DevOps Shack

# Git

# Comprehensive Guide

Batch-9 | By DevOps Shack

# Git Comprehensive Guide

## 🔍 What is Git?

Git is a **Distributed Version Control System** where every developer has:

- A full copy of the repository (.git directory)
- Complete history including all branches, commits, and tags
- Ability to work offline, merge changes, and push to a central server

🔥 Unlike older systems (CVS, SVN), Git doesn't rely on a central repository. It gives full power and flexibility to every contributor.

---

## 🔄 Centralized vs Distributed

| Feature | Centralized VCS | Git (DVCS) |
|---|---|---|
| Server dependency | Mandatory | Not required |
| Offline work | ❌ No | ✅ Yes |
| Performance | Slower | Faster |
| History stored | Server-only | Local |
| Collaboration | Limited branching | Unlimited branching, rebasing |

---

## 🧠 Why Git Changed the Game

- Snapshots instead of diffs
- Immutable commit history
- Fast operations via local metadata
- Advanced branching + merging strategies
- SHA-1 content hashing (ensures data integrity)
- Efficient file storage using content deduplication

---

## 📘 Git Terminology – Technical Explanation

| Term | Purpose | Behind-the-scenes |
|---|---|---|
| git init | Creates .git/ | Initializes metadata directories and refs |
| Working Directory | Where devs modify files | Tracked vs Untracked files |
| Staging Area | Area to prepare commits | .git/index file tracks staged file metadata |
| Commit | Saves current state | Stores a new object with SHA-1 and links |
| Push | Uploads commits to remote | Affects .git/refs/heads/branch remotely |
| HEAD | Pointer to the current branch | Contents of .git/HEAD |
| Refs | Branch, tag, and HEAD pointers | Stored in .git/refs/ and packed-refs |

---

## 🧱 Git's Core Object Model

Every piece of data in Git is stored as an object in the .git/objects directory. These objects are:

| Object Type | Description |
|---|---|
| **Blob** | Stores file contents |
| **Tree** | Stores directory structure |
| **Commit** | Stores snapshot reference, metadata, and parent(s) |
| **Tag** | Reference to a commit or object |

---

## 📁 Blob Object

- Raw content of a file (no filename)
- Stored in a compressed format
- SHA-1 hash used as ID

echo "Hello World" > file.txt

git hash-object file.txt

💡 This generates a **blob** object stored in .git/objects/ab/123456...

---

### 🌳 Tree Object

- Maps directory names to blob/tree hashes
- Stores metadata (mode, type, name)

Use:

git ls-tree HEAD

This lists the contents of the current commit tree.

---

### 🧱 Commit Object

- Links to a tree object
- Stores author, committer, date, message
- Links to parent commit(s)

git cat-file -p HEAD

Sample output:

tree 48ab23...

parent a1c9e7...

author Aditya <adi@devops.com>

date Tue Apr 9 20:31:54 2024 +0530

Initial commit

---

### 🧠 Git Commit DAG (Directed Acyclic Graph)

Git doesn't use a linear history — it uses a **DAG** of commits:

```
A --- B --- C --- D  (main)
       \
        E --- F      (feature)
```

- Each commit points to its **parent(s)**
- Merges have multiple parents
- The **DAG ensures consistency**, no circular references

📌 Git resolves branches and history using these relationships.

### 🔗 SHA-1 Hashing

- Git uses SHA-1 (e.g., 9fceb02c...) to uniquely identify:
  - Blobs
  - Trees
  - Commits
  - Tags

💡 This makes Git **secure and content-addressable**.

Even a single character change produces a new hash.

---

### ⚙️ What Happens During Git Commands

🔸 **git add file.txt**

- File is added to **staging area**
- Its content is hashed
- Blob is created (if not already exists)
- Index file is updated

🔸 **git commit -m "msg"**

- Git creates:
  - A tree object for the directory
  - A commit object pointing to the tree and the parent commit
- Moves HEAD to this new commit

🔸 **git push**

- Transfers commits and refs to remote repo via HTTP/SSH

---

### 📂 Inside the .git Directory

```
.git/
├── HEAD            # Pointer to current branch
├── config          # Repo configuration
├── index           # Staging area (binary file)
├── objects/        # All Git objects (blobs, trees, commits, tags)
├── refs/           # Branch and tag pointers
|   ├── heads/      # Local branches
|   ├── tags/       # Tags
```

### 🔬 Explore Git Internals Live

Try these commands during a live session:

```
# Create repo and file
git init
echo "Hello Git" > hello.txt
git add hello.txt
git commit -m "Initial commit"

# Inspect Git internals
ls .git/objects
git cat-file -p HEAD
git cat-file -p <commit hash>
git cat-file -p <tree hash>
```

---

### 🧠 Git = Snapshots + DAG + Content Hashing

| Core Concept | Benefit |
|---|---|
| Snapshot-based | Ensures complete project state at every commit |
| DAG structure | Enables powerful merge/rebase/branch operations |
| SHA-1 hashing | Ensures data integrity and deduplication |
| Local repository | Fast operations, offline support |
| Immutable commits | Clean audit trail and history |

### ✅ Summary (Key Takeaways)

- Git is not just a VCS — it's a **snapshot engine with a content-addressed filesystem**
- Every commit = complete tree state + metadata
- Blobs store content, trees organize it, commits record it, refs track it
- Git’s internals (SHA-1 + DAG) make it robust, fast, and scalable

## Section 2: Git Branch Management & Git Diff (Ultra-Detailed Guide)

### 🧩 Part 1: Git Branching – Concepts, Syntax, Strategies, and Internals

---

#### 🔹 What is a Git Branch?

A **branch in Git** is simply a lightweight movable pointer to a specific commit. It's not a copy of files or directories — it's a **reference to a snapshot** in your project's history.

Think of a Git branch like a bookmark in the commit timeline. You can create a new one, move it, or merge it without duplicating code.

---

#### ⚙️ How Git Branches Work Under the Hood

- Git maintains a special pointer called **HEAD**, which references your current branch.
- Each branch is a file in .git/refs/heads/branch-name, containing the SHA-1 hash of the latest commit.
- When you create a branch, Git **copies the current commit hash** into a new file in .git/refs/heads/.

#### 🔍 Example:

```
git branch feature-x
cat .git/refs/heads/feature-x
```

You'll see a commit hash like:

```
3adf23d5d61e2e6aee43b00e93e93e1e9e32a012
```

This means feature-x is pointing to that commit. As you commit more, Git updates that pointer.

---

#### 🛠️ Basic Branching Commands (with Scenarios)

| Task | Command | Scenario |
|---|---|---|
| Create branch | git branch feature-x | You want to work on a new login feature |
| Switch to branch | git checkout feature-x or git switch feature-x | Start working on the login feature |
| Create + switch | git checkout -b feature-x | Combine both actions |

| Task | Command | Scenario |
|---|---|---|
| List branches | git branch | See all local branches |
| Delete branch | git branch -d feature-x | After merge, cleanup |
| Force delete | git branch -D feature-x | When a branch isn't merged but you want to delete it anyway |
| Rename branch | git branch -m old-name new-name | Rename for clarity |
| See remote branches | git branch -r | Collaborators' branches |
| See all branches | git branch -a | Local + remote branches |

---

🧠 **Advanced Branching Internals**

When you commit in a new branch:

- Git writes a new commit object with a new SHA-1
- Updates the current branch pointer (e.g., feature-x) to this new hash
- The HEAD also moves with it since you're on that branch

This allows **non-linear development**, where branches diverge from and merge back into mainline code.

---

## Small Assignment:

```
git init
echo "Hello World" > hello.txt
git add . && git commit -m "Initial commit"

# Create branch
git branch feature-login
git checkout feature-login
echo "Login Page" > login.html
git add . && git commit -m "Add login page"
```

Then, go back to main and see that login.html doesn't exist — that's branch isolation.

---

## 🚀 Git Branch Naming Conventions in Teams

Use **clear, consistent names** that reflect purpose and scope.

| Branch Type | Example | Purpose |
|---|---|---|
| main or master | - | Production-ready code |
| develop | - | Staging area for integration |
| feature/* | feature/payment-integration | New feature development |
| bugfix/* | bugfix/login-crash | Fix minor issues not in prod |
| hotfix/* | hotfix/critical-vuln | Patch production issues immediately |
| release/* | release/v1.2.0 | Final stabilization before release |

## 🧭 Corporate Git Branching Strategies

There are multiple branching models used across companies. Let's break them down:

---

### 🔁 1. Git Flow (Vincent Driessen Model)

Most detailed and process-heavy strategy. Great for release-based projects.

**Branches Used:**

- main (production)
- develop (integration)
- feature/*
- release/*
- hotfix/*

**Flow:**

1. Start from develop, create feature/x
2. When done → merge back to develop
3. To release → create release/1.0
4. Final release → merge release/1.0 into main and tag it
5. Hotfixes start from main and also go into develop

---

### 💉 Git Flow Example:

```
# Create feature
git checkout develop
git checkout -b feature/cart-page

# Work, commit, then:
git checkout develop
git merge feature/cart-page

# Prepare release
git checkout -b release/v1.0
# Tag and test...

# Final release
git checkout main
git merge release/v1.0
git tag v1.0
```

```
# Bring back to develop
git checkout develop
git merge release/v1.0
```

---

### 🚀 2. GitHub Flow (Simpler, for Continuous Deployment)

Great for startups or SaaS.

