Intermediate Git for Biostatistics

A Three-Day Boot Camp

The rgtlab Curriculum Project

2026-11-13

Intermediate Git for Biostatistics

A Three-Day Boot Camp

The rgtlab Curriculum Project

Welcome

This is the online version of **Intermediate Git for Biostatistics: A Three-Day Boot Camp** by The rgtlab Curriculum Project. It is a short second course in version control. We have situated it between the boot camp and the graduate sequence proper.

The companion volume *Git and GitHub for Biostatistics: A One-Week Boot Camp* takes a student from never having made a commit to collaborating through pull requests. This bridge picks up there. It teaches the intermediate skills a student reaches for within days of finishing the boot camp: committing a clean subset of a messy working directory, shelving work in progress, curating a branch's history into something a reviewer can read, and recovering when an operation goes wrong. These are the skills that separate a tidy, reproducible project history from one that is merely serviceable.

The bridge is deliberately short, **three days** rather than five, at the same daily cadence: one hour of lecture, two hours of homework, no examinations. It is not intended as a survey of everything Git can do. We leave the infrastructure that surrounds version control in professional practice, namely continuous integration, containerization, dependency pinning, and data-versioning tooling, to the *Biostatistics Practicum*, which this bridge feeds directly.

The organizing principle is unchanged from the boot camp: version control is a research-integrity tool. Where the boot camp teaches a student to record history, this bridge teaches a student to curate and to recover it. The aim, in both cases, is a commit log that remains a faithful and legible account of how an analysis was built.

The scope of each day:

1. **Selective staging and work in progress.** Committing part of a file with patch-mode staging, discarding selective changes, and shelving unfinished work with the stash.
2. **Rewriting history.** Amending, interactive rebase, cherry-picking, and the discipline that governs when rewriting is safe and when it is forbidden.
3. **Recovery and forensics.** The reflog as a safety net, `git bisect` to locate the commit that changed a result, and `git blame` and the pickaxe to investigate history.

Position in the sequence

This bridge is a preparatory companion, alongside the two boot camps, to the graduate sequence:

- *Git and GitHub for Biostatistics*, the prerequisite boot camp.
- *R for Biostatistics*, the parallel R boot camp.
- *Biostatistics Practicum*, workflow infrastructure, which this bridge feeds.

License

This book is licensed under Creative Commons Attribution 4.0 International.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

Intermediate Git for Biostatistics

A Three-Day Boot Camp.

Draft, November 13, 2026. Content may change.

Copyright

Intermediate Git for Biostatistics: A Three-Day Boot Camp by rgtlab curriculum team is licensed under a Creative Commons Attribution 4.0 International License.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

To cite this book, please use:

> rgtlab curriculum team (2026). *Intermediate Git for Biostatistics: A Three-Day Boot Camp*. Available at https://git-intermediate.rgtlab.org.

Table of contents

Position in the sequence 2
License 2

Copyright **5**

Preface **11**
Why a bridge, and why only three days 11
What this bridge covers 12
What this bridge does not cover 12
How this bridge is meant to be used 13
Prerequisites 13
Acknowledgments 13

Conventions **15**
Commands and output 15
Callouts 16
Diagrams of history 16
Cross-references 17
Chapter structure 17

How to use this book **19**
The daily cadence 19
A note on danger 19
A running example 20
Conventions 20
Companion volumes 21

I. Three Days **23**

1. Day 1: Selective Staging and Work in Progress **25**
1.1. Learning objectives 25
1.2. Lecture 25
1.2.1. The problem: a mixed working directory 25
1.2.2. Hunks and patch mode 26
1.2.3. Splitting and editing hunks 28
1.2.4. Discarding and unstaging selectively 29
1.2.5. Shelving work in progress with git stash 30
1.2.6. Stash or commit? A rule of thumb 32
1.3. Worked example: two changes, one interruption 32
1.4. Homework 34
1.5. Solutions 35
1.6. What's next 38

2. Day 2: Rewriting History **39**
2.1. Learning objectives 39
2.2. Lecture 40
2.2.1. Rewriting means replacing 40
2.2.2. Amending the last commit 40
2.2.3. Referring to commits 42
2.2.4. Interactive rebase 42
2.2.5. The autosquash workflow 46
2.2.6. Cherry-picking a single commit 47
2.2.7. The governing rule: do not rewrite published history 47
2.2.8. Updating a pushed branch with force-with-lease . . 49
2.3. Worked example: cleaning a feature branch before review . 50
2.4. Homework 52
2.5. Solutions 54
2.6. What's next 57

3. Day 3: Recovery and Forensics **59**
3.1. Learning objectives 59
3.2. Lecture 59
3.2.1. The safety net beneath Day 2 59
3.2.2. What the reflog records 60
3.2.3. Reading the reflog 60
3.2.4. Recovery 1: an accidental reset –hard 61

Draft, November 13, 2026. Content may change.

3.2.5. Recovery 2: a commit orphaned by interactive rebase 63
3.2.6. Recovery 3: a deleted branch 64
3.2.7. From recovery to forensics 65
3.2.8. Finding the culprit commit with git bisect 65
3.2.9. Reading a line's history with git blame 68
3.2.10. The pickaxe and line-log 69
3.2.11. Reading the commit with git show 70
3.3. Worked example: a number that moved 71
3.4. Homework 74
3.5. Solutions 75
3.6. What's next 78
3.6.1. Where this bridge hands off 79

References **81**

Appendices **83**

Credits **83**

Colophon **85**

Preface

This volume is a three-day bridge between *Git and GitHub for Biostatistics: A One-Week Boot Camp* and the graduate sequence that follows it. The boot camp is a prerequisite; this bridge assumes the reader is fluent in the everyday workflow it taught.

Why a bridge, and why only three days

The boot camp is deliberately narrow. It teaches the irreducible daily workflow: `init`, `add`, `commit`, `status`, `log`, `diff`, the undo commands, branching and merging, remotes, and the pull-request cycle on GitHub. A student who completes it can work reproducibly and collaborate. Within days of finishing, however, that student meets a predictable set of problems that the boot camp did not cover. We list them here because each recurs often enough to warrant a chapter of its own.

- The working directory contains two unrelated changes, and the student wants each in its own commit rather than one muddled commit.
- A colleague asks for a change, but the student is halfway through something else and cannot cleanly commit yet.
- A feature branch has accumulated six commits with messages like 'wip', 'fix', and 'oops', and a reviewer is about to read it.
- A `git reset --hard` discarded the wrong work, and the student needs it back.
- A number in a table changed between two analyses and nobody can say which commit changed it.

Each of these problems has a standard, learnable solution, and none belongs in a first week. Nor do they belong in the *Practicum*, whose subject is the infrastructure around version control (Docker, `renv`, continuous integration,

CDISC deliverables) rather than the version-control craft itself. The three-day bridge fills exactly this gap, and no more.

What this bridge covers

Three days, one chapter each:

1. **Selective staging and work in progress.** Patch-mode staging with `git add -p` and `git restore -p`; composing a clean commit from a mixed working directory; splitting an over-large change; shelving and restoring unfinished work with `git stash`.
2. **Rewriting history.** `git commit --amend` in depth; interactive rebase (`git rebase -i`) to reword, reorder, squash, fixup, and drop commits; `git cherry-pick`; `git push --force-with-lease`; and the governing rule, namely that published history is rewritten only under strict conditions.
3. **Recovery and forensics.** `git reflog` as the safety net beneath every history-altering operation; `git bisect` to locate the commit that introduced a change in a result; `git blame`, the pickaxe (`git log -S` and `-G`), and `git log -L` to investigate how a line or a value came to be.

Each chapter is approximately 1 hour of reading and worked examples plus 2 hours of homework problems with worked solutions provided.

What this bridge does not cover

The bridge stops short of the infrastructure that the *Practicum* owns:

- Continuous integration and GitHub Actions.
- Pre-commit hooks and automated checks.
- Containerized, dependency-pinned reproducibility (Docker, `renv` in depth).
- Data and large-artifact versioning (Git LFS, DVC).
- Regulatory version-control practice for clinical trials (CDISC deliverables, validated pipelines).

Draft, November 13, 2026. Content may change.

- Team-scale collaboration policy (branch protection, `CODEOWNERS`, review requirements, `rerere`).

We name each of these where it naturally arises, with a pointer to the *Practicum*.

How this bridge is meant to be used

The three-day cadence matches the boot camp: roughly three hours of focused work per day, one hour reading with a terminal open and two hours on the homework. Every command should be run, not merely read. This is especially true of the history-altering operations, which are learned only by performing them on a scratch repository and observing the result.

The commands are shown for a Unix-style shell (macOS Terminal, Linux, or Git Bash on Windows), consistent with the boot camp.

Prerequisites

The bridge assumes:

- Completion of *Git and GitHub for Biostatistics*, or equivalent fluency: comfort with `add`, `commit`, `status`, `log`, `diff`, branching, merging, remotes, and pull requests.
- Git 2.40 or later and, for the GitHub-facing material, the `gh` command-line tool authenticated as in the boot camp.

Acknowledgments

This bridge consolidates the intermediate patterns that recur in reproducible statistical practice. Scott Chacon and Ben Straub's *Pro Git* and Jenny Bryan's *Happy Git and GitHub for the useR* provide the technical scaffolding. The selection of topics reflects experience with the specific problems that incoming students meet in the weeks after their first commit.

Conventions

This page summarizes the visual conventions used throughout the book. They are identical to those of the companion boot camp, repeated here for convenience.

Commands and output

This book is command-line first. Shell commands appear in syntax-highlighted blocks. A command the reader types is shown after a `$` prompt; the prompt itself is not typed. Output appears on the following lines, without a prompt:

```
$ git --version
git version 2.45.2
```

When a sequence of commands is shown without output, the prompts are omitted for readability:

```
git add -p analysis.R
git commit -m 'Fix na.rm in HbA1c mean'
```

Interactive tools, namely `git add -p`, `git rebase -i`, and `git bisect`, prompt for input during operation. Their prompts and the reader's single-letter responses are shown inline, with the response after the tool's `?`-style prompt.

Inline commands, file names, and Git objects are set in `monospace`. Git subcommands are named in full (`git rebase`, not merely `rebase`). Placeholders the reader must replace are written in angle brackets, for example `git cherry-pick <hash>`. The angle brackets are not typed.

Callouts

Three callout types appear:

> **Tip**
>
> A small practical recommendation.

> **Check your understanding: example**
>
> A short question testing comprehension of the just-read material. Click to expand the answer.

> **Warning**
>
> A pitfall the reader may otherwise hit. In this volume the warnings matter more than in the boot camp, because several operations rewrite or discard history irrecoverably.

Diagrams of history

Commit history is drawn as a left-to-right graph, with each node a commit labeled by a short hash and time flowing rightward. Branch names are pointers attached to commits:

```
        feature
           |
A -- B -- C -- D
     |
    main
```

These diagrams are conceptual; the real hashes are the forty-character identifiers Git assigns. When an operation changes history, the before and after graphs are shown side by side so that the effect is explicit.

Draft, November 13, 2026. Content may change.

Cross-references

Within this book, sections are referenced by their Quarto label (`@sec-day1`, `@sec-interactive-rebase`). These resolve to clickable links in HTML and section numbers in PDF.

References to the companion volumes *Git and GitHub for Biostatistics* and *Biostatistics Practicum* use **prose pointers** rather than Quarto cross-references, because cross-references do not resolve across separate books.

