Devops Shack

WHY EVERY DEVOPS ENGINEER MUST MASTER Git

By Devops Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Why Every DevOps Engineer Must Master Git

Table of Contents

- Adding a remote repository (git remote add origin)
- Pushing changes (git push)
- Pulling updates (git pull)

7. **Undoing and Reverting Changes**
   - Undoing last commit (git reset --soft, git reset --hard)
   - Removing files from staging (git reset)
   - Reverting a commit (git revert)
8. **Stashing Changes**
   - Temporarily saving changes (git stash)
   - Viewing and applying stashed changes (git stash list, git stash pop)
9. **Git Best Practices**
   - Writing meaningful commit messages
   - Keeping commits small and focused
   - Using branches for new features
10. **Advanced Git Tips & Troubleshooting**

- Viewing commit history (git log, git reflog)
- Fixing merge conflicts
- Recovering lost commits

1. Introduction to Git

What is Git?

Git is a **distributed version control system (VCS)** that helps developers track changes in their code, collaborate efficiently, and manage different versions of a project. Unlike traditional version control systems, Git allows multiple people to work on the same codebase without conflicts.

Why Use Git?

- **Tracks Changes** – Keeps a history of all modifications.
- **Collaboration** – Multiple developers can work on the same project.
- **Branching & Merging** – Create separate workspaces without affecting the main project.
- **Backup & Recovery** – Easy to revert to previous versions.
- **Works Offline** – No need for an internet connection to work locally.

Git vs. GitHub

- **Git** – A tool for version control that runs on your local system.
- **GitHub** – A cloud-based platform that hosts Git repositories for collaboration and backup.

Basic Git Workflow

A typical Git workflow follows these steps:

1. **Initialize a repository** – Start tracking a project with git init or clone an existing one with git clone.
2. **Make changes** – Modify files in your project.
3. **Stage changes** – Add updated files to the staging area using git add.
4. **Commit changes** – Save changes to Git's history with git commit -m "message".
5. **Push changes** – Upload changes to a remote repository (git push).
6. **Pull updates** – Get the latest updates from a remote repository (git pull).

Key Git Commands for This Section

```
git init          # Initialize a new Git repository
git clone <url>   # Clone an existing repository
git add <file>    # Add a file to the staging area
git commit -m "message"  # Commit changes with a message
git push origin main  # Push changes to remote repository
git pull origin main  # Pull the latest changes from remote
```

Would you like any additional explanations or examples before moving to the

1. Introduction to Git (Expanded Explanation &

Examples) What is Git? (In Simple Terms)

Imagine you're working on a project—whether it's coding, writing, or designing. You make changes, but then realize you need to go back to an older version. Without Git, you'd have to manually save copies like:

☑ project_v1 → project_v2_final → project_v2_final_final → project_v2_final_FINAL

With Git, you don't need to do this. Git tracks changes automatically, so you can **switch between versions anytime without losing anything.**

Why Use Git? (Real-World Benefits)

1. **Tracks Everything Automatically**
    - You'll never lose work. If something breaks, you can always go back.
2. **Team Collaboration is Easy**
    - Instead of sending files over email, multiple developers can work on the same project at once.
3. **Branches Keep Work Organized**
    - You can create a separate branch to test a new feature without affecting the main project.

4. **Works Offline**
    - Unlike other version control systems, Git lets you work without an internet connection.

Git vs. GitHub (What's the Difference?)

Feature	Git	GitHub
What it is?	A tool that manages version control on your local system.	A platform that stores Git repositories online for collaboration.
Where it runs?	On your local computer.	On the web (GitHub, GitLab, Bitbucket, etc.).
Main Use?	Tracks and manages code versions.	Enables team collaboration, issue tracking, and pull requests.
Example?	git init starts tracking your project.	git push sends your project to GitHub.

Think of **Git** as your personal notebook where you track changes and **GitHub** as a shared online workspace where everyone can access and collaborate.

