Git from Scratch

A Hands-On Tutorial for SRE Engineers

Built for Siddharth's SRE Bootcamp

References: git-scm.com/doc | git-scm.com/book

Every section = Type the commands. See the output. Build muscle memory.

Lab 1: Install & Configure Git

Step 1: Verify Installation

Open your terminal and check if Git is installed:

```
$ git --version
# Expected: git version 2.x.x

# If not installed:
# Ubuntu/Debian:  sudo apt install git
# macOS:          brew install git
# Windows:        Download from git-scm.com/downloads
```

Step 2: Set Your Identity (Required)

Git tags every commit with your name and email. This is NOT optional.

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

# Verify what you set:
$ git config --global user.name
Siddharth
$ git config --global user.email
your.email@example.com
```

Step 3: Set Your Default Editor & Branch

```
# Set VS Code as default editor (or vim, nano, etc.)
$ git config --global core.editor "code --wait"

# Set default branch name to 'main' (modern standard)
$ git config --global init.defaultBranch main

# See ALL your config settings:
$ git config --list
```

SRE Tip: Config Levels

Git has 3 config levels: --system (all users), --global (your user), --local (this repo only). Local overrides global, global overrides system. In production, you'll often set different emails for work vs personal repos using --local.

Checkpoint

You should be able to answer: What command shows your Git username? What's the difference between --global and --local config?

Lab 2: Create Your First Repository

Step 1: Initialize a New Repo

```
$ mkdir my-sre-project
$ cd my-sre-project
$ git init
# Output: Initialized empty Git repository in .../my-sre-project/.git/

# Look at what git created:
$ ls -la
# You'll see a hidden .git/ directory

$ ls .git/
# HEAD  config  description  hooks/  info/  objects/  refs/
```

Important

The .git/ directory IS your repository. Everything Git knows lives here. Delete it and you lose all history. Never manually edit files inside .git/ unless you know exactly what you're doing.

Step 2: Understand the Three Areas

Git has 3 areas you MUST understand. Everything else builds on this:

```
Working Directory    -->    Staging Area    -->    Repository (.git/)
(your files)                (git add)              (git commit)

Think of it as:
  Desk               -->    Box             -->    Warehouse
  (edit files)               (pack items)           (ship & store)
```

Step 3: Your First Commit Cycle

```
# 1. Create a file
$ echo '# My SRE Project' > README.md
$ echo 'print("hello from SRE")' > app.py

# 2. Check status -- see what Git knows
$ git status
# Both files show as 'Untracked files' (red)

# 3. Stage files (move to staging area)
$ git add README.md
$ git status
# README.md is green (staged), app.py still red (untracked)

# 4. Stage everything
$ git add .
$ git status
```

```
# Both files green (staged, ready to commit)

# 5. Commit -- save a snapshot
$ git commit -m "Initial commit: add README and app"
# Output: [main (root-commit) abc1234] Initial commit...

# 6. View your commit
$ git log
# Shows commit hash, author, date, message
```

Hands-On Exercise

Create 3 more files: config.yaml, Dockerfile, and monitoring.py. Stage them one at a time, running 'git status' after each 'git add'. Watch how files move from red (untracked) to green (staged). Then commit all three in one commit.

Lab 3: Tracking Changes -- add, diff, log

Step 1: Make Changes and See the Diff

```
# Edit an existing tracked file
$ echo 'import os' >> app.py
$ echo 'print(os.getcwd())' >> app.py

# See what changed (unstaged changes)
$ git diff
# Shows + lines (additions) in green, - lines (deletions) in red

# Stage the changes
$ git add app.py

# Now diff shows nothing! Changes are staged.
$ git diff
# (empty)

# To see STAGED changes (what will go in next commit):
$ git diff --staged
# Shows your staged additions
```

Common Mistake

'git diff' shows UNSTAGED changes only. 'git diff --staged' shows what's queued for commit. Many beginners get confused when 'git diff' shows nothing after 'git add'. This is correct behavior!

Step 2: The Complete Modify-Stage-Commit Cycle

