Git & GitHub
Complete Reference Guide

Version Control Systems

A Comprehensive Guide to Git Commands,
GitHub Workflows, and Best Practices

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to Git

1.1 What is Git?

Git is a **distributed version control system** (DVCS) designed to manage source code efficiently. It enables developers to:

- Track changes in code over time
- Collaborate effectively with multiple developers
- Support flexible and non-linear development workflows
- Maintain complete project history locally and remotely

1.2 Key Features of Git

1. **Version Control**: Complete history of all changes
2. **Distributed Nature**: Every developer has a full repository copy
3. **Branching & Merging**: Parallel development support
4. **Fast Performance**: Efficient operations on large codebases
5. **Data Integrity**: SHA-based commit identification

1.3 Why Use Git?

1.3.1 Bug Fixing and Code Versioning

- Maintains complete history of code changes
- Tracks *who* made changes, *when*, and *why*
- Enables reverting to previous stable versions
- Supports multiple software versions simultaneously
- Allows safe testing without affecting main codebase

1.3.2 Collaboration

- Multiple developers work on same project simultaneously
- Manages and resolves conflicts between changes
- Each developer works on local repository copy
- Pull requests and code reviews improve quality
- Enhances teamwork in large projects

1.3.3 Non-Linear Development

- Branching supports parallel development
- Separate branches for features, bug fixes, experiments
- Independent work without disturbing main code
- Tested branches merge into main branch
- Improves flexibility and development speed

2 Core Git Concepts

2.1 Distributed Version Control System (DVCS)

- Every developer has a complete repository copy
- Full project history available locally
- Supports offline work
- Improves collaboration and redundancy
- Example: Git, Mercurial

2.2 Repository

A **repository** (or ”repo”) is a storage location for a Git project containing:

- All project files and directories
- Complete commit history
- All branches and tags
- Configuration and metadata

Repositories can be:

- **Local**: Stored on developer’s machine
- **Remote**: Hosted on servers (GitHub, GitLab, Bitbucket)

2.3 Commit

A **commit** is a snapshot of staged changes:

- Each commit has a unique SHA ID (hash)
- Contains commit message describing changes
- Forms the project’s version history
- Immutable once created

2.4 Branch

A **branch** is a parallel line of development:

- Allows working on features independently
- Main/master branch contains stable code
- Feature branches for new development
- Can be merged back into main branch

2.5 Working Directory

The **working directory** is where files are created and modified:

- Contains current project files
- Changes not tracked until staged
- Reflects current branch state

2.6 Staging Area (Index)

The **staging area** is an intermediate preparation zone:

- Prepares changes for commit
- Allows selective file tracking
- Files added using `git add`
- Review changes before committing

2.7 Merge

Merging combines changes from different branches:

- Integrates parallel development work
- Creates merge commit (in complex merges)
- May require conflict resolution

2.8 Clone

Cloning creates a local copy of remote repository:

- Includes all files, branches, and history
- Automatically sets up remote reference
- Enables local development

3 Git Framework and Workflow

3.1 The Four-Stage Architecture

Git follows a structured workflow through four main areas:

1. **Workspace (Working Directory)**
   - Where files are created, modified, or deleted
   - Changes not tracked until staged
   - Example: Creating/editing `index.html`
2. **Staging Area (Index)**
   - Intermediate area for preparing commits
   - Allows selective change tracking
   - Command: `git add <file>`
3. **Local Repository**
   - Stores committed project history locally
   - Each commit is a snapshot of staged changes
   - Command: `git commit -m "message"`
4. **Remote Repository**
   - Hosted on servers (GitHub, GitLab)
   - Enables team collaboration
   - Command: `git push origin main`

3.2 Typical Workflow Example

1. Developer edits `index.html` in workspace
2. File added to staging area: `git add index.html`
3. Staged changes committed to local repository
4. Commit pushed to remote repository

Key Point

The Git framework ensures systematic version control by organizing changes from development to collaboration through distinct stages.

4 Git Configuration and Setup

4.1 Initial Configuration

Before using Git, configure user identity. This information is associated with every commit.

4.1.1 Setting Username

```
git config --global user.name "Your Name"
```

- Username appears in all commits
- `--global` applies to all repositories

4.1.2 Setting Email

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

- Email associates commits with user
- Should match email on Git platforms

4.1.3 Verifying Configuration

```
git config --global user.name
git config --global user.email
```

These commands display currently saved Git configuration values.

