Git for DevOps

Mastering Git Commands, Workflows & Real-Time Scenarios

Author: Sainath Shivaji Mitalakar

Edition 2025

"A practical handbook for DevOps professionals"

In today's DevOps era, velocity and collaboration separate the teams that ship from the teams that stall. This eBook is not a dry command reference — it's a battle-tested playbook for engineers who move fast and stay safe. Packed with terminal snippets, workflow blueprints, and real-world scenarios, it translates complex Git mechanics into crisp actions you can use the same day. Whether you're starting out or leveling up, consider this your day-to-day guide to shipping with confidence.

What you will get from this book:

- Concise, copy-pasteable command snippets for real tasks.
- Clear walkthroughs of branching, merging, rebasing, and recovery.
- Practical workflows tailored to DevOps, CI/CD and GitOps.
- Troubleshooting tips and safe recovery techniques for the inevitable mistakes.

1 What is Git? Why DevOps Needs It

Git is a *distributed version control system (DVCS)* created by Linus Torvalds in 2005. It tracks changes in source code, enables collaboration across teams, and ensures every developer has a full copy of the project history.

1.1 Centralized vs Distributed Version Control

Centralized VCS (e.g., SVN)

- Single central server stores all versions.
- If server goes down, no commits are possible.
- Developers rely on network connectivity.

Distributed VCS (e.g., Git)

- Every developer has a complete history of the repo.
- Work offline, commit locally, sync later.
- Faster, safer, and highly scalable for teams.

1.2 Why Git Matters in DevOps

In a DevOps pipeline, speed, collaboration, and traceability are non-negotiable. Git provides:

- **Collaboration:** Teams work on features simultaneously without blocking each other.
- **Traceability:** Every commit is logged with author, timestamp, and changes.
- **Rollback:** Easy recovery from mistakes with `git revert` or `git reset`.
- **Automation:** Git triggers CI/CD pipelines (Jenkins, GitHub Actions, GitLab CI).
- **GitOps:** Infrastructure and deployments managed declaratively through Git.

1.3 Check if Git is Installed

Verify Git installation:

```
$ git --version
```

Output:

```
git version 2.43.0
```

Tip: Always keep Git updated to the latest stable version for new features and security fixes.

2 Installing Git

Git can be installed on all major operating systems. Below are the steps for Linux, macOS, and Windows.

2.1 Linux Installation

For Debian/Ubuntu:

```
$ sudo apt update
$ sudo apt install git -y
```

For RHEL/CentOS/Fedora:

```
$ sudo dnf install git -y
```

Verify installation:

```
$ git --version
git version 2.43.0
```

2.2 macOS Installation

Option 1: Install via Homebrew

```
$ brew install git
```

Option 2: Install Xcode Command Line Tools

```
$ xcode-select --install
```

Tip: Homebrew is the preferred method for flexibility and easy upgrades.

2.3 Windows Installation

Step 1: Download Git for Windows Go to: `https://git-scm.com/download/win`
Step 2: Run the installer Choose recommended settings (Git Bash, PATH integration).
Step 3: Verify installation

```
> git --version
git version 2.43.0.windows.1
```

Note: On Windows, Git Bash provides a Unix-like terminal that makes Git usage easier for DevOps engineers.

3 First-time Git Configuration

After installing Git, it's important to configure your identity and preferences. These settings help track commits and improve your Git experience.

3.1 Set Your Identity

```
$ git config --global user.name "Your Name"
$ git config --global user.email "youremail@example.com"
```

Tip: Always use the same email ID linked to your GitHub/GitLab account for seamless commits.

3.2 Set Default Editor

```
# For Vim
$ git config --global core.editor "vim"

# For Nano
$ git config --global core.editor "nano"

# For VS Code
$ git config --global core.editor "code --wait"
```

Pro Tip: VS Code is a popular choice among DevOps engineers for editing commit messages and resolving merge conflicts.

3.3 Enable Helpful Colors

```
$ git config --global color.ui auto
```

Now Git highlights important information with colors in the terminal, making it easier to spot changes.

3.4 Check Your Configuration

```
$ git config --list
```

Example output:

```
user.name=Your Name
user.email=youremail@example.com
core.editor=code --wait
color.ui=auto
```

4 Creating Your First Git Repository

There are two main ways to start working with Git:

- Initialize a brand-new repository for a project.

- Clone an existing repository from a remote source.

