next: README