5 Essential Git Commands

5.1 Repository Initialization

5.1.1 `git init`

Initializes a new Git repository.

```
git init
```

- Creates hidden `.git` directory
- Stores version control data
- Execute once at project start

5.2 Status and Tracking

5.2.1 `git status`

Displays current repository state.

```
git status
```

- Shows modified, staged, and untracked files
- Indicates which changes are ready to commit
- Helps understand current working state

5.3 Staging Files

5.3.1 `git add .`

Stages all modified and untracked files.

```
git add .
```

5.3.2 `git add <file_name>`

Stages specific file.

```
git add index.html
```

5.4 Committing Changes

5.4.1 `git commit -m "<message>"`

Saves staged changes to local repository.

```
git commit -m "Add homepage design"
```

- `-m` flag provides commit message
- Creates project snapshot
- Message should be descriptive

5.5 Unstaging Files

5.5.1 `git restore --staged <file_name>`

Removes file from staging area.

```
git restore --staged index.html
```

- Does not delete file or changes
- Useful when file staged accidentally
- Changes remain in workspace

5.6 The .gitignore File

Specifies files and folders Git should ignore.

```
# Example .gitignore entries
node_modules/
*.log
.env
build/
*.tmp
```

- Prevents tracking of build files, logs, secrets
- One pattern per line
- Use wildcards for pattern matching

6 Viewing and Analyzing History

6.1 Basic Log Commands

6.1.1 `git log`

Displays complete commit history.

```
git log
```

- Shows full commit details
- Includes SHA ID, author, date, message
- Press `q` to exit

6.1.2 `git log --oneline`

Displays condensed commit history.

```
git log --oneline
```

- One commit per line
- Shows shortened SHA and message
- Quick history overview

6.1.3 `git log --stat`

Shows file change statistics.

```
git log --stat
```

- Displays files changed per commit
- Shows lines added/deleted
- Useful for understanding commit scope

6.1.4 `git log -p`

Displays detailed patch information.

```
git log -p
```

- Shows line-by-line changes
- Full diff for each commit
- Useful for detailed code review

6.2 Branch History

6.2.1 `git log --oneline --all`

Shows history of all branches.

```
git log --oneline --all
```

6.2.2 `git log --oneline --all --graph`

Displays graphical branch representation.

```
git log --oneline --all --graph
```

- Visualizes branching structure
- Shows merge points
- Helps understand development flow

6.3 Viewing Specific Commits

6.3.1 `git show <sha_id>`

Displays detailed commit information.

```
git show a3f8e2
```

- First 6 digits of SHA sufficient
- Shows commit metadata and changes
- Includes author, date, diff

6.4 Comparing Changes

6.4.1 `git diff`

Shows differences between files.

```
git diff
```

- Compares workspace vs staging area
- Shows unstaged changes
- Review before committing

7 Branching and Navigation

7.1 Understanding Branches

Branches are snapshots of a repository allowing independent development:

- Main branch contains stable version
- Feature branches for new development
- Enables parallel work by multiple developers
- Changes tested before merging to main

7.2 Branch Commands

7.2.1 `git branch`

Lists all available branches.

```
git branch
```

- Current branch marked with asterisk (*)
- Shows local branches only

7.2.2 `git branch <name>`

Creates new branch.

```
git branch feature-login
```

7.2.3 `git branch <name> <sha_id>`

Creates branch at specific commit.

```
git branch hotfix a3f8e2
```

- Useful for starting from older commit
- Enables targeted development

7.3 Switching Branches

7.3.1 `git checkout <branch>`

Switches to specified branch.

```
git checkout feature-login
```

- Updates workspace to match branch
- Changes HEAD pointer
- Files may appear/disappear based on branch

7.3.2 `git checkout <sha_id>`

Moves to specific commit.

```
git checkout a3f8e2
```

- Enters *detached HEAD* state
- Used for inspecting old versions
- Not for making new commits

7.3.3 `git checkout main/master`

Returns to main branch.

```
git checkout main
```

7.4 Branch Switching Behavior

Important Note

When switching branches:

- Files from previous branch may be removed from workspace
- Files are **NOT permanently deleted**
- They are restored when switching back
- Git updates workspace to match selected branch

7.5 Deleting Branches

7.5.1 `git branch -d <branch>`

Deletes merged branch safely.

```
git branch -d feature-login
```

7.5.2 `git branch -D <branch>`

Force deletes branch (even if unmerged).

```
git branch -D experimental-feature
```

Branch Deletion Rule

A branch currently in use cannot be deleted. Switch to another branch first to avoid data loss.