Chapter structure

Every content chapter follows the same five-section template inherited from the boot camp:

1. **Learning objectives.** What you will be able to do after the day.
2. **Lecture.** The substantive content, with commands to run in your own terminal.
3. **Worked example.** A realistic biostatistical scenario that uses the day's content end to end.
4. **Homework.** Problems organized from easier to harder.
5. **Solutions.** Worked solutions to all homework problems. Read only after attempting the problem.

Each chapter closes with a short **What's next** pointer.

How to use this book

We begin with a short orientation, briefer than the boot camp's because the reader arrives already knowing the cadence and conventions.

The daily cadence

Each day has the same shape as the boot camp:

- **Lecture content (~1 hour).** Read the chapter, running each command in your own terminal on a scratch repository as you encounter it. The history-altering operations of Days 2 and 3 are learned only by performing them.
- **Homework (~2 hours).** Problems with worked solutions at the end of the chapter. Attempt each before checking the solution.

Three consecutive days complete the bridge.

A note on danger

The boot camp's operations were almost all additive: a new commit, a new branch, a merge. They rarely destroyed anything. This bridge is different. Several of its operations rewrite history (Day 2), and some can discard uncommitted work (Day 1). One chapter exists precisely to recover from disaster (Day 3).

Two habits make this safe:

- **Work on a scratch repository while learning.** Do not practice interactive rebase for the first time on a project you care about. Create a throwaway repository, as the homework directs, and experiment there.
- **Learn the reflog early.** Day 3 covers `git reflog`, the log of everywhere `HEAD` has been. It is the safety net beneath nearly every operation in this book. Almost anything that has been committed can be recovered through it. The one exception, uncommitted work destroyed by `git reset --hard` or `git restore`, is flagged wherever it arises.

> **Tip**
>
> If an operation in this book frightens you, commit your work first. A clean working tree with everything committed is recoverable through the reflog no matter how badly a subsequent rewrite goes. The danger is almost always in uncommitted changes, not in committed history.

A running example

The worked examples follow a single project across the three days: `trial-analysis`, an R analysis repository for a hypothetical randomized trial of a glucose-lowering agent, with a primary endpoint of twelve-week change in HbA1c. The repository begins Day 1 in the state a boot-camp graduate would leave it: a working `analysis.R`, a clean history on `main`, and the everyday workflow already in hand. Each day develops it further.

Conventions

The visual cues are described on the Conventions page and are identical to the boot camp. In brief: commands you type follow a `$` prompt; output has no prompt; placeholders you replace are written in `<angle-brackets>`; interactive-tool prompts are shown inline.

Draft, November 13, 2026. Content may change.

Companion volumes

This bridge sits between the boot camp and the *Practicum*:

- *Git and GitHub for Biostatistics* (the prerequisite), https://git-bootcamp.rgtlab.org
- *R for Biostatistics* (the parallel R boot camp), https://r-bootcamp.rgtlab.org
- *Biostatistics Practicum* (what comes next), https://rgtlab.org/practicum

The end of Day 3 describes how the *Practicum* extends the skills of this bridge into a full reproducible pipeline.

Part I.

Three Days

Draft, November 13, 2026. Content may change.

1. Day 1: Selective Staging and Work in Progress

1.1. Learning objectives

By the end of this day you should be able to:

- Explain why a commit should contain one logical change, and why a mixed working directory obstructs that.
- Stage part of a file with patch-mode `git add -p`, responding to the hunk prompt and splitting or editing hunks as needed.
- Compose two or more clean commits from a single working directory that mixes unrelated changes.
- Discard selected changes with `git restore -p` and unstage selected changes with `git restore --staged -p`.
- Shelve unfinished work with `git stash`, inspect the stash list, and restore work with `git stash pop` or `git stash apply`.
- Judge when to stash and when to make a work-in-progress commit instead.

1.2. Lecture

1.2.1. The problem: a mixed working directory

The boot camp established a principle: a good commit records a single logical change. The history then reads as a sequence of intelligible steps, and any one of them can be understood, reverted, or cited in isolation. The boot camp also gave you the tool for the simple case, namely `git add <file>` to stage a whole file.

Draft, November 13, 2026. Content may change.

Real editing sessions rarely respect that principle. You sit down to fix a bug in `analysis.R` and, while you are there, you also improve an unrelated figure caption and add a line to the README. Your working directory now contains three unrelated changes. `git add analysis.R` would bundle the bug fix and the caption change into one commit, and a fourth `git add .` would sweep in the README too. The result is a commit whose message cannot honestly describe it, because it does more than one thing.

There are two ways out. Committing more often is good advice, but it does not help once the changes are already interleaved. The alternative is to stage selectively: tell Git precisely which changes belong in the next commit, down to the individual line, and leave the rest in the working directory for a later commit. That is the subject of the first half of today.

1.2.2. Hunks and patch mode

Git describes a change as a set of **hunks**. A hunk is a contiguous block of altered lines together with a few lines of unchanged context around it. When you run `git diff`, the blocks you see, each introduced by an `@@ ... @@` header, are hunks. Patch-mode staging works one hunk at a time.

Suppose `analysis.R` has two unrelated edits. Near the top you fixed a missing `na.rm`; near the bottom you added a new subgroup table. `git diff` shows two hunks:

```
$ git diff analysis.R
diff --git a/analysis.R b/analysis.R
index 3a1f8c2..b7e4d90 100644
--- a/analysis.R
+++ b/analysis.R
@@ -4,7 +4,7 @@ dat <- read.csv("data-raw/trial.csv")

-mean_hba1c <- mean(dat$hba1c)
+mean_hba1c <- mean(dat$hba1c, na.rm = TRUE)

 n_total <- nrow(dat)
@@ -18,3 +18,5 @@ summary_table <- dat |>
   summarise(mean_change = mean(change, na.rm = TRUE))
```


Draft, November 13, 2026. Content may change.

```
+
+by_sex <- dat |> group_by(sex) |>
+  summarise(mean_change = mean(change, na.rm = TRUE))
```

These are two logically distinct changes that happen to sit in one file. To commit them separately, stage them separately with `git add -p` (equivalently `git add --patch`):

```
$ git add -p analysis.R
```

Git presents the first hunk and prompts:

```
@@ -4,7 +4,7 @@ dat <- read.csv("data-raw/trial.csv")
-mean_hba1c <- mean(dat$hba1c)
+mean_hba1c <- mean(dat$hba1c, na.rm = TRUE)
 n_total <- nrow(dat)
(1/2) Stage this hunk [y,n,q,a,d,s,e,?]?
```

The letters are the commands. The essential ones:

- `y` stage this hunk.
- `n` do not stage this hunk.
- `q` quit; stage nothing further.
- `a` stage this hunk and all later hunks in the file.
- `d` do not stage this hunk or any later hunk in the file.
- `s` split the hunk into smaller hunks (only when the change has a gap Git can split on).
- `e` edit the hunk by hand (for the hard cases).
- `?` print help.

Here you want the `na.rm` fix but not the subgroup table, so answer `y` to the first hunk and `n` to the second:

```
(1/2) Stage this hunk [y,n,q,a,d,s,e,?]? y
(2/2) Stage this hunk [y,n,q,a,d,s,e,?]? n
```


Draft, November 13, 2026. Content may change.

Confirm what you have staged with `git diff --staged`, which shows only the `na.rm` fix, and `git status`, which now lists `analysis.R` as both staged and modified, exactly the split state from the boot camp's staging discussion:

```
$ git diff --staged
diff --git a/analysis.R b/analysis.R
@@ -4,7 +4,7 @@ dat <- read.csv("data-raw/trial.csv")
-mean_hba1c <- mean(dat$hba1c)
+mean_hba1c <- mean(dat$hba1c, na.rm = TRUE)
```

Commit the fix on its own, then stage and commit the remaining change:

```
$ git commit -m 'Fix missing na.rm in HbA1c mean'
[main 5c7e9a1] Fix missing na.rm in HbA1c mean
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git add analysis.R
$ git commit -m 'Add change-in-HbA1c table by sex'
[main 7d9f0b3] Add change-in-HbA1c table by sex
 1 file changed, 3 insertions(+)
```

One messy working directory has become two clean, single-purpose commits.

> **Tip**
>
> Reach for `git add -p` even when you think a file holds a single change. Reviewing each hunk before you stage it is the cheapest code review available: it is the last moment you see the change before it enters history, and it routinely catches a stray `print()`, a commented-out line, or a debugging edit you meant to remove.

1.2.3. Splitting and editing hunks

Two adjacent changes sometimes fall within a single hunk, with no unchanged line between them for Git to split on. Pressing `s` then does nothing

Draft, November 13, 2026. Content may change.

useful. The `e` command is the fallback: it opens the hunk in your editor, where each line is prefixed by `+` (an added line), `-` (a removed line), or a space (context). To stage only some lines, delete the lines you do not want from the patch, following the rule Git prints in the editor: to exclude an added (`+`) line, delete it; to keep a removed (`-`) line out of the commit, change its `-` to a space. Save and close, and Git stages exactly the edited patch.

Editing hunks by hand is fiddly and error-prone. Reserve it for hunks that genuinely cannot be split. In our experience, committing more often in the first place avoids the need most of the time.

Check your understanding: what does -p operate on?

You run `git add -p` with no filename in a repository where three files have changed. What happens?
Git walks the hunks of all three changed tracked files in turn, prompting for each. The `-p` flag is not tied to a single file; naming a file (`git add -p analysis.R`) simply restricts the walk to that file. When you have edits across several files and want to assemble one logical commit that spans some of them, running `git add -p` with no argument lets you pick the relevant hunks from each. Note that patch mode considers only tracked files; a brand-new, untracked file has no diff to present and must be added whole with `git add <file>`.

1.2.4. Discarding and unstaging selectively

Patch mode is not limited to staging. Two companions apply the same hunk-by-hunk selection to the reverse operations.

To **unstage** selected hunks, that is, to move them back out of the staging area while keeping the changes in your working directory, use patch mode on `git restore --staged`:

```
$ git restore --staged -p analysis.R
```

The prompt now asks, for each staged hunk, whether to unstage it (`y` unstages). Nothing in your working directory is lost; only the staging area

Draft, November 13, 2026. Content may change.

changes.

To **discard** selected hunks, that is, to throw away uncommitted changes in the working directory, use patch mode on `git restore`:

```
$ git restore -p analysis.R
```

Here `y` discards the hunk. This is destructive. A discarded uncommitted hunk is gone, with no reflog entry to recover it, because it was never committed. Use it to drop an experimental edit you have decided against, but read the warning below first.

> ⚠ Warning
>
> `git restore -p` and `git restore <file>` permanently discard uncommitted changes. Unlike almost everything in Day 2 and Day 3, this cannot be undone through the reflog, because uncommitted work is not in the object database at all. Before discarding anything you are unsure about, either commit it to a scratch branch or stash it (below); both are recoverable, whereas a discard is not.

1.2.5. Shelving work in progress with git stash

The second recurring problem of the early weeks is the interruption. You are halfway through an edit, the code does not yet run, and something urgent arrives. A colleague needs a quick fix on `main`, or you must switch branches to check a result. Your working directory is too unfinished to commit, but switching branches would carry the mess with you.