```
# Make another change
$ echo '# Configuration' > config.yaml
$ echo 'log_level: INFO' >> config.yaml

# Check, stage, commit
$ git status           # see what changed
$ git diff             # see the actual changes
$ git add config.yaml  # stage it
$ git diff --staged    # verify what you're committing
$ git commit -m "Add logging configuration"
```

Step 3: Reading Git Log Like a Pro

```
# Full log
$ git log

# Compact one-line view (you'll use this daily)
$ git log --oneline
# abc1234 Add logging configuration
```

```
# def5678 Initial commit: add README and app

# Show what changed in each commit
$ git log --oneline --stat

# Beautiful graph (useful with branches later)
$ git log --oneline --graph --all

# Show last 3 commits only
$ git log -3

# Show commits that touched a specific file
$ git log -- app.py
```

SRE Real-World Use

When debugging a production incident, 'git log --oneline --since="2 hours ago"' shows you what changed recently. 'git log --author="name"' helps find who made a change. These are invaluable for incident response.

Lab 4: Undoing Things -- restore, reset, revert

Step 1: Discard Unstaged Changes

```
# Make a change you want to undo
$ echo 'BROKEN CODE' >> app.py
$ git diff  # see the damage

# Discard the change (restore to last committed version)
$ git restore app.py
$ cat app.py  # file is back to normal

# Old syntax (still works): git checkout -- app.py
```

Step 2: Unstage a File

```
# Accidentally staged something
$ echo 'secret_key=abc123' > secrets.txt
$ git add secrets.txt
$ git status  # secrets.txt is staged (green)

# Unstage it (move back from staging to working directory)
$ git restore --staged secrets.txt
$ git status  # secrets.txt is untracked again (red)

# Old syntax (still works): git reset HEAD secrets.txt
```

Step 3: Amend the Last Commit

```
# Oops, typo in commit message
$ git commit --amend -m "Add logging configuration (corrected)"

# Or forgot to add a file to the last commit
$ echo '# Monitoring' > monitoring.py
$ git add monitoring.py
$ git commit --amend --no-edit
# Adds monitoring.py to the previous commit, same message
```

Step 4: Revert a Commit (Safe for Shared History)

```
# First, make a commit to revert
$ echo 'bad config' > bad.txt
$ git add bad.txt && git commit -m "Add bad config"

# Revert it -- creates a NEW commit that undoes the changes
$ git revert HEAD
# Editor opens. Save the default revert message.
```

```
$ git log --oneline
# You'll see: Revert "Add bad config" as the newest commit
$ ls bad.txt  # File is gone
```

revert vs reset

git revert: SAFE. Creates a new commit that undoes changes. Use on shared branches. git reset: DANGEROUS on shared branches. Rewrites history. Use only on local, unpushed commits. Rule: If you've pushed it, revert it. If it's local only, you can reset.

Step 5: Git Reset (Local Only!)

```
# Make 2 test commits
$ echo 'test1' > test1.txt && git add . && git commit -m "test1"
$ echo 'test2' > test2.txt && git add . && git commit -m "test2"
$ git log --oneline  # see both commits

# --soft: undo commit, keep changes staged
$ git reset --soft HEAD~1
$ git status  # test2.txt is staged (green)

# --mixed (default): undo commit, keep changes unstaged
$ git reset HEAD~1
$ git status  # test1.txt is modified (red)

# --hard: undo commit, DELETE all changes (nuclear option)
$ git reset --hard HEAD~1
$ git status  # clean, changes are GONE
```

Hands-On Exercise

Create 3 commits (file1, file2, file3). Use 'git reset --soft HEAD~1' and check status. Then commit again. Now try '--mixed'. Finally try '--hard'. Notice how each mode treats your files differently.

Lab 5: Branching -- The Core of Git Workflows

Step 1: Create and Switch Branches