4.1 Initialize a New Repository

```
$ mkdir myproject
$ cd myproject
$ git init
```

Initialized empty Git repository in /myproject/.git/

Tip: After `git init`, a hidden `.git/` folder is created. This folder stores all Git data — never delete it unless you want to remove version control.

4.2 Clone an Existing Repository

```
$ git clone https://github.com/example/project.git
```

Cloning into 'project'... remote: Enumerating objects: 42, done. remote: Counting objects: 100Receiving objects: 100

Pro Tip: Use `git clone --branch <branchname>` if you only need a specific branch.

4.3 Verify Repository

```
$ git status
```

On branch master
No commits yet
nothing to commit (create/copy files and use "git add" to track)

5 Branching in Git

Branching allows you to work on multiple features or fixes simultaneously without affecting the main codebase. Think of branches as parallel timelines in your project.

5.1 Creating Branches

```
$ git branch feature-login
```

This creates a new branch named 'feature-login' based on the current branch.

Tip: Name branches clearly based on the feature, bug, or task. Example: `feature-payment-api`, `bugfix-typo`, `hotfix-urgent`.

—

5.2 Listing Branches

```
$ git branch
```

```
* main feature-login
```

The asterisk (*) indicates the current branch you are on.

—

5.3 Switching Branches

Old method:

```
$ git checkout feature-login
```

Modern method:

```
$ git switch feature-login
```

```
Switched to branch 'feature-login'
```

Tip: Use `git switch` for clarity, especially when collaborating in DevOps pipelines.

—

6 Merging Branches

Merging combines changes from one branch into another. Typically, you merge a feature branch into the main branch.

6.1 Merge Example

1. Switch to the branch you want to merge into:

```
$ git switch main
```

2. Merge the feature branch:

```
$ git merge feature-login
```

```
Updating 1a2b3c4..5d6e7f8 Fast-forward app.py — 12 ++++++++++++ 1 file changed, 12 insertions(+)
```

—

6.2 Handling Merge Conflicts

Sometimes Git cannot merge automatically. Conflicts occur when the same lines in a file are modified in both branches.

Example conflict markers in a file:

```
<<<<<<< HEAD
print("Hello from main")
=======
print("Hello from feature-login")
>>>>>>> feature-login
```

Steps to resolve:

1. Open the conflicted file. 2. Decide which changes to keep. 3. Remove the conflict markers (‘iiiiiii‘, ‘=======‘, ‘¿¿¿¿¿¿¿‘). 4. Stage the resolved file:

```
$ git add app.py
```

5. Complete the merge with a commit:

```
$ git commit -m "Resolved merge conflict in app.py"
```

Tip: Always communicate with team members when resolving conflicts to ensure no important changes are lost. In DevOps workflows, resolving conflicts quickly is critical to avoid pipeline failures.

—

6.3 Deleting Branches After Merge

After merging a feature branch, it's good practice to delete it:

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

```
Deleted branch feature-login (was 5d6e7f8).
```

Tip: Use `-D` to force delete if the branch hasn't been merged yet, but only when sure.

7 Working with Remote Repositories

Remote repositories allow collaboration by hosting your Git repository on platforms like GitHub, GitLab, or Bitbucket. You can share changes, fetch updates, and maintain synchronization with your team.

7.1 Adding Remote Repositories

To connect your local repository to a remote repository:

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

```
This adds a remote repository named 'origin' pointing to your GitHub project.
```

Tip: Use descriptive names if you have multiple remotes. Example: `git remote add upstream <url>`.

—

7.2 Listing Remote Repositories

Check the configured remotes:

```
$ git remote -v
```

```
origin https://github.com/username/project.git (fetch) origin https://github.com/username/project.git (push)
```

Tip: Always verify your remote URLs before pushing to avoid accidental commits to the wrong repository.

—

7.3 Fetching Changes from Remote

Fetching downloads commits, files, and refs from a remote repository but does not merge them automatically:

```
$ git fetch origin
```

remote: Enumerating objects: 5, done. remote: Counting objects: 100remote: Compressing objects: 100Unpacking objects: 100

Tip: Use `git fetch` regularly to stay updated with remote changes without modifying your working directory.

—

7.4 Pulling Changes from Remote

Pulling fetches and merges remote changes into your current branch:

```
$ git pull origin main
```

Updating 1a2b3c4..5d6e7f8 Fast-forward app.py — 5 +++++ 1 file changed, 5 insertions(+)

Tip: Pull frequently before starting work to avoid merge conflicts in DevOps team environments.