`git stash` solves this. It records your uncommitted changes (both staged and unstaged) onto a stack, then reverts the working directory to the last commit, leaving you clean:

```
$ git status --short
 M analysis.R
 M report.qmd
$ git stash
```


Draft, November 13, 2026. Content may change.

```
Saved working directory and index state WIP on main: 7d9f0b3 Add cha
$ git status
On branch main
nothing to commit, working tree clean
```

Your changes are not lost; they are on the stash stack. Do the urgent work, commit it, then bring your changes back with `git stash pop`, which reapplies the most recent stash and removes it from the stack:

```
$ git stash pop
On branch main
Changes not staged for commit:
    modified:   analysis.R
    modified:   report.qmd
Dropped refs/stash@{0} (a3c5e70...)
```

You are back exactly where you left off. Useful variants:

- `git stash list` shows the stack; each entry is `stash@{0}`, `stash@{1}`, and so on, newest first.
- `git stash push -m 'half-done sensitivity analysis'` records a stash with a description, so a stash you keep for more than a few minutes remains intelligible.
- `git stash push <path>` stashes only the named files.
- `git stash -u` also stashes untracked files (by default they stay in the working directory).
- `git stash apply` reapplies a stash but leaves it on the stack, in case you need it again; `git stash drop` then removes it explicitly.
- `git stash branch <name>` creates a new branch from the commit the stash was made on and applies the stash there, which is the clean way to resume a stash that no longer applies to a moved-on branch.

> **Warning**
>
> The stash is a stack, and it is easy to forget. A stash has no branch, no message unless you supplied one, and does not appear in `git status`. Work stashed ‘for a minute’ and left for a week is a common way to

lose track of changes. If an interruption will last longer than the walk to the coffee machine, prefer a work-in-progress commit on a branch, as described next; it is visible, named, and recoverable.

1.2.6. Stash or commit? A rule of thumb

Both stashing and a work-in-progress commit let you set work aside, and they differ mainly in visibility and durability. A stash is convenient and quick, but it is invisible to `git status` and detached from any branch. A work-in-progress commit, by contrast, is a first-class, named, recoverable object.

The working rule: **stash for a brief context switch you will reverse within minutes; commit for anything longer.** When you must switch branches to answer a colleague's question and will return immediately, stash. When you are stopping for the day, or the interruption is substantial, make a commit on your feature branch with a frank message:

```
$ git commit -am 'WIP: subgroup analysis, do not review'
```

A work-in-progress commit is not a blemish; it can be folded into a clean commit later with the interactive rebase you will learn on Day 2. The history you publish is what must be clean, not every intermediate state on your own branch.

1.3. Worked example: two changes, one interruption

The running `trial-analysis` repository is on `main` with a clean history. You begin a working session on `analysis.R` and make two unrelated edits. You correct a filtering threshold that was excluding eligible participants, and, separately, you start a new age-subgroup table that is not yet finished. Then a co-investigator emails to say that the enrollment count in yesterday's summary looks wrong, and asks you to check it now.

Inspect the state. Two hunks, one finished and one not:

Draft, November 13, 2026. Content may change.

```
$ git diff --stat
 analysis.R | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)
```

First, commit only the finished change, the threshold fix, using patch mode:

```
$ git add -p analysis.R
(1/2) Stage this hunk [y,n,q,a,d,s,e,?]? y
(2/2) Stage this hunk [y,n,q,a,d,s,e,?]? n
$ git commit -m 'Correct age-eligibility threshold to >= 18'
[main 9a1c3e5] Correct age-eligibility threshold to >= 18
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The unfinished subgroup table remains in the working directory. You cannot commit it, since it does not yet run, but you must switch context to check the co-investigator's number. Stash the remainder:

```
$ git stash push -m 'unfinished age-subgroup table'
Saved working directory and index state On main: unfinished age-subg
$ git status
On branch main
nothing to commit, working tree clean
```

With a clean tree you investigate the enrollment count, find and commit a fix:

```
$ git add -p summary.R
(1/1) Stage this hunk [y,n,q,a,d,s,e,?]? y
$ git commit -m 'Count enrolled, not screened, participants'
[main b3d5f70] Count enrolled, not screened, participants
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now resume the subgroup work exactly where you left it:

Draft, November 13, 2026. Content may change.

```
$ git stash pop
On branch main
Changes not staged for commit:
    modified:   analysis.R
Dropped refs/stash@{0} (c7e9a12...)
$ git log --oneline -3
b3d5f70 Count enrolled, not screened, participants
9a1c3e5 Correct age-eligibility threshold to >= 18
7d9f0b3 Add change-in-HbA1c table by sex
```

The history records two clean, separately intelligible fixes. The co-investigator's question was answered without losing the in-progress table, and no unrelated changes were bundled together. This is the everyday value of selective staging and the stash.

1.4. Homework

Work in a scratch repository. Create one and seed it with a short R script so that you have something to edit:

```
mkdir stash-practice
cd stash-practice
git init
printf 'x <- 1\ny <- 2\nz <- 3\n' > script.R
git add script.R
git commit -m 'Initial script'
```

1. **Two hunks, two commits.** Edit `script.R` so that it has two separated changes: change the first line and, leaving the middle line untouched, change the last line. Use `git add -p` to stage and commit each change as its own commit with a distinct message. Confirm with `git log --oneline` that you produced two commits.
2. **Stage across a gap.** In a single edit, add a new line after line 1 and another new line after line 3. Run `git add -p` and stage only the first of the two new lines. What does `git diff --staged` show, and what does `git diff` (the unstaged remainder) show?

Draft, November 13, 2026. Content may change.

3. **Unstage selectively.** Stage all changes to `script.R` with `git add script.R`, then use `git restore --staged -p` to unstage one hunk while keeping the other staged. Confirm the split with `git status`. Did any change leave your working directory?

4. **Discard, and reflect on danger.** Make an edit to `script.R` that you decide you do not want. Discard it with `git restore script.R`. Then attempt to recover it. Can you? Explain, in one or two sentences, why or why not.

5. **A clean interruption.** Begin an edit to `script.R`, leaving it unfinished. Without committing, stash it with a descriptive message. Create a file `hotfix.txt`, commit it, then restore your stashed work and confirm it is back. Show the output of `git stash list` before and after the restore.

6. **Stash only some files.** Modify both `script.R` and a new tracked file `notes.txt` (create and commit it first). Stash only `notes.txt` using `git stash push notes.txt`. Confirm that `script.R` remains modified in the working directory while `notes.txt` is clean.

7. **Stash or commit?** For each of the following, state whether you would stash or make a work-in-progress commit, and why: (a) you need to switch branches for thirty seconds to read a colleague's code; (b) you are stopping work for the weekend mid-change; (c) you want to run the test suite against a clean version of the last commit without losing your current edits.

1.5. Solutions

Problem 1.

```
# Edit script.R: line 1 'x <- 1' -> 'x <- 10'
#                line 3 'z <- 3' -> 'z <- 30'
git add -p script.R
#> (1/2) Stage this hunk? y
#> (2/2) Stage this hunk? n
git commit -m 'Change x to 10'
```


Draft, November 13, 2026. Content may change.

```
git add script.R
git commit -m 'Change z to 30'
git log --oneline
#> 3rd... Change z to 30
#> 2nd... Change x to 10
#> 1st... Initial script
```

The unchanged middle line gives Git the gap it needs to present the two edits as separate hunks. Patch mode can then stage them independently.

Problem 2. After staging only the first new line, `git diff --staged` shows the insertion after line 1, and `git diff` shows the still-unstaged insertion after line 3. The single edit has been split across the staging boundary: part is staged, part remains in the working directory. This is the same staged-versus-modified split the boot camp introduced, now applied at the level of individual hunks rather than whole files.

Problem 3.

```
git add script.R
git restore --staged -p script.R
#> (1/2) Unstage this hunk? y
#> (2/2) Unstage this hunk? n
git status
#> Changes to be committed:
#>  modified:   script.R
#> Changes not staged for commit:
#>  modified:   script.R
```

No change left the working directory. `git restore --staged` moves changes from the staging area back to the working directory; it never discards them. The file appears under both headings because one hunk is staged and one is not.

Problem 4. You cannot recover it. `git restore script.R` overwrites the working-directory copy with the version from the last commit, and the discarded edit was never committed or stashed. It exists nowhere in Git's object database. The reflog, which rescues committed work on Day 3, cannot

Draft, November 13, 2026. Content may change.

help here, because it tracks commits and branch tips, not uncommitted working-directory changes. This is the one truly unrecoverable class of loss in Git, and it is why the lecture warns against `restore` on changes you are unsure about.

Problem 5.

```
# begin editing script.R, leave unfinished
git stash push -m 'half-done refactor of script.R'
git stash list
#> stash@{0}: On main: half-done refactor of script.R
echo 'urgent fix' > hotfix.txt
git add hotfix.txt
git commit -m 'Add hotfix'
git stash pop
git stash list
#> (empty)
```

Before the restore the stash list has one entry; after `git stash pop` it is empty, because `pop` reapplies the stash and removes it from the stack. The unfinished edit to `script.R` is back in the working directory.

Problem 6.

```
printf 'first note\n' > notes.txt
git add notes.txt
git commit -m 'Add notes'
# modify both script.R and notes.txt
git stash push notes.txt
git status --short
#>  M script.R
```

Only `notes.txt` was stashed, so `script.R` remains modified in the working directory and `notes.txt` is clean. Path- limited stashing is the tool for setting aside one file's changes while continuing on another.

Problem 7.

(a) **Stash.** A thirty-second context switch you will reverse immediately is the exact case the stash is for.

(b) **Commit** (a work-in-progress commit on your branch). An interruption spanning a weekend is far too long for an invisible stash; a named commit is visible in the log, attached to the branch, and recoverable, and it can be folded into a clean commit later with Day 2's interactive rebase.

(c) **Stash** (or `git stash --keep-index`). You want a clean working tree matching the last commit temporarily, then your edits back. Stashing gives you exactly that; `git stash pop` afterwards restores the edits. A commit would also work but is heavier than the moment requires.

1.6. What's next

Today you learned to assemble clean commits as you go and to set work aside safely. Day 2 turns to history that is already recorded but not yet clean: amending the last commit, and using interactive rebase to reword, reorder, squash, and drop earlier commits, so that a branch of work-in-progress commits becomes a sequence a reviewer can read. It also sets out the boundary that governs all of this, namely when rewriting is safe and when it is forbidden. Keep the `trial-analysis` repository, since Day 2 continues from here.

Draft, November 13, 2026. Content may change.

2. Day 2: Rewriting History

2.1. Learning objectives

By the end of this day you should be able to:

- Explain that rewriting a commit produces a new commit with a new hash, and that the old commit is replaced rather than edited in place.
- Repair the most recent commit with `git commit --amend`: correct its message, add a forgotten file, or fold in a further edit, including the `--amend --no-edit` form.
- Refer to commits for rewriting by `HEAD~n`, by short hash, and by range.
- Run an interactive rebase with `git rebase -i HEAD~n`, and use `pick`, `reword`, `edit`, `squash`, `fixup`, `drop`, and line reordering to reshape a messy branch.
- Squash a cluster of work-in-progress commits into one clean, reworded commit, and abort cleanly when a rebase goes wrong.
- Prepare a fixup with `git commit --fixup <hash>` and apply it with `git rebase -i --autosquash`.
- Copy a single commit onto the current branch with `git cherry-pick <hash>`.
- State the governing rule of history rewriting and apply it: rewrite only local, unpublished commits, and update a pushed personal branch with `git push --force-with-lease`, never a bare `--force` on a shared branch.