```
# See current branch
$ git branch
# * main

# Create a new branch
$ git branch feature/add-monitoring

# Switch to it
$ git switch feature/add-monitoring
# Old syntax: git checkout feature/add-monitoring

# Shortcut: create AND switch in one command
$ git switch -c feature/add-alerting
# Old syntax: git checkout -b feature/add-alerting

# List all branches
$ git branch
# * feature/add-alerting
#   feature/add-monitoring
#   main
```

Step 2: Work on a Branch

```
# Make sure you're on your feature branch
$ git branch  # star (*) shows current branch

# Create work on this branch
$ cat > alerting.py << 'EOF'
import requests

def send_alert(message):
    print(f'ALERT: {message}')
    # TODO: integrate with PagerDuty
EOF

$ git add alerting.py
$ git commit -m "feat: add alerting module skeleton"

# This commit exists ONLY on feature/add-alerting
$ git log --oneline

# Switch back to main -- alerting.py disappears!
$ git switch main
$ ls alerting.py  # No such file!
```

```
# Switch back -- it's there again
$ git switch feature/add-alerting
$ ls alerting.py  # It's back!
```

Why This Matters for SRE

In real teams, you NEVER commit directly to main. The pattern is: create branch -> do work -> open PR -> get review -> merge. Branch naming conventions matter: feature/, bugfix/, hotfix/, release/ are common prefixes.

Step 3: Visualize Branches

```
# The most useful git command for understanding branches:
$ git log --oneline --graph --all

# Example output:
# * abc1234 (feature/add-alerting) feat: add alerting module
# | * def5678 (feature/add-monitoring) feat: add monitoring
# |/
# * ghi9012 (main) Initial commit
```

Lab 6: Merging Branches

Step 1: Fast-Forward Merge (Simple Case)

```
# Setup: make sure feature branch has commits ahead of main
$ git switch main
$ git log --oneline --all --graph  # visualize before merge

# Merge feature into main
$ git merge feature/add-alerting
# Output: Fast-forward

$ git log --oneline --graph
# main pointer simply moved forward -- no merge commit needed
```

Step 2: Three-Way Merge (Both Branches Changed)

```
# Create diverging branches
$ git switch -c feature/logging
$ echo 'import logging' > logger.py
$ git add logger.py && git commit -m "feat: add logger"

# Go back to main and make a different change
$ git switch main
$ echo '# Updated README' > README.md
$ git add README.md && git commit -m "docs: update README"

# Now main and feature/logging have DIVERGED
$ git log --oneline --graph --all

# Merge -- this creates a merge commit
$ git merge feature/logging
# Editor opens for merge commit message. Save it.

$ git log --oneline --graph
# You'll see the branches joining back together
```

Step 3: Handling Merge Conflicts

Conflicts happen when both branches changed the SAME lines. Let's create one on purpose:

```
# Setup: create a conflict
$ git switch -c branch-a
$ echo 'log_level: DEBUG' > config.yaml
$ git add . && git commit -m "Set DEBUG logging"

$ git switch main
$ echo 'log_level: WARNING' > config.yaml
$ git add . && git commit -m "Set WARNING logging"
```

```
# Try to merge -- CONFLICT!
$ git merge branch-a
# CONFLICT (content): Merge conflict in config.yaml
# Automatic merge failed; fix conflicts and then commit.

# See what's conflicted:
$ git status  # config.yaml shows as 'both modified'

# Open the file -- you'll see conflict markers:
$ cat config.yaml
# <<<<<<< HEAD
# log_level: WARNING
# =======
# log_level: DEBUG
# >>>>>>> branch-a
```

Step 4: Resolve the Conflict

```
# Edit config.yaml -- remove markers, keep what you want:
$ echo 'log_level: INFO' > config.yaml  # compromise!

