Mastering Git

A comprehensive visual guide to professional version control for developers

Start Learning

If you've ever struggled with managing code versions using folders named "FINAL", "FINAL_v2", "FINAL_FINAL_FOR_REAL" — you know the pain. Git solves this chaos permanently.

This isn't another boring command reference. I'll walk you through Git **the way you'll actually use it at work** — from cloning the company repo to getting your code merged into production.

What Makes This Guide Different

- ✅ Structured around the **actual workflow** companies use
- ✅ Commands introduced **when you need them** (Just-in-Time Learning)
- ✅ **Undo operations** at every step (because mistakes happen)
- ✅ The "why" behind every command, not just the "what"

What is Git?

Git is a distributed version control system that tracks changes in your code.

In plain English: Git is a **time machine + collaboration manager** for your code.

"Version Control System" means:

- It tracks every change you make to your code
- It remembers what your code looked like yesterday, last week, or last year
- It lets you go back to any previous version instantly
- It shows you exactly what changed between versions

"Distributed" means:

- Everyone on your team has a complete copy of the entire project history
- You don't need internet to work or save your progress
- If the server crashes, anyone can restore everything
- No single point of failure

Why Every Company Uses Git

Reason 1: Time Travel for Your Code

Remember those 23 "FINAL" folders? That never happens with Git.

bash Copy

```bash
# See all previous versions
git log --oneline

# Compare today with last week
git diff HEAD~7
```

```
# Go back to any version
git checkout abc123
```

Reason 2: Safe Collaboration

When two developers edit the same file, Git doesn't silently delete anyone's work. Instead:

1. Developer A edits checkout.js and commits
2. Developer B edits checkout.js and tries to push
3. **Git says:** "Hold on! There are changes you haven't seen. Let me help you merge them."
4. Both changes are preserved

