Advanced Git & Best Practices

git cherry-pick, git reflog (especially as a recovery mechanism for lost commits), git blame, aliases, good branch/commit conventions, and a comparison tying together merge/rebase/squash/revert. Git best practices for a development team.

Intermediate 30 min

Last updated: August 27, 2026
Read online: https://www.learnforgex.com/en/topics/advanced-git-and-best-practices

---

This lesson rounds out the course with a handful of genuinely useful tools you'll reach for less often than `add` / `commit` / `push`, but which matter a great deal on the day you actually need them — especially `git reflog`, which can save you when everything else in this course would tell you the work is gone. It closes with a set of team-level practices that tie every earlier lesson together.

git cherry-pick

`git cherry-pick` copies **one specific commit** from anywhere in the repository onto your current branch — useful when exactly one fix from another branch is relevant to yours, without merging or rebasing the whole branch:

```
git cherry-pick a1b2c3d
```

A common real scenario: a critical bug fix lands on `main`, and you need that exact fix on a `release` branch too, without pulling in everything else that's happened on `main` since the release branched off.

git reflog

Every time HEAD moves — a commit, a checkout, a reset, an amend, a rebase step — Git silently records it in the **reflog**, a local, personal log of everywhere your HEAD has pointed:

```
git reflog
```

```
a9f3e21 HEAD@{0}: commit: Add password validation
8a1b2c3 HEAD@{1}: reset: moving to HEAD~1
3c9d1a2 HEAD@{2}: commit: Add password validation (first attempt)
```

The reflog is **local only** — it's never pushed, never shared, and it's specific to your own clone. But it means almost nothing you do in Git is ever truly, permanently gone the moment it happens; it just becomes hard to find through normal commands like `log`.

Recovering Lost Commits with reflog

This is the single most valuable emergency skill in this course. Say you ran `git reset --hard` and immediately realized it discarded a real commit:

```
#!/bin/sh
# Recovering a commit that git reset --hard accidentally discarded.

git log --oneline -1
# Output:
# a9f3e21 Add password validation

# Meant to undo the LAST commit before this one -- typed one too many resets.
git reset --hard HEAD~2
# Output:
# HEAD is now at 3c9d1a2 Add UserService with findById stub

# "Add password validation" (a9f3e21) is nowhere in `git log` anymore.
git log --oneline -3
# Output:
# 3c9d1a2 Add UserService with findById stub
# ... (a9f3e21 is gone from this view)

# But it's not actually gone -- the reflog remembers every place HEAD has been.
git reflog
# Output:
# 3c9d1a2 HEAD@{0}: reset: moving to HEAD~2
# a9f3e21 HEAD@{1}: commit: Add password validation
# 8a1b2c3 HEAD@{2}: commit: Add UserService with findById method

# HEAD@{1} is exactly the commit we wanted back. Reset onto it directly.
git reset --hard a9f3e21

git log --oneline -1
# Output:
# a9f3e21 Add password validation
```

Emphasize this: `git reflog` is your recovery mechanism for accidentally lost commits — a bad `reset --hard`, an amend you regret, even a branch you deleted too soon. As long as the commit was ever actually committed on your machine, its hash still exists in Git's object database until garbage collection eventually cleans it up (which normally takes weeks), and the reflog is how you find that hash again.

git blame

`git blame` shows who last changed each line of a file, and in which commit:

```
git blame PasswordValidator.java
```

```
a1b2c3d4 (Ada Lovelace  2026-08-10 14:22:03 +0300  12) if (password.length() < 8) {
e5f6a7b8 (Grace Hopper  2026-08-15 09:10:41 +0300  13)     throw new
WeakPasswordException();
```

This is invaluable when you're trying to understand *why* a specific line exists — pair it with `git show <hash>` on the commit it names to see the full context and commit message behind that change.

Conventional Commits (Recap)

You already met Conventional Commits in "Working With Commits" — prefixing messages with a type like `feat:` / `fix:` / `docs:` / `refactor:`. Worth repeating here specifically as a **team practice**: when everyone on a team follows it consistently, `git log` becomes scannable at a glance, and tooling can auto-generate changelogs or determine version bumps from the commit history alone.