# Mark as resolved
$ git add config.yaml

# Complete the merge
$ git commit -m "merge: resolve config conflict, set INFO level"

# Verify
$ git log --oneline --graph
```

Conflict Resolution Rules

1. NEVER commit files with conflict markers (<<<, ===, >>>) -- your code will break. 2. Always run 'git status' to check ALL conflicted files before committing. 3. Test your code after resolving. 4. If overwhelmed, abort the merge: 'git merge --abort'

Lab 7: Working with Remotes (GitHub/GitLab)

Step 1: Set Up SSH Keys (Do This Once)

SSH keys let you push/pull without typing your password every time. This is the professional way to authenticate.

```
# 1. Generate an SSH key pair
$ ssh-keygen -t ed25519 -C "your.email@example.com"
# Press Enter for default location (~/.ssh/id_ed25519)
# Enter a passphrase (optional but recommended)

# 2. Start the SSH agent
$ eval "$(ssh-agent -s)"
# Agent pid 12345

# 3. Add your key to the agent
$ ssh-add ~/.ssh/id_ed25519

# 4. Copy your PUBLIC key (this goes to GitHub/GitLab)
$ cat ~/.ssh/id_ed25519.pub
# ssh-ed25519 AAAAC3NzaC1... your.email@example.com

# 5. Add to GitHub:
#    Go to GitHub.com -> Settings -> SSH and GPG Keys -> New SSH key
#    Paste the public key and save

# 6. Test the connection
$ ssh -T git@github.com
# Hi Siddharth! You've successfully authenticated...
```

SSH vs HTTPS

HTTPS URLs look like: https://github.com/user/repo.git (requires password/token every push). SSH URLs look like: git@github.com:user/repo.git (uses your SSH key, no password needed). For daily work, SSH is far more convenient. You can switch an existing repo with: git remote set-url origin git@github.com:user/repo.git

Step 2: Clone a Repository

```
# Clone creates a local copy of a remote repo
$ git clone git@github.com:YOUR_USER/YOUR_REPO.git
$ cd YOUR_REPO

# See the remote connection
$ git remote -v
# origin  git@github.com:YOUR_USER/YOUR_REPO.git (fetch)
# origin  git@github.com:YOUR_USER/YOUR_REPO.git (push)

# Clone with a custom folder name
$ git clone git@github.com:YOUR_USER/YOUR_REPO.git my-local-name
```

```
# Clone only a specific branch (faster for large repos)
$ git clone --branch develop --single-branch git@github.com:user/repo.git
```

Step 3: Add a Remote to Existing Local Repo

```
# If you already have a local repo and want to connect it
$ cd my-sre-project
$ git remote add origin git@github.com:YOUR_USER/my-sre-project.git

# Verify
$ git remote -v

# See detailed info about a remote
$ git remote show origin

# Rename a remote
$ git remote rename origin upstream

# Remove a remote
$ git remote remove upstream

# Change remote URL (e.g., HTTPS to SSH)
$ git remote set-url origin git@github.com:YOUR_USER/my-sre-project.git
```

Step 4: Pushing for the First Time

When you push a branch for the first time, Git doesn't know which remote branch to track. You MUST set the upstream.

```
# Push main to remote for the first time
$ git push -u origin main
# -u (or --set-upstream) links local 'main' to 'origin/main'
# After this, just 'git push' works (no need to specify origin main)

# What happens if you forget -u?
$ git push
# ERROR: fatal: The current branch main has no upstream branch.
# To push the current branch and set the remote as upstream, use:
#     git push --set-upstream origin main

# Verify the upstream tracking
$ git branch -vv
# * main abc1234 [origin/main] Initial commit
#   ^              ^-- tracking remote branch
```

Step 5: Pushing Feature Branches

```
# Create and work on a feature branch
$ git switch -c feature/add-monitoring
$ echo 'import prometheus_client' > monitoring.py
$ git add . && git commit -m "feat: add Prometheus monitoring"

# Push the feature branch to remote (first time)
$ git push -u origin feature/add-monitoring
# Creates 'feature/add-monitoring' on GitHub/GitLab
# Output includes a link to create a Pull Request!