—

7.5 Pushing Changes to Remote

After committing changes locally, push them to the remote repository:

```
$ git push origin main
```

Enumerating objects: 3, done. Counting objects: 100Writing objects: 100Total 3 (delta 0), reused 0 (delta 0) To https://github.com/username/project.git 1a2b3c4..5d6e7f8 main -¿ main

Tip: If you push for the first time, you may need to set upstream:

```
$ git push -u origin main
```

This links your local branch to the remote.

—

7.6 Tracking Branches with Remotes

To see which remote branch your local branch is tracking:

```
$ git branch -vv
```

```
* main 5d6e7f8 [origin/main] Merge login feature feature1 3a4b5c6 [origin/feature1] Work in progress
```

Tip: Tracking remote branches helps DevOps engineers understand which branch is synced and avoid accidental pushes or merges.

8 Advanced Git Commands

8.1 Tagging

Tags are used to mark specific points in history as important (usually releases).

```
$ git tag v1.0
```

```
This creates a lightweight tag named v1.0 at the current commit.
```

Annotated tag (recommended for releases):

```
$ git tag -a v1.0 -m "Release version 1.0"
```

```
Tagged commit with message "Release version 1.0"
```

Tip: Use annotated tags for release points in DevOps pipelines to provide metadata like author, date, and message.

—

8.2 Reverting Commits

Revert creates a new commit that undoes a previous commit without rewriting history:

```
$ git revert 5d6e7f8
```

```
[main 8f9g0h1] Revert "Added login feature" 1 file changed, 10 deletions(-)
```

Tip: Use `git revert` in shared repositories to safely undo changes without altering commit history.

—

8.3 Reset vs. Rebase

Reset: Moves the current branch pointer to a specific commit.

Soft reset (keeps changes staged):

```
$ git reset --soft HEAD~1
```

Hard reset (discards all changes):

```
$ git reset --hard HEAD~1
```

Rebase: Re-applies commits on top of another base branch to maintain a linear history:

```
$ git rebase main
```

Tip: Rebase before merging feature branches to keep history clean, but avoid rebasing public branches to prevent conflicts.

—

8.4 Cherry-Pick

Apply a specific commit from one branch into another:

```
$ git cherry-pick 3a4b5c6
```

```
[feature-login 9d0e1f2] Cherry-picked commit 3a4b5c6
```

Tip: Cherry-pick is useful in DevOps when you want to apply hotfixes or select features to another branch without merging all commits.

9 Real-Time Collaboration Scenarios

9.1 Feature Branches and Pull Requests

In a team environment, feature branches isolate work. Once completed, a pull request (PR) is created to merge into the main branch.

Tip: Always pull the latest main branch before creating a feature branch to reduce conflicts.

9.2 Handling Conflicts in a Team

When multiple developers work on the same files, conflicts may occur.

Steps to resolve:

1. Pull the latest changes:

```
$ git pull origin main
```

2. Resolve conflicts manually in the affected files. 3. Stage resolved files:

```
$ git add app.py
```

4. Commit the resolution:

```
$ git commit -m "Resolved merge conflicts in app.py"
```

Tip: Communicate with team members and document resolutions to maintain transparency in DevOps workflows.

—

9.3 Code Review Best Practices

- Always create pull requests for review. - Keep commits small and descriptive. - Review code for functionality, security, and maintainability. - Use CI/CD to run automated tests on PRs.

Tip: Integrate automated code quality tools (like SonarQube) in PR pipelines for DevOps teams to ensure robust production-ready code.

10 Git Hooks & Automation

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

10.1 Pre-Commit Hooks

Pre-commit hooks run before a commit is finalized. They are useful for code linting, formatting, and validation.

```
# Create a pre-commit hook
$ nano .git/hooks/pre-commit
```

```
#!/bin/sh
# Lint Python files before commit
flake8 *.py
if [ $? -ne 0 ]; then
  echo "Linting failed. Commit aborted."
  exit 1
fi
```

If code fails linting, the commit is rejected.

Tip: Use pre-commit hooks to maintain code quality in DevOps pipelines, preventing bad code from entering the repository.