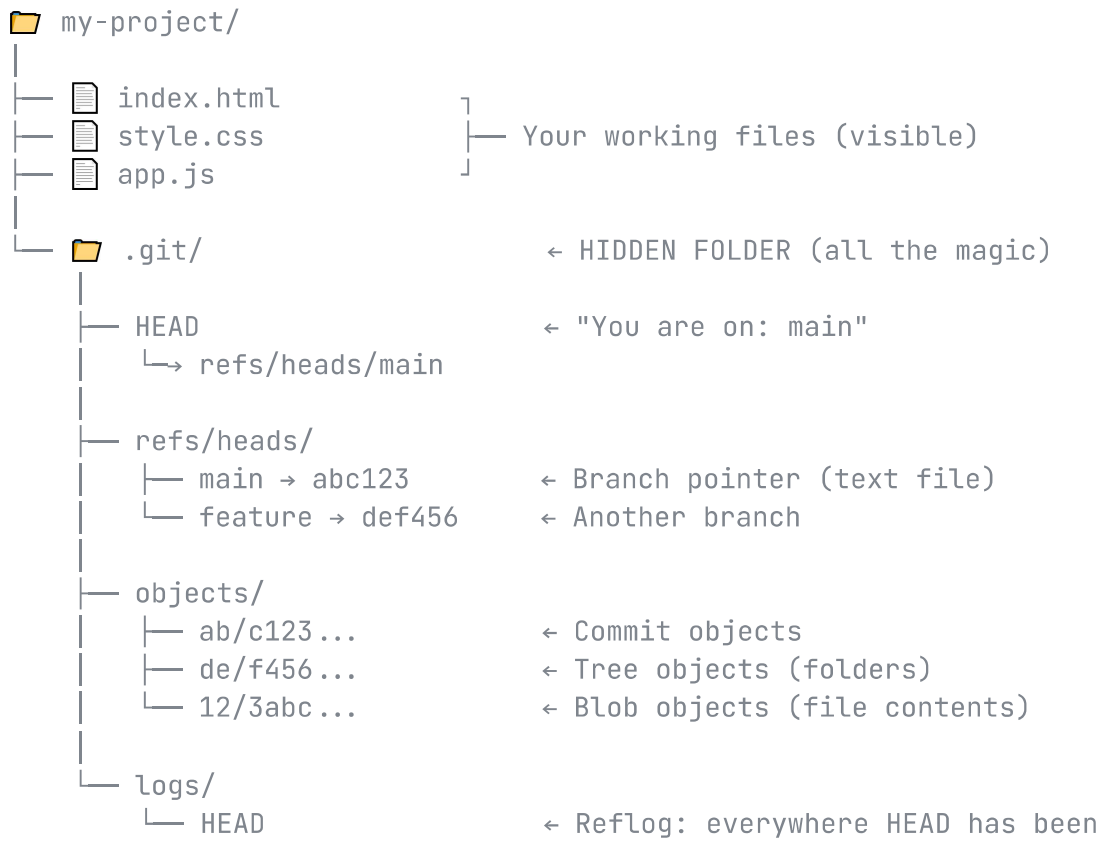
Reason 3: Experiment Without Fear

Want to try a risky redesign? Create a branch, experiment freely. If it fails, delete the branch. If it works, merge it in. **Main code stays safe either way.**

Understanding the .git Folder

Look inside your cloned project. There's a hidden `.git` folder that contains **everything:**