Draft, November 13, 2026. Content may change.

2.2. Lecture

2.2.1. Rewriting means replacing

Yesterday you learned to assemble clean commits as you go. Today concerns the commits you did not assemble cleanly: the branch that already carries three commits named ‘wip’, ‘fix’, and ‘oops’, or the commit whose message has a typo, or the file you forgot to stage before committing. Git lets you reshape recorded history so that what a reviewer reads is a deliberate sequence rather than a transcript of your false starts.

One fact governs everything that follows. A commit is immutable: its hash is a checksum of its content, its message, its author, and its parent. You cannot edit a commit. What the commands below actually do is build a **new** commit from the old one plus your change, give it a new hash, and move the branch pointer to it. The original commit is not modified. It is abandoned, left with no branch pointing at it, and can be recovered through the reflog (Day 3) until it is eventually garbage-collected. Keep this in mind: every ‘edit’ of history in this chapter is a replacement.

2.2.2. Amending the last commit

The simplest rewrite targets only the most recent commit. `git commit --amend` replaces `HEAD` with a new commit built from whatever is currently staged, plus the old commit’s content, opening an editor on the old message so you can revise it.

The commonest use is a message you want to fix. Suppose you have just committed and see a typo:

```
$ git log --oneline -1
a1b2c3d Ad change-in-HbA1c tabel by sex
$ git commit --amend
```

Your editor opens on the old message; correct it, save, and close:

Draft, November 13, 2026. Content may change.

```
[main e4f5a6b] Add change-in-HbA1c table by sex
 1 file changed, 3 insertions(+)
```

Note the hash changed, from `a1b2c3d` to `e4f5a6b`. This is the replacement at work: the reworded commit is a different object.

The second use is the forgotten file. You committed `analysis.R` but forgot that the change also needed a new helper, `R/subgroup.R`. Rather than a second commit reading 'add file I forgot', fold it into the commit it belongs to:

```
git add R/subgroup.R
git commit --amend --no-edit
```

The `--no-edit` flag keeps the existing message unchanged, which is what you want when the content changes but the message still describes it correctly. The same pattern folds in a further edit: stage the additional change with `git add -p` from Day 1, then `git commit --amend --no-edit` to absorb it into the last commit.

Check your understanding: what does amend replace?

You commit, then run `git add forgotten.R` and `git commit --amend --no-edit`. Afterwards, how many commits does `git log --oneline` show compared with before, and has the hash of the top commit changed?
The count is unchanged. Amend does not add a commit; it replaces the existing top commit. The hash of that top commit **has** changed, because the new commit contains an extra file and therefore a different content checksum. The old commit still exists in the object database, reachable through the reflog, but no branch points to it. This is why amend, like rebase, is safe only on unpublished commits. Any collaborator who had fetched the old hash now holds a commit your branch no longer contains.

> **Warning**
>
> `git commit --amend` rewrites the last commit and so is subject to the governing rule below. If the commit you are amending has already been pushed to a shared branch, amending it and pushing again forces a divergence that every collaborator must reconcile. Amend freely while the commit is still local, but think twice once it has left your machine.

2.2.3. Referring to commits

Rewriting commands need a way to name the commits they act on. Three notations from the boot camp suffice, recapped briefly.

- **`HEAD`** is the current commit; **`HEAD~1`** (or `HEAD~`) is its parent, **`HEAD~2`** its grandparent, and in general `HEAD~n` is the commit `n` steps back along the first-parent line.
- A **short hash**, the seven-or-so characters `git log --oneline` prints, names a commit directly and unambiguously in all but the largest repositories.
- A **range** `A..B` denotes the commits reachable from `B` but not from `A`, that is, the commits after `A` up to and including `B`. Interactive rebase uses this implicitly: `git rebase -i HEAD~3` replays the three commits after `HEAD~3`, namely `HEAD~2`, `HEAD~1`, and `HEAD`.

When you plan a rebase, count the commits you intend to reshape and name the commit just before them. To tidy the last three commits, that base is `HEAD~3`.

2.2.4. Interactive rebase

Interactive rebase is the central tool of this chapter. It takes a run of consecutive commits, presents them as an editable ‘to-do’ list, and replays them according to the instructions you leave. With it you can reword messages, reorder commits, combine several into one, split work across a stop, and delete commits entirely, all before anyone else sees them.

Draft, November 13, 2026. Content may change.

Start it by naming the base commit, the one just before the first commit you want to touch:

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

Git opens your editor on a to-do list. The commits are listed **oldest first**, the reverse of `git log`, because they will be replayed in that order:

```
pick 7a1c9e0 wip
pick 3f2b8d1 fix
pick c5e4a90 oops

# Rebase 9a1c3e5..c5e4a90 onto 9a1c3e5 (3 commands)
#
# Commands:
# p, pick <commit>   = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit>   = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit>  = like "squash", but discard this commit's log m
# d, drop <commit>   = remove commit
# ...
```

You reshape history by editing this list. Each line begins with an **action**:

- **`pick`** replay the commit unchanged. This is the default for every line.
- **`reword`** replay the commit but stop to edit its message.
- **`edit`** stop after replaying the commit, with it as `HEAD`, so you can amend its content or split it, then resume.
- **`squash`** meld this commit into the one **above** it, combining their messages into one you get to edit.
- **`fixup`** like `squash`, but discard this commit's message and keep only the one above.
- **`drop`** discard the commit entirely (equivalently, delete its line).

You can also **reorder** commits by moving whole lines up or down, and Git will replay them in the new order. Reordering is safe when the commits

touch different files; when they touch the same lines it can produce a conflict you resolve as in a merge.

Consider the three messy commits above, the residue of a Day 1 session where you committed 'wip' early, then 'fix' and 'oops' as you corrected it. They are one logical change recorded in three untidy steps. You want a single, well-named commit. Edit the to-do list so the first commit is reworded and the other two are folded into it with `fixup` (their messages, 'fix' and 'oops', are worthless and can be discarded):

```
reword 7a1c9e0 wip
fixup 3f2b8d1 fix
fixup c5e4a90 oops
```

Save and close. Because the first line says `reword`, Git now opens a second editor for its message. Replace 'wip' with something a reviewer can read:

```
Add age-subgroup HbA1c table with tests
```

Save and close again. Git replays the three commits as one and reports:

```
[detached HEAD b8d2f31] Add age-subgroup HbA1c table with tests
 Date: Wed Jul 22 09:14:02 2026 -0700
 2 files changed, 41 insertions(+)
Successfully rebased and updated refs/heads/main.
```

The log now shows one clean commit where three messy ones stood:

```
$ git log --oneline -2
b8d2f31 Add age-subgroup HbA1c table with tests
9a1c3e5 Correct age-eligibility threshold to >= 18
```

2.2.4.1. Stopping to amend: the edit action

`squash` and `fixup` combine commits; `edit` does the opposite work of letting you change one commit in the middle of a run. When the rebase reaches a line marked `edit`, it replays that commit and then stops, leaving it as `HEAD` with a clean working tree, and prints instructions:

Draft, November 13, 2026. Content may change.

```
Stopped at 3f2b8d1...  fix
You can amend the commit now, with
  git commit --amend
Once you are satisfied with your changes, run
  git rebase --continue
```

At this stop you do ordinary work: edit files, stage with `git add`, and fold the change in with `git commit --amend`, exactly as earlier in this chapter. You might also split the commit here, resetting it and making two smaller commits in its place. When the commit is as you want it, resume the replay:

```
git rebase --continue
```

Git applies the remaining to-do lines and finishes.

2.2.4.2. Aborting a rebase

If a rebase goes wrong, whether you mistyped an action, hit a conflict you are not ready to resolve, or simply changed your mind, you are never trapped. At any pause, restore the branch to exactly its pre-rebase state:

```
git rebase --abort
```

This returns `HEAD` and the working tree to where they were before you started, as if the rebase had never run. Reach for it whenever a rebase confuses you; then start again more carefully.

> **Warning**
>
> An interactive rebase rewrites every commit it replays, giving each a new hash, including commits you left as `pick` if they follow a rewritten one, because their parent changed. Never run `git rebase -i` across commits you have already pushed to a branch others share. On your own unpushed work, or a personal feature branch nobody has based work on, rebase freely; the reflog (Day 3) makes even a botched rebase

recoverable.

> **Tip**
>
> If your editor opens on the to-do list and you cannot recall what an action does, the comment block Git prints beneath the list is a full reference, and closing the editor with the list unchanged (all `pick`) simply replays history as it was, a harmless no-op. Save an empty or all-comment list, or run `git rebase --abort` at the next prompt, to back out without effect.

2.2.5. The autosquash workflow

Choosing to squash while editing a to-do list works, but Git offers a way to mark the intent at commit time, which is tidier when you notice a fix belongs to an earlier commit. If you have just written a correction that properly belongs in commit `9a1c3e5`, commit it as a fixup of that commit:

```
git add analysis.R
git commit --fixup 9a1c3e5
```

This creates an ordinary commit whose message is `fixup! Correct age-eligibility threshold to >= 18`, echoing the target's subject. It records, in the message, that this commit is destined to be folded into `9a1c3e5`. Later, run an interactive rebase with `--autosquash`:

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

Git prepares the to-do list with the fixup commit already moved directly beneath its target and pre-marked `fixup`, so you review and save rather than reorder by hand. The result is identical to a manual squash, but the bookkeeping was done when the correction was fresh in your mind.

Draft, November 13, 2026. Content may change.

2.2.6. Cherry-picking a single commit

Rebase reshapes a run of commits in place. Sometimes you instead want one specific commit from **elsewhere**, applied onto your current branch. `git cherry-pick <hash>` does exactly that: it takes the change introduced by one commit and replays it as a new commit on `HEAD`.

Suppose a colleague fixed a units bug in a shared helper on another branch, in commit `d17a4c2`, and you need that fix on your feature branch now, without merging the whole branch:

```
$ git cherry-pick d17a4c2
[subgroup-analysis 2e9f0a7] Fix mmol/mol to percent conversion
 1 file changed, 2 insertions(+), 2 deletions(-)
```

The result is a new commit on your branch with the same change and message as the original, but a new hash, because its parent differs. Cherry-pick is the tool for lifting one useful commit out of context: a hotfix you need on two branches, or a single good commit from an abandoned line of work.

A cherry-pick can conflict, exactly as a merge can, when the change does not apply cleanly to your branch. Git pauses and marks the conflicting files with the same `<<<<<<<`, `=======`, `>>>>>>>` markers you resolved in the boot camp. Edit the files to resolve, stage them, then continue:

```
git add <resolved-file>
git cherry-pick --continue
```

As with rebase, `git cherry-pick --abort` backs the operation out entirely if you would rather not proceed.

2.2.7. The governing rule: do not rewrite published history

Every operation in this chapter replaces commits with new ones. That is safe when the commits are yours alone, and dangerous the moment they are shared. The rule is short and absolute:

Draft, November 13, 2026. Content may change.

> Never rewrite history that has been published, that is, pushed to a branch that others may have pulled. Rewrite only local, unpushed commits, or commits on a personal branch that nobody else has based work on.

The reason is mechanical. Rewriting a pushed commit replaces it with one of a different hash, so your branch and the remote disagree about what commits exist. A collaborator who pulled the original still holds it, along with any work built on top of it. When your rewritten branch reaches them, Git cannot fast-forward. Their downstream commits, now parented on a commit that no longer exists on the remote, can be duplicated or lost. A single rewrite of a shared branch can cost a team an afternoon.