**Branches:**

- main only (no develop)
- Feature branches → Pull Requests (PRs)

**Flow:**

1. Create feature branch from main
2. Open PR early for visibility
3. Merge after review/tests

**Command:**

```
git checkout -b feature/signup-form main
```

---

### 🔁 3. Trunk-Based Development

Used by large-scale teams like Google or Facebook.

- Developers commit frequently to main or trunk
- Feature flags used to hide incomplete features
- Encourages continuous integration + testing

---

### ❓ When to Use What?

| Team Type | Strategy |
|---|---|
| Enterprise, releases, QA cycles | Git Flow |
| Startups, speed-focused | GitHub Flow |
| Large-scale CI/CD with feature flags | Trunk-Based |

---

## 🎯 Best Practices for Branch Management

- ✅ Keep branch names meaningful (feature/signup-form)
- ✅ Delete stale branches regularly
- ✅ Avoid working too long in isolation (merge often)
- ✅ Rebase locally, merge remotely
- ✅ Use git fetch --prune to remove stale remote refs
- ✅ Use Pull Request templates to enforce code quality

---

## 🔍 Troubleshooting Branching Issues

| Issue | Fix |
|---|---|
| Accidentally created branch from wrong base | git rebase correct-base |
| Wrong commits in branch | git reset or git cherry-pick |
| Merge conflict | Resolve manually → git add → git commit |
| Outdated remote refs | git fetch --prune |

---

## 🧪 Hands-on Practice Ideas

1. Simulate a complete Git Flow cycle from develop to main
2. Introduce a merge conflict between branches and resolve it
3. Create and delete 5 dummy branches, both local and remote
4. Use GitHub/GitLab to create a PR from a branch

---

## 🔄 Part 2: Git Diff (Deep Dive)

---

### 🔹 What is git diff?

git diff shows the **line-by-line changes** between two versions of a file or branch. It compares:

- Working Directory ↔ Staging Area
- Staging Area ↔ Last Commit
- One commit ↔ Another
- One branch ↔ Another

---

### 🔎 Git Diff Basics

| Command | Compares | Use Case |
|---|---|---|
| git diff | Working dir ↔ Staging | See unstaged changes |
| git diff --cached | Staging ↔ Last commit | See what's ready to commit |
| git diff HEAD | Working dir ↔ Last commit | See all uncommitted changes |
| git diff A B | Commit A ↔ Commit B | Compare two commits |
| git diff branch1..branch2 | branch1 ↔ branch2 | See difference between branches |

---

### 🧪 Practical Examples

```
# Before staging
git diff

# After staging
git diff --cached

# Between two commits
git diff 4e9f5d1 29c3dd2

# Between branches
git diff develop..feature-xyz
```

---

## 🧠 Git Diff Output Anatomy

diff

diff --git a/index.html b/index.html

index 83db48f..bf2692e 100644

--- a/index.html

+++ b/index.html

@@ -5,7 +5,7 @@

<title>Homepage</title>

-<h1>Hello World</h1>

+<h1>Welcome to Git Course</h1>

| Symbol | Meaning |
|---|---|
| --- / +++ | File names before/after change |
| @@ | Line number ranges |
| - | Removed line |
| + | Added line |

---

## 🛠 Advanced Diff Usage

- Compare last 3 commits:

```
git diff HEAD~3 HEAD
```

- Compare staged vs HEAD:

```
git diff --cached
```

- See changes word-by-word:

```
git diff --word-diff
```

- See stat summary:

```
git diff --stat
```

---

💡 **Tips & Tricks**

- Use git difftool to compare with an external diff viewer (e.g., meld, Beyond Compare)
- Set up aliases:

git config --global alias.diffs "diff --stat --cached"

# Section 3: Git Merge & Git Rebase (With Demos)

---

### 📌 Why This Section Is Crucial

Merge and Rebase are two of the **most powerful yet misunderstood Git commands**, especially in corporate DevOps environments. Understanding their internal workings, use cases, pros and cons, and conflict resolution techniques is critical for:

- Clean code history
- Collaborative development
- Simplified debugging
- Safe release cycles

---

### 🧠 Part 1: What is Git Merge?

---

#### 🔹 Definition

git merge combines two branches by creating a **new commit** that brings together the histories of both branches.

```
git merge feature-x
```

This merges feature-x into the current branch (usually main or develop) and **preserves the commit history** of both branches.

---

### 🏗️ How Git Merge Works Internally

1. Git finds the **common ancestor** of the two branches (called the **merge base**).
2. It performs a **three-way merge** between:
    - Current branch (e.g., main)
    - Target branch (e.g., feature-x)
    - The merge base
3. A new **merge commit** is created with **two parent commits**.

🧬 Merge commits are special because they have multiple parents. This is what enables non-linear history.

---

### 🧪 Visual Example

```
A---B---C (main)
     \
      D---E (feature-x)
```

After git merge feature-x:

```
A---B---C---------F (main)
     \       /
      D---E---/  (feature-x)
```

- F is the new merge commit
- History from both branches is preserved

---

### 🧰 When to Use Git Merge

| Situation | Why Use Merge |
|---|---|
| Large teams | Keeps traceability of feature work |
| Long-lived branches | Better to show feature timelines |
| Finalizing features | Easy to visualize contribution via PRs |
| CI/CD pipelines | Merges trigger builds, not rebases |

---

### 👍 Pros of Git Merge

- Easy to understand
- Preserves complete history
- Safe for production merges
- Great for team collaboration (PRs)

---

### 👎 Cons of Git Merge

- Can create **messy history** with too many branches and merge commits
- Merge conflicts must be resolved manually
- Merge commits can be noisy for simple changes

---

## ⚙️ Git Merge – Real World Demo

---

### ✅ Step-by-Step:

```
# 1. Initialize repo
git init
echo "Hello" > file.txt
git add . && git commit -m "Initial commit"

# 2. Create feature branch
git checkout -b feature-login
echo "Login Feature" >> file.txt
git add . && git commit -m "Add login feature"

# 3. Go back to main
git checkout main
echo "Main Change" >> file.txt
git add . && git commit -m "Main branch update"

# 4. Merge feature
git merge feature-login
```

If there are conflicts, Git will pause the merge until resolved.

---

### 🔥 If Conflict Occurs

```
# Resolve file.txt manually

git add file.txt

git commit
```

---

## 🔄 Part 2: What is Git Rebase?

---

### 🔹 Definition

git rebase takes a branch and **reapplies its commits** on top of another branch.

git rebase main

This **rewrites the commit history** by changing the parent of your commits, making it look like they were created after the latest commit on main.

---

### 🔍 How Git Rebase Works Internally

1. Git finds the **common ancestor** between feature-x and main.
2. It temporarily **removes** all commits on feature-x after that point.
3. It **reapplies** each commit from feature-x on top of main, one by one.
4. Each commit gets a **new SHA-1 hash** (because history is rewritten).

---

### 🧪 Visual Example (Before Rebase)

```
A---B---C (main)
     \
      D---E (feature-x)
```

After git checkout feature-x && git rebase main:

```
            D'--E' (feature-x)
           /
A---B---C (main)
```

- D and E are rewritten as D' and E'
- Looks like the feature was developed after C

---

### 🧰 When to Use Git Rebase

| Situation | Why Use Rebase |
|---|---|
| Before merge | Clean history without merge commits |
| Solo developer | Rewriting own history is safe |

| Situation | Why Use Rebase |
|---|---|
| Local changes | Rebase before pushing |
| Linear project history | Easier to debug with git bisect |

---

### 👍 Pros of Git Rebase

- Keeps history **linear and clean**
- Easier to navigate with git log
- Avoids noisy merge commits
- Ideal before merging to main

---

### 👎 Cons of Git Rebase

- **Rewrites history** — dangerous if shared with others
- **Conflicts** can occur with each commit replayed
- Should not rebase **public branches**

---

### ⚠️ Golden Rule of Rebase

🚨 **Never rebase shared/public branches. Only rebase local commits that haven't been pushed.**

---

### ⚙️ Git Rebase – Real World Demo

---

### ✅ Step-by-Step:

```
# 1. Create main commit
git init
echo "App start" > app.txt
git add . && git commit -m "Initial commit"

# 2. Create feature branch
git checkout -b feature-ui
echo "UI code" >> app.txt
git add . && git commit -m "UI change 1"
echo "UI fix" >> app.txt
```

```
git add . && git commit -m "UI fix"

# 3. Switch to main and make changes
git checkout main
echo "Main update" >> app.txt
git add . && git commit -m "Main update"

# 4. Rebase feature onto updated main
git checkout feature-ui
git rebase main
```

---

### 🔥 If Conflicts Happen

```
# Manually fix file
git add file.txt
git rebase --continue
```

---

### 🧨 Rebase Abort

```
git rebase --abort  # Roll back rebase if needed
```

---

### ⚔️ Merge vs Rebase: Battle of Titans

| Feature | Merge | Rebase |
|---|---|---|
| Commit History | Preserves full tree | Rewrites history |
| Merge Commit | Always adds one | No merge commit |
| Use In Teams | Preferred for shared code | Preferred for local branches |
| Rewriting History | No | Yes |
| Traceability | Better | Cleaner but less traceable |
| Conflict Resolution | Once | Multiple times (per commit) |

---

### 🧪 Practical Scenario

Your teammate merges feature-a into main. You're working on feature-b.