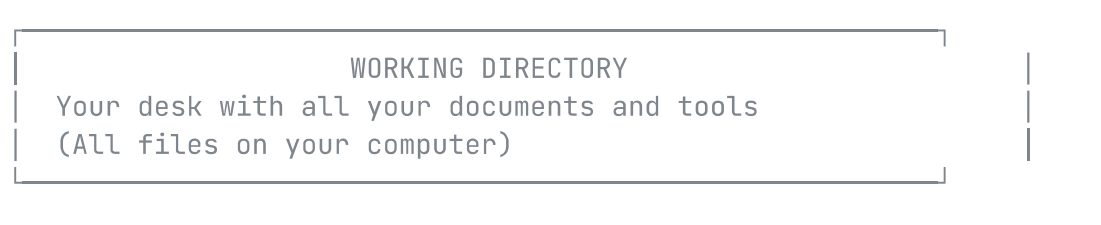
Local Repository Structure

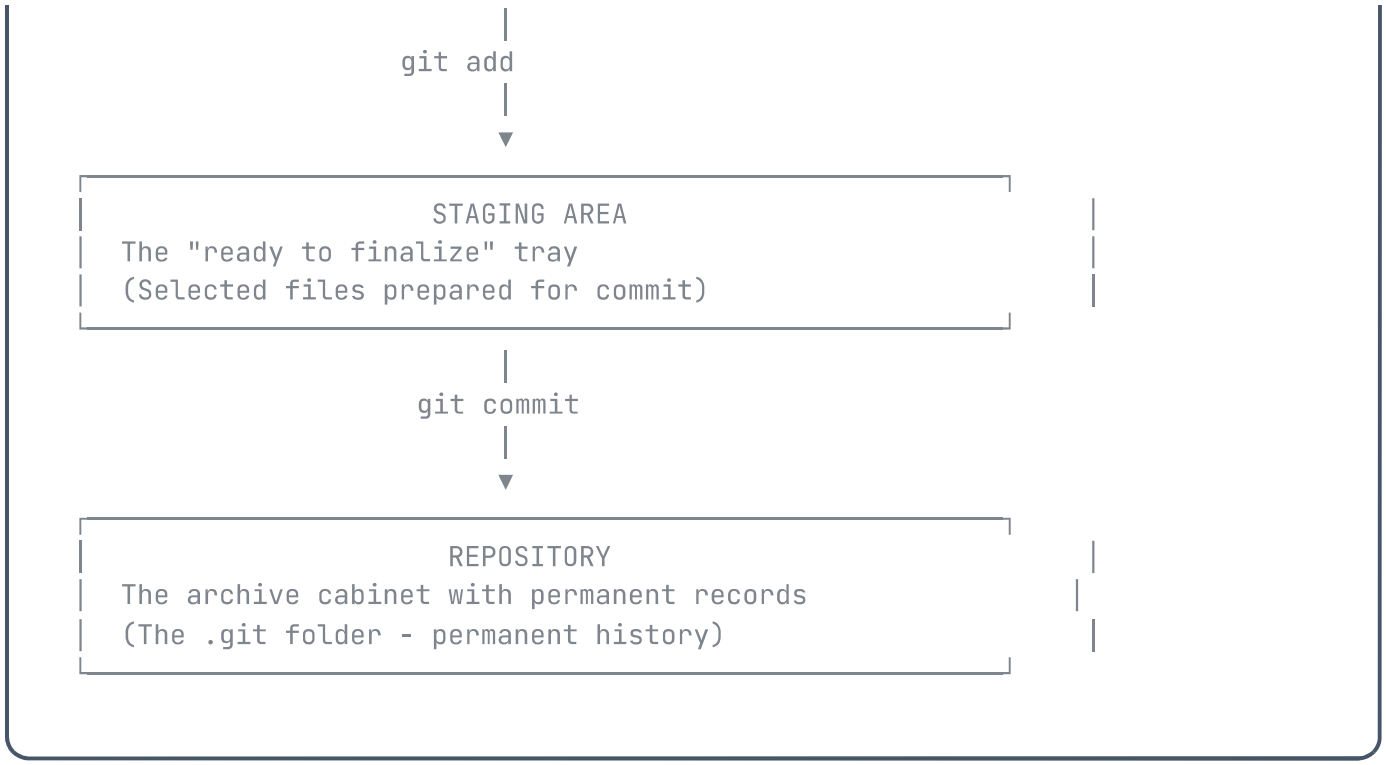


> **NOTE**
>
> **A branch is just a 40-character text file.** That's it. Branches are incredibly lightweight—creating one takes microseconds.

The Three Stages of Git

Think of it like working in an office:





- **Working Directory** = Your desk with all your documents and tools (all files that exist on your computer)
- **Staging Area** = The stack of papers you put in the "ready to finalize" tray (you choose specific changes to prepare for submission)
- **Repository** = The archive cabinet where final approved documents are stored (this is the permanent record)

Configure Your Identity

Every commit you make will be stamped with this information:

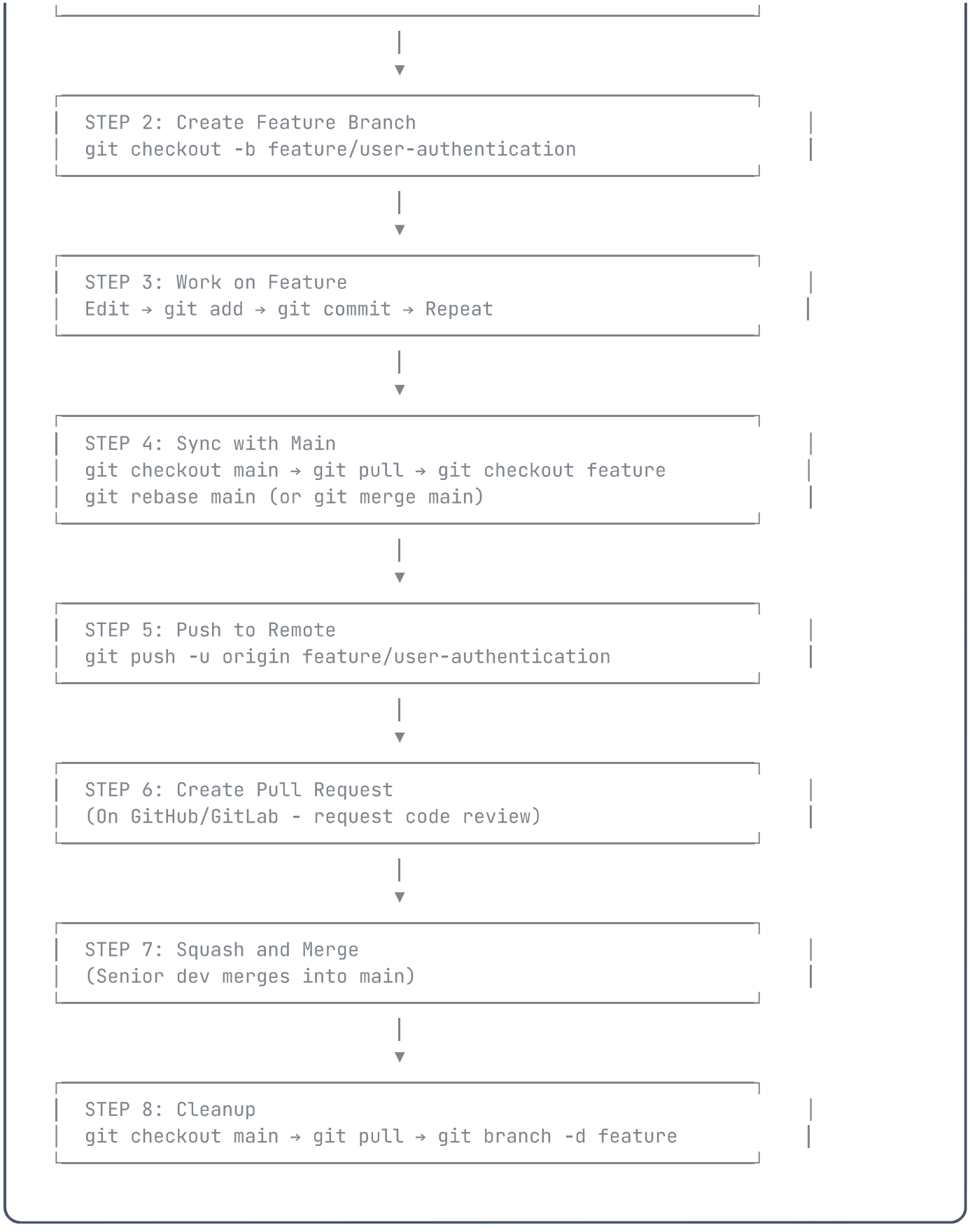
bash Copy

```
# Set your name (appears in commits)
git config --global user.name "Your Name"

# Set your email (appears in commits)
git config --global user.email "your.email@company.com"

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

# Verify your settings
git config --list

# Set default code editor to VS Code
git config --global core.editor "code --wait"
```