Basic Git Workflow (Step-by-Step Example)

Imagine you're working on a website project, and you want to track changes using Git.

Step 1: Initialize a Git Repository

Start tracking a new project by running:

```
git init
```

This creates a hidden .git folder that stores the history of your project.

Step 2: Create & Modify Files

Let's create a simple HTML file:

```
echo "<h1>Hello, Git!</h1>" > index.html
```

Step 3: Check the Status of Your Repository

Git keeps track of changes. To see the status of your files:

```
git status
```

- **Expected Output:**

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

This means Git sees the file but isn't tracking it yet.

Step 4: Add the File to Staging

To tell Git to start tracking index.html:

```
git add index.html
```

Now, index.html is in the **staging area**, meaning it's ready to be committed.

Step 5: Commit the Changes

A commit is like a "snapshot" of your project at a certain point. To save the staged file:

```
git commit -m "Initial commit: Added index.html"
```

- **Expected Output:**

```
[main (root-commit) abc1234] Initial commit: Added index.html
 1 file changed, 1 insertion(+)
```

Now, Git has officially saved this version.

Step 6: View Commit History

To see a list of all saved

git log --oneline

- **Expected Output:**

abc1234 Initial commit: Added index.html

Each commit has a **unique ID (hash)** that helps track changes.

Step 7: Pushing Changes to GitHub (Optional for Teams & Backup)

If you want to save your work online, you can push it to GitHub.

First, connect your local project to a remote repository:

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

Then, push the changes to GitHub:

git push -u origin main

- Now your code is safely stored online!

Quick Recap: Essential Commands from This Section

Command	Description
git init	Initializes a new Git repository.
git status	Shows the current state of files.
git add <file>	Stages a file for commit.
git commit -m "message"	Saves changes with a description.
git log --oneline	Shows commit history in a single line.
git remote add origin <url>	Connects a local repo to GitHub.
git push origin main	Pushes local changes to GitHub.

2. Setting Up Git

Before using Git, you need to install and configure it on your system. This section covers:

☑ Installing Git on different operating systems
☑ Configuring Git with your name and email
☑ Verifying installation

Installing Git

On Windows

1. Download the latest Git for Windows from git-scm.com.
2. Run the installer and follow the default settings.
3. Open **Git Bash** (installed with Git).

On macOS

- Using Homebrew

(recommended): brew install git

- Verify installation:

git --version

On Linux

- Ubuntu/Debian:

sudo apt update && sudo apt install git

- Fedora:

sudo dnf install

git

- Verify installation:

git --version

Configuring Git

After installing Git, configure it with your **name** and **email** (required for commits).

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

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

- **Example:**

git config --global user.name "John Doe"

git config --global user.email "john@example.com"

To check your current Git settings:

git config --list

- **Expected Output:**

user.name=John Doe

user.email=john@example.com

Setting Up a Default Text Editor

By default, Git uses Vim for commit messages, but you can set your preferred editor:

- **VS Code** (Recommended)

git config --global core.editor "code --wait"

- **Nano (Simple & Easy to Use)**

git config --global core.editor "nano"

- **Vim (For Advanced Users)**

git config --global core.editor "vim"

Setting Up Git Credential Cache (Optional)

If you're using GitHub or any remote repository, Git may prompt for your username/password each time you push or pull changes. You can cache your credentials for convenience:

git config --global credential.helper cache

This stores your credentials temporarily so you don't have to enter them every time.

Verifying Git Setup

Run the following command to ensure everything is set up correctly:

git config --list

You should see your name, email, and editor settings.

Quick Recap: Essential Commands from This Section

Command	Description
git --version	Checks if Git is installed.
git config --global user.name "Your Name"	Sets your Git username.
git config --global user.email "your.email@example.com"	Sets your Git email.
git config --list	Displays current Git settings.
git config --global core.editor "code --wait"	Sets the default text editor for Git.
git config --global credential.helper cache	Caches Git credentials for easier access.