8 Merging Branches

8.1 The Merge Process

8.1.1 `git merge <branch>`

Merges specified branch into current branch.

```
git checkout main
git merge feature-login
```

- Combines changes from branches
- Always merge *into* current branch
- May require conflict resolution

8.2 Types of Merging

8.2.1 Fast-Forward Merge

Occurs when current branch has no new commits since branching.

- Git simply moves branch pointer forward
- No merge commit created
- Results in linear history
- Clean and simple

Scenario:

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

After merge:
main:     A---B---C---D
```

8.2.2 Complex Merge (Three-Way Merge)

Occurs when both branches have new commits.

- Git creates new merge commit
- Combines histories of both branches
- May result in merge conflicts
- Requires manual conflict resolution

Scenario:

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

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

8.3 Branching Strategies

Common strategies for organizing development:

- **Feature Branching**: One branch per feature
- **Git Flow**: Structured workflow with multiple branch types
- **Release Branching**: Separate branches for releases
- **Trunk-Based**: Short-lived branches, frequent merging

9 Working with Remote Repositories

9.1 Connecting to Remote

9.1.1 `git remote add origin <url>`

Adds remote repository reference.

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

- Links local repository to remote server
- `origin` is conventional name
- Can have multiple remotes

9.1.2 `git remote -v`

Displays all remote repositories.

```
git remote -v
```

- Shows remote names and URLs
- Displays fetch and push URLs

9.1.3 `git remote rename <old> <new>`

Renames existing remote.

```
git remote rename origin upstream
```

9.2 Pushing Changes

9.2.1 `git push origin <branch>`

Uploads local commits to remote.

```
git push origin main
```

- Synchronizes local with remote
- Requires write permissions
- Updates remote branch

9.2.2 `git push origin <branch> --force`

Forcefully pushes commits, overwriting remote.

```
git push origin main --force
```

Warning

Use with extreme caution! Force push overwrites remote history and can cause data loss for collaborators.

9.3 Fetching and Pulling

9.3.1 `git clone <url>`

Creates local copy of remote repository.

```
git clone https://github.com/username/repo.git
```

- Downloads complete repository
- Automatically sets up remote reference
- Creates directory with repo name

9.3.2 `git fetch`

Downloads changes without merging.

```
git fetch origin
```

- Updates remote-tracking branches
- Does not modify working directory
- Safe operation for reviewing changes

9.3.3 `git pull origin <branch>`

Fetches and automatically merges changes.

```
git pull origin main
```

- Equivalent to `git fetch` + `git merge`
- Updates current branch
- May cause merge conflicts

9.4 Understanding Origin and Upstream

- **origin**: User's fork/personal repository
- **upstream**: Original repository (in fork workflow)
- **main**: Default primary branch name
- **HEAD**: Pointer to current branch/commit

10 Advanced Operations

10.1 Reverting Changes

10.1.1 `git revert <sha_id>`

Creates new commit that reverses changes.

```
git revert a3f8e2
```

- Does not delete commit history
- Safer than reset in shared repos
- Preserves project timeline
- Recommended for public branches

10.2 Viewing Differences with Remote

10.2.1 `git log upstream/main --oneline`

Displays upstream commit history.

```
git log upstream/main --oneline
```

10.2.2 `git diff upstream/main`

Shows differences with upstream.

```
git diff upstream/main
```

- Compares current branch with upstream
- Helps identify sync status
- Useful before creating pull requests

10.3 Syncing with Upstream

10.3.1 `git pull upstream main`

Fetches and merges from original repository.

```
git pull upstream main
```

- Updates fork with original changes
- Essential for staying current
- Prevents merge conflicts

10.3.2 `git remote add upstream <url>`

Adds original repository as upstream.