- To catch up:

# Merge

```
git checkout feature-b
git merge main
```

Or:

```
# Rebase
git checkout feature-b
git rebase main
```

Rebase gives a clean history; merge preserves context.

---

### 💣 Common Merge/Rebase Errors & Fixes

| Error | Cause | Fix |
|---|---|---|
| Merge conflict | Overlapping file edits | Fix file → git add → git commit |
| "Refusing to rebase" | Uncommitted changes | git stash or commit first |
| "No rebase in progress" | Wrong usage | Use git rebase --abort |
| Rebased wrong branch | Mistake in base branch | Use reflog: git reflog, then git reset |
| Accidentally rebased public branch | Broke team sync | Coordinate and force-push (use with care) |

---

### 🧠 Pro Tips

- Use git pull --rebase to sync your branch linearly with upstream
- Enable auto-setup rebase:

```
git config --global pull.rebase true
```

- Visualize history before/after with:

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

---

### 🛠 Advanced Techniques

- **Interactive Rebase**:

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

  - You can squash, reorder, rename, or drop commits

- **Squash with rebase**:

```
git rebase -i main
```

  - Use pick for first commit, squash for the rest

---

### 📈 Corporate DevOps Relevance

| Stage | Merge or Rebase? | Why? |
|---|---|---|
| Local development | Rebase | Clean commit history |
| Pull Request | Merge | Trackable contribution |
| Production release | Merge | CI/CD traceability |
| Bugfix hotpatch | Merge | Safety and visibility |
| Feature sync | Rebase | No merge noise |

# Section 4: Git Stash & Pop

---

### 🧠 What is Git Stash?

git stash is a **powerful Git feature** that lets you **temporarily save** (or "stash away") your **uncommitted changes**, so you can **work on something else** without losing progress.

Think of it like placing your work on a shelf for safekeeping while you clean your workspace or switch tasks.

---

### 🧩 Why Do Developers Use Git Stash?

| Situation | Why Stash Helps |
|---|---|
| Mid-task switch | Need to quickly switch branches without committing half-done work |
| Code review | Clean up your working directory before reviewing someone else's branch |
| Experimentation | Try a quick fix without affecting your ongoing task |
| Pulling latest changes | Stash changes, pull latest code, re-apply stashed changes on top |
| Avoiding unnecessary commits | Save changes temporarily without polluting history with incomplete commits |

---

### ⚙️ Git Stash Syntax and Core Commands

| Command | Description |
|---|---|
| git stash | Stash both staged and unstaged changes |
| git stash save "msg" | Save stash with a custom message |
| git stash list | View all saved stashes |
| git stash show | Show changes in most recent stash (summary) |
| git stash show -p | Show patch (diff) of stash |

| Command | Description |
|---|---|
| git stash apply | Apply most recent stash without deleting it |
| git stash pop | Apply most recent stash and remove it |
| git stash drop stash@{0} | Delete a specific stash |
| git stash clear | Delete all stashes |
| git stash branch <branch> | Create a new branch from a stash |

---

### 🔍 Under the Hood: How Git Stash Works Internally

git stash doesn't just save a "working copy" of files — it creates **three commits** in the background:

1. **Commit A**: Your changes in the working directory (unstaged + staged)
2. **Commit B**: The staged index (only what was added)
3. **Commit C**: A merge commit combining them, stored under refs/stash

This stash reference is just like a regular commit — you can use git log or git show to explore it.

```
git log refs/stash
```

✅ Git stash uses the same internal mechanisms as commits — which means it's powerful, scriptable, and robust.

---

## 🔧 Demo: Real-World Scenario

---

### 🎯 Scenario:

You're working on a feature in feature/cart branch. Suddenly, you need to switch to develop to fix a bug reported by QA — but you don't want to commit half-done code. What do you do?

---

### ✅ Step-by-Step Demo

# 1. Create a repo and branch

```
git init stash-demo
cd stash-demo
echo "Version 1" > app.js
git add . && git commit -m "Initial commit"
git checkout -b feature/cart
```

# 2. Make changes (but don't commit)

```
echo "Cart logic WIP" >> app.js
```

# 3. Stash the changes

```
git stash save "WIP: cart logic implementation"
```

At this point, your working directory is **clean**, and you can switch branches:

```
git checkout develop
```

---

### 🔄 Restore Your Work

Once the bug is fixed, go back to your branch and restore your work:

```
git checkout feature/cart
git stash pop
```

Boom 💥 — your changes are back.

---

### 📂 Anatomy of a Stash Entry

git stash list

Output:

stash@{0}: WIP on feature/cart: 1a2b3c4 Initial commit

This means:

- stash@{0} is the latest stash
- It came from feature/cart
- 1a2b3c4 is the base commit

You can inspect it further:

git stash show stash@{0}

git stash show -p stash@{0}

---

### 🧱 Stash Structure Internals

Each stash is stored under:

.git/logs/refs/stash

And internally composed of:

- A commit of working directory
- A commit of staged changes
- A reference to the original HEAD

These are created using low-level plumbing like:

git write-tree

git commit-tree

---

### 🔍 Stashing with Only Tracked or Untracked Files

| Goal | Command |
|---|---|
| Stash only tracked files | git stash -k or --keep-index |
| Include untracked files | git stash -u |
| Include ignored files too | git stash -a |

### 🧪 Advanced Demo: Stash While Pulling Latest Code

# 1. You've edited app.js but want to pull new changes from remote

git stash

git pull origin main

git stash pop

This avoids:

- Merge conflicts with your incomplete changes
- Creating a "junk commit" just for syncing

---

### 🎯 Stash Naming: Best Practices

Give your stashes clear names — avoid "WIP" spam in history.

git stash save "cart: UI state logic cleanup"

Use hooks/scripts to auto-stash with branch names and timestamps if needed.

---

### 🔍 Stash Application Techniques

### 🔁 Apply but Keep

git stash apply stash@{0}

This re-applies your stash **without deleting it**. Useful when you're experimenting.

### 💣 Pop (Apply + Delete)

git stash pop stash@{0}

Safe for one-time restoration.

---

### 🚨 Conflict Handling During stash pop

If the stash conflicts with your current branch:

- Git will halt the pop operation
- You'll need to **manually resolve conflicts**
- Then run:

git add .

git stash drop stash@{0}

### 🛠 Tools That Work With Git Stash

| Tool | Integration |
|---|---|
| VS Code | Shows stash entries in Git sidebar |
| GitKraken | Visual stash management |
| SourceTree | UI-based stash pop/apply/drop |
| Git GUI | Basic stash support |

---

### 💉 Git Stash + Branch: Experimental Power

Let's say your stash becomes worth saving in a proper branch:

git stash branch experimental-ui stash@{0}

This does:

1. Creates experimental-ui branch
2. Checks it out
3. Applies the stash
4. Drops it from stash list

💡 This is perfect for rapid prototypes that turned into features.

---

### 🧠 Git Aliases for Stashing

Speed up your workflow:

git config --global alias.ss "stash save"

git config --global alias.sp "stash pop"

git config --global alias.sl "stash list"

Now run git ss "msg" instead of git stash save.

---

## 📜 Summary: Stash Command Cheatsheet

| Action | Command |
|---|---|
| Save stash | git stash or git stash save "msg" |
| List stashes | git stash list |
| Show changes | git stash show or -p |
| Apply stash | git stash apply stash@{0} |
| Pop stash | git stash pop stash@{0} |
| Drop stash | git stash drop stash@{0} |
| Clear all | git stash clear |
| Branch from stash | git stash branch branch-name |

---

## 💬 Real-World DevOps Use Cases

| Situation | Stash Strategy |
|---|---|
| CI/CD pipeline failing during manual fix | Stash uncommitted debug work |
| Emergency hotfix needed | Stash incomplete feature and switch |
| Developer context switching | Save WIP progress with message |
| Rebuilding corrupted branches | Use stash branch for temporary experiments |
| Training new devs | Use stash to simulate change recovery scenarios |

---

## 🔥 Corporate Best Practices

- Never stash without a message
- Avoid leaving stashes around forever
- Regularly git stash clear or integrate into branches
- Use stash branch for transitioning temp work into PRs
- Educate your team to treat stashes like **drafts**, not garbage bins

# Section 5: Git Cherry-pick

### What is git cherry-pick?

git cherry-pick is a **Git command that allows you to apply a specific commit from one branch to another**, **without merging the full branch**.