> **Why this matters**
>
> When something breaks, the team needs to know who wrote that code—not to blame, but to ask questions.

This is how professional development teams work. Follow this flow for every feature you build.

Professional Git Workflow



Step 1: Clone the Main Repository

What it does: Downloads the complete project (including all history) to your computer.

bash Copy

```bash
# Clone via HTTPS
git clone https://github.com/company/repo-name.git

# Clone via SSH (recommended for frequent use)
git clone git@github.com:company/repo-name.git

# This creates:
# project-name/
#   ├── all project files
#   └── .git/          ← Complete history lives here
```

What happens:

1. Creates a folder with the repository name
2. Downloads all files and complete history
3. Sets up a connection called "origin" pointing to the company repo
4. Checks out the default branch (usually main)

> 🚨 **UNDO**
>
> Cloned to wrong location? Just delete the folder and clone again. Nothing on the server is affected.

> ✨ **SSH vs HTTPS**
>
> **Why SSH?** SSH uses key-based authentication, so you won't need to enter your password for every push. Set up SSH keys once, and Git remembers you.

Step 2: Create Your Feature Branch

The Golden Rule: Never work directly on main. Always create a feature branch.

```bash
# First, make sure you have the latest main
git checkout main
git pull origin main

# Create AND switch to a new branch
git checkout -b feature/user-authentication

# Or the modern way (Git 2.23+)
git switch -c feature/user-authentication
```

Branch Naming Conventions

Type	Example	When to Use
`feature/`	`feature/add-login`	New features
`fix/`	`fix/payment-bug`	Bug fixes
`hotfix/`	`hotfix/security-patch`	Urgent production fixes
`refactor/`	`refactor/auth-module`	Code improvements
`docs/`	`docs/api-guide`	Documentation

Step 3: Work on Your Feature

This is where you spend most of your time. The cycle is: **Edit** → **Stage** → **Commit** → **Repeat**.

Check Status (Do This Constantly!)

bash Copy

```
git status

# Compact format
git status -s

# Output:
#  M src/App.js           # Modified, not staged
# M  src/index.js         # Modified and staged
# ?? src/NewFile.js       # Untracked
# A  src/AddedFile.js     # New file, staged
```

Stage Your Changes

bash Copy

```
# Stage specific file
git add src/login.js

# Stage multiple files
git add src/login.js src/auth.js

# Stage all changes in current directory
git add .

# Stage part of a file (interactive—powerful!)
git add -p src/login.js
```

Review What You're About to Commit

bash Copy

```bash
# See unstaged changes
git diff

# See staged changes (what will be committed)
git diff --staged

# Output shows:
# - lines (deleted)
# + lines (added)
```

Commit Your Changes

A commit is a **permanent snapshot** of your staged changes.

bash Copy

```bash
git commit -m "feat: add user login form validation"
```

Conventional Commit Format

Type	Example	When to Use
`feat:`	`feat: add dark mode`	New features
`fix:`	`fix: resolve login timeout`	Bug fixes
`docs:`	`docs: update API examples`	Documentation
`style:`	`style: format code`	Code formatting
`refactor:`	`refactor: simplify auth logic`	Code improvements
`test:`	`test: add login tests`	Adding tests

Step 4: Sync with Main

While you were working, your teammates pushed changes to main. Your branch is now outdated. Before pushing, you **must** sync.