```
git remote add upstream https://github.com/original/repo.git
```

11 Git Hosting Platforms

11.1 GitHub

GitHub is a web-based hosting service for Git repositories.

- Owned by Microsoft
- Largest developer community
- Provides collaboration tools:
  - Pull requests
  - Issue tracking
  - Project boards
  - GitHub Actions (CI/CD)
  - GitHub Pages (hosting)
- Free public and private repositories
- Popular for open-source projects

11.2 GitLab

GitLab is a complete DevOps platform.

- Self-hosted or cloud-based
- Integrated CI/CD pipelines
- Built-in container registry
- Issue tracking and boards
- Wiki and documentation
- Security scanning features
- Can be deployed on-premises

11.3 SSH Protocol

SSH (Secure Shell) provides secure authentication.

- Encrypted communication protocol
- Password-less authentication via keys
- More secure than HTTPS with passwords
- Required for many Git operations
- SSH keys consist of public/private pair

12 Fork and Pull Request Workflow

12.1 Understanding Forks

A **fork** is a personal copy of another user's repository.

- Allows independent development
- Does not affect original repository
- Common in open-source projects
- Changes proposed via pull requests

12.2 Clone vs Fork

Cloning	**Forking**
Creates local copy of repository	Creates copy in your GitHub account
Directly linked to original	Independent copy on remote
Used for any repository access	Used for contributing to others' projects
Command: `git clone <url>`	Done via GitHub/GitLab interface

12.3 Pull Requests

A **pull request** (PR) proposes changes for review.

- Requests code review and approval
- Allows discussion about changes
- Ensures code quality before merging
- Can be approved, rejected, or requested changes
- Commonly used workflow in teams

12.4 Git Fork Workflow

1. **Fork** the original repository on GitHub
2. **Clone** your fork locally:

   ```
   git clone https://github.com/yourname/repo.git
   ```

3. **Add upstream** remote:

   ```
   git remote add upstream https://github.com/original/repo.git
   ```

4. **Create feature branch**:

   ```
   git checkout -b feature-name
   ```

5. **Make changes** and commit
6. **Push to origin**:

   ```
   git push origin feature-name
   ```

7. **Create pull request** from your fork to upstream/main
8. **Wait for review** and address feedback
9. After approval, changes are **merged** by maintainer

12.5 Keeping Fork Updated

```
# Fetch upstream changes
git fetch upstream

# Merge upstream/main into your local main
git checkout main
git merge upstream/main

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

13 Terminal and Directory Commands

13.1 Basic Directory Operations

13.1.1 `mkdir`

Creates new directory.

```
mkdir project-folder
```

13.1.2 `cd`

Changes current directory.

```
cd project-folder      # Enter directory
cd ..                  # Go up one level
cd ~                   # Go to home directory
```

13.2 Terminal Control Commands

13.2.1 `cls` (Command Prompt)

Clears Command Prompt screen.

```
cls
```

13.2.2 `clear` (PowerShell/Bash)

Clears terminal screen.

```
clear
```

13.2.3 `q`

Quits paginated output (git log, help pages).

```
q
```

14 Comprehensive Glossary

Branch	A separate line of development allowing developers to work on features independently without affecting the main codebase.
Clone	A local copy of a remote Git repository created on a developer's computer, including all history and branches.
Commit	A snapshot of the project's state at a specific point, with a message describing changes made.
Continuous Delivery (CD)	Automated software movement through development lifecycle, ensuring code can be released reliably anytime.
Continuous Integration (CI)	Practice where developers frequently integrate code changes into shared codebase, often multiple times daily.
Distributed Version Control System (DVCS)	System tracking code changes regardless of storage location. Each user has complete repository copy.
Fork	Copy of existing repository in user's GitHub account, allowing independent development without affecting original.
Git	Free, open-source distributed version control system under GNU General Public License. Enables local project maintenance worldwide.
GitHub	Web-hosted platform providing Git repository hosting and collaboration tools like pull requests and issue tracking.
GitHub Branches	Store all repository files, used to isolate code changes. Completed changes merge back to main branch.
GitLab	Complete DevOps platform offering Git repository hosting, source code management, and CI/CD tools in single application.
HEAD	Pointer to currently checked-out branch or commit. Represents current working position in repository.
Merge	Process combining changes from one branch into another, typically merging feature branch into main.
Origin	Default name for remote repository. Points to main remote location where code is pushed/pulled.
Pull Request	Process requesting review and approval of code changes before merging into main branch.
Repository	Data structure storing project files, folders, and version history. Configured for version control.
SSH Protocol	Secure method for remote login and communication, commonly used for authenticating Git operations.