✅ It's like copying a cherry (commit) from one cake (branch) and placing it on another cake (branch) without transferring the whole cake.

---

### Syntax

git cherry-pick <commit-hash>

You can also cherry-pick a **range of commits**, **multiple commits**, or even cherry-pick with options for conflict resolution, message editing, or no-commit:

```
git cherry-pick A B C                  # Multiple commits
git cherry-pick A^..B                  # Commit range
git cherry-pick --no-commit <hash>     # Apply changes without committing
git cherry-pick --edit <hash>          # Edit message before commit
git cherry-pick --signoff <hash>       # Add sign-off line
```

---

### How Cherry-pick Works Internally

Cherry-pick operates by:

1. Identifying the **diff** introduced by a specific commit.
2. Applying that **diff** to your current working tree.
3. Staging the result and **creating a new commit** with the same content but a **new hash**.

🚨 The commit history is **not preserved**. Git re-applies the change, not the commit.

---

### 🎯 Use Cases for Cherry-pick

| Scenario | Why Use Cherry-pick |
|---|---|
| Apply a hotfix from hotfix/ to main and develop | Share a fix without merging the full branch |
| Backport a feature from main to release/1.2 | Deliver a selective feature to older versions |
| Pick a specific bug fix into QA | Isolate fixes without unnecessary changes |
| Move only relevant commits from feature/ into develop | Avoid experimental commits |

---

### 💡 Scenario:

You're on the hotfix/login-crash branch. You fixed a production bug and now need to:

- Apply the fix to main
- Also apply it to develop

---

### ✅ Step-by-Step Setup

# 1. Start with repo and main branch

```
git init cherry-demo
cd cherry-demo
echo "Login system" > app.js
git add . && git commit -m "Initial commit"
```

# 2. Create develop and hotfix branches

```
git checkout -b develop
```

```
echo "Working dev version" >> app.js
```

```
git commit -am "Dev enhancements"
```

```
git checkout -b hotfix/login-crash main
```

```
echo "Hotfix for login crash" >> app.js
```

```
git commit -am "Fix: login crash on null token"
```

---

### 📝 View Commit History

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

Sample:

e12b987 (hotfix/login-crash) Fix: login crash on null token

f74c8d3 (develop) Dev enhancements

a7f45dc (main) Initial commit

---

### 🍒 Cherry-pick the Fix onto Main

`git checkout main`

`git cherry-pick e12b987`

— now your main has the fix without touching anything else from hotfix/login-crash.

---

### 🌀 Now Cherry-pick to Develop

`git checkout develop`

`git cherry-pick e12b987`

Now the hotfix is reflected in all relevant branches — clean, safe, and traceable.

---

### 🔍 Inspecting Cherry-picked Commits

`git log --oneline`

You'll see:

- A **new commit hash**
- Same commit message
- History is **linear**, no merge commits

Git creates a **new commit** with the same patch but a different parent — it's a fresh snapshot.

---

## 🔥 Advanced Cherry-pick Options

---

### ➕ --edit

Allows you to change the commit message during cherry-pick:

```
git cherry-pick --edit <commit-hash>
```

Ideal when the fix needs to reflect the context of the new branch.

---

### 🚫 --no-commit

Applies changes without committing. Useful for combining multiple commits into one:

```
git cherry-pick --no-commit <hash>
# Make more changes or cherry-pick more
git commit -m "Combined fix for auth issues"
```

---

### 🖋️ --signoff

Adds a line at the bottom of the commit message indicating who cherry-picked it:

```
git cherry-pick --signoff <hash>
```

Used in open-source or regulated environments for audit purposes.

---

### 🧪 Cherry-pick a Range of Commits

```
git cherry-pick A^..C
```

Picks all commits between A and C (inclusive).

💡 Use git log --oneline to identify hashes.

---

### 💥 Handling Cherry-pick Conflicts

Cherry-pick can **fail with conflicts** just like merge or rebase:

CONFLICT (content): Merge conflict in app.js

**Resolve in 3 Steps:**

1. Open the file, fix the conflict
2. Stage it:

```
git add app.js
```

3. Continue cherry-pick:

```
git cherry-pick --continue
```

Abort if needed:

```
git cherry-pick --abort
```

---

### 📜 Cherry-pick vs Merge vs Rebase

| Feature | Cherry-pick | Merge | Rebase |
|---|---|---|---|
| Copy commits | ✅ | ❌ | ❌ |
| Preserve history | ❌ | ✅ | ❌ |
| Conflict potential | Medium | High | High |
| Use case | Isolate commits | Combine branches | Rewrite history |
| Commit hash changes | ✅ | ❌ | ✅ |
| New commit created | ✅ | ✅ (merge commit) | ✅ |

---

### 📂 Cherry-pick in DevOps Pipelines

| Use Case | Where It Helps |
|---|---|
| Hotfixes | Apply fixes to both main and develop |
| QA sync | Backport specific bug fixes |
| Production rollbacks | Extract working commits from feature branches |
| Reusable code patterns | Move small reusable commits across microservices |

---

### 🔐 Best Practices

- ✅ Use meaningful commit messages to ease cherry-picking later
- ✅ Always test cherry-picked changes in their new context
- ✅ Avoid cherry-picking merge commits (they are complex)

- ✅ Prefer cherry-pick over merge for small, isolated fixes
- 🚫 Never cherry-pick long chains of complex interdependent commits

---

### 🚨 Common Pitfalls & Fixes

| Problem | Cause | Solution |
|---|---|---|
| Cherry-pick fails | Conflicting code | Manually resolve → git cherry-pick --continue |
| Cherry-picked multiple times | Duplicate commits | Use git log, prune with reflog |
| Cherry-pick of merge commit fails | Complex history | Use patch instead or squash the merge |
| Wrong commit cherry-picked | Context mismatch | Use git revert or reset the branch |

---

### 🧠 Real-World Analogy

Imagine a restaurant kitchen (your repo). You create a new “special recipe” (commit) for pasta in the *Italian* section (branch). The dessert section (another branch) wants just that pasta recipe, not everything else from the Italian section.
Cherry-pick allows you to copy just that recipe.

---

### 👥 Collaborative Scenarios

- Teammates pick critical bugfixes into staging branches for QA testing
- QA engineers maintain a QA-only branch with selected stable commits
- Managers request targeted bugfix rollouts without full deploys

---

### 💡 Git Aliases for Cherry-pick

Speed things up:

```
git config --global alias.cp 'cherry-pick'
```

```
git config --global alias.cpedit 'cherry-pick --edit'
```

Now you can run:

`git cp <hash>`

`git cpedit <hash>`

---

### 🧪 Interactive Practice Ideas

1. Create 3 branches, each with different features.
2. Cherry-pick a bug fix from one into the others.
3. Do it with --edit and --no-commit.
4. Simulate a conflict, resolve, and continue.
5. Undo a cherry-pick using:

`git log`

`git revert <cherry-picked-hash>`

---

### 📜 Summary: Git Cherry-pick Power Cheatsheet

| Task | Command |
|---|---|
| Basic usage | git cherry-pick <hash> |
| Range of commits | git cherry-pick A^..B |
| Without auto-commit | git cherry-pick --no-commit |
| Edit message | git cherry-pick --edit |
| Signoff | git cherry-pick --signoff |
| Resolve conflict | git cherry-pick --continue |
| Abort cherry-pick | git cherry-pick --abort |

---

### 🔚 Final Thoughts

Git Cherry-pick is an **essential tool** for precision control in version history. It lets you surgically move code changes **without carrying full branches**, enabling agile fixes, targeted deployments, and safe backporting.

# Section 6: Git Tags

### 🧠 What is a Git Tag?

A **Git tag** is a **label** or **marker** attached to a specific commit in your repository. It's used to **mark important points in history** — usually for **releases**, **milestones**, or **production deployments**.

📌 Unlike branches, tags are immutable — they don't move. Once a tag is created, it always points to the same commit unless manually deleted or re-created.