# Now continue working and pushing is simple:
$ echo 'from prometheus_client import Counter' >> monitoring.py
$ git add . && git commit -m "feat: add Counter metric"
$ git push  # upstream already set, just 'git push' works

# Shortcut: push and set upstream in one step
$ git push -u origin HEAD
# HEAD = current branch name (saves typing long branch names)
```

Step 6: Viewing Remote Branches

```
# List only local branches
$ git branch
# * feature/add-monitoring
#   main

# List only remote branches
$ git branch -r
# origin/main
# origin/feature/add-monitoring
# origin/feature/dashboard
# origin/bugfix/login-error

# List ALL branches (local + remote)
$ git branch -a

# See which local branches track which remote branches
$ git branch -vv
# * feature/add-monitoring abc123 [origin/feature/add-monitoring] feat: add
monitoring
#   main                  def456 [origin/main] Initial commit

# Checkout a remote branch that someone else created
$ git fetch origin
$ git switch feature/dashboard
# Git auto-creates local branch tracking origin/feature/dashboard
```

Step 7: Fetch vs Pull

```
# fetch: download changes but DON'T apply them
$ git fetch origin

# See what's new on remote that you don't have locally
$ git log --oneline main..origin/main
# abc1234 feat: someone else's commit
# def5678 fix: another team member's fix

# See differences between your branch and remote
$ git diff main origin/main

# NOW merge if you want (fetch + manual merge = safe)
$ git merge origin/main

# --- OR ---

# pull: fetch + merge in one step (convenience)
$ git pull origin main

# pull with rebase (cleaner history, preferred by many teams)
$ git pull --rebase origin main

# Make rebase the default for pull (recommended):
$ git config --global pull.rebase true
```

When to Use fetch vs pull

Use 'git fetch' when you want to SEE what changed before applying (safer, more control). Use 'git pull' when you trust the changes and want to apply immediately. In SRE work, 'git fetch' first is safer -- you can inspect before merging, especially on shared infrastructure repos.

Step 8: Deleting Remote Branches

```
# After your PR is merged, clean up the remote branch
$ git push origin --delete feature/add-monitoring
# Deletes 'feature/add-monitoring' on GitHub/GitLab

# Also delete your local branch
$ git switch main
$ git branch -d feature/add-monitoring
# -d = delete (only if fully merged)
# -D = force delete (even if not merged -- be careful!)

# Clean up stale remote tracking references
# (branches deleted on remote but still showing locally)
$ git fetch --prune
# or
$ git remote prune origin

# Make pruning automatic on every fetch:
```

```
$ git config --global fetch.prune true
```

Step 9: Force Push (After Rebase)

After rebasing or amending a pushed commit, you need to force push because you rewrote history.

```
# Scenario: You rebased your feature branch
$ git switch feature/my-task
$ git rebase main

# Normal push fails because history was rewritten
$ git push
# ERROR: rejected -- non-fast-forward
# hint: Updates were rejected because the tip of your current
# hint: branch is behind its remote counterpart.

# BAD: Don't use --force (overwrites everything blindly)
# $ git push --force   # DANGEROUS!

# GOOD: Use --force-with-lease (safe force push)
$ git push --force-with-lease
# This checks that no one else pushed to this branch since
# your last fetch. If they did, it REFUSES to push.