—

10.2 Post-Commit Hooks

Post-commit hooks run after a commit is made. They are useful for notifications, updating logs, or triggering CI/CD pipelines.

```
# Create a post-commit hook
$ nano .git/hooks/post-commit
```

```
#!/bin/sh
# Notify team on Slack after commit
curl -X POST -H 'Content-type: application/json' \
--data '{"text":"New commit pushed to repository"}' \
https://hooks.slack.com/services/TOKEN
```

Tip: Post-commit hooks are powerful for DevOps teams to automate notifications and integrate Git with CI/CD tools.

—

10.3 Integrating Git with CI/CD Pipelines

Hooks can trigger automated pipelines in Jenkins, GitHub Actions, GitLab CI/CD, or other DevOps tools.

Example: Run tests automatically on commit $git commit -m "Update feature X" Pre-commit runs linting Post-commit triggers CI/CD pipeline$

Tip: Automating Git tasks ensures consistency, faster feedback, and improved collaboration in DevOps workflows.

11 Git Best Practices for DevOps

Following best practices ensures efficient collaboration, maintainable code, and smooth DevOps workflows.

11.1 Commit Message Conventions

- Use **imperative mood**: "Add feature X", not "Added feature X" - Include **ticket/issue IDs**: "Fix 123: Correct login bug" - Keep messages **concise yet descriptive**

Example commit message $git commit -m "Add user authentication module 45"$

Tip: Consistent commit messages make tracking changes easier in CI/CD and code review processes.

—

11.2 Branching Strategies

Git Flow: - Main (stable), Develop (integration), Feature, Release, Hotfix branches - Ideal for larger DevOps teams

GitHub Flow: - Simple model: main branch + feature branches - Pull request merges directly into main - Ideal for continuous deployment pipelines

Tip: Choose a branching strategy that fits your team size, release frequency, and DevOps workflow.

—

11.3 Repository Structure and Organization

- Keep **related code together** (e.g., /src, /tests, /docs) - Use **clear naming conventions** for folders and files - Avoid committing large binaries; use Git LFS if needed - Include README, LICENSE, and CONTRIBUTING files

Tip: A well-organized repository simplifies onboarding, collaboration, and automation in DevOps environments.

12 Git Troubleshooting

12.1 Detached HEAD State

A detached HEAD occurs when you checkout a specific commit instead of a branch.

$gitcheckout 5d6e7f8$

Note: You are in 'detached HEAD' state. Changes here will not affect any branch.

To recover and create a branch from detached HEAD $gitcheckout - bnew - feature$

Always attach to a branch to preserve work when in detached HEAD state.

—

12.2 Undoing Pushed Commits

Revert a commit safely $gitrevert < commit - hash >$

Creates a new commit that undoes changes from the specified commit.

Force push (use cautiously) $gitreset --hard < commit - hash >$ git push origin main –force

Avoid force-pushing on shared branches to prevent breaking teammates' work.

—

12.3 Recovering Deleted Branches

List all recent commits to find deleted branch $gitreflog$

Restore branch from commit hash $gitcheckout - brestored - branch < commit - hash >$

Git reflog is essential for recovering lost commits or branches.

—

12.4 Resolving Corrupted Repositories

Check repository integrity $gitfsck$

Checking objects: 100No problems detected

Re-clone repository if local repo is broken $gitclone < remote - url > new - repo$

Regularly back up your repository and use remotes for safe recovery.

13 Git Troubleshooting

13.1 Detached HEAD State

A detached HEAD occurs when you checkout a specific commit instead of a branch. Any commits made here are not attached to a branch unless you create one.

```
$ git checkout 5d6e7f8       # Checkout a specific commit (detached HEAD)
$ git checkout -b new-feature # Create a branch from detached HEAD to save work
```

13.2 Undoing Pushed Commits

Use ‘git revert‘ to safely undo commits in shared branches; ‘git reset‘ can be used for local history rewrite but requires caution when pushing.

```
$ git revert <commit-hash>          # Creates a new commit that undoes a previous co
$ git reset --hard <commit-hash>    # Reset branch to specific commit, discarding ch
$ git push origin main --force      # Force push rewritten history to remote
```

13.3 Recovering Deleted Branches

Recover lost branches using ‘git reflog‘, which logs all moves of HEAD.

```
$ git reflog                            # Shows the history of HEAD movements
$ git checkout -b restored-branch <commit-hash>  # Restore branch from previous comm
```