3. Initializing and Cloning Repositories

This section covers:

☑ How to create a new Git repository (git init)
☑ How to clone an existing repository (git clone)

Creating a New Git Repository

If you're starting a new project, you need to **initialize** a Git repository.

Step 1: Navigate to Your Project Directory

Open the terminal and go to your project folder:

```
cd path/to/your/project
```

Example:

```
cd Documents/MyProject
```

Step 2: Initialize Git

Run:

```
git init
```

- **Expected Output:**

```
Initialized empty Git repository in /path/to/your/project/.git/
```

This creates a hidden .git folder inside your project, where Git will track changes.

Step 3: Verify Initialization

To check if Git is tracking the repository:

```
git status
```

Since no files are tracked yet, you'll see:

```
On branch
main No
commits yet
```

Untracked files:

(use "git add <file>..." to include in what will be committed)

Cloning an Existing Repository

Instead of starting from scratch, you can copy an existing project from GitHub or another Git server using git clone.

Step 1: Find the Repository URL

On GitHub, GitLab, or Bitbucket, go to the project and copy its **HTTPS or SSH URL**. Example:

https://github.com/user/repository.git

Step 2: Clone the Repository

Run the following command:

git clone <repository-url>

Example:

git clone https://github.com/user/repository.git

- **Expected Output:**

Cloning into 'repository'...

remote: Enumerating objects: 100, done.

remote: Counting objects: 100% (100/100), done.

Receiving objects: 100% (100/100), 500 KiB | 1.2 MiB/s, done.

This will create a new folder with the project files inside.

Step 3: Navigate to the Cloned Repository

Move into the cloned folder:

cd repository

Step 4: Check Remote Repository

To confirm that Git is connected to the remote repository:

git remote -v

- **Expected Output:**

origin https://github.com/user/repository.git (fetch)

origin https://github.com/user/repository.git (push)

Quick Recap: Essential Commands from This Section

Command	Description
git init	Initializes a new Git repository in a project folder.
git clone <repository-url>	Copies an existing repository to your local machine.
git status	Shows the status of the working directory.
git remote -v	Lists the remote repositories linked to your project.

4. Understanding Git Staging and Committing

In this section, we'll cover:

- ☑ How Git tracks changes (Working Directory → Staging → Commit)
- ☑ Adding files to staging (git add)
- ☑ Committing changes (git commit)
- ☑ Viewing commit history (git log)

Understanding Git's Three States

Git operates in three main areas:

1. **Working Directory** – Where you modify files (untracked/modified).
2. **Staging Area** – Files prepared for the next commit.
3. **Repository (Commits)** – Where committed changes are stored permanently.

- ◇ **Flow Diagram:**

[Working Directory] → git add → [Staging Area] → git commit → [Repository]

Adding Files to Staging

After modifying or creating new files, Git **does not automatically track changes**. You need to add them to the **staging area**.

Step 1: Check the Current Status

```
git status
```

- ◇ **Expected Output (if you added a new file index.html)**

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

Step 2: Add a Single File to Staging

```
git add index.html
```

Step 3: Add Multiple Files to Staging

```
git add file1.txt file2.txt
```

Step 4: Add All Files to Staging

```
git add .
```

Step 5: Verify Staging Status

```
git status
```

- **Expected Output:**

```
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)
       new file:  index.html
```

Committing Changes

A **commit** permanently saves the staged changes into the repository. Think of it as a **checkpoint** in your project.

Step 1: Commit Changes with a Message

```
git commit -m "Added homepage"
```

- **Expected Output:**

```
[main abc1234] Added homepage
1 file changed, 10 insertions(+)
 create mode 100644 index.html
```

Step 2: Commit All Staged Files in One Command

```
git commit -a -m "Updated multiple files"
```

⚠ This command **only works for modified files** that were previously committed. New files still need git add.