# Even safer: force-with-lease with expected value
$ git push --force-with-lease=feature/my-task
```

Force Push Safety Rules

1. NEVER force push to main, develop, or any shared branch. 2. ONLY force push to YOUR OWN feature branches after rebase/amend. 3. ALWAYS use --force-with-lease instead of --force (it checks if someone else pushed first). 4. If --force-with-lease rejects, do 'git fetch' first and re-inspect.

Step 10: The Complete Daily Workflow

```
# === START OF YOUR DAY ===

# 1. Get latest from main
$ git switch main
$ git pull origin main

# 2. Create your feature branch
$ git switch -c feature/JIRA-1234-add-health-check

# 3. Do work, commit often with good messages
$ # ... edit files ...
$ git add .
$ git commit -m "feat: add /healthz endpoint"
$ # ... more edits ...
$ git add .
```

```
$ git commit -m "test: add health check integration tests"

# 4. Before pushing, rebase onto latest main
$ git fetch origin
$ git rebase origin/main
# Resolve any conflicts if needed

# 5. Push your branch
$ git push -u origin HEAD
# (or 'git push --force-with-lease' if you rebased after a prior push)

# 6. Open Pull Request on GitHub/GitLab (web UI)
#    - Add reviewers
#    - Link to JIRA ticket
#    - Describe what changed and why

# === AFTER PR IS APPROVED & MERGED ===

# 7. Clean up
$ git switch main
$ git pull origin main
$ git branch -d feature/JIRA-1234-add-health-check
$ git push origin --delete feature/JIRA-1234-add-health-check
$ git fetch --prune
```

SRE GitOps Context

In SRE teams using GitOps (ArgoCD, Flux), pushing to a Git branch can trigger automatic deployments. 'git push' to main might deploy to production. This is why branch protection, PR reviews, and CI/CD checks are critical. Never push directly to main in a team setting. Force pushing to main in a GitOps setup can cause rollbacks and outages.

Step 11: Working with Multiple Remotes (Forks)

When contributing to open-source or when your team uses forks, you'll have multiple remotes:

```
# Typical fork setup: 'origin' = your fork, 'upstream' = original repo
$ git remote add upstream git@github.com:ORIGINAL_ORG/repo.git

$ git remote -v
# origin    git@github.com:YOUR_USER/repo.git (fetch)
# origin    git@github.com:YOUR_USER/repo.git (push)
# upstream  git@github.com:ORIGINAL_ORG/repo.git (fetch)
# upstream  git@github.com:ORIGINAL_ORG/repo.git (push)

# Keep your fork up to date with the original repo
$ git fetch upstream
$ git switch main
$ git merge upstream/main
$ git push origin main
```

```
# Create feature branch from the latest upstream
$ git fetch upstream
$ git switch -c feature/my-contribution upstream/main
# ... do work ...
$ git push -u origin feature/my-contribution
# Then open a PR from YOUR fork to the ORIGINAL repo
```

Lab 8: Stashing -- Save Work Without Committing

```
# Scenario: You're mid-work and need to switch branches urgently
$ echo 'work in progress...' >> app.py
$ git status  # modified app.py

# Can't switch -- Git warns about uncommitted changes
# Solution: STASH your changes
$ git stash
# Saved working directory and index state WIP on main: abc1234...

$ git status   # Clean! Changes are safely stashed
$ cat app.py   # Your WIP line is gone (safely stored)

# Now you can switch branches freely
$ git switch feature/urgent-fix
$ # ... do urgent work ... commit ...
$ git switch main

# Bring your stashed changes back
$ git stash pop
$ cat app.py   # Your WIP line is back!

# === STASH MANAGEMENT ===
$ git stash list         # see all stashes
$ git stash show         # see what's in top stash
$ git stash pop          # apply & remove top stash
$ git stash apply        # apply but KEEP in stash list
$ git stash drop         # discard top stash
$ git stash clear        # delete ALL stashes

# Named stash (easier to find later):
$ git stash push -m "WIP: alerting feature half done"
```

Lab 9: Rebase -- Clean Linear History

Step 1: What Rebase Does

```
# MERGE creates:     REBASE creates:
#     * merge          * C' (feature)
#    / \               * B' (feature)
#   *   * (feature)    * A  (main)
#   *   *
#    \ /               # Linear! Clean! Easy to read!
#     * (main)
```

Step 2: Hands-On Rebase