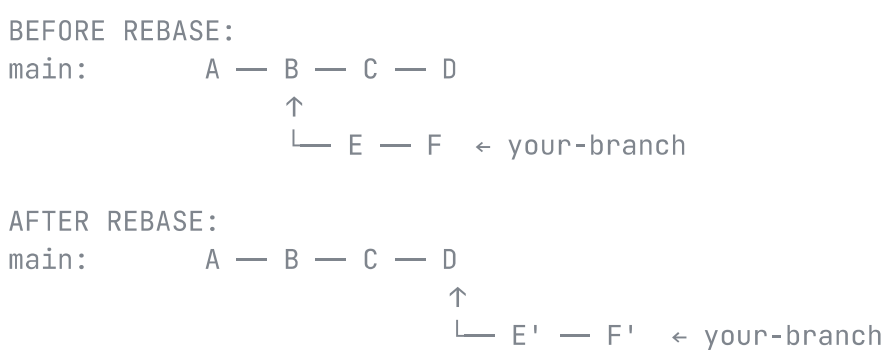
> ⚠️ **The Golden Rule of Conflicts**
>
> You should **never** leave conflicts for the Senior Dev or Maintainer to solve. They don't know why you changed a specific line of code.

Approach A: Rebase (Preferred for Clean History)

bash Copy

```bash
# 1. Switch to main and get latest
git checkout main
git pull origin main

# 2. Switch back to your branch
git checkout feature/user-authentication

# 3. Rebase onto main
git rebase main
```

Rebase vs Merge



```
------------------------------------------------

BEFORE MERGE:
main:       A — B — C — D
                  ↑
                  └— E — F  ← your-branch

AFTER MERGE:
main:       A — B — C — D ——————————┐
                  ↑                 ↓
                  └— E — F ———————— M ← your-branch
                                  (merge commit)
```

If Conflicts Occur During Rebase

bash Copy

```bash
# Git stops and shows conflicted files
git status

# 1. Open each conflicted file
# 2. Look for conflict markers:
# <<<<<<< HEAD
# code from main
# =======
# your code
# >>>>>>> your-commit

# 3. Edit to keep what you want, remove markers
# 4. Stage the resolved file
git add resolved-file.js

# 5. Continue rebase
git rebase --continue
```

🚨 UNDO

> Rebase going badly? `git rebase --abort` returns everything to how it was before rebase started.

Step 5: Push to Remote

bash Copy

```bash
# First-time push (sets upstream tracking)
git push -u origin feature/user-authentication

# Subsequent pushes
git push

# If you rebased after already pushing
git push --force-with-lease origin feature/user-authentication
```

> 🚨 **CAUTION**
>
> **NEVER force push to main or any shared branch!** This rewrites history for everyone.

Step 6: Create Pull Request

On GitHub/GitLab, create a Pull Request to merge your branch into main.

A Good PR Includes:

- Clear title describing the change
- Description of what and why

- Screenshots for UI changes
- Links to related issues

Step 7: Squash and Merge

This is typically done by the approver (senior dev or team lead), not you.

```
YOUR PR COMMITS:
E: "feat: add login form"
F: "fix: typo in label"
G: "style: adjust button padding"
H: "fix: forgot validation"

AFTER SQUASH:
S: "feat: add user authentication (#123)"

All your messy WIP commits become one clean commit on main.
```

Step 8: Cleanup

bash Copy

```bash
# Switch to main
git checkout main

# Get the merged changes
git pull origin main

# Delete local feature branch (it's merged, you don't need it)
git branch -d feature/user-authentication

# Delete remote feature branch (optional)
git push origin --delete feature/user-authentication
```

Mistakes happen. Git has your back.

The Reflog: Your Time Machine

The reflog tracks everywhere HEAD has been—even "deleted" commits.

bash Copy

```
git reflog

# Output:
# e4f5g6h HEAD@{0}: reset: moving to HEAD~3
# a1b2c3d HEAD@{1}: commit: my important work    ← "deleted" commit!
# 9876543 HEAD@{2}: commit: previous work

# Recover "lost" commits:
git reset --hard a1b2c3d
```

> **NOTE**
>
> Reflog entries expire after **30 days** for unreachable commits and **90 days** for reachable ones. Recover lost work promptly!