Viewing Commit History

To check past commits:

Step 1: View Commit History (Simple View)

```
git log --oneline
```

- **Expected Output:**

```
abc1234 Added homepage
def5678 Fixed navbar issue
```

Step 2: View Full Commit History

```
git log
```

- This will show details like author, date, and commit message.

Undoing Staged Changes

Sometimes, you add files to staging but want to remove them **before committing**.

Step 1: Remove a File from Staging

```
git reset HEAD index.html
```

- The file will move back to the **working directory**, meaning it's no longer staged.

Step 2: Remove All Staged Files

```
git reset
```

Quick Recap: Essential Commands from This Section

Command	Description
git add <file>	Adds a specific file to the staging area.
git add .	Stages all modified and new files.
git status	Shows which files are staged or untracked.
git commit -m "message"	Commits staged changes with a message.
git log --oneline	Shows a short commit history.

Command	Description
git reset HEAD <file>	Removes a file from the staging area before committing.

5. Working with Branches in Git

In this section, we'll cover:

- ☑ What branches are and why they are useful
- ☑ Creating and switching branches (git branch, git checkout, git switch)
- ☑ Merging branches (git merge)
- ☑ Deleting branches (git branch -d)

What is a Git Branch?

A branch in Git is like a separate line of development. It allows you to work on new features **without affecting the main project**.

- ◇ **Example Workflow:**

```
(main) ──────────┐
                 ├── (feature-branch) ── New Feature
                 └── (bugfix-branch) ── Bug Fix
```

By default, Git starts with a branch called **main** or **master**.

Creating and Switching Branches

Step 1: View Existing Branches

```
git branch
```

- ◇ **Expected Output (if on main branch):**

```
* main
```

The asterisk (*) shows the current active branch.

Step 2: Create a New Branch

```
git branch feature-1
```

This creates a branch named feature-1 but **does not switch to it**.

Step 3: Switch to the New Branch

- Using checkout (older method):

git checkout feature-1

- Using switch

(recommended): git switch feature-1

◇ **Expected Output:**

Switched to branch 'feature-1'

Step 4: Create and Switch in One Command

git checkout -b feature-1 OR

git switch -c feature-1

Merging Branches

Once a feature is complete, merge it into the main branch.

Step 1: Switch to the main Branch

git switch main

Step 2: Merge Another Branch into main

git merge feature-1

◇ **Expected Output (if no conflicts):**

Updating abc1234..def5678

Fast-forward

new file: feature.txt

Handling Merge Conflicts

If the same file is edited in both branches, Git **cannot automatically merge** them.

◇ **Example Conflict Message:**

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

How to Fix a Merge Conflict

1. Open the conflicting file in a text editor.
2. Git marks conflicts like this:

<<<<<<< HEAD

Code from main branch

=======

Code from feature-1 branch

>>>>>>> feature-1

3. **Manually edit the file** to keep the correct version.
4. Add the resolved

file: git add index.html

5. Commit the merge:

git commit -m "Resolved merge conflict"

Deleting Branches

Step 1: Delete a Local Branch

Once merged, you can delete the branch:

git branch -d feature-1

- If the branch is not merged yet, Git will warn you. To force delete:

git branch -D feature-1

Step 2: Delete a Remote Branch

git push origin --delete feature-1

Quick Recap: Essential Commands from This Section

Command	Description
git branch	Lists all local branches.
git branch <branch-name>	Creates a new branch.
git switch <branch-name>	Switches to a branch (recommended).
git checkout <branch-name>	Switches to a branch (older method).
git checkout -b <branch-name>	Creates and switches to a new branch.
git merge <branch-name>	Merges a branch into the current branch.
git branch -d <branch-name>	Deletes a merged branch.
git push origin --delete <branch-name>	Deletes a remote branch.

6. Working with Remote Repositories

In this section, we'll cover:

- ☑ Connecting a local repository to a remote repository
- ☑ Pushing and pulling changes (git push, git pull)
- ☑ Fetching updates without merging (git fetch)
- ☑ Working with multiple collaborators

Adding a Remote Repository

Before pushing changes, you need to connect your local repository to a remote one (e.g., GitHub, GitLab, or Bitbucket).

