Git and GitHub for Biostatistics

A One-Week Boot Camp for incoming graduate students

The rgtlab Curriculum Project

2026-11-13

Git and GitHub for Biostatistics

A One-Week Boot Camp

The rgtlab Curriculum Project

Welcome

This is the online version of **Git and GitHub for Biostatistics: A One-Week Boot Camp** by The rgtlab Curriculum Project. We designed it as a short preparatory course for incoming graduate students in biostatistics and adjacent programs.

We cover everything an entering masters-level biostatistician needs to place analytical work under version control on day one of the academic program. That is, we take up installing and configuring Git, recording a project's history in commits, navigating and undoing that history, branching and merging, publishing to GitHub, and collaborating through pull requests. The book is designed for **five consecutive days** of work: one hour of lecture content each day, two hours of homework, no examinations.

Our organizing principle is that version control is a research-integrity tool, not a software-engineering nicety. A biostatistician who cannot say which version of a script produced a published table has a reproducibility problem, not a tooling preference. Every chapter opens with a data-analysis problem, develops the Git mechanism that addresses it, and returns to the problem to judge what the mechanism did, and did not, solve.

The boot camp is a preparatory companion to a five-volume graduate sequence:

- **R for Biostatistics: A One-Week Boot Camp** (https://r-bootcamp.rgtlab.org), the parallel preparatory volume for R.
- *Biostatistics Practicum*, workflow infrastructure (Git in depth, Docker, renv, Quarto, CDISC) that builds directly on this boot camp.
- *Statistical Computing in the Age of AI*, introductory methods (linear models, GLM, mixed models, survival, Bayesian, bootstrap, simulation).

- *Advanced Statistical Computing in the Age of AI*, advanced computing (numerical stability, MCMC, HPC, high-dimensional methods, ML, software engineering).
- *Applied Generative AI for Health Sciences Research*, capability classes, RAG, agents, evaluation, regulation, deployment.

Together with *R for Biostatistics*, this volume forms the two-part pre-program preparation: one week of R, one week of version control. A student who completes both arrives at the *Practicum* ready to work in a reproducible pipeline from the first session.

The reader is referred to the Preface for the design rationale, and to the Conventions page for the visual cues used throughout.

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.

Git and GitHub for Biostatistics

A One-Week Boot Camp for incoming graduate students.

Draft, November 13, 2026. Content may change.

Copyright

Git and GitHub for Biostatistics: A One-Week 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). *Git and GitHub for Biostatistics: A One-Week Boot Camp*. Available at https://git-bootcamp.rgtlab.org.

Table of contents

License 2

Copyright **5**

Preface **11**
Why version control belongs in a biostatistics curriculum 11
What this book covers 12
What this book does not cover 13
How this book is meant to be used 13
Prerequisites 14
Acknowledgments 14

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

How to use this book **19**
The daily cadence 19
Chapter template 19
A note on the two tools 20
Conventions 20
Companion volumes 21

I. Five Days **23**

1. Day 1: Why Version Control, Setup, First Repository **25**
1.1. Learning objectives 25

1.2. Lecture 25
1.2.1. What problem does version control solve? 25
1.2.2. Installing Git 27
1.2.3. A brief command-line refresher 27
1.2.4. Configuring your identity 28
1.2.5. The three areas 29
1.2.6. Creating your first repository 29
1.2.7. Making your first commit 30
1.2.8. The add-commit cycle 32
1.2.9. Reading history with git log 33
1.2.10. Ignoring files with .gitignore 34
1.3. Worked example: versioning a small analysis from scratch . 35
1.4. Homework 37
1.5. Solutions 38
1.6. What's next 41

2. Day 2: Recording History: Commits, Diffs, and Undoing 43
2.1. Learning objectives 43
2.2. Lecture 43
2.2.1. Comparing versions with git diff 44
2.2.2. Comparing commits 45
2.2.3. Inspecting a single commit with git show 46
2.2.4. Reading history in depth 47
2.2.5. Referring to commits 48
2.2.6. The anatomy of a good commit 49
2.2.7. Undoing changes safely 51
2.3. Worked example: catching and fixing an analytical mistake 55
2.4. Homework 58
2.5. Solutions 59
2.6. What's next 64

3. Day 3: Branching and Merging 65
3.1. Learning objectives 65
3.2. Lecture 65
3.2.1. What a branch is 65
3.2.2. Listing, creating, and switching branches 67
3.2.3. How HEAD moves and what happens to your files . 68
3.2.4. Deleting a branch 69

Draft, November 13, 2026. Content may change.