13.4 Resolving Corrupted Repositories

Check repo integrity and clone a fresh copy if needed.

```
$ git fsck                              # Verify repository integrity
$ git clone <remote-url> new-repo    # Re-clone repository if corruption detected
```

14 Git Cheat Sheets & Commands Reference

14.1 Repository Setup

```
$ git init                             # Initialize a new Git repository
$ git clone <url>                       # Clone a remote repository to local machine
$ git remote add origin <url>           # Add remote repository URL
$ git remote -v                         # List remote URLs
```

14.2 Configuration

```
$ git config --global user.name "Your Name"      # Set your Git username globally
$ git config --global user.email "you@example.com" # Set your email globally
$ git config --global core.editor nano           # Set default editor for Git commit
$ git config --global color.ui auto              # Enable colored Git output
$ git config --list                              # View all Git configuration settin
```

14.3 Staging and Committing

```
$ git status                # Show current branch status and changes
$ git add <file>            # Stage a specific file
$ git add .                 # Stage all changes
$ git commit -m "Message"   # Commit staged changes with a message
$ git commit -am "Message" # Stage and commit tracked files in one step
$ git diff                  # Show unstaged changes
$ git diff --staged         # Show staged changes
```

14.4 Branching and Merging

```
$ git branch                # List all local branches
$ git branch <name>         # Create a new branch
$ git checkout <name>       # Switch to branch
$ git switch <name>         # Alternative to checkout
$ git merge <branch>        # Merge a branch into current branch
$ git rebase <branch>       # Reapply commits on top of another branch
$ git branch -d <branch>    # Delete a branch safely (merged)
$ git branch -D <branch>    # Force delete a branch
```

14.5 Remote Repositories

```
$ git fetch origin          # Fetch changes from remote without merging
$ git pull origin main      # Fetch and merge changes from remote main branch
$ git push origin main      # Push local commits to remote main branch
$ git push origin --all     # Push all branches
$ git push origin --tags    # Push all tags
```

14.6 Tips for Fast DevOps Workflows

- Always pull before starting work.
- Keep feature branches short-lived.
- Commit frequently with descriptive messages.
- Use stash to temporarily save unfinished work.
- Automate tasks with hooks and CI/CD integration.
- Tag releases and use branches for hotfixes.
- Regularly backup repositories and push to remotes.
- Review changes using ‘git diff‘ before committing.

15 Git Security & Permissions

15.1 SSH Keys for Authentication

SSH keys allow secure, passwordless authentication with Git remotes.

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"   # Generate a new SSH key
$ eval "$(ssh-agent -s)"                              # Start the SSH agent
$ ssh-add ~/.ssh/id_ed25519                            # Add private key to agent
$ cat ~/.ssh/id_ed25519.pub                           # Display public key to add to
```

15.2 HTTPS vs SSH for Remotes

- **HTTPS**: Simple, may require username/password or token - **SSH**: Secure, key-based authentication, recommended for automation

```
$ git remote set-url origin git@github.com:user/repo.git   # Switch remote to SSH
$ git remote set-url origin https://github.com/user/repo.git # Switch remote to HTTP
```

15.3 Managing Repository Permissions

Assign read, write, or admin permissions to users in the remote repository (GitHub, GitLab, Bitbucket).

Tip: Use team-based permissions for better control in organizations.

15.4 Signing Commits

Signing commits ensures authenticity and integrity.

```
$ gpg --full-generate-key                                    # Generate a GPG key
$ git config --global user.signingkey <key-id>              # Configure Git to use this
$ git commit -S -m "Signed commit message"                  # Create a signed commit
$ git log --show-signature                                   # Verify signed commits
```

16 Git Workflows & DevOps Integration

16.1 Git Workflows

Different workflows help teams collaborate efficiently.

```
# Git Flow: Feature -> Develop -> Release -> Master
# GitHub Flow: Feature branches -> Pull Request -> Main branch
# Trunk-Based Development: Small short-lived branches merged to main/trunk
```

16.2 Feature Branching, Release Branching, Hotfixes

```
$ git checkout -b feature/login                    # Create a feature branch
$ git checkout -b release/v1.0                     # Create a release branch
$ git checkout -b hotfix/urgent-fix                # Create a hotfix branch
$ git merge feature/login                           # Merge feature to develop/release
```

16.3 Integrating Git with CI/CD Pipelines

Automate builds, tests, and deployments using Git triggers.