Step 1: Check Existing Remote Repositories

```
git remote -v
```

- **Expected Output (if no remotes are set yet):**

```
(no output)
```

Step 2: Add a Remote Repository

```
git remote add origin https://github.com/user/repository.git
```

- Replace origin with any name if needed, but origin is the default.

Step 3: Verify the Remote Repository

```
git remote -v
```

- **Expected Output:**

```
origin  https://github.com/user/repository.git (fetch)
origin  https://github.com/user/repository.git (push)
```

Pushing Changes to a Remote Repository

Step 1: Push the First Commit (Set Upstream Branch)

If this is your first time pushing, run:

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

- The -u flag links your local branch with the remote

Step 2: Push Subsequent Commits

Once the upstream is set, simply use:

```
git push
```

- If working with another branch:

```
git push origin feature-branch
```

Pulling Changes from a Remote Repository

Step 1: Fetch and Merge Latest Changes

To get the latest updates from the remote repository:

```
git pull origin main
```

- This **fetches** new commits and **merges** them into your local branch.

Step 2: Pull Without Merging (Fetch Only)

To see updates without merging:

```
git fetch origin
```

- This downloads updates but **does not merge them**.

Cloning a Remote Repository

If you need to copy an entire repository from GitHub:

```
git clone https://github.com/user/repository.git
```

- This downloads the repository and sets origin as the remote.

Working with Multiple Collaborators

Step 1: Pull the Latest Changes Before Working

Before making new changes, always update your local branch:

```
git pull origin main
```

Step 2: Resolve Conflicts (If Any)

If there are merge conflicts, Git will ask you to resolve them manually.
Steps:

1. Open the conflicting file and edit it.
2. Use git add <file> to stage the resolved file.
3. Commit the changes with git commit -m "Resolved conflict".
4. Push the

changes: git push origin

main

Quick Recap: Essential Commands from This Section

Command	Description
git remote -v	Lists remote repositories linked to the project.
git remote add origin <repo-url>	Adds a remote repository.
git push -u origin main	Pushes the main branch to the remote repository for the first time.
git push	Pushes commits to the remote repository.
git pull origin main	Fetches and merges changes from the remote repository.
git fetch origin	Fetches changes without merging.
git clone <repo-url>	Clones a remote repository.

7. Undoing Changes and Reverting Commits

In this section, we'll cover:

☑ Undoing local changes before committing
☑ Resetting and reverting commits (git reset, git revert)
☑ Checking out previous commits (git checkout <commit-hash>)
☑ Stashing changes (git stash)

Undoing Local Changes (Before Committing)

If you made changes to a file but haven't staged them yet, you can **discard** the modifications.

Step 1: Discard Changes in a Specific File

```
git checkout -- filename.txt
```

⚠ This command **cannot be undone** and will restore the file to the last committed version.

Step 2: Discard Changes in All Files

```
git checkout -- .
```

Unstaging Files (Before Committing)

If you staged a file using git add, but haven't committed it yet, you can **unstage** it.

```
git reset HEAD filename.txt
```

◇ This removes the file from **staging** but keeps the changes in the working directory.

To unstage all files:

```
git reset
```

Undoing a Commit (Before Pushing to Remote)

If you already committed changes but **haven't pushed yet**, you can undo last commit.

Step 1: Soft Reset (Undo Commit but Keep Changes Staged)

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

- This **undoes the last commit** but keeps your files **staged**.

Step 2: Mixed Reset (Undo Commit and Unstage Changes)

```
git reset --mixed HEAD~1
```

- This **undoes the commit and unstages** the changes, but keeps them in the working directory.

Step 3: Hard Reset (Undo Commit and Discard Changes)

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

⚠ **WARNING:** This will **delete** all changes and cannot be undone.