```
# Setup
$ git switch main
$ echo 'main change 1' >> README.md
$ git add . && git commit -m "main: update 1"

$ git switch -c feature/rebase-demo
$ echo 'feature work' > feature.txt
$ git add . && git commit -m "feature: add feature.txt"

# Meanwhile, main gets more commits
$ git switch main
$ echo 'main change 2' >> README.md
$ git add . && git commit -m "main: update 2"

# Now rebase feature onto latest main
$ git switch feature/rebase-demo
$ git rebase main
# Your feature commit is now ON TOP of main's latest

$ git log --oneline --graph --all
# Beautiful linear history!
```

Golden Rule of Rebase

NEVER rebase commits that have been pushed and shared with others. Rebase rewrites commit hashes. If others based work on the old hashes, you'll create a mess. Safe: rebase YOUR local feature branch onto main before pushing. Dangerous: rebase main or any shared branch.

Lab 10: .gitignore -- Keep Repos Clean

```
# Create a .gitignore file
$ cat > .gitignore << 'EOF'
# Python
__pycache__/
*.pyc
.env
venv/

# IDE
.vscode/
.idea/
*.swp

# OS
.DS_Store
Thumbs.db

# Secrets (CRITICAL for SRE!)
*.pem
*.key
secrets.yaml
.env.local

# Build artifacts
dist/
build/
*.tar.gz

# Logs
*.log
logs/
EOF

$ git add .gitignore
$ git commit -m "Add .gitignore"

# Test it -- create files that should be ignored
$ echo 'SECRET=abc123' > .env
$ mkdir __pycache__ && touch __pycache__/test.pyc
$ git status  # These files do NOT show up!
```

Already-Tracked Files

.gitignore only works on UNTRACKED files. If you already committed a file and then add it to .gitignore, Git keeps tracking it. Fix: 'git rm --cached filename' (removes from tracking, keeps the file on disk).

```
# Remove a file from tracking (but keep on disk)
$ echo 'oops' > secrets.txt
$ git add secrets.txt && git commit -m "accidentally tracked"
```

```
# Add to .gitignore first, then:
$ git rm --cached secrets.txt
$ git commit -m "Stop tracking secrets.txt"
$ ls secrets.txt  # File still exists locally!
```

Lab 11: Essential SRE Git Commands

git blame -- Who Changed This Line?

```
# See who last modified each line of a file
$ git blame app.py
# abc1234 (Siddharth 2025-02-08) import os
# def5678 (Siddharth 2025-02-08) print(os.getcwd())

# Blame a specific line range
$ git blame -L 1,5 app.py
```

git cherry-pick -- Grab a Specific Commit

```
# Apply one specific commit from another branch
$ git log --oneline feature/add-alerting
# abc1234 feat: add alerting module skeleton

$ git switch main
$ git cherry-pick abc1234
# That one commit is now on main too
```

git bisect -- Find Which Commit Broke Things

```
# Binary search through commits to find a bug
$ git bisect start
$ git bisect bad          # current commit is broken
$ git bisect good abc123  # this old commit was working

# Git checks out the midpoint. Test it, then:
$ git bisect good  # if this commit works
# or
$ git bisect bad   # if this commit is broken

# Repeat until Git finds the exact commit that broke things
# When done:
$ git bisect reset
```

git tag -- Mark Release Versions

```
# Lightweight tag
$ git tag v1.0.0

# Annotated tag (preferred -- includes metadata)
$ git tag -a v1.0.0 -m "Release version 1.0.0"

# List tags
```

```
$ git tag

# Push tags to remote
$ git push origin v1.0.0
$ git push origin --tags  # push ALL tags
```

SRE Context

Tags are used to mark release versions. In GitOps workflows, ArgoCD can be configured to deploy specific tags. 'git bisect' is invaluable during incident response when you need to find which deployment introduced a regression.

Lab 12: Interactive Rebase -- Cleaning Commits Before PR