---

### 📈 Why Tags Matter in Software Development

- Semantic Versioning (SemVer): v1.0.0, v2.3.4
- Snapshot releases for QA or Production
- Deployment rollbacks or reproducibility
- Dependency tracking (e.g., Python, Go modules, Docker)
- CI/CD triggers
- Audit & compliance checkpoints

---

### 📂 Types of Git Tags

| Tag Type | Description | Use Case |
|---|---|---|
| **Lightweight** | Just a name pointing to a commit (like a branch, no metadata) | Quick personal markers |
| **Annotated** | A full Git object with metadata (author, message, timestamp, GPG signature) | Public releases, production tags |

---

### 🔸 1. Lightweight Tags

git tag v1.0

- Does **not store metadata**
- Just a direct reference to the commit hash
- Equivalent to:

```
git tag -l
git show v1.0
```

### 🔸 2. Annotated Tags

git tag -a v1.1 -m "Release v1.1"

- Creates a new tag object stored in .git/objects/
- Includes:
  - Tag message
  - Tagger (author)
  - Date
  - GPG signature (if configured)
- Better for public releases and audits

You can verify:

git show v1.1

---

### ⚙️ Tag Creation – Full Syntax Reference

| Action | Command |
|---|---|
| Create lightweight tag | git tag v1.0 |
| Create annotated tag | git tag -a v1.1 -m "Release v1.1" |
| Tag a specific commit | git tag -a v1.2 <commit-hash> |
| Sign a tag (GPG) | git tag -s v1.3 -m "Signed release" |
| List all tags | git tag |
| Search tags | git tag -l "v1.*" |
| Show tag content | git show v1.1 |
| Delete tag locally | git tag -d v1.0 |
| Delete tag remotely | git push origin --delete v1.0 |
| Push tags to remote | git push origin --tags |
| Push single tag | git push origin v1.2 |

---

## 🧪 Git Tag Demo – Step-by-Step Walkthrough

---

### 🔨 Step 1: Setup

```
git init tag-demo
cd tag-demo
echo "Initial app version" > app.js
git add . && git commit -m "Initial commit"
```

---

### 🔨 Step 2: Tag the Commit

```
git tag -a v1.0 -m "First official release"
```

---

### 🔨 Step 3: Simulate New Features

```
echo "Feature A" >> app.js
git commit -am "Add feature A"
echo "Feature B" >> app.js
git commit -am "Add feature B"
```

---

### 🔨 Step 4: Add New Tag

```
git tag -a v1.1 -m "Second release with feature A & B"
```

---

### 🔨 Step 5: Push to Remote

```
git remote add origin <your-repo-url>
git push origin --tags
```

---

### 🧱 Behind the Scenes: How Tags Work Internally

Git stores tags inside .git/refs/tags/ (like branches are in .git/refs/heads/).

Each tag file contains the SHA-1 hash of the commit (or tag object in case of annotated).

You can inspect the object:

```
git cat-file -p <tag-hash>
```

---

### 🔍 Annotated Tag Structure:

- Tag object references the commit object
- Stored in objects/
- Contains:

object <commit-hash>

type commit

tag v1.0

tagger John Doe <john@company.com> ...

message

---

### 🔐 Signed Tags for Security

git tag -s v1.2 -m "Signed tag"

Requires GPG setup:

gpg --list-secret-keys

git config --global user.signingkey <key>

Verify a signed tag:

git tag -v v1.2

Used in high-compliance environments for **traceability** and **authenticity**.

---

### 🔄 Tagging Older Commits

Use Git log to find the hash:

git log --oneline

Then:

git tag -a v0.9 <commit-hash> -m "Pre-release"

---

### 🏷️ Naming Convention Best Practices

| Tag Type | Format |
|---|---|
| Releases | v1.0.0, v2.3.1 |
| Milestones | mvp-1, rc-2 |
| Hotfix | hotfix-2024-04-10 |
| Signed tags | signed-v1.0 |

Follow **Semantic Versioning**:

- MAJOR.MINOR.PATCH = Breaking, Feature, Fix
- Example: v2.1.4

---

### 🔧 Tags in CI/CD Pipelines

| CI/CD Use Case | How Tags Help |
|---|---|
| Trigger deploys | e.g., only deploy on new v* tag |
| Rollbacks | Easily revert to v1.3 tag |
| Docker builds | Use tags as docker build -t myapp:v1.0 . |
| Artifact versioning | Save JARs, ZIPs, etc. as release-v1.0.zip |
| Immutable builds | Tags guarantee identical code for all environments |

---

### 💡 Practical Real-World Use Cases

| Role | Use Case |
|---|---|
| Developer | Marks final build before sending to QA |
| QA | Tags builds as "tested-v1.2" |
| DevOps | Tags production-ready builds for CI/CD |
| Release Manager | Uses signed annotated tags for release notes |
| Project Manager | Reviews milestone delivery via tags |

---

### 🛠 Undo or Manage Tags

| Task | Command |
|---|---|
| Delete locally | git tag -d v1.0 |
| Delete remote | git push origin --delete v1.0 |
| Update tag (delete + re-create) | Delete, then git tag -f v1.0 <hash> |
| Clone repo with tags | git clone --branch v1.0 <url> |

---

### 🚨 Gotchas & Safety Tips

- 🧨 **Tags are not pushed automatically!**
  You must explicitly git push origin --tags.
- 🧨 **Avoid re-tagging history**: This can confuse CI/CD, cause production mishaps
- 🧨 **Do not move annotated tags on public repos**
- 🧨 **Protect release tags** using GitHub/GitLab tag protection rules

---

### 🧪 Deep Technical Exercise

# Tag current commit

git tag -a v2.0 -m "Major release"

# View objects

git rev-parse v2.0

git cat-file -p <tag-hash>

Observe:

- Tag object
- Commit object it points to
- Differences from lightweight tag

---

**Tags + Branch Strategy**

| Task | Strategy |
|---|---|
| Finalize release | Tag release/x → merge to main |
| Hotfix production | Tag hotfix commit for rollback |
| Archive old builds | Tag and archive v0.9 |
| Continuous delivery | Use tags for semver release notes |

---

### 📜 Tag Management Cheatsheet

| Action | Command |
|---|---|
| Create lightweight | git tag v1.0 |
| Create annotated | git tag -a v1.1 -m "msg" |
| Sign tag | git tag -s v1.2 -m "msg" |
| List all | git tag |
| Show tag info | git show v1.1 |
| Push tags | git push origin --tags |
| Push one tag | git push origin v1.1 |
| Delete local | git tag -d v1.0 |
| Delete remote | git push origin --delete v1.0 |
| Tag old commit | git tag -a v0.9 <hash> |

---

### 💬 Real Team Stories

- **Startup Release Flow**: Tags trigger GitHub Actions that deploy Docker containers to Kubernetes.
- **Enterprise App**: Signed tags required by infosec team for every release.
- **Monorepo Setup**: Each team tags their microservice inside the same repo.
- **Banking App**: Tags mapped to regulatory testing environments.

**Final Thoughts**

Git tags are simple to use but incredibly powerful when used strategically. They enable:

- ✅ Release tracking
- ✅ Rollback safety
- ✅ Semantic versioning
- ✅ Deployment automation
- ✅ Immutable history

# Section 7: Git Squash

### 🧠 What is Git Squash?

**Git Squash** is the process of **combining multiple commits into a single one** to create a **cleaner, more readable, and more meaningful commit history**.

It's most commonly used when:

- A feature branch has many tiny or WIP commits
- You want to merge that feature into main with **one polished commit**

✅ Squashing is typically done using **interactive rebase**, not the merge command.

---

### 🔎 Why Squash Matters in Teams

| Situation | Why Squash? |
|---|---|
| Feature development | Developers make many WIP commits. Squashing summarizes them. |
| Code reviews | A single clean commit is easier to review |
| PR history cleanup | Keep Git history readable and focused |
| CI/CD pipelines | Avoid triggering builds on each micro-commit |
| Open-source contributions | Maintainers prefer 1 commit per feature/fix |

---

### 🎨 Real Example Before Squash

commit c5e1b4a Add more spacing to form

commit a6d9e87 Fix typo

commit 3b1aa84 Working login page

commit 8af2930 Create login page

🧠 All part of a single feature. After squash:

commit 41a9b70 Add login page feature

---

### 🧱 Internals: What Happens During a Squash?

Under the hood:

- Git takes a list of commits
- Applies them one by one in memory
- Rewrites history with only the first commit (message + content)
- Merges content of the other commits
- Deletes the rest

This is possible via **interactive rebase**, which rewrites history.

---

### ⚙️ Interactive Rebase for Squash

```
git rebase -i HEAD~4
```

This opens your default editor (e.g., Vim, Nano, VS Code) with:

```
pick 8af2930 Create login page
pick 3b1aa84 Working login page
pick a6d9e87 Fix typo
pick c5e1b4a Add more spacing to form
```

Change to:

```
pick 8af2930 Create login page
squash 3b1aa84 Working login page
squash a6d9e87 Fix typo
squash c5e1b4a Add more spacing to form
```

Then:

- Save and close the editor
- Git will prompt you to **edit the new commit message**
- Finalize the message → save → done ✅

---

## 🧪 Step-by-Step Git Squash Demo

---

### 🎯 Scenario:

You're working on feature/login with 4 commits. You want to squash them before merging to main.