```
# Example Jenkinsfile trigger for push
pipeline {
    triggers { pollSCM('* * * * *') }
    stages { stage('Build') { steps { sh 'make build' } } }
}

# GitHub Actions example workflow
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm test
```

16.4 Automation Hooks for Deployments

Git hooks can automate tasks like tests, formatting, or deployments.

```
$ ls .git/hooks/                                  # List all available hooks
$ cp pre-commit.sample .git/hooks/pre-commit   # Enable pre-commit hook
$ chmod +x .git/hooks/pre-commit               # Make hook executable
```

Tip: Combine hooks with CI/CD pipelines to ensure quality and reduce manual errors.

17 Appendix & References

17.1 Glossary of Git Terms

- **HEAD:** Points to the current commit in your working branch.
- **Staging Area (Index):** Temporary area where changes are added before committing.
- **Commit:** Snapshot of your project at a point in time.
- **Branch:** Parallel version of the repository for features, fixes, or experiments.

- **Remote:** A version of your repository hosted on a server (GitHub, GitLab, Bitbucket).
- **Merge:** Combining changes from one branch into another.
- **Rebase:** Reapply commits on top of another branch to create a linear history.
- **Tag:** Named reference to a specific commit, often used for releases.
- **Cherry-Pick:** Apply a commit from one branch onto another.
- **Detached HEAD:** State when HEAD points directly to a commit instead of a branch.

17.2 Quick Reference Tables for Commands

Repository Setup:

```
$ git init                  # Initialize a new repository
$ git clone <url>           # Clone an existing repository
$ git remote add origin <url> # Add remote repository
$ git remote -v             # Show remote URLs
```

Staging & Committing:

```
$ git status                # Show modified files and staging area
$ git add <file>            # Stage a specific file
$ git add .                 # Stage all changes
$ git commit -m "Message" # Commit staged changes
$ git commit -am "Message" # Stage and commit tracked files
```

Branching & Merging:

```
$ git branch                # List branches
$ git branch <name>         # Create new branch
$ git checkout <name>       # Switch branch
$ git switch <name>         # Alternative switch branch
$ git merge <branch>        # Merge another branch
$ git rebase <branch>       # Reapply commits onto branch
```

Remotes:

```
$ git fetch origin        # Fetch changes from remote
$ git pull origin main    # Fetch and merge changes
$ git push origin main    # Push local commits to remote
$ git push origin --tags  # Push all tags
```

17.3 Resources and Further Learning

- Official Git Documentation – Comprehensive reference
- GitHub Guides – Beginner to advanced tutorials
- Pro Git Book – Free eBook, excellent examples
- Atlassian Git Tutorials – Visual and practical guides
- Cheat Sheets: GitHub PDF Cheat Sheet Tower Git Cheat Sheet Online

17.4 Acknowledgments and Notes

Note: This ebook is designed to make you a confident Git and DevOps practitioner. The examples and workflows are based on real-world DevOps scenarios. Remember, practice is key: the more you work with Git daily, the more fluent you become. Stay curious, keep experimenting, and share knowledge with your team.

Final Tip: Combine the commands and workflows in this ebook with CI/CD pipelines, hooks, and best practices to maximize productivity and minimize errors in real DevOps environments.

18 Top 20 Git DevOps Interview Questions

1. What is Git and why is it used in DevOps?
2. Explain the difference between Git and SVN.
3. What is a branch in Git and why is it important?
4. How do you resolve merge conflicts in Git?
5. What is the difference between git pull and git fetch?
6. Explain git rebase and when to use it.
7. What is a detached HEAD in Git?
8. How do you undo a commit that has already been pushed?
9. What are Git hooks and how are they used?
10. How do you manage multiple remote repositories?
11. Explain Git stash and when to use it.
12. How do you sign a commit with GPG in Git?
13. What is Git Flow, GitHub Flow, and trunk-based development?
14. How do you revert a commit without losing history?
15. What are tags in Git and how are they different from branches?
16. How do you recover a deleted branch?
17. What are best practices for commit messages?
18. How do you check repository integrity and fix corruption?
19. How do you integrate Git with CI/CD pipelines?
20. Explain the differences between reset, revert, and checkout.
21. What is the difference between a fast-forward merge and a three-way merge?
22. How do you squash multiple commits into one?
23. Explain the difference between Git rebase and Git merge.
24. How do you handle large binary files in Git repositories?
25. What are the advantages of using Git tags versus branches for releases?
26. How do you revert a merge commit?
27. What is Git reflog and how is it useful?
28. How do you handle conflicts when rebasing a branch?
29. Explain shallow clones and when to use them.
30. How can you securely store Git credentials for automation or CI/CD pipelines?

19 Git Cheat Sheet: 60+ Most Useful Commands