The safe domain of rewriting is therefore precise:

- **Local commits you have not pushed.** Reshape them freely; nobody else has seen them.
- **A personal feature branch** that you pushed only to back up work or open a draft pull request, and that no collaborator has branched from or built upon. This is the one case where you may rewrite already-pushed commits, and it requires the careful force-push below.
- **Never** `main`, a shared `develop`, a release branch, or any branch on which others are actively working.

When in doubt, do not rewrite; a slightly untidy shared history is vastly preferable to a rewrite that strands a colleague's work.

Check your understanding: which rewrites are safe?

For each, decide whether rewriting is safe: (a) three local commits you have not pushed; (b) the last commit on `main`, which you pushed an hour ago and a co-author has since pulled; (c) your branch `subgroup-analysis`, pushed only to open a draft pull request nobody has reviewed or branched from.

(a) **Safe.** Unpushed local commits are yours alone; rebase or amend at will. (b) **Unsafe.** The commit is on a shared branch and a collaborator already holds it; rewriting it would force a reconciliation on them and risk their downstream work. Correct it with a new commit instead. (c) **Safe, with care.** A personal

Draft, November 13, 2026. Content may change.

branch nobody has built upon may be rewritten, but because it is already on the remote you must update it with `git push --force-with-lease`, described next, rather than an ordinary push.

2.2.8. Updating a pushed branch with force-with-lease

Once you rewrite commits you have already pushed, an ordinary `git push` is rejected. The remote and your rewritten branch have diverged, and Git will not fast-forward. You must tell Git to replace the remote branch with yours. The naive tool is `git push --force`, and it is dangerous, since it overwrites the remote branch unconditionally, discarding any commit a collaborator pushed in the interval that you have not seen.

Prefer `git push --force-with-lease`. It force-pushes only if the remote branch is still where you last observed it; if someone else has pushed to it since your last fetch, the push is refused and you are warned, so you can fetch, inspect their work, and avoid destroying it:

```
$ git push --force-with-lease origin subgroup-analysis
 + b8d2f31...c5e4a90 subgroup-analysis -> subgroup-analysis (forced
```

The distinction is the whole point. `--force` says ‘make the remote match me, whatever is there’. `--force-with-lease` says ‘make the remote match me, but only if it is still what I last saw’. The second refuses in exactly the case where the first would silently destroy work, namely an unexpected push by someone else.

> **Warning**
>
> Never `git push --force` (or `--force-with-lease`) to a shared branch such as `main`. Force-pushing is reserved for your own feature branch, and even there `--force-with-lease` is the only acceptable form. Configure a memorable habit: if the branch name is not one you created and control alone, no force-push of any kind is permitted. Branch protection, which the *Practicum* covers, exists precisely to make this mistake impossible on important branches.

2.3. Worked example: cleaning a feature branch before review

The `trial-analysis` repository continues from Day 1. You have been working on a feature branch, `subgroup-analysis`, and pushed it early to open a **draft** pull request so a co-investigator could follow along. Nobody has reviewed it or branched from it. The branch carries four commits, three of them the work-in-progress residue Day 1 warned about, plus one genuine, well-named commit made afterward:

```
$ git log --oneline -4
9c3a1f7 Add unit tests for subgroup function
b2e8d40 oops
a7f1c93 wip
5d0b6a2 wip sex subgroup
```

Read bottom to top, this is one coherent piece of work, the sex-subgroup analysis and its tests, recorded as four commits, three of them noise. You want the reviewer to see two clean commits: one adding the analysis function, one adding its tests. The before-and-after shape:

```
before:  ... -- 5d0b6a2 -- a7f1c93 -- b2e8d40 -- 9c3a1f7
              (wip)      (wip)      (oops)     (tests)

after:   ... -- F1 -------- F2
              (function)  (tests)
```

The three work-in-progress commits, `5d0b6a2`, `a7f1c93`, and `b2e8d40`, all build the subgroup function and belong together as commit F1; the test commit `9c3a1f7` becomes F2. Start an interactive rebase over the four commits:

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

Git opens the to-do list, oldest first:

Draft, November 13, 2026. Content may change.

```
pick 5d0b6a2 wip sex subgroup
pick a7f1c93 wip
pick b2e8d40 oops
pick 9c3a1f7 Add unit tests for subgroup function
```

Reword the first commit to name the function it introduces, fold the two later work-in-progress commits into it with `fixup`, and leave the test commit as its own `pick`:

```
reword 5d0b6a2 wip sex subgroup
fixup a7f1c93 wip
fixup b2e8d40 oops
pick 9c3a1f7 Add unit tests for subgroup function
```

Save and close. Because the first line is `reword`, Git opens an editor for its message; replace 'wip sex subgroup' with a description a reviewer can read:

```
Add change-in-HbA1c analysis by sex subgroup
```

Save and close. Git replays the branch and reports:

```
[detached HEAD 4a9e2c1] Add change-in-HbA1c analysis by sex subgroup
 2 files changed, 28 insertions(+)
Successfully rebased and updated refs/heads/subgroup-analysis.
```

Verify the result with a graph log. Four commits have become two, each single-purpose and clearly named:

```
$ git log --oneline --graph -3
* f70b3d8 Add unit tests for subgroup function
* 4a9e2c1 Add change-in-HbA1c analysis by sex subgroup
* 9a1c3e5 Correct age-eligibility threshold to >= 18
```

Note that the test commit's hash changed, from `9c3a1f7` to `f70b3d8`, although you left it as `pick`: its parent was rewritten, so it too is a new commit. This is the replacement principle in action.

The branch is now clean, but the remote still holds the old four-commit version from the draft pull request. Because this is your personal branch and nobody has built upon it, you may update it, using `--force-with-lease` so that the push is refused if the co-investigator happened to push something you have not seen:

```
$ git push --force-with-lease origin subgroup-analysis
 + 9c3a1f7...f70b3d8 subgroup-analysis -> subgroup-analysis (forced upd
```

The pull request now shows two intelligible commits. The reviewer reads a deliberate sequence, the analysis then its tests, with no trace of the work-in-progress path that produced it. This is the everyday purpose of rewriting: the history you were free to make messily becomes the history you publish cleanly.

2.4. Homework

Work in a scratch repository so that no rewrite can harm work you care about. Create one and seed it with a short R script:

```
mkdir rebase-practice
cd rebase-practice
git init
printf 'a <- 1\n' > script.R
git add script.R
git commit -m 'Initial script'
```

1. **Amend a message.** Add a line to `script.R`, commit it with a deliberately misspelled message, then fix the message with `git commit --amend`. Use `git log --oneline` before and after to confirm the commit's hash changed while the commit count did not.

Draft, November 13, 2026. Content may change.

2. **Amend in a forgotten file.** Create `helper.R` but forget to stage it; commit a change to `script.R` alone. Then stage `helper.R` and fold it into that same commit with `git commit --amend --no-edit`. Confirm with `git show --stat` that the one commit now contains both files.

3. **Reword with rebase.** Make three commits with the messages 'first', 'second', and 'third'. Run `git rebase -i HEAD~3` and reword only the middle commit. Show the resulting `git log --oneline`.

4. **Squash three into one.** Make three commits named 'wip', 'more', and 'done', each adding one line to `script.R`. Use an interactive rebase to combine them into a single commit with a clear message, discarding the 'more' and 'done' messages. Confirm the log shows one commit where three stood.

5. **Drop a commit.** Make three commits; the middle one adds a line `debug <- TRUE` you never wanted. Use `git rebase -i HEAD~3` to drop that commit, and confirm with `git show HEAD~1` and the file contents that the debug line is gone while the other two commits remain.

6. **Autosquash.** Commit a change, then commit a correction to it with `git commit --fixup <hash>` naming the first commit. Run `git rebase -i --autosquash HEAD~2` and observe that Git has pre-arranged the to-do list. Save it and confirm the two commits became one.

7. **Cherry-pick across branches.** From `main`, create a branch `feature` and add a commit. Return to `main` and note a commit hash on `feature` with `git log feature`. Cherry-pick that one commit onto `main` and confirm with `git log --oneline` that `main` now carries the change under a new hash.

8. **Reason about safety.** For each, state whether you may rewrite, and if you push afterward, which push form you would use: (a) two local commits you have not pushed; (b) a commit on `main` you pushed yesterday and a co-author has pulled; (c) your own branch `draft-pr`, pushed for a draft pull request nobody has reviewed or branched from.

Draft, November 13, 2026. Content may change.

2.5. Solutions

Problem 1.

```
# edit script.R, then:
git commit -am 'Ad a second lien'
git log --oneline -1
#> 1a2b3c4 Ad a second lien
git commit --amend
# fix the message in the editor to 'Add a second line'
git log --oneline -1
#> 9f8e7d6 Add a second line
```

The commit count is unchanged; amend replaces the top commit rather than adding one. The hash changed because the message is part of the commit's checksum, so a reworded commit is a new object.

Problem 2.

```
printf 'h <- 2\n' > helper.R
# forget to stage helper.R
git commit -am 'Add helper logic'
git add helper.R
git commit --amend --no-edit
git show --stat
#> Add helper logic
#>  helper.R | 1 +
#>  script.R | 1 +
#>  2 files changed, 2 insertions(+)
```

The `--no-edit` flag keeps the original message, which still describes the commit correctly. The single commit now records both files, as if `helper.R` had been staged all along.

Problem 3.

Draft, November 13, 2026. Content may change.

```
git rebase -i HEAD~3
# in the to-do list, change 'pick' to 'reword' on the
# middle line only:
#   pick   ... first
#   reword ... second
#   pick   ... third
# Git then opens an editor on the middle commit's message.
git log --oneline -3
#> ... third
#> ... second (reworded)
#> ... first
```

Only the middle commit's message changed, but note that its hash and the hash of every commit after it changed too, because a rewritten commit alters the parent of all its descendants.

Problem 4.

```
git rebase -i HEAD~3
# edit the to-do list to:
#   pick  ... wip
#   fixup ... more
#   fixup ... done
# no message editor opens, because 'fixup' discards the
# folded messages; to reword the survivor use 'reword' on
# the first line, or 'squash' on the others.
git log --oneline
#> ... wip
#> ... Initial script
```

Using `fixup` on the second and third lines discards their messages and melds their changes into the first commit. To give the survivor a clean message in the same operation, change its `pick` to `reword`.

Problem 5.

Draft, November 13, 2026. Content may change.

```
git rebase -i HEAD~3
# delete the middle line entirely, or change its action to
# 'drop':
#   pick ... first
#   drop ... add debug flag
#   pick ... third
git show HEAD~1
# the debug commit is gone; 'first' is now HEAD~1
grep debug script.R
#> (no output)
```

Dropping a commit removes its change from history entirely. The remaining two commits are replayed with new hashes onto the unchanged base, and the `debug <- TRUE` line never existed in the rebuilt history.

Problem 6.

```
git commit -am 'Add analysis step'      # say this is 3c4d5e6
# make a correction, then:
git add script.R
git commit --fixup 3c4d5e6
git log --oneline -2
#> ... fixup! Add analysis step
#> 3c4d5e6 Add analysis step
git rebase -i --autosquash HEAD~2
# the to-do list arrives pre-arranged:
#   pick  3c4d5e6 Add analysis step
#   fixup ....... fixup! Add analysis step
# save without editing.
git log --oneline -1
#> 7e8f9a0 Add analysis step
```

`--autosquash` reads the `fixup!` prefix, moves the fixup commit beneath its target, and pre-marks it `fixup`, so the squash needs no manual reordering. The two commits become one, under the target's message.