---

### ✅ Step-by-Step

# 1. Start with fresh repo

```
git init squash-demo
cd squash-demo
```

# 2. Create commits

```
echo "login base" > login.js
git add . && git commit -m "Create login page"

echo "login WIP" >> login.js
git commit -am "Working login page"

echo "Fix typo" >> login.js
git commit -am "Fix typo"

echo "Spacing update" >> login.js
git commit -am "Add more spacing to form"
```

# 3. Squash last 4 commits

```
git rebase -i HEAD~4
```

Edit from:

```
pick a1
pick a2
pick a3
pick a4
```

To:

pick a1
squash a2
squash a3
squash a4

---

💡 **Final Commit Message Prompt:**

# Combine commit messages

Create login page

# Working login page

# Fix typo

# Add more spacing to form

Clean it up as:

Add login page feature with layout, fixes, and spacing updates

---

🔁 **Git Squash Variants**

| Method | Command |
|---|---|
| Interactive Rebase | git rebase -i HEAD~N |
| Squash during Merge | git merge --squash feature-branch |
| Auto-squash with Fixup | git rebase -i --autosquash |
| CI squash in GitLab | Merge request → squash before merge |

---

⚒️ **Git Merge with --squash**

git checkout main

git merge --squash feature/login

git commit -m "Add login feature"

✅ Good for keeping commit history clean **without rebasing** the branch.

---

## Advanced: Autosquash with Fixup

**Step 1: Tag a fixup commit**

git commit --fixup <commit-hash>

**Step 2: Auto-squash**

git rebase -i --autosquash HEAD~N

Git reorders commits and marks them for squashing automatically.

---

### 👷 Real DevOps Use Case

You develop a feature across 6 commits:

- Add component
- Fix logic
- Rename variable
- Change structure
- Final cleanup
- Add test cases

Before merging to develop, squash into 1 commit:

git rebase -i HEAD~6

Now, reviewers and release notes see 1 clean summary commit instead of messy mid-dev changes.

---

### 🚨 Squashing & Shared Branches

❗ Squashing rewrites history → **NEVER squash commits on a shared branch** unless coordinated.

If others have pulled the branch:

- Use git push --force-with-lease (safer than --force)
- Or, squash before pushing at all

---

**Why Use HEAD~N?**

HEAD~N tells Git to go N commits back from the current commit. For example:

- HEAD~3 = last 3 commits before HEAD
- HEAD~1 = the parent of HEAD

💡 You can also squash from a **specific commit**:

git rebase -i <base-commit>

---

### 🛠 Best Practices

| Tip | Reason |
|---|---|
| Squash before merging PRs | Clean history for release & review |
| Use --autosquash for fixups | Saves time in long branches |
| Don't squash shared history | Avoid breaking collaborators |
| Test after squashing | Conflicts can lead to broken merges |
| Add meaningful messages | Merged commit should reflect all the changes |

---

### ⚙️ Squash in Pull Requests

#### 🔷 GitHub

- Enable "Squash and merge" in PR options
- Auto-squash PR commits into one on merge

#### 🔷 GitLab

- Use "Squash commits when merging"
- Default per project

#### 🔷 Bitbucket

- Set PR merge strategy to squash in settings

---

### 📜 Git Squash Commands Summary

| Task | Command |
|---|---|
| Rebase interactively | git rebase -i HEAD~N |
| Mark commits for squash | Change pick → squash |
| Auto-squash | git commit --fixup <hash> + git rebase -i --autosquash HEAD~N |
| Merge with squash | git merge --squash <branch> |
| Force push after squash | git push --force-with-lease |
| Undo squash | Use git reflog and git reset |

---

### 💬 Real Team Story

At SHACKVERSE, developers were making 15–20 commits per feature, including:

- Code commits
- Fixes
- Console logs
- Refactors

Before each PR merge, they squash into a single commit. This helped:

- Simplify history
- Speed up code reviews
- Improve release changelogs
- Prevent build trigger overloads in CI

---

### Git Squash vs Git Revert

| Action | Git Squash | Git Revert |
|---|---|---|
| Combine commits | ✅ | ❌ |
| Keep history intact | ❌ | ✅ |
| Rewrites commit IDs | ✅ | ❌ |

| Action | Git Squash | Git Revert |
|---|---|---|
| Safe for shared branches | ❌ | ✅ |
| Reverses changes | ❌ | ✅ |

---

### 🛑 Avoid These Mistakes

- ❌ Squashing commits **after push** without team sync
- ❌ Leaving vague commit messages like “fix”
- ❌ Squashing bugfixes before they’re reviewed
- ❌ Forgetting to test after squash
- ❌ Forcing push without checking shared usage

---

### 🧪 Suggested Practice

1. Create a feature branch
2. Make 5+ commits (WIP, fix, rename, etc.)
3. Squash interactively
4. Test output
5. Force-push to remote
6. Revert using reflog if you mess up

---

### 🧠 Mental Model

Imagine writing a book:

- Every sentence is a commit
- Your feature branch is a draft
- Squashing is like writing the final chapter: clear, edited, no typos

You don’t publish every note — just the best version.

# Section 8: Git Merge Conflicts

### 🧠 What is a Merge Conflict?

A **merge conflict** in Git occurs when Git **can't automatically determine which version of a file to keep** during a merge, rebase, or cherry-pick operation.

💥 The most common scenario: **both branches edited the same line of the same file differently.**

---

### 🔄 Typical Operations That Can Trigger Conflicts

| Operation | Description |
|---|---|
| git merge | Merging two branches with conflicting edits |
| git rebase | Replaying commits over another branch |
| git cherry-pick | Applying a commit that touches the same lines as existing code |
| git pull | Pulling remote changes when you have local edits |
| git stash pop | Applying stashed changes on top of edited files |

---

### 📂 Anatomy of a Merge Conflict

Let's break this down visually. Suppose we have:

**main branch**

console.log("User logged in");

**feature/login-enhancement branch**

console.log("Login successful");

Now if we try to merge feature/login-enhancement into main, Git cannot decide:

- Which line should be kept?
- Are they meant to be merged?
- Is one newer or more important?

---

### 💣 When Git Triggers Conflict

Git identifies that **the same part of a file** was modified **differently in both branches**. Since Git is not semantic-aware (it doesn't understand programming logic), it flags the conflict and **asks you to resolve it manually**.

---

### ⚔️ Conflict Markers in the File

Git updates the conflicting file like this:

```
<<<<<<< HEAD
console.log("User logged in");
=======
console.log("Login successful");
>>>>>>> feature/login-enhancement
```

**Markers:**

- <<<<<<< HEAD → Your branch (current)
- ======= → Separator
- >>>>>>> branch-name → Incoming branch (to be merged)

Your job is to edit the file and **remove the conflict markers**, choosing (or combining) the right code.

---

### Merge Conflict Demo – Step-by-Step

---

**Scenario:**

You are on main. You and a teammate made conflicting edits on the same file.

---

### ✅ Step-by-Step Conflict Demo

```
# 1. Create repo
git init conflict-demo
cd conflict-demo
echo "console.log('Init');" > app.js
git add . && git commit -m "Initial commit"
```

# 2. Create new branch and edit

git checkout -b feature-logging

echo "console.log('User logged in');" > app.js

git commit -am "Add login log"

# 3. Go back to main and make conflicting change

git checkout main

echo "console.log('Login successful');" > app.js

git commit -am "Change login message"

# 4. Merge feature-logging into main

git merge feature-logging

💥 Git will show:

Auto-merging app.js

CONFLICT (content): Merge conflict in app.js

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

---

📄 **Check Conflict Status**

git status

You'll see:

both modified: app.js

---

✍️ **Resolving Conflicts Manually**

Open app.js, you'll see:

```
<<<<<<< HEAD
console.log("Login successful");
=======
console.log("User logged in");
>>>>>>> feature-logging
```

Edit to:

console.log("User logged in successfully");

Then:

git add app.js

git commit -m "Merge feature-logging with conflict resolved"

Done ✅

---

### 🧠 Tools to Resolve Conflicts

| Tool | Command |
|---|---|
| VS Code Merge Tool | Built-in UI |
| Git mergetool (CLI) | git mergetool |
| Meld | git config --global merge.tool meld |
| KDiff3 | GUI |
| SourceTree / GitKraken | Click-based conflict resolution |
| IntelliJ/GitHub Desktop | Visual merge helpers |

---

**Example: git mergetool**

git mergetool

It will launch the configured merge tool for each conflict.

---

### 🚀 Tips for Clean Conflict Resolution

| Tip | Reason |
|---|---|
| Edit the file carefully | Never leave <<<<<<< or ======= markers |
| Test after resolving | Conflicts can introduce bugs |
| Use git diff to verify changes | Ensure only expected differences |
| Use git log --merge | View conflicting commits side-by-side |

| Tip | Reason |
| --- | --- |
| Avoid auto-generated files in merges | Stash or discard them before merging |

---

### 🔍 View History of Conflicts

You can see the commits involved in the conflict:

git log --merge

This helps you understand **why the conflict occurred**.

---

### 🔄 Conflict Recovery & Undo

| Action | Command |
| --- | --- |
| Abort merge | git merge --abort |
| Abort rebase | git rebase --abort |
| Restart merge | Delete MERGE_HEAD |
| Undo changes | git reset --hard |
| Go back | git reflog + git reset <old-hash> |

---

### 🛡 Strategies to Avoid Merge Conflicts

| Practice | Benefit |
| --- | --- |
| Pull before working | Start from latest code |
| Communicate with team | Coordinate shared files |
| Use smaller branches | Easier merges |
| Avoid long-lived branches | Less divergence |
| Prefer feature toggles over massive changes | Incremental updates reduce conflicts |
| Use git blame to understand who changed what | Helps during resolution |

---

**Advanced Conflict Scenarios**

**1. Binary Files**

Git cannot auto-merge binary files.