```
# --- Configuration ---
$ git config --global user.name "Your Name"          # Set username
$ git config --global user.email "you@example.com"   # Set email
$ git config --global core.editor "code --wait"      # Set editor
$ git config --list                                  # Show configs

# --- Repository ---
$ git init                                           # Initialize repo
$ git clone <url>                                    # Clone repo

# --- Status & Logs ---
$ git status                                         # Show changes
$ git log                                            # Full commit history
$ git log --oneline                                  # Short commit history
$ git log --graph                                    # Commit graph
$ git show <commit>                                  # Show commit details
$ git diff                                           # Show unstaged changes
$ git diff --staged                                  # Show staged changes

# --- Staging & Committing ---
$ git add <file>                                     # Stage file
$ git add .                                          # Stage all
$ git commit -m "Message"                            # Commit changes
$ git commit -am "Message"                           # Stage & commit tracked files
$ git commit --amend                                 # Amend last commit

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

# --- Remote Repositories ---
$ git remote -v                                      # Show remotes
$ git remote add origin <url>                        # Add remote
```

```
$ git fetch                                    # Fetch from remote
$ git pull                                     # Fetch + merge
$ git push                                     # Push commits
$ git push origin --delete <branch>            # Delete remote branch

# --- Undoing Changes ---
$ git restore <file>                           # Discard changes
$ git reset <file>                             # Unstage file
$ git reset --soft <commit>                    # Reset to commit, keep staged
$ git reset --hard <commit>                    # Reset to commit, discard changes
$ git revert <commit>                          # Undo commit safely
$ git clean -fd                                # Remove untracked files

# --- Tags ---
$ git tag                                      # List tags
$ git tag -a v1.0 -m "Message"                 # Annotated tag
$ git push origin --tags                       # Push tags

# --- Stash ---
$ git stash                                    # Save changes
$ git stash list                               # List stashes
$ git stash apply                              # Apply stash
$ git stash drop <stash>                       # Delete stash

# --- Cherry-pick & Rebase ---
$ git cherry-pick <commit>                     # Apply specific commit
$ git rebase <branch>                          # Reapply commits
$ git rebase --abort                           # Abort rebase

# --- Ignoring Files ---
$ echo "*.log" >> .gitignore                   # Ignore log files
$ git rm --cached <file>                       # Remove from repo, keep locally

# --- Hooks & Automation ---
# .git/hooks/pre-commit                          # Pre-commit script

# --- SSH & Authentication ---
$ ssh-keygen -t rsa -b 4096 -C "you@example.com"   # Generate SSH key
$ ssh-add ~/.ssh/id_rsa                             # Add SSH key
```

```
# --- Inspection ---
$ git reflog                                    # Reference logs
$ git fsck                                      # Repository health check
$ git blame <file>                              # Show last modifier per line

# --- Miscellaneous ---
$ git shortlog                                  # Commits per author
$ git describe                                  # Describe commit with tag
$ git remote show origin                         # Remote details
$ git branch -r                                 # List remote branches
$ git branch -a                                 # List all branches
$ git archive -o latest.zip HEAD                # Export repo as zip
$ git bisect start                              # Start bug search
$ git bisect good <commit>                      # Mark good commit
$ git bisect bad <commit>                       # Mark bad commit
$ git bisect reset                               # End bisect
```

Thank You for Reading!

"DevOps is not just a set of tools, it's a culture of collaboration, learning, and delivering excellence continuously."

APJ Abdul Kalam once said:

"Excellence happens not by accident. It is a process of continuous improvement and learning."

Key Takeaways:

- Commit small, meaningful changes frequently.
- Automate repetitive tasks for faster, reliable delivery.
- Branch, merge, and collaborate smartly.
- Use CI/CD pipelines to maintain quality.
- Keep learning — DevOps is a continuous journey.

Connect with me to share ideas, learn together, and grow as DevOps professionals!

https://www.linkedin.com/in/sainathmitalakar/

Sainath Shivaji Mitalakar

Keep exploring, keep automating, and keep innovating. The journey never ends.