Staging Area	Intermediate preparation zone where changes are prepared before committing to repository.
Upstream	Remote reference to original repository (used in fork workflows). Distinguished from personal fork (origin).
Version Control	System tracking file changes over time, allowing recovery of previous versions if errors occur.
Working Directory	Directory on local file system containing files and subdirectories associated with Git repository.

15 Quick Reference Guide

15.1 Essential Commands Summary

Setup & Configuration

```
git config --global user.name "Name"
git config --global user.email "email@example.com"
git init
```

Basic Workflow

```
git status
git add <file>              # Stage specific file
git add .                   # Stage all changes
git commit -m "message"
git push origin main
```

Branching

```
git branch                  # List branches
git branch <name>           # Create branch
git checkout <branch>       # Switch branch
git merge <branch>          # Merge branch
git branch -d <branch>      # Delete merged branch
git branch -D <branch>      # Force delete branch
```

Remote Operations

```
git clone <url>
git remote add origin <url>
git remote -v
git pull origin main
git push origin main
git fetch origin
```

History & Inspection

```
git log
git log --oneline
git log --oneline --all --graph
git show <sha-id>
git diff
```

Undoing Changes

```
git restore --staged <file>  # Unstage file
git revert <sha-id>          # Revert commit
git checkout <sha-id>        # View old commit
```

15.2 Common Workflows

15.2.1 Starting New Project

1. Create directory: `mkdir project-name`
2. Navigate: `cd project-name`
3. Initialize: `git init`
4. Create files and make changes
5. Stage: `git add .`
6. Commit: `git commit -m "Initial commit"`
7. Add remote: `git remote add origin <url>`
8. Push: `git push origin main`

15.2.2 Contributing to Existing Project

1. Fork repository on GitHub
2. Clone your fork: `git clone <your-fork-url>`
3. Add upstream: `git remote add upstream <original-url>`
4. Create branch: `git checkout -b feature-name`
5. Make changes and commit
6. Push to fork: `git push origin feature-name`
7. Create pull request on GitHub

15.2.3 Keeping Fork Updated

1. Fetch upstream: `git fetch upstream`
2. Switch to main: `git checkout main`
3. Merge upstream: `git merge upstream/main`
4. Push to fork: `git push origin main`

15.3 Best Practices

- **Commit often** with meaningful messages
- **Pull before push** to avoid conflicts
- **Use branches** for features and fixes
- **Review changes** before committing
- **Write clear commit messages** (50 char summary, detailed description)
- **Keep commits atomic** (one logical change per commit)
- **Use .gitignore** to exclude unnecessary files

- **Test before committing** to maintain working code
- **Avoid force push** on shared branches
- **Document your code** and repository

15.4 Commit Message Guidelines

Good Commit Messages

Format:

```
Short summary (50 chars or less)

More detailed explanation if needed. Wrap at 72 characters.
Explain what and why, not how.

- Bullet points okay
- Use present tense: "Add feature" not "Added feature"
```

Examples:

- `Add user authentication system`
- `Fix bug in payment processing`
- `Refactor database connection logic`
- `Update README with installation steps`

16 Conclusion

Git is an essential tool in modern software development that enables:

- **Effective version control** with complete history tracking
- **Smooth collaboration** among distributed teams
- **Non-linear development** through branching and merging
- **Code quality** through review processes
- **Project management** via platforms like GitHub and GitLab

Mastering Git requires practice and understanding of its core concepts. This reference guide provides the foundation needed to work effectively with Git in professional software development environments.

Remember

- Git is distributed - every copy is a full backup
- Commits are permanent - think before you commit
- Branches are cheap - use them liberally
- Communication is key - write clear commit messages
- When in doubt, consult `git --help`

16.1 Additional Resources

- Official Git Documentation: `https://git-scm.com/doc`
- GitHub Guides: `https://guides.github.com`
- GitLab Documentation: `https://docs.gitlab.com`
- Interactive Git Tutorial: `https://learngitbranching.js.org`
- Git Cheat Sheet: `https://education.github.com/git-cheat-sheet-education.pdf`

End of Git & GitHub Complete Reference Guide