Problem 7.

Draft, November 13, 2026. Content may change.

```
git branch feature
git switch feature
printf 'f <- 9\n' >> script.R
git commit -am 'Add feature line'
git switch main
git log feature --oneline -1
#> abc1234 Add feature line
git cherry-pick abc1234
git log --oneline -1
#> def5678 Add feature line
```

The change from `feature` now sits on `main` as a new commit with a different hash (`def5678`, not `abc1234`), because its parent on `main` differs from its parent on `feature`. Cherry-pick copies a change, it does not move the original commit.

Problem 8.

(a) **May rewrite; ordinary push (or none).** Unpushed local commits are yours alone. If you push for the first time after rewriting, an ordinary `git push` suffices, because there is nothing on the remote to diverge from.

(b) **Must not rewrite.** The commit is on a shared branch and a co-author has pulled it; rewriting would force a painful reconciliation and risk their work. Correct it with a new commit and an ordinary push.

(c) **May rewrite; push with `--force-with-lease`.** A personal branch nobody has built upon is safe to rewrite, but because it is already published you must update the remote with `git push --force-with-lease`, never a bare `--force`.

2.6. What's next

Today you learned to reshape recorded history: amending the last commit, squashing and rewording a branch of work-in-progress commits into a clean sequence with interactive rebase, lifting a single commit with cherry-pick,

and updating a personal branch safely with `--force-with-lease`. All of this is bounded by the rule that published history is not rewritten. Every one of these operations replaces commits, and a replaced commit is not destroyed but merely unreferenced. Day 3 makes that safety net explicit. It covers `git reflog`, which records everywhere `HEAD` has been and lets you recover a commit abandoned by a rebase or an amend, and then `git bisect` and `git blame` for finding when and where a result changed. Keep the `trial-analysis` repository, since Day 3 continues from here.

Draft, November 13, 2026. Content may change.

3. Day 3: Recovery and Forensics

3.1. Learning objectives

By the end of this day you should be able to:

- Explain what `git reflog` records, namely every position `HEAD` and each branch tip has held, and why this makes most committed work recoverable.
- Read a reflog listing, interpreting `HEAD@{0}`, `HEAD@{1}`, and the operation labels beside each entry.
- Recover from an accidental `git reset --hard`, from a commit orphaned by an interactive rebase, and from a deleted branch, using the reflog to find the lost commit.
- State precisely the one class of loss the reflog cannot reverse, namely uncommitted working-directory changes.
- Locate the commit that introduced a change in a computed result with `git bisect`, both by hand and with `git bisect run`.
- Investigate how a line reached its current form with `git blame`, the pickaxe (`git log -S` and `git log -G`), and `git log -L`.

3.2. Lecture

3.2.1. The safety net beneath Day 2

Day 2 asked you to rewrite history: to amend, reorder, squash, and drop commits. Each of those operations replaces a commit with a new one and leaves the original with no branch pointing at it. If `git rebase -i` builds the wrong result, or a `git reset --hard` lands on the wrong commit, where does the displaced work go?

Draft, November 13, 2026. Content may change.

Nowhere. A commit that no branch and no tag points to is unreferenced, not deleted. Git keeps it in the object database until garbage collection eventually prunes it, and it records the fact that `HEAD` once pointed there. That record is the reflog, and it is the reason the rewriting of Day 2 is safe. Today you learn to read it and recover through it. We then turn from recovery to forensics, that is, the commands that reconstruct how a repository reached its present state.

3.2.2. What the reflog records

The **reflog** (reference log) is a local, per-repository journal of where references have pointed over time. There is one for `HEAD` and one for each branch. Every time `HEAD` moves, whether on a commit, checkout, reset, merge, rebase step, amend, or pull, Git appends a line to the `HEAD` reflog recording the new position and the operation that caused the move.

The essential properties to fix in mind:

- It records **commits and reference positions**, not uncommitted edits. Work you never committed is not in it.
- It is **local**. The reflog lives in `.git/logs/` and is never pushed or fetched. A fresh clone has an empty reflog; your collaborator cannot see yours.
- Entries **expire**. By default, reachable entries are kept 90 days and unreachable ones 30 days (`gc.reflogExpire` and `gc.reflogExpireUnreachable`). Recovery is a prompt operation, not an archival one.

3.2.3. Reading the reflog

Run `git reflog` (a shorthand for `git reflog show HEAD`) in the `trial-analysis` repository:

```
$ git reflog
b3d5f70 (HEAD -> main) HEAD@{0}: commit: Count enrolled, not screened,
9a1c3e5 HEAD@{1}: commit: Correct age-eligibility threshold to >= 18
7d9f0b3 HEAD@{2}: commit: Add change-in-HbA1c table by sex
```


Draft, November 13, 2026. Content may change.

```
5f3e9c1 HEAD@{3}: commit: Add primary endpoint computation
2b7c1a0 HEAD@{4}: commit (initial): Import trial data and codebook
```

Read it as newest-first. Each line has three parts: the commit the reference moved to, the selector `HEAD@{n}` naming the position `n` moves ago, and a label describing the operation. `HEAD@{0}` is always where `HEAD` is now; `HEAD@{1}` is where it was one move ago, and so on. The selector is usable anywhere a commit is expected, so `git show HEAD@{2}` shows the commit `HEAD` pointed at two moves back, and `git diff HEAD@{3} HEAD@{0}` compares then with now.

The labels matter, because they tell you what happened. `commit` is an ordinary commit, and you will also see `reset:`, `rebase (start):`, `rebase (finish):`, `checkout:`, `commit (amend):`, and `pull:`. When you are hunting for lost work, the label is how you recognize the move that lost it.

Check your understanding: what does amend leave in the reflog?

You commit, then immediately run `git commit --amend` to fix the message. How many entries does this add to the `HEAD` reflog, and can you still reach the pre-amend commit?
Two moves are recorded: the original `commit`, then a `commit (amend):` entry for the replacement. Amend does not edit a commit in place; it writes a new commit and moves `HEAD` to it, orphaning the original. The original is still in the object database and still named by its reflog entry, so `git show HEAD@{1}` (or the original's hash) reaches it. This is exactly why an amend you regret is recoverable: the superseded commit is orphaned, not destroyed.

3.2.4. Recovery 1: an accidental reset –hard

The classic disaster. You mean to discard the last commit but mistype the count, or you run `git reset --hard` against the wrong target, and two commits of work vanish from the branch:

Draft, November 13, 2026. Content may change.

```
$ git reset --hard HEAD~2
HEAD is now at f81d2a9 Update report figures and caption
$ git log --oneline -1
f81d2a9 Update report figures and caption
```

The two commits are gone from `main`, but not from the reflog. Look:

```
$ git reflog
f81d2a9 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~2
3c9a1f7 HEAD@{1}: commit: Document corrected eligibility threshold
a70e5d2 HEAD@{2}: commit: Add unit test for threshold filter
f81d2a9 HEAD@{3}: commit: Update report figures and caption
```

`HEAD@{1}` is the branch tip as it stood immediately before the reset, namely the commit you wanted to keep. The reset moved only the `main` pointer; the commits themselves are untouched and still named by the reflog:

```
before:  f81d2a9 -- a70e5d2 -- 3c9a1f7
                                  |
                                main  (HEAD@{1})

after:   f81d2a9 -- a70e5d2 -- 3c9a1f7   (orphaned, in reflog)
            |
          main  (HEAD@{0})
```

Two ways to recover, one destructive-looking and one cautious.

The direct route moves the branch back to that tip:

```
$ git reset --hard HEAD@{1}
HEAD is now at 3c9a1f7 Document corrected eligibility threshold
```

The cautious route, preferable when you are not certain, does not touch `main` at all. It creates a new branch at the lost commit so you can inspect it before deciding:

Draft, November 13, 2026. Content may change.

```
$ git branch recovered HEAD@{1}
$ git log --oneline recovered -2
3c9a1f7 Document corrected eligibility threshold
a70e5d2 Add unit test for threshold filter
```

Recovery never requires typing a forty-character hash from memory. If you can point at the commit in the reflog, you can point a branch at it.

3.2.5. Recovery 2: a commit orphaned by interactive rebase

Day 2 warned that dropping a line in the `git rebase -i` to-do list deletes that commit from the branch. Suppose during a squash you deleted the wrong line and lost a commit named `Add subgroup sensitivity analysis`. It is not on the rebased branch, but the rebase recorded its passage:

```
$ git reflog
d4f0a19 (HEAD -> main) HEAD@{0}: rebase (finish): returning to refs/
d4f0a19 HEAD@{1}: rebase (pick): Update report figures and caption
6b2c8e3 HEAD@{2}: rebase (start): checkout main~4
e8a1c05 HEAD@{3}: commit: Add subgroup sensitivity analysis
c1e8f42 HEAD@{4}: commit: Add sensitivity analysis on complete cases
```

The `rebase (start):` entry marks where the rebase began; the commit just above it in real time, `e8a1c05`, is the one the rebase omitted. Recover it by cherry-picking it back onto the current branch:

```
$ git cherry-pick e8a1c05
[main 5a9d3c1] Add subgroup sensitivity analysis
 1 file changed, 12 insertions(+)
```

If you would rather examine it in isolation first, branch it as before with `git branch recovered e8a1c05` and inspect, then merge or cherry-pick when satisfied. Either way, the reflog turned an irreversible-looking rebase mistake into a one-line fix.

Draft, November 13, 2026. Content may change.

3.2.6. Recovery 3: a deleted branch

Deleting a branch removes only the pointer, not the commits it named. If you delete a feature branch that was never merged, Git even warns you and prints the tip hash:

```
$ git branch -D age-subgroups
Deleted branch age-subgroups (was 7c1b9f4).
```

Should you want it back, the tip hash in that message is enough:

```
$ git branch age-subgroups 7c1b9f4
```

If the message has scrolled away, the `HEAD` reflog still holds the `checkout:` entries from when you worked on the branch, and each branch keeps its own reflog until deletion. Searching the `HEAD` reflog for the last commit you made there recovers the tip:

```
$ git reflog | grep age-subgroups
7c1b9f4 HEAD@{9}: commit: Add age-stratified change table
```

Point a branch at `7c1b9f4` and the work is restored.

> **Warning**
>
> The reflog cannot recover uncommitted work. Changes you never committed, whether discarded by `git restore`, overwritten by `git checkout`, or wiped by `git reset --hard`, were never written to the object database, and they leave no reflog entry. This is the same irreducible loss Day 1 flagged for `git restore -p`, and it is the only class of loss in this book that no command reverses. The reflog protects commits, not working-directory edits. If work matters, commit it, if only to a scratch branch, before running any `--hard` operation.

Draft, November 13, 2026. Content may change.

> **Warning**
>
> Do not treat the reflog as a backup. It is local, so it vanishes with the machine or the working copy; it is not pushed, so a collaborator's clone cannot restore your lost commit; and its entries expire, so a rewrite you regret in four months may already be pruned. The reflog is a short-horizon safety net for the operations of this book, not a substitute for pushing to a remote and, in due course, the backup and archival practice the Practicum covers.