```
# Create messy commits (realistic scenario)
$ git switch -c feature/messy-work
$ echo 'v1' > feature.py && git add . && git commit -m "WIP"
$ echo 'v2' > feature.py && git add . && git commit -m "more WIP"
$ echo 'v3' > feature.py && git add . && git commit -m "fix typo"
$ echo 'v4' > feature.py && git add . && git commit -m "actually works now"

# Squash these 4 commits into 1 clean commit
$ git rebase -i HEAD~4

# Editor opens with:
# pick abc1234 WIP
# pick def5678 more WIP
# pick ghi9012 fix typo
# pick jkl3456 actually works now

# Change to:
# pick abc1234 WIP
# squash def5678 more WIP
# squash ghi9012 fix typo
# squash jkl3456 actually works now

# Save & close. Editor opens for combined message.
# Write: 'feat: add feature module (clean implementation)'
# Save & close.

$ git log --oneline
# One clean commit instead of 4 messy ones!

# Now push (force-with-lease because history was rewritten)
$ git push --force-with-lease
```

Quick Reference Cheat Sheet

Setup

```
git config --global user.name "Name"    # Set identity
git config --global user.email "email"   # Set email
git init                                  # New repo
git clone <url>                           # Copy remote repo
ssh-keygen -t ed25519 -C "email"          # Generate SSH key
```

Daily Commands

```
git status                    # What's changed?
git add <file>                # Stage file
git add .                     # Stage everything
git commit -m "message"       # Commit staged changes
git pull                      # Get latest from remote
git push                      # Send commits to remote
```

Branching

```
git branch                    # List local branches
git branch -r                 # List remote branches
git branch -a                 # List all branches
git branch -vv                # Show tracking info
git switch -c <name>          # Create & switch
git switch <name>             # Switch branch
git merge <branch>            # Merge into current
git branch -d <name>          # Delete merged branch
git branch -D <name>          # Force delete branch
```

Remotes & Pushing

```
git remote -v                           # List remotes
git remote add <name> <url>             # Add remote
git remote set-url origin <url>         # Change remote URL
git push -u origin <branch>             # First push (set upstream)
git push                                # Push (after upstream set)
git push -u origin HEAD                 # Push current branch
git push origin --delete <branch>       # Delete remote branch
git push --force-with-lease             # Safe force push
git fetch origin                        # Download without merge
git fetch --prune                       # Clean stale remotes
git pull --rebase                       # Pull with rebase
```

Inspection

```
git log --oneline             # Compact history
```

```
git log --oneline --graph     # Visual branch graph
git diff                      # Unstaged changes
git diff --staged             # Staged changes
git blame <file>              # Who changed what
```

Undo

```
git restore <file>            # Discard changes
git restore --staged <file>   # Unstage
git commit --amend            # Fix last commit
git revert <commit>           # Safe undo (new commit)
git reset --hard HEAD~1       # Nuclear undo (local only!)
git stash / git stash pop     # Temporarily shelve work
```

What's Next?

You've now covered the core 90% of Git that you'll use daily as an SRE. To deepen your skills:

1. Practice the PR workflow: Create a GitHub repo, make branches, open Pull Requests, merge them.

2. Learn Git hooks: Pre-commit hooks can run linters and tests automatically before every commit.

3. Explore GitOps: Combine Git with ArgoCD for automated Kubernetes deployments.

4. Learn git worktree: Work on multiple branches simultaneously without stashing.

5. Read the Pro Git book: Free at git-scm.com/book -- the definitive reference.

Daily Drill (5 minutes/day)

Every day this week, do this: create a repo, make 3 branches, commit to each, merge them into main, resolve at least one conflict. Push all branches to a remote repo. Delete the merged branches both locally and on the remote. By Friday, branching, merging, and pushing will be second nature.