The Complete Undo Cheatsheet

bash Copy

```
# ──────────────────────────────────────────────────────────
# UNSTAGE A FILE (after git add, before commit)
# ──────────────────────────────────────────────────────────
git reset <file>
# or (Git 2.23+)
git restore --staged <file>

# ──────────────────────────────────────────────────────────
# DISCARD CHANGES IN A FILE (before staging)
# ──────────────────────────────────────────────────────────
git checkout -- <file>
# or (Git 2.23+)
git restore <file>

# ──────────────────────────────────────────────────────────
# UNDO LAST COMMIT (keep changes staged)
# ──────────────────────────────────────────────────────────
git reset --soft HEAD~1

# ──────────────────────────────────────────────────────────
# UNDO LAST COMMIT (keep changes unstaged)
# ──────────────────────────────────────────────────────────
git reset HEAD~1

# ──────────────────────────────────────────────────────────
# UNDO LAST COMMIT (DELETE changes completely) ⚠
# ──────────────────────────────────────────────────────────
git reset --hard HEAD~1

# ──────────────────────────────────────────────────────────
# UNDO A PUSHED COMMIT (safe - creates "undo" commit)
# ──────────────────────────────────────────────────────────
git revert <commit-hash>

# ──────────────────────────────────────────────────────────
# RECOVER "LOST" COMMITS
# ──────────────────────────────────────────────────────────
git reflog
git reset --hard <commit-from-reflog>
```

Understanding Reset Modes

Mode	Moves HEAD?	Clears Staging?	Clears Working Dir?
`--soft`	✓	✗	✗
`--mixed` (default)	✓	✓	✗
`--hard` ⚠	✓	✓	✓

Reset vs Revert: When to Use Which

Scenario	Use	Why
Undo local commit (not pushed)	`git reset`	Rewrites history safely since no one else has it
Undo pushed commit	`git revert`	Creates new "undo" commit, preserves history
Unstage files	`git reset <file>`	Safe, only affects staging area
Discard all local changes	`git reset --hard`	Nuclear option - use with caution

Common Scenarios

"I committed to the wrong branch!"

bash Copy

```bash
# You're on main but should be on feature branch

# 1. Create the correct branch (with your commit)
git branch correct-branch

# 2. Reset main back to before your commit
git reset --hard HEAD~1

# 3. Switch to correct branch
git checkout correct-branch
```

"I have uncommitted changes and need to switch branches!"

bash Copy

```bash
# Option 1: Stash (if you want to keep changes)
git stash push -m "WIP: description"
git checkout other-branch
# ... do work ...
git checkout original-branch
git stash pop

# Option 2: Commit (if changes are ready)
git add .
git commit -m "WIP: save progress"
git checkout other-branch
```

Daily Commands

bash Copy

```
git status                  # What's changed?
git add .                   # Stage all
git commit -m "message"     # Save snapshot
git push                    # Upload to remote
git pull                    # Download and merge
git log --oneline -10       # Recent history
```

Branch Operations

bash Copy

```
git branch                  # List branches
git checkout -b new-branch  # Create and switch
git checkout main           # Switch to main
```

```
git merge feature-branch      # Merge into current
git branch -d old-branch      # Delete branch
```

Sync Operations

bash Copy

```
git fetch origin              # Download without merging
git pull origin main          # Download and merge
git rebase main               # Replay commits on main
git push -u origin branch     # First push with tracking
git push --force-with-lease   # Force push (after rebase)
```

Stashing

bash Copy

```
git stash                     # Save temporarily
git stash pop                 # Restore and delete
```

```
git stash list                # Show all stashes
git stash apply stash@{0}     # Restore without deleting
```

- ✅ **The Mental Model:** Git as time machine + collaboration manager
- ✅ **The Three Stages:** Working Directory → Staging → Repository
- ✅ **The .git Folder:** What HEAD, refs, and objects actually are
- ✅ **The Company Workflow:** Clone → Branch → Work → Sync → Push → PR → Merge → Cleanup
- ✅ **Undo Operations:** Reset, revert, restore, and reflog for every situation
- ✅ **Best Practices:** Conventional commits, branch naming, never force-push to main

Next Steps

1. **Practice the workflow** on a real project
2. **Make mistakes intentionally** and practice recovering
3. **Use git status constantly** until it's second nature
4. **Read error messages**—Git's errors are actually helpful

> ✨ **Pro tip**
>
> Every expert developer was once confused by Git. The difference is they kept practicing. You've got this! 🚀

Created with ❤️ for developers learning Git

Based on the comprehensive guide by Mostafa Hatem