3.2.5. Merging: fast-forward and three-way 70
3.2.6. The feature-branch workflow 72
3.2.7. Merge conflicts 73
3.3. Worked example: a subgroup analysis on a feature branch . 75
3.4. Homework 78
3.5. Solutions 79
3.6. What's next 85

4. Day 4: GitHub: Remotes and Publishing 87
4.1. Learning objectives 87
4.2. Lecture 87
4.2.1. What GitHub is, and what it is not 87
4.2.2. What a remote is 88
4.2.3. Installing the GitHub CLI 89
4.2.4. Authenticating with gh auth login 90
4.2.5. Publishing a local repository in one step 91
4.2.6. Public, private, and protected health information . . 92
4.2.7. The remote mechanics gh sets up 92
4.2.8. Pushing later commits 93
4.2.9. Getting changes: fetch and pull 94
4.2.10. Cloning an existing repository 95
4.2.11. The README as the front page 96
4.2.12. Browsing a repository from the terminal 97
4.2.13. Issues as a tracked to-do list 98
4.3. Worked example: publishing the baseline-summary repository 99
4.4. Homework 102
4.5. Solutions 103
4.6. What's next 108

5. Day 5: Collaboration and Reproducible Research 109
5.1. Learning objectives 109
5.2. Lecture 109
5.2.1. The collaboration problem 109
5.2.2. Two collaboration models 110
5.2.3. The pull-request workflow with gh 111
5.2.4. Keeping your local main current 115
5.2.5. Forking and syncing a fork 115
5.2.6. When your push is rejected 116

Draft, November 13, 2026. Content may change.

5.2.7. Git through the RStudio IDE 118
5.2.8. A .gitignore for R projects 120
5.2.9. Best-practices recap 121
5.3. Worked example: a full collaborative cycle 122
5.4. Homework 125
5.5. Solutions 126
5.6. What's next 130

References **133**

Appendices **135**

Credits **135**

Colophon **137**

Draft, November 13, 2026. Content may change.

Preface

This volume is a one-week boot camp for incoming graduate students in biostatistics who arrive with no or minimal prior exposure to version control. It is the second half of the pre-program preparation. The companion volume *R for Biostatistics* covers the R language over an equivalent week. Where that book makes a student functional in R, this one makes a student's work reproducible and shareable. Both are prerequisites that every subsequent volume in the sequence assumes.

The design constraint is severe: five lectures, ten hours of homework, and the student arrives at the academic program able to place any analysis under version control and collaborate on GitHub. We meet the constraint through ruthless prioritization. Git is a large and idiosyncratic tool, and we omit the great majority of its surface area deliberately. What remains is the irreducible core that a practicing biostatistician uses every day.

Why version control belongs in a biostatistics curriculum

The case for teaching version control to statisticians is not a case about software engineering. It is a case about research integrity. We consider three failures that every data analyst has seen or committed:

- A results table in a manuscript cannot be reproduced because the script was edited after the table was made, and no record of the earlier version survives.
- A folder contains `analysis.R`, `analysis_final.R`, `analysis_final_v2.R`, and `analysis_FINAL_use_this.R`, and nobody can say which produced the submitted figures.

- Two collaborators email a script back and forth, each edits it, and their changes silently overwrite one another.