- Use ours or theirs strategy:

```
git checkout --ours file.pdf
git add file.pdf
```

📌 **2. Line Endings (Windows vs Unix)**

Standardize with .gitattributes:

```
*.js text eol=lf
```

📌 **3. White Space Differences**

Avoid unnecessary conflicts by ignoring whitespace:

```
git diff --ignore-space-change
git merge -Xignore-space-change branch
```

---

📈 **Conflict Frequency by Merge Strategy**

| Strategy | Conflict Likelihood |
|---|---|
| Rebase | High (commits replayed one by one) |
| Merge | Medium (1-time conflict resolution) |
| Cherry-pick | Medium (depends on overlap) |
| Squash Merge | Low (single commit applied) |
| Trunk-based Dev | Low (short-lived branches) |

---

📌 **CI/CD Merge Conflict Handling**

- CI checks PR mergeability
- Rebase + squash in merge strategy
- Auto-fail on unresolved conflicts
- Auto-close stale PRs to reduce risk

GitHub/GitLab auto-block PRs that can’t merge cleanly into the base branch.

### 🛠 Aliases to Speed Up Conflict Workflows

```
git config --global alias.conflict "diff --name-only --diff-filter=U"
```

```
git config --global alias.resolve "add -u && commit -m 'Resolve conflicts'"
```

Then:

```
git conflict
```

```
git resolve
```

---

### 📘 Merge Conflict Resolution in UI Tools

| Tool | How |
|---|---|
| VS Code | Shows inline conflict resolution options ("Accept Incoming", etc.) |
| GitHub | Web-based conflict editor in PR |
| GitLab | Merge conflict editor |
| SourceTree | Side-by-side diff viewer |
| IntelliJ | 3-way merge with visual diff |

---

### 📃 Merge Conflict Command Recap

| Task | Command |
|---|---|
| Merge branches | git merge branch-name |
| View conflicts | git status |
| View conflict history | git log --merge |
| Resolve manually | Edit files, remove markers |
| Use mergetool | git mergetool |
| Abort merge | git merge --abort |
| Add resolved file | git add <file> |
| Complete merge | git commit |

**Final Thoughts**

Merge conflicts aren't evil — they're Git asking you for **clarification**. The key is:

- Know why it happened
- Use the right tools
- Communicate with your team
- Resolve carefully and test

# Section 9: Git Revert

### 🧠 What is git revert?

git revert is a **safe, history-preserving command** used to **undo the effect of a previous commit** by creating a new commit that **reverses the changes** introduced by the original one.

✅ Unlike git reset, revert does **not rewrite commit history** — making it **ideal for shared/public branches**.

---

### 🧩 Why Revert?

| Scenario | Why Use Revert |
|---|---|
| A deployed change broke production | Revert safely without removing history |
| A feature merged prematurely | Revert it from main while fixing it |
| Remove a bad commit without a force push | Shared branches remain stable |
| Legal/audit compliance | Retain traceability while fixing mistakes |

---

### 🧪 How Git Revert Works Internally

- Git looks at the **diff (changes)** introduced by the target commit.
- Then it **creates a new commit** that **reverses** that diff.
- The new commit is **added on top** of the current branch.

**Example:**

git revert <commit-hash>

This creates a new commit with a message like:

Revert "Add new payment processor"

---

### ⚙️ git revert vs git reset vs git checkout

| Command | Behavior | Safe for Teams? |
|---|---|---|
| git revert | Adds a new commit to undo changes | ✅ Yes |

| Command | Behavior | Safe for Teams? |
|---|---|---|
| git reset | Rewrites commit history | 🚫 No (unsafe on shared branches) |
| git checkout (legacy) / git restore | Changes files in working directory | ✅ Yes |

---

🔧 **Git Revert: Basic Syntax**

git revert <commit>

More options:

| Option | Use |
|---|---|
| --no-commit | Stage changes only, don't commit yet |
| --edit | Open editor to modify the commit message |
| --mainline | Used to revert merge commits (see below) |
| -n or --no-commit | Combine with multiple reverts before one commit |
| --signoff | Append signoff footer (used in audits or signed projects) |

---

**Demo: Git Revert in Action**

---

🎯 **Scenario:**

You're in main, and a recent commit caused a bug in production. You want to **undo** it without touching previous work or rewriting history.

---

✅ **Step-by-Step Demo**

# 1. Initialize repo

git init revert-demo

cd revert-demo

echo "Initial content" > app.js

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

# 2. Add a bad change

echo "console.log('Breaking change');" >> app.js

git commit -am "Add breaking log"

# 3. Get commit history

git log --oneline

You'll see something like:

c3d5a1a Add breaking log

0a7b3c2 Initial commit

---

⏪ **Step 4: Revert the Bad Commit**

git revert c3d5a1a

Git opens your default editor:

Revert "Add breaking log"

This reverts commit c3d5a1a.

You can edit or accept and save.

Git then creates:

[main 8b31a73] Revert "Add breaking log"

Done ✅. History is intact, code is fixed.

---

🧠 **What Happens Under the Hood**

- Git internally computes:

diff <parent-of-c3d5a1a> c3d5a1a

- Then it **reverses that diff** and applies it to the current working directory
- A new commit is created with reversed changes

---

### 🔄 Resulting Commit History

```
git log --oneline
```

Output:

8b31a73 Revert "Add breaking log"

c3d5a1a Add breaking log

0a7b3c2 Initial commit

💡 The original "bad" commit is **still in the history**, but now reversed by a new commit.

---

### 🔧 Revert Without Committing Immediately

Use this if you want to **modify or combine multiple reverts**:

```
git revert --no-commit c3d5a1a
```

# Make changes, then:

git commit -m "Revert bad change and update logic"

---

### ⚔️ Reverting Merge Commits (Advanced)

Git cannot revert a merge commit without extra info.

```
git revert -m 1 <merge-commit-hash>
```

Here:

- -m 1 specifies the **mainline parent** (usually the base branch like main)
- Needed because merge commits have **multiple parents**

⚠️ Dangerous if misunderstood — recommended only for advanced users

---

### 🛠 Use Cases by Role

| Role | Use Case |
|---|---|
| Developer | Undo a bad patch or experimental commit |
| DevOps Engineer | Roll back a CI/CD-deployed change |
| QA Engineer | Remove buggy code added by accident |

| Role | Use Case |
|---|---|
| Project Manager | Undo PR merge while keeping traceability |
| Open-source Contributor | Clean up public history without rebasing |

---

### 🧪 Multi-Commit Revert (Sequential)

```
git revert commit1 commit2 commit3
```

Git will revert them in order and pause on each (if conflicts occur).

To batch and commit together:

```
git revert --no-commit commit1 commit2
```

```
git commit -m "Revert feature A and B"
```

---

### 🧱 Revert in GUI Tools

| Tool | Behavior |
|---|---|
| VS Code | Right-click commit → Revert |
| GitHub | On PR → Click “Revert” button |
| GitLab | Same as above |
| GitKraken | Click commit → Revert |
| IntelliJ | Commit window → right-click → Revert |

---

### ⚠️ Common Revert Errors & Fixes

| Error | Cause | Fix |
|---|---|---|
| "Your local changes would be overwritten" | Uncommitted changes | Stash or commit them first |
| "Reverting a merge needs mainline specified" | Trying to revert a merge | Use -m 1 with caution |
| "Conflict during revert" | Overlapping changes | Manually resolve, then git revert --continue |

| Error | Cause | Fix |
|---|---|---|
| Accidentally reverted wrong commit | Wrong hash used | Use git reflog to find previous HEAD and reset |

---

### 💬 Revert vs Reset vs Rebase

| Feature | Revert | Reset | Rebase |
|---|---|---|---|
| Safe for team use | ✅ Yes | ❌ No | ❌ No (on shared branches) |
| History rewrite | ❌ No | ✅ Yes | ✅ Yes |
| Commit added | ✅ Yes | ❌ No | Depends |
| Use case | Undo public commit safely | Remove local commits | Reorder & rewrite local commits |
| Example | Rollback prod | Cleanup WIP | Linearize PR |

---

### 🛡 Git Revert in Production

- Used to **quickly fix issues without downtime**
- GitOps platforms like ArgoCD or FluxCD can monitor Git → Deploy rollback commit automatically
- Maintains full **audit trail**

---

### ⚙️ Git Aliases for Revert Workflow

```
git config --global alias.undo 'revert HEAD'
git config --global alias.rlog 'log --grep=Revert'
```

Now run:

```
git undo     # Reverts the latest commit
git rlog     # Shows all revert commits
```

---

## 📜 Git Revert Cheatsheet

| Action | Command |
|---|---|
| Revert single commit | git revert <hash> |
| Revert with no commit | git revert --no-commit <hash> |
| Revert merge commit | git revert -m 1 <merge-hash> |
| View all reverts | git log --grep=Revert |
| Undo a revert | git revert <revert-hash> (re-reverts) |
| Abort revert | git revert --abort |

---

## 🧪 Practice Ideas

1. Create a repo, make 5 commits
2. Revert the 3rd commit
3. Revert a range of commits
4. Revert a revert (double revert)
5. Revert a merge (advanced)
6. Check diffs between original and revert

---

## 🧬 How Git Revert Affects History Graph

Before:

**A --- B --- C (MAIN)**

After reverting B:

**A --- B --- C --- D (MAIN)**

****

**(REVERT OF B)**

Git adds a new commit; B is still part of history but its changes are undone in D.