3.2.7. From recovery to forensics

The reflog answers 'how do I get my work back'. The rest of today answers a different question: 'how did the repository come to be as it is'. In statistical practice the trigger is almost always a number that moved. A figure in this week's report disagrees with last week's, a count in a table is not what a co-investigator remembers, or a p-value crossed a threshold. Some commit changed it, and you must find which, understand what it did, and decide whether it was correct. Git provides three complementary instruments for this task: `git bisect` to find the commit, `git blame` and the pickaxe to find the line and its history, and `git show` to read the commit once you have it.

3.2.8. Finding the culprit commit with git bisect

When a result changed and you do not know which of many commits changed it, `git bisect` performs a binary search over the history. You mark one commit where the result was correct (`good`) and one where it is wrong (`bad`); Git checks out the midpoint, you test it and report `good` or `bad`, and Git halves the remaining range each time. Over a hundred commits it converges in about seven tests.

Frame it concretely. The mean twelve-week change in HbA1c was reported as `-0.84` last week and is `-0.71` today, and you want the commit that moved it. Start a bisect, mark the current (broken) state bad and last week's tagged release good:

Draft, November 13, 2026. Content may change.

```
$ git bisect start
$ git bisect bad
$ git bisect good 2b7c1a0
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[b3d5f70] Count enrolled, not screened, participants
```

Git has checked out the midpoint commit `b3d5f70`. Test it, that is, run the analysis and read the value:

```
$ Rscript analysis.R
[1] -0.71
```

Still wrong, so this commit is bad:

```
$ git bisect bad
Bisecting: 1 revision left to test after this (roughly 1 step)
[7d9f0b3] Add change-in-HbA1c table by sex
```

Test the new midpoint:

```
$ Rscript analysis.R
[1] -0.84
```

Correct, so this commit is good:

```
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[9a1c3e5] Correct age-eligibility threshold to >= 18
```

One commit remains between the last good and the first bad. Test it; the value is `-0.71`, so it is bad, and Git has its answer:

```
$ git bisect bad
9a1c3e5 is the first bad commit
commit 9a1c3e5...
    Correct age-eligibility threshold to >= 18
 analysis.R | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```


Draft, November 13, 2026. Content may change.

The search is over. When you finish, always end the session, which returns `HEAD` to where you started and restores the branch:

```
$ git bisect reset
Previous HEAD position was 9a1c3e5 Correct age-eligibility threshold
Switched to branch 'main'
```

Note that `git bisect` does not tell you the change was wrong, only which commit is responsible. Whether lowering the age-eligibility threshold to 18 was correct is a scientific judgment; bisect has narrowed a hundred candidates to one so you can make it.

3.2.8.1. Automating with git bisect run

Testing each midpoint by hand is tedious and, worse, error-prone. A moment's inattention and you mark a commit `good` that was bad, corrupting the search. If you can express the test as a script that exits `0` when the state is good and non-zero when it is bad, `git bisect run` performs every step for you. Write a check script, `check-mean.R`, that recomputes the value and sets the exit status:

```
source("analysis.R")
target <- -0.84
q(status = if (abs(mean_change - target) < 0.01) 0 else 1)
```

Then hand it to bisect after marking the endpoints:

```
$ git bisect start
$ git bisect bad
$ git bisect good 2b7c1a0
$ git bisect run Rscript check-mean.R
```

Git checks out each midpoint, runs the script, reads its exit status as the `good`/`bad` verdict, and drives the search to completion unattended, printing `<hash> is the first bad commit` at the end. Follow with `git bisect reset` as before. An automated bisect over a script is the closest Git comes to answering 'which commit broke this' with a single command.

Draft, November 13, 2026. Content may change.

Check your understanding: which endpoint is good?

A result was correct in an old commit and is wrong now. When you start a bisect, which endpoint do you mark `good` and which `bad`, and what would happen if you swapped them?
Mark the current, wrong state `bad` and the old, correct commit `good`; here `bad` means 'exhibits the change you are tracing' and `good` means 'does not yet'. The labels are arbitrary tokens for the two sides of a transition, not moral judgments, but they must be consistent with how your test decides. If you swap them, Git searches the wrong direction, assumes the property appeared going backwards in time, and either reports a nonsensical 'first bad commit' or complains that the good commit is a descendant of the bad one. With `git bisect run`, swapping is equivalent to inverting the script's exit status, and the reported culprit is meaningless.

3.2.9. Reading a line's history with git blame

Bisect finds the commit; `git blame` finds the line. `git blame <file>` annotates every line of a file with the commit, author, and timestamp that last changed it. On a long file the full output is unwieldy, so restrict it to a range with `-L <start>,<end>`. Suppose you are asking 'why is the eligibility threshold 18, and when did it become that':

```
$ git blame -L 8,11 analysis.R
9a1c3e5a (Rex Thomas 2026-07-18 09:14:02 -0700  8) dat <- dat |>
9a1c3e5a (Rex Thomas 2026-07-18 09:14:02 -0700  9)   filter(age >= 18)
5f3e9c10 (Rex Thomas 2026-07-10 11:02:55 -0700 10)
5f3e9c10 (Rex Thomas 2026-07-10 11:02:55 -0700 11) mean_change <- dat |
```

Line 9, the `filter(age >= 18)`, was last touched by `9a1c3e5`, on 18 July, the same commit bisect identified. The neighboring lines are older, from `5f3e9c1`. Blame has told you when the threshold reached its current value and in which commit to look for the reasoning.

Blame reports only the **last** change to each line. If the threshold was 21, then 18, then back to 18 after a revert, blame shows only the most recent

Draft, November 13, 2026. Content may change.

commit, and the earlier history is invisible to it. To follow a line across every change, use `git log -L`, below. Note also what blame does not provide. The commit's author and message may explain why the change was made, but blame itself answers only who, what, and when, never why. For why you must read the commit.

> **Tip**
>
> If blame points at a commit that merely reformatted the file, reindented it, or renamed a variable throughout, the answer you want is one change further back. `git blame` accepts a starting commit: `git blame 9a1c3e5~1 -L 8,11 analysis.R` re-runs the blame as of the parent of the noise commit, skipping past it. Repeating this walks backward through cosmetic changes to the commit that last altered the logic.

3.2.10. The pickaxe and line-log

Blame starts from a line and looks back one step. Often you want the opposite: start from a string or a value and find every commit that touched it. This is the **pickaxe**.

`git log -S '<string>'` lists the commits that changed the number of occurrences of the exact string, that is, the commits that added or removed it. To find when `na.rm = TRUE` first entered the analysis:

```
$ git log -S 'na.rm = TRUE' --oneline -- analysis.R
5c7e9a1 Fix missing na.rm in HbA1c mean
```

One commit added the string, so one commit is reported. `-S` counts occurrences, so a commit that moves the string without changing how many times it appears is, deliberately, not reported; you are asking 'when did this appear or disappear', not 'when was this line touched'.

For a pattern rather than a fixed string, `git log -G '<regex>'` reports every commit whose diff contains an added or removed line matching the regular expression, whether or not the count changed. To trace all edits touching the eligibility threshold, whatever its value:

Draft, November 13, 2026. Content may change.

```
$ git log -G 'filter\(age' --oneline -- analysis.R
9a1c3e5 Correct age-eligibility threshold to >= 18
5f3e9c1 Add primary endpoint computation
```

Both commits that touched the filter are reported: the one that introduced it and the one that changed the threshold.

Finally, `git log -L <start>,<end>:<file>` reconstructs the complete history of a specific line range. It shows each commit that changed it together with the diff it applied, and so answers 'show me everything that ever happened to these lines':

```
$ git log -L 9,9:analysis.R
commit 9a1c3e5...
    Correct age-eligibility threshold to >= 18
@@ -9,1 +9,1 @@
-  filter(age >= 21)
+  filter(age >= 18)

commit 5f3e9c1...
    Add primary endpoint computation
@@ -0,0 +1,1 @@
+  filter(age >= 21)
```

Now the full story is visible: `5f3e9c1` introduced the line with a threshold of 21, and `9a1c3e5` changed it to 18. This is what blame could not show, because blame reports only the most recent change. Between the pickaxe, which finds commits by content, and `-L`, which follows a line through time, you can reconstruct the provenance of any value in an analysis.

3.2.11. Reading the commit with git show

Bisect, blame, and the pickaxe all end by handing you a commit hash. The command that reads it is `git show <hash>`, from the boot camp: it prints the commit's metadata, message, and full diff. Having identified `9a1c3e5`, read it:

Draft, November 13, 2026. Content may change.

```
$ git show 9a1c3e5
commit 9a1c3e5...
Author: Rex Thomas <rgthomas@ucsd.edu>
Date:   Sat Jul 18 09:14:02 2026 -0700

    Correct age-eligibility threshold to >= 18

diff --git a/analysis.R b/analysis.R
@@ -8,3 +8,3 @@ dat <- read.csv("data-raw/trial.csv")
 dat <- dat |>
-  filter(age >= 21)
+  filter(age >= 18)
```

The message states the intent and the diff shows the act. The threshold moved from 21 to 18, which admitted the 18-to-20 age band into the analysis population and shifted the mean change. Whether that was the correct eligibility criterion is now a question for the protocol, not for Git, and the investigation is complete.

3.3. Worked example: a number that moved

The `trial-analysis` repository is on `main`. In this week's report the primary result, the mean twelve-week change in HbA1c, reads `-0.71`; last week's archived report gave `-0.84`. No one recalls changing the analysis. You will find the responsible commit, understand it, and, along the way, recover a commit a collaborator lost to a stray reset.

First, confirm the current value and reproduce the discrepancy:

```
$ Rscript analysis.R
[1] -0.71
```

Last week's value came from the release tagged `2b7c1a0`. Use it as the known-good endpoint and bisect:

Draft, November 13, 2026. Content may change.

```
$ git bisect start
$ git bisect bad
$ git bisect good 2b7c1a0
Bisecting: 2 revisions left to test after this (roughly 2 steps)
[b3d5f70] Count enrolled, not screened, participants
$ Rscript analysis.R
[1] -0.71
$ git bisect bad
Bisecting: 1 revision left to test after this (roughly 1 step)
[7d9f0b3] Add change-in-HbA1c table by sex
$ Rscript analysis.R
[1] -0.84
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[9a1c3e5] Correct age-eligibility threshold to >= 18
$ Rscript analysis.R
[1] -0.71
$ git bisect bad
9a1c3e5 is the first bad commit
```

Three tests converged on `9a1c3e5`. End the session:

```
$ git bisect reset
Switched to branch 'main'
```

Now understand what `9a1c3e5` did. Read the commit, then confirm the line with a focused blame:

```
$ git show 9a1c3e5 --stat
    Correct age-eligibility threshold to >= 18
 analysis.R | 2 +-
$ git blame -L 8,9 analysis.R
9a1c3e5a (Rex Thomas 2026-07-18 09:14:02 -0700  8) dat <- dat |>
9a1c3e5a (Rex Thomas 2026-07-18 09:14:02 -0700  9)   filter(age >= 18)
```

The commit lowered the eligibility threshold from 21 to 18, enlarging the analysis population and moving the mean. The number moved for a

Draft, November 13, 2026. Content may change.

documented reason; the report needs only a note that the eligibility criterion changed between the two analyses. The forensic question is answered.

While preparing the correction, a collaborator on the same clone runs a reset against the wrong commit and reports that `Document corrected eligibility threshold` has disappeared from `main`:

```
$ git log --oneline -1
f81d2a9 Update report figures and caption
```

The commit is not on the branch, but the reflog holds the tip as it stood before the reset:

```
$ git reflog
f81d2a9 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~2
3c9a1f7 HEAD@{1}: commit: Document corrected eligibility threshold
a70e5d2 HEAD@{2}: commit: Add unit test for threshold filter
f81d2a9 HEAD@{3}: commit: Update report figures and caption
```

`HEAD@{1}` is the lost tip. Recover cautiously by branching it first, verifying the contents, then fast-forwarding `main`:

```
$ git branch recovered HEAD@{1}
$ git log --oneline recovered -2
3c9a1f7 Document corrected eligibility threshold
a70e5d2 Add unit test for threshold filter
$ git reset --hard recovered
HEAD is now at 3c9a1f7 Document corrected eligibility threshold
$ git branch -d recovered
```

Both commits are back on `main`. In one session you located the commit that moved a headline result, established that the move was intended, and recovered a colleague's work that a mistaken reset had displaced. The forensic commands found the cause; the reflog undid the accident.

Draft, November 13, 2026. Content may change.

3.4. Homework

Work in a scratch repository with a history long enough to search. The following seeds one where a computed value changes at a known commit, so you can rediscover it:

```
mkdir bisect-practice
cd bisect-practice
git init
printf 'threshold <- 21\nx <- c(19, 20, 22, 25, 30)\ncat(mean(x[x >= th
git add compute.R
git commit -m 'Add computation with threshold 21'
printf 'first\n' > notes.txt && git add notes.txt && git commit -m 'Add
printf 'second\n' >> notes.txt && git commit -am 'Extend notes'
sed 's/threshold <- 21/threshold <- 18/' compute.R > tmp && mv tmp comp
git commit -am 'Lower threshold to 18'
printf 'third\n' >> notes.txt && git commit -am 'Extend notes again'
```

Running `Rscript compute.R` prints `25.75` at the tip and `25.66667` before the threshold change. The first commit is the known-good state.

1. **Read your own history.** Run `git reflog` and identify the entry for the very first commit and the entry for the current tip. What selector (`HEAD@{n}`) names each, and what does the label on the tip entry say?

2. **Undo a reset with the reflog.** Note the current tip hash. Run `git reset --hard HEAD~2`, confirm with `git log --oneline` that two commits are gone, then use the reflog to restore the branch to its previous tip. Show the reflog line you used.

3. **Recover without moving the branch.** Repeat the reset from Problem 2, but this time recover by creating a branch at the lost tip rather than by moving `main`. Confirm the lost commits are reachable from the new branch while `main` stays reset.

4. **Bisect by hand.** With the branch whole again, use `git bisect` to find the commit where `Rscript compute.R` first printed `25.75`. Mark the initial commit good and the tip bad, test each midpoint, and report the first bad commit. Finish with `git bisect reset`.

Draft, November 13, 2026. Content may change.

5. **Bisect automatically.** Write a check script that sources or runs `compute.R` and exits `0` when the printed value is `25.66667` and non-zero otherwise. Use `git bisect run` to locate the same commit as in Problem 4 without testing any midpoint by hand.

6. **Blame and the pickaxe.** Use `git blame` to find the commit that last set the `threshold` line, then use `git log -S '18'` (or a more specific string) to find the commit that introduced the value 18. Do the two commands report the same commit? Explain briefly why they must.

7. **Follow a line through time.** Use `git log -L 1,1:compute.R` to show every change to the first line of `compute.R`. How many commits appear, and what does each diff show? Contrast this with what `git blame -L 1,1 compute.R` alone would have told you.

8. **The limit of recovery.** Make an uncommitted edit to `compute.R`, then run `git reset --hard HEAD`. Attempt to recover the edit through the reflog. Can you? Explain in one or two sentences why the reflog does or does not help, connecting your answer to Day 1's warning about `git restore`.

3.5. Solutions

Problem 1.

```
git reflog
#> e1a2b3c (HEAD -> main) HEAD@{0}: commit: Extend notes again
#> ...
#> 9f0d1e2 HEAD@{5}: commit (initial): Add computation with threshol
```

The tip is named `HEAD@{0}` and its label reads `commit: Extend notes again`. The first commit is named by the highest-numbered selector, here `HEAD@{5}`, and its label is `commit (initial):`, the form Git uses for the first commit in a repository. The exact number depends on how many moves your reflog holds.

Problem 2.

Draft, November 13, 2026. Content may change.

```
git log --oneline -1
#> e1a2b3c Extend notes again
git reset --hard HEAD~2
git log --oneline -1
#> 7c3d4e5 Lower threshold to 18
git reflog
#> 7c3d4e5 (HEAD -> main) HEAD@{0}: reset: moving to HEAD~2
#> e1a2b3c HEAD@{1}: commit: Extend notes again
git reset --hard HEAD@{1}
git log --oneline -1
#> e1a2b3c Extend notes again
```

`HEAD@{1}` names the tip as it stood immediately before the reset. `git reset --hard HEAD@{1}` moves `main` back to it, undoing the reset. The commits were never destroyed; only the branch pointer had moved.

Problem 3.

```
git reset --hard HEAD~2
git branch rescue HEAD@{1}
git log --oneline main -1
#> 7c3d4e5 Lower threshold to 18
git log --oneline rescue -1
#> e1a2b3c Extend notes again
```

`git branch rescue HEAD@{1}` creates a pointer at the lost tip without touching `main`. The two 'lost' commits are now reachable from `rescue`, while `main` remains where the reset left it. This is the cautious recovery: nothing on `main` changes until you have inspected `rescue` and decided. Restore `main` afterwards with `git reset --hard rescue` if desired.

Problem 4.

```
git bisect start
git bisect bad
git bisect good 9f0d1e2      # the initial commit
#> Bisecting: ... [<midpoint>]
Rscript compute.R            # inspect the printed value
```


Draft, November 13, 2026. Content may change.

```
git bisect good              # if it printed 25.66667
# ... or ...
git bisect bad               # if it printed 25.75
#> 7c3d4e5 is the first bad commit
git bisect reset
```

The search converges on `Lower threshold to 18`, the commit that changed the printed value from `25.66667` to `25.75`. Marking each midpoint by the value it prints drives the binary search to that commit in two or three tests.

Problem 5.

```
# check.R
val <- as.numeric(system("Rscript compute.R", intern = TRUE))
q(status = if (abs(val - 25.66667) < 0.001) 0 else 1)
```

```
git bisect start
git bisect bad
git bisect good 9f0d1e2
git bisect run Rscript check.R
#> 7c3d4e5 is the first bad commit
git bisect reset
```

The script exits `0` while the value is still `25.66667` (good) and non-zero once it becomes `25.75` (bad). `git bisect run` reads those exit statuses as the verdicts and reaches the same commit as Problem 4 with no manual testing.

Problem 6.

```
git blame -L 1,1 compute.R
#> 7c3d4e5a (You 2026-07-25 ... 1) threshold <- 18
git log -S '18' --oneline -- compute.R
#> 7c3d4e5 Lower threshold to 18
```

Both report `7c3d4e5`. They must, because in this history the value 18 was introduced by exactly one commit, and that same commit is therefore the

last to have touched the line. Blame finds the most recent change to the line; the pickaxe finds the commit that changed the count of the string `18`; here they coincide. Had 18 been introduced, removed, and reintroduced, `git log -S` would list several commits while blame still showed only the most recent.

Problem 7.

```
git log -L 1,1:compute.R
#> commit 7c3d4e5...  Lower threshold to 18
#> -threshold <- 21
#> +threshold <- 18
#> commit 9f0d1e2...  Add computation with threshold 21
#> +threshold <- 21
```

Two commits appear. The first diff shows the line changing from 21 to 18; the second shows the line first being created with value 21. `git log -L` reconstructs the entire history of the line. `git blame -L 1,1` alone would have shown only `7c3d4e5`, the most recent change, hiding the fact that the line existed earlier with a different value. The line-log is the tool when you need the full provenance rather than the last edit.

Problem 8. You cannot recover it. `git reset --hard HEAD` overwrites the working directory with the committed version, and the uncommitted edit was never written to the object database, so no reflog entry names it; the reflog records commits and reference positions, not working-directory state. This is the identical loss Day 1 flagged for `git restore`: the safety net protects committed history, and the one thing it cannot catch is work that was never committed. Commit before any `--hard` operation and this class of loss cannot occur.

3.6. What's next

You have completed the bridge. Over three days you added selective staging and the stash to your daily workflow, and you made the rewriting of history a controlled and reversible tool rather than a hazard. Today you learned the reflog that made that rewriting safe all along, together with the forensic

Draft, November 13, 2026. Content may change.

commands that reconstruct how any value in an analysis came to be. With the boot camp behind you and this bridge complete, your preparation in version control proper is finished. You can compose clean history, revise it deliberately, recover from the accidents that revision invites, and investigate the provenance of a result down to the line and the commit.

3.6.1. Where this bridge hands off

The remaining subjects are no longer about Git the tool but about the infrastructure built around it, and they belong to the *Biostatistics Practicum*. The Practicum takes up, in order:

- Git internals: the object model of blobs, trees, and commits underlying everything in this book, and the plumbing commands that manipulate them.
- Hooks and continuous integration: pre-commit checks and GitHub Actions that run tests and validation automatically on every push.
- Containerized, dependency-pinned reproducibility: Docker and `renv` in depth, so that an analysis runs identically on any machine and in any year.
- Data and large-artifact versioning: Git LFS and DVC for the datasets and model outputs that do not belong in ordinary Git.
- Team-scale collaboration policy: branch protection, `CODEOWNERS`, review requirements, and the regulatory version-control practice of clinical-trial deliverables.

Each extends a skill this bridge established into the full reproducible pipeline that professional statistical work requires. Keep the `trial-analysis` repository, since the Practicum begins where Day 3 leaves it. In closing, carry the reflog with you. Whatever you attempt next, committed work is recoverable, and that assurance is what lets you work boldly.

Draft, November 13, 2026. Content may change.

References

Credits

The three-day scope was chosen to fill the specific gap between the companion boot camp and the *Biostatistics Practicum*: the version-control craft that a student needs within days of the first commit, but that belongs neither in a first week nor in an infrastructure course. The topic selection draws on Scott Chacon and Ben Straub's *Pro Git*, Jenny Bryan's *Happy Git and GitHub for the useR*, and the intermediate portions of the Software Carpentry version-control lessons.

Cover artwork generated procedurally by the R script at `images/build-cover.R`. Palette: deep slate through steel and pale mist, anchored on a brand steel blue `#2c4a5e` chosen to sit alongside, and one register cooler than, the boot camp's rust. Typography: Avenir family (Apple system font); body text in Source Serif 4.

Colophon

This book was produced with Quarto, typeset in Source Serif 4, with code blocks set in JetBrains Mono. The command examples target Git 2.40 or later and the GitHub CLI (`gh`) 2.40 or later.

The book is hosted at https://git-intermediate.rgtlab.org on Netlify, with continuous deployment via GitHub Actions from the `rgt47/git-intermediate` repository on every push to `main`. The deployment recipe is in `HOSTING.md`.

The cover is generated procedurally by `images/build-cover.R` from a slate gradient anchored on `#2c4a5e` (the volume's brand steel blue). Re-running the script regenerates the cover; the procedural approach makes it reproducible and editable in version control rather than dependent on a one-shot image-generation step.

Last rendered: see the build timestamp on the homepage.