Each of these is a version-control failure, and each threatens the reproducibility that is the ethical foundation of quantitative science. Git addresses all three directly. It records every version of every file with an author, a timestamp, and a message. It makes the history navigable. And it merges concurrent edits rather than losing them. A biostatistician who internalizes this treats the commit history as a lab notebook, which is precisely what it is.

What this book covers

Five days, one chapter each:

1. **Why version control, setup, first repository.** What version control is and why it is a reproducibility tool; installing and configuring Git; the working directory, staging area, and repository; `git init`, `git status`, `git add`, `git commit`, `git log`; the `.gitignore` file.
2. **Recording history: commits, diffs, undoing.** The commit as a snapshot; `git diff` and `git show`; reading and navigating history; writing commit messages that earn their place; undoing safely with `git restore`, `git revert`, and `git reset`; amending.
3. **Branching and merging.** Branches as movable pointers; `git branch`, `git switch`, `git merge`; the feature-branch workflow; merge conflicts and how to resolve them without fear.
4. **GitHub: remotes and publishing.** What GitHub is; authenticating with the `gh` command-line tool; `git remote`, `git push`, `git pull`, `git fetch`, `git clone`; creating repositories with `gh repo create`; READMEs and issues.
5. **Collaboration and reproducible research.** The fork-and-pull-request workflow; opening and reviewing pull requests with `gh pr`; resolving remote conflicts; using Git through the RStudio IDE; `.gitignore` for R projects, `renv`, and data; the handoff to the *Biostatistics Practicum* volume.

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

Draft, November 13, 2026. Content may change.

What this book does not cover

We deliberately omit, and point elsewhere for, nearly everything beyond the daily core:

- Git internals (the object database, the reflog, plumbing commands), interactive rebasing, cherry-picking, submodules, and hooks. See the *Biostatistics Practicum*.
- Continuous integration, automated testing, and GitHub Actions. See the *Practicum* and *Advanced Statistical Computing in the Age of AI*.
- Containerised and dependency-pinned reproducibility (Docker, `renv` in depth). See the *Practicum*.
- Regulatory version-control practice for clinical trials (CDISC deliverables, validated pipelines). See the *Practicum* and *Applied Statistical Methods for Health Sciences Research*.

The boot camp is intentionally narrow. The student finishes Day 5 able to work fluently in the everyday Git workflow. This leaves the *Practicum* free to teach the infrastructure that surrounds it.

How this book is meant to be used

The five-day cadence is the spine of the book. A student starting the week having never made a commit can finish the week collaborating on GitHub. The cadence assumes roughly three hours of focused work per day:

- **Hour 1:** Read the chapter, running every command in your own terminal as you reach it. Git is learned by typing, not by reading.
- **Hours 2-3:** Complete the homework problems. Solutions are at the end of each chapter; check yourself only after attempting each problem.

We show the commands in this book for a Unix-style shell (macOS Terminal, Linux, or Git Bash on Windows). Where a step differs on Windows, we note the difference.

Prerequisites

We assume:

- A working laptop running macOS, Windows, or Linux.
- Basic familiarity with the command line: opening a terminal, changing directories with `cd`, listing files with `ls`. A short refresher is provided on Day 1.
- No prior exposure to Git, GitHub, or any version-control system.

Prior completion of *R for Biostatistics* is helpful, since the worked examples version R analysis scripts. It is not strictly required, however. The Git mechanics are independent of the language of the files under version control.

Acknowledgments

This boot camp consolidates patterns developed over two decades of teaching reproducible workflows to incoming biostatistics students. Jenny Bryan's *Happy Git and GitHub for the useR* and the Software Carpentry version-control lessons provide the technical scaffolding. The boot-camp pedagogy itself reflects experience with what incoming students actually need in their first week, as distinct from the full capabilities of the tool.

Conventions

This page summarizes the visual conventions used throughout the book.