---

### 🔁 Revert a Revert: “Undo the Undo”

git revert <revert-commit-hash>

Git sees the previous revert and re-applies the original changes.

💡 Useful when you revert something, then realize it wasn’t broken after all.

---

### 📚 Real-World Use Case: SHACKVERSE Team

- Developer merges an experimental feature to main by mistake.
- CI/CD deploys it → production breaks.
- DevOps uses:

git revert <feature-commit>

- Rollback happens in Git.
- ArgoCD redeploys the fixed version.
- Postmortem includes the revert ID for traceability.

---

### ✅ Final Thoughts

Git Revert is the **safest, cleanest** way to undo changes on a shared branch.

✅ Safe in production
✅ Maintains team trust
✅ Keeps history readable
✅ Works with CI/CD

# Section 10: Git

### 🧠 What is git reset?

git reset is a **powerful Git command** used to **move the HEAD pointer and branch reference to a new commit**, potentially modifying the staging area (index) and working directory depending on the mode.

⚠️ Depending on how you use it, git reset can either be **harmless** or **dangerously destructive**.

---

### 📍 Three Core Components of Git State

To fully understand git reset, you must understand the **three main states** Git manages:

| Component | Description |
|---|---|
| HEAD | Points to the current commit |
| Index (Staging Area) | Files added with git add |
| Working Directory | Actual files on disk |

---

### Git Reset Modes Explained

| Mode | Command | What It Does |
|---|---|---|
| **Soft** | git reset --soft HEAD~1 | Moves HEAD, keeps staging and working directory intact |
| **Mixed** (default) | git reset --mixed HEAD~1 | Moves HEAD, resets staging area, working directory untouched |
| **Hard** | git reset --hard HEAD~1 | Moves HEAD, resets staging area, and working directory |

---

### 🔬 Under the Hood: What Git Reset Actually Does

- Git moves the HEAD pointer to a different commit (usually a previous one).
- Then, based on the mode:
    - **Soft**: Index and working tree are untouched.
    - **Mixed**: Index is reset to match HEAD; working directory untouched.

- **Hard**: Everything (HEAD, index, working tree) is reset.

---

**Visual Representation**

**Before Reset:**

**HEAD -> C**

Index: matches C

Working Dir: matches C

**After --soft HEAD~1:**

**HEAD -> B**

Index: matches C

Working Dir: matches C

**After --mixed HEAD~1:**

**HEAD -> B**

Index: matches B

Working Dir: matches C

**After --hard HEAD~1:**

**HEAD -> B**

Index: matches B

Working Dir: matches B

---

**Git Reset Use Cases**

| Use Case | Best Mode |
|---|---|
| Undo latest commit but keep changes staged | Soft |
| Unstage a file accidentally added | Mixed |
| Rollback working directory and commits completely | Hard |
| Rewind to clean state before a bad rebase | Hard |
| Remove sensitive data from recent commit | Hard (after removing file and rewriting history) |

**Git Reset: Demo Setup**

**Step-by-Step**

```
git init reset-demo
cd reset-demo
echo "Line 1" > file.txt
git add . && git commit -m "Initial commit"
```

Make 2 more commits:

```
echo "Line 2" >> file.txt
git add . &&  git commit -am "Add line 2"

echo "Line 3" >> file.txt
git add . && git commit -am "Add line 3"
```

Now check history:

```
git log --oneline
```

You'll see:

c3d Add line 3

b2a Add line 2

a1b Initial commit

---

**1. Soft Reset Demo**

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

Effect:

- HEAD now points to Add line 2
- **Index still contains changes from "Add line 3"**
- git status: changes ready to commit

```
git status
```

# On branch main

# Changes to be committed:

#  modified: file.txt

✅ Perfect if you want to **reword** or **amend** the last commit:

```
git commit -m "Add line 3 with corrections"
```

---

### 🖌️ 2. Mixed Reset Demo (Default)

```
git reset --mixed HEAD~1
# or just:
git reset HEAD~1
```

Effect:

- HEAD moves back one commit
- **Index is reset**
- Working directory still has changes

```
git status
# Changes not staged for commit:
#   modified: file.txt
```

✅ Best for **unstaging files**:

```
git add file.txt   # re-stage selectively
```

---

### 💣 3. Hard Reset Demo

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

Effect:

- HEAD moves back one commit
- Index and working directory are both reset
- The commit **and changes** from "Add line 3" are **gone**

```
git log --oneline   # commit "Add line 3" gone
cat file.txt        # only Line 1 and Line 2
```

⚠️ Danger! You've lost both the commit and its changes unless backed up.

---

**Safety Precautions with Git Reset**

| Safety Tip | Reason |
|---|---|
| Use git reflog | You can recover lost commits |
| Always stash or commit before hard reset | Prevent data loss |
| Never reset --hard on shared branches | May overwrite others' work |
| Use --soft when amending commit messages | Keeps your code safe |

---

🧱 **Reflog: Recovering After Mistakes**

git reflog

Shows history of HEAD movements:

6e9 HEAD@{0}: reset: moving to HEAD~1

c3d HEAD@{1}: commit: Add line 3

Recover lost commit:

git checkout c3d

# Or

git reset --hard c3d

---

🛠 **Reset a Specific File (Not Entire Commit)**

git reset HEAD file.txt

- Removes file.txt from staging area
- Keeps changes in working directory

Useful when you added a file by mistake and want to **unstage** without losing it.

---

### Git Reset vs Revert vs Checkout

| Feature | Reset | Revert | Checkout |
|---|---|---|---|
| Rewrite history | ✅ | ❌ | ❌ |
| Safe for teams | ❌ (on shared branches) | ✅ | ✅ |
| Affects commit history | ✅ | ✅ (non-destructive) | ❌ |
| Undo last commit | ✅ | ✅ | ❌ |
| Restore file | ✅ | ❌ | ✅ |

---

### 🧠 Advanced Reset Scenarios

#### 💡 Reset to Specific Commit

```
git reset --mixed <commit-hash>
```

Move your HEAD to an earlier commit by SHA.

---

#### 💡 Clean up WIP commits

```
git reset --soft HEAD~3
git commit -m "Refactor: consolidated changes"
```

Useful before opening a Pull Request.

---

#### 💡 Partial Hard Reset

```
git checkout HEAD file.txt
```

Just reset one file to its last committed state. Does not touch other files.

---

### Real Use Cases from DevOps

| Role | Use Case |
|---|---|
| Developer | Reset HEAD~1 after accidental commit |
| DevOps Engineer | Hard reset cloned repo to clean state |

| Role | Use Case |
|---|---|
| QA Engineer | Reset feature branch before retesting |
| Release Manager | Roll back to known commit before tagging |
| GitOps Pipeline | Auto-reset state before re-applying configuration |

---

**Common Git Reset Mistakes**

| Mistake | Fix |
|---|---|
| Used --hard on wrong branch | Use reflog to find lost commit |
| Staged wrong files and committed | Use --soft, re-stage |
| Hard reset shared branch | Use git revert instead |
| Lost important work | Check .git/lost-found or reflog |

---

**🛠 Git Reset + Git Reflog = Recovery Superpowers**

1. Make a bad reset:

```
git reset --hard HEAD~2
```

2. Realize mistake:

```
git reflog
```

3. Recover:

```
git reset --hard <previous-head-hash>
```

---

**📄 Git Reset Command Summary**

| Task | Command |
|---|---|
| Undo last commit, keep staged | git reset --soft HEAD~1 |
| Undo last commit, keep code only | git reset --mixed HEAD~1 |
| Wipe commit + code | git reset --hard HEAD~1 |
| Reset specific file | git reset HEAD file.js |

| Task | Command |
| --- | --- |
| Recover from reset | git reflog + git reset |

---

**Real World Example**

A XYZ engineer mistakenly committed an environment secret file. Instead of rewriting public history, the engineer:

1. Created a new commit removing the file
2. Reset his local branch:

git reset --soft HEAD~1

3. Squashed everything into one clean commit:

git commit -m "Remove secret file + cleanup"

---

✅ **Final Thoughts**

Git Reset is **one of the most powerful and misunderstood Git commands**.

✅ Use it to fix mistakes
✅ Use --soft for clean commit history
✅ Use --mixed to unstage
❗ Use --hard with backups or in safe branches

When used properly, it will **supercharge your Git workflow** and give you **confidence to manage any branch state**.