Useful Git Aliases

Git lets you define shortcuts for commands you type constantly:

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.lg "log --oneline --graph --all"
```

After this, `git st` runs `git status`, and `git lg` gives you a compact, visual branch graph. Aliases are entirely personal — they live in your own Git config, not the project, so there's no risk of imposing them on teammates.

Good Branch Naming

A consistent branch naming convention makes `git branch -a` scannable and often integrates with tooling (some CI systems, or issue trackers, parse branch names):

```
feature/add-password-reset
fix/login-null-pointer
chore/upgrade-spring-boot
```

The exact prefixes vary by team, but the pattern — a type, then a short, hyphenated description — is close to universal.

Good Commit Practices (Recap)

From "Working With Commits": small, focused commits; imperative-mood summary lines; explaining *why*, not *what*. Worth restating here as a team-level habit, not just an individual one — a team that consistently writes good commit messages produces a codebase that documents its own history, which pays off enormously the first time someone needs to understand a years-old decision.

Merge vs Rebase vs Squash vs Revert

A single table tying together every history-changing operation from this course:

- `git merge` — combines branches, preserving all commits. Rewrites shared history? No — safe on shared branches.
- `git rebase` — replays commits onto a new base. Rewrites shared history? Yes — only on unshared, local commits.
- **Squash** (via `rebase -i` or GitHub) — combines multiple commits into one. Rewrites shared history? Yes — same rule as rebase.
- `git revert` — undoes a commit by adding a new, opposite commit. Rewrites shared history? No — safe on shared/pushed commits.

Git Best Practices for a Development Team

Bringing every lesson in this course together, the habits that separate smooth team collaboration from constant Git friction:

- Small, focused commits with messages explaining *why*.
- Feature branches for everything — never commit directly to `main`.
- Pull Requests with real review before merging, enforced by branch protection rules.
- Rebase your own unshared branches to keep history clean; never rebase shared ones.
- `--force-with-lease`, never plain `--force`, on the rare occasion a force push is genuinely needed.
- `.gitignore` configured from the start of a project (build output, IDE files, `.env`/secrets — never committed).

Common Mistakes

- **Reaching for `cherry-pick` when a proper merge or rebase would be clearer** — cherry-pick is for genuinely isolated, one-off fixes, not a general substitute for merging.
- **Not knowing `reflog` exists**, and assuming a bad `reset --hard` or accidental branch deletion is unrecoverable — it usually isn't, within the reflog's retention window.

- **Personal aliases leaking into project documentation** — remember aliases are local to your machine; don't write setup docs assuming a teammate has your `git lg`.

Best Practices

- Learn `git reflog` before you need it — the time you actually need it is not the time to be reading documentation for the first time.
- Keep commit messages and branch names consistent across the team — small conventions compound into a genuinely easier-to-navigate history.
- Default to the safest operation that solves the problem — `revert` over `reset` on shared commits, `--force-with-lease` over `--force`, merge over rebase on shared branches.

Summary, Cheat Sheet, and Glossary

Summary

- `git cherry-pick` copies one specific commit onto your current branch.
- `git reflog` is a local, personal record of everywhere HEAD has pointed — your recovery mechanism when a reset, amend, or rebase goes wrong.
- `git blame` shows who last touched each line and in which commit, useful for understanding *why* code looks the way it does.
- Team-level Git practices — consistent commit style, feature branches, reviewed PRs, careful rebase/force-push discipline — are what actually make Git collaboration smooth in practice.

Cheat Sheet

```
git cherry-pick <hash>          # copy one commit onto the current branch
git reflog                      # see everywhere HEAD has pointed
git reset --hard <hash-from-reflog>   # recover to a commit found via reflog
git blame <file>                # see who last changed each line
git config --global alias.<name> "<command>"   # define a personal shortcut
```

Glossary

- **Reflog** — a local, personal log of every place HEAD has pointed; the primary recovery mechanism for "lost" commits.
- **Cherry-pick** — copying a single commit from elsewhere onto the current branch.
- **Blame** — showing which commit (and author) last changed each line of a file.