Reverting a Commit (After Pushing to Remote)

If you **already pushed a commit** and want to undo it without deleting history, use git revert.

```
git revert HEAD
```

- This creates a **new commit that reverses** the last commit.

To revert a specific commit:

```
git revert <commit-hash>
```

- Use git log --oneline to find the commit hash.

Checking Out a Previous Commit

To temporarily view an old version of the project:

```
git checkout <commit-hash>
```

- Use git log --oneline to find the commit hash.

To return to the latest commit:

```
git switch main
```

Stashing Changes (Save Changes Temporarily)

If you need to **switch branches** but don't want to commit yet, you can stash changes.

Step 1: Save Changes in a Stash

```
git stash
```

- This saves your changes and restores a clean working directory.

Step 2: View Stashed Changes

```
git stash list
```

Step 3: Apply the Last Stash

```
git stash apply
```

- This restores the last stashed changes but **keeps them in stash**.

Step 4: Apply and Remove the Stash

```
git stash pop
```

Step 5: Remove All Stashes

```
git stash clear
```

Quick Recap: Essential Commands from This Section

Command	Description
git checkout -- <file>	Discards local changes in a file.
git reset HEAD <file>	Unstages a file.
git reset --soft HEAD~1	Undo last commit but keep files staged.
git reset --mixed HEAD~1	Undo last commit and unstage files.
git reset --hard HEAD~1	Undo commit and delete changes permanently.

Command	Description
git revert HEAD	Creates a new commit that undoes the last commit.
git checkout <commit- hash>	Temporarily switch to a previous commit.
git stash	Save current changes temporarily.
git stash pop	Restore last stashed changes and remove the stash.

8. Git Log, Aliases, and Advanced Tips

In this section, we'll cover:

☑ Viewing commit history with git log
☑ Using aliases for efficiency
☑ Advanced Git tips for productivity

Viewing Commit History

Git keeps track of every commit made to a repository. You can inspect the history using git log.

Basic Log Command

git log

- This displays a list of commits, including:
  - Commit hash
  - Author name
  - Commit date
  - Commit

message **View Logs in**

One Line git log --

oneline

- **Example Output:**

a1b2c3d Fix login page bug

e4f5g6h Add user authentication

i7j8k9l Initial commit

This makes it easier to scan through commit history quickly.

View Logs with Graph

git log --oneline --graph --all

- This helps visualize branch history.

Filtering Logs

- **Show commits by a specific author:**

git log --author="John Doe"

- **Search for a keyword in commit messages:**

git log --grep="bugfix"

- **Show commits within a date range:**

git log --since="2024-01-01" --until="2024-03-01"

Using Git Aliases

Git allows you to create shortcuts for frequently used commands.

Step 1: Set Up Aliases

git config --global alias.st status

git config --global alias.co checkout

git config --global alias.br branch

git config --global alias.cm "commit -m"

git config --global alias.hist "log --oneline --graph --all --decorate"

Step 2: Use the Aliases

- Instead of git status, just

type: git st

- Instead of git checkout,

type: git co <branch>

- Instead of git log --oneline --graph --all,

type: git hist

Advanced Git Tips for Productivity

1. View Last Commit Details

git show

- This displays the last commit's changes.

2. Find Who Changed a Line in a File

```
git blame filename.txt
```

- This shows the author and commit hash for each line in a file.

3. Restore a Deleted File

If you accidentally delete a file and haven't committed yet:

```
git checkout -- filename.txt
```

If the file was deleted in a commit:

```
git checkout HEAD~1 filename.txt
```

4. Amend the Last Commit

```
git commit --amend -m "Updated commit message"
```

- This changes the last commit message **without creating a new commit**.

5. Clean Untracked Files

If you want to remove untracked files (files not in Git):

```
git clean -f
```

- To remove untracked directories:

```
git clean -fd
```

Quick Recap: Essential Commands from This Section

Command	Description
git log	View commit history.
git log --oneline	View commit history in one line.
git log --graph --all	Show commit history with a graph.
git config --global alias.<name> "<command>"	Create a Git alias.

Command	Description
git show	View details of the last commit.
git blame <file>	Show who modified each line of a file.
git checkout -- <file>	Restore a deleted or modified file.
git commit --amend -m "<new message>"	Edit the last commit message.
git clean -f	Remove untracked files.

9. Git Best Practices

In this section, we'll cover:

☑ Writing clear commit messages
☑ Structuring branches effectively
☑ Avoiding common mistakes
☑ Keeping repositories clean

1. Writing Clear Commit Messages

A good commit message helps team members understand the changes. Follow these best practices:

☑ **Follow the Conventional Format**

<type>: <short description>

[Optional] Detailed explanation of the change

- **Example:**

feat: Add user authentication

fix: Resolve login page bug

☑ **Use Present Tense and Be Concise**

❌ Bad: *Fixed login issue*
☑ Good: *Fix login issue*

2. Structuring Branches Effectively

Branching strategies help keep the codebase clean and manageable.

☑ **Use a Clear Naming Convention**

Branch Type	Naming Example
Main Branch	main

Branch Type	Naming Example
Feature Branch	feature/user-authentication
Bug Fix Branch	fix/login-bug
Hotfix Branch	hotfix/critical-fix
Release Branch	release/v1.2.0

☑ **Use Feature Branches**

Never work directly on main. Instead, create a feature branch:

```
git checkout -b feature/new-feature
```

After finishing the work, merge it:

```
git checkout main
```

```
git merge feature/new-feature
```

3. Avoiding Common Mistakes

❌ **Committing Sensitive Data**

Always ignore API keys, passwords, or configuration files using .gitignore:

```
echo "config.json" >> .gitignore
```

❌ **Pushing Large Files**

Use Git Large File Storage (LFS) for large files:

```
git lfs track "*.mp4"
```

❌ **Merging Without Pulling First**

Before merging, always pull the latest changes:

```
git pull origin main
```

❌ **Using git reset --hard Carelessly**

This command **permanently deletes** commits. Instead, use:

```
git revert <commit-hash>
```

This keeps the history intact.

4. Keeping Repositories Clean

☑ Delete Merged Branches

Once a feature is merged, delete its branch:

```
git branch -d feature/new-feature
```

If it's a remote branch:

```
git push origin --delete feature/new-feature
```

☑ Rebase Instead of Merging (When Appropriate)

If working solo, rebasing keeps history cleaner:

```
git rebase main
```

However, avoid rebasing shared branches!

☑ Squash Small Commits Before Merging

Instead of multiple small commits:

```
git rebase -i HEAD~3
```

Then choose squash for minor commits.

Quick Recap: Best Practices from This Section

Best Practice	Command/Guideline
Write clear commit messages	Use feat:, fix:, docs: prefixes
Use feature branches	git checkout -b feature/branch-name
Avoid committing sensitive files	Use .gitignore
Keep repositories clean	Delete merged branches with git branch -d
Use rebase for cleaner history	git rebase main

Best Practice	Command/Guideline
Squash small commits before merging	git rebase -i HEAD~3

Conclusion

Git is an essential tool for developers, enabling efficient collaboration, version control, and code management. In this guide, we covered:

☑ **Setting Up Git** – Installing Git, configuring user details, and initializing repositories.
☑ **Basic Git Workflow** – Adding, committing, and pushing changes to remote repositories.
☑ **Branching and Merging** – Creating feature branches, merging changes, and resolving conflicts.
☑ **Undoing Changes** – Resetting, reverting, and stashing changes safely.
☑ **Advanced Commands & Productivity Tips** – Using aliases, viewing commit history, and managing repositories efficiently.
☑ **Best Practices** – Writing clear commit messages, maintaining a structured branching strategy, and keeping repositories clean.

By following these Git fundamentals and best practices, you can improve your workflow, minimize errors, and collaborate effectively. Keep practicing, experiment with commands, and refine your Git skills over time. Happy coding!