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 analysis.R
git commit -m 'Add baseline analysis'
```

Inline commands, file names, and Git objects are set in `monospace`. Git subcommands are named in full (`git commit`, not merely `commit`) so that they are unambiguous. GitHub operations use the official `gh` command-line tool, shown the same way:

```
$ gh repo view --web
Opening github.com/rgt47/trial-analysis in your browser.
```

Placeholders that the reader must replace with their own values are written in angle brackets, for example `git clone <url>` or `gh repo create <name>`. 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, most often an operation that discards work 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 and tag names are pointers attached to commits:

```
        main
         |
A -- B -- C
```

Here `C` is the most recent commit and `main` points to it. These diagrams are conceptual; the real hashes are the forty-character identifiers Git assigns.

Cross-references

Within this book, sections are referenced by their Quarto label (`@sec-day1`, `@sec-merge-conflicts`). These resolve to clickable links in HTML and

Draft, November 13, 2026. Content may change.

section numbers in PDF.

References to the companion volumes *R for Biostatistics* and *Biostatistics Practicum* use **prose pointers** rather than Quarto cross-references, because cross-references do not resolve across separate books. For example: 'see the Docker chapter of the companion *Biostatistics Practicum* volume'.

Chapter structure

Every content chapter follows the same five-section template:

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 as you read.
3. **Worked example.** A small but 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. The pattern repeats deliberately; by the third chapter you know where to find each component.

How to use this book

We begin with a short orientation. In this chapter we set out the daily cadence, the chapter template, and the conventions used throughout, so that the reader knows what to expect.

The daily cadence

Each day has the same shape:

- **Lecture content (~1 hour).** Read the chapter, running each command in your own terminal as you encounter it. Do not skip the commands; the chapter is designed to be read with a terminal open and a scratch repository at hand.
- **Homework (~2 hours).** Five to eight problems, with worked solutions at the end of the chapter. Attempt each problem before checking the solution.

The cadence assumes one chapter per day for five consecutive days. The boot camp may be compressed (two chapters per day across two-and-a-half days) or expanded (one chapter per week across a month) depending on the student's other commitments; we leave the choice to the reader.

Chapter template

Each chapter follows a five-section template, and we shall keep to it consistently:

1. **Learning objectives.** A bulleted list of what the student should be able to do by the end of the day.

Draft, November 13, 2026. Content may change.

2. **Lecture.** The substantive content for the day, with commands to run.
3. **Worked example.** A small but realistic biostatistical scenario that uses the day's content end to end, most often placing an R analysis script under version control.
4. **Homework.** Problems organized from easier to harder.
5. **Solutions.** Worked solutions to all homework problems. Read only after attempting the problem.

There are no quizzes, no examinations, and no tests in this book. Self-assessment proceeds entirely through the homework solutions.

A note on the two tools

We teach two related but distinct tools:

- **Git** is the version-control system. It runs on your own machine and records the history of your files. Git is the subject of Days 1 through 3.
- **GitHub** is a website that hosts Git repositories and adds collaboration features (issues, pull requests, review). GitHub is not Git; it is one of several services that host Git repositories. GitHub is the subject of Days 4 and 5, operated from the terminal with the official `gh` command-line tool.

Keeping the distinction clear from the outset prevents a common beginner confusion in which the two are treated as a single thing.

Conventions

The visual cues used throughout the book are described on the Conventions page. In brief:

- Commands you type follow a `$` prompt; output has no prompt.
- Placeholders you replace are written in `<angle-brackets>`.
- GitHub operations use the `gh` command-line tool.
- Command lines and file listings are wrapped at 78 characters.

Draft, November 13, 2026. Content may change.

Companion volumes

This boot camp is the version-control half of a two-part preparation. The R half is *R for Biostatistics*. After both, the reader may continue with the four follow-on volumes:

- *Biostatistics Practicum* (https://rgtlab.org/practicum)
- *Statistical Computing in the Age of AI* (https://scai.rgtlab.org)
- *Advanced Statistical Computing in the Age of AI* (https://scai-advanced.rgtlab.org)
- *Applied Generative AI for Public Health and Biostatistics* (https://applied-genai.rgtlab.org)

The end of Chapter 5 describes how the *Practicum* picks up the version-control thread and extends it into a full reproducible pipeline.

Part I.

Five Days

Draft, November 13, 2026. Content may change.

1. Day 1: Why Version Control, Setup, First Repository

1.1. Learning objectives

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

- Explain what version control is and why it is a research-integrity tool rather than a software-engineering convenience.
- Install Git on macOS, Windows, or Linux and confirm the installation.
- Configure your Git identity (name, email, default branch, editor).
- Describe Git's three areas: the working directory, the staging area (index), and the repository.
- Create a repository with `git init` and record snapshots with `git add` and `git commit`.
- Read the state of a repository with `git status` and its history with `git log`.
- Exclude files from version control with a `.gitignore` file.

1.2. Lecture

1.2.1. What problem does version control solve?

Every data analyst has lived some version of this folder:

```
project/
  analysis.R
  analysis_final.R
  analysis_final_v2.R
```


Draft, November 13, 2026. Content may change.

```
  analysis_FINAL_use_this.R
  analysis_reviewer_comments.R
```

The folder is a manual, error-prone version-control system. It records some history, since there are several versions, but it records it badly. The ordering is ambiguous, the reason for each version is lost, and nothing connects a given file to the results it produced. When a reviewer asks 'which script made Table 2?', the honest answer is often 'I am not sure'.

A **version-control system** (VCS) replaces this with a disciplined record. It keeps one file, `analysis.R`, and stores every historical version of it internally, each stamped with an author, a time, and a message explaining the change. You can view any past version, compare any two, and recover any of them exactly. The folder above collapses to:

```
project/
  analysis.R      # plus a complete, queryable history
```

Git is the version-control system we teach in this book. It is the de facto standard in data science, statistics, and software. For a biostatistician the payoff is specific and serious: version control is what makes an analysis reproducible over time. If a manuscript's results were produced by commit `a1b2c3d`, you can return to that exact commit years later and rerun it. That is a claim about research integrity, not about convenience.

Check your understanding: Git versus GitHub

You may have heard 'Git' and 'GitHub' used interchangeably. Are they the same thing?
No. **Git** is the version-control software that runs on your own computer and records history locally. It needs no internet connection and no account. **GitHub** is a commercial website that hosts Git repositories and adds collaboration tools (issues, pull requests, code review). GitHub is one of several hosts for Git repositories; alternatives include GitLab and Bitbucket. Days 1 through 3 use only Git, entirely offline. GitHub arrives on Day 4.

Draft, November 13, 2026. Content may change.

1.2.2. Installing Git

Git is a command-line program. Install it as follows, then confirm the installation in a terminal.

macOS. The simplest route is to install the Xcode command-line tools, which include Git:

```
xcode-select --install
```

Alternatively, if you use Homebrew, `brew install git` gives a newer version.

Windows. Download and run the installer from https://git-scm.com/download/win. This installs both Git and **Git Bash**, a Unix-style terminal. Use Git Bash for every command in this book; the commands are written for a Unix shell and will not all work in the classic Windows Command Prompt.

Linux. Use the system package manager, for example `sudo apt install git` on Debian or Ubuntu, or `sudo dnf install git` on Fedora.

Confirm the installation in a terminal:

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

Any version 2.40 or later is suitable for this book. If the command is not found, the installation did not complete; revisit the step for your operating system.

1.2.3. A brief command-line refresher

Git is driven from a terminal. Four commands cover almost everything you need this week:

```
pwd                 # print working directory (where am I?)
ls                  # list files in the current directory
cd project          # change into the 'project' directory
cd ..               # go up one directory
```

If these are unfamiliar, spend ten minutes navigating your filesystem with them before continuing. Git operates on 'the repository in the current directory', so knowing where you are is a prerequisite.

1.2.4. Configuring your identity

Git stamps every commit with an author name and email. Configure these once, globally, before your first commit:

```
git config --global user.name 'Jane Investigator'
git config --global user.email 'jane@university.edu'
```

Use the email you will later associate with your GitHub account (Day 4), so that your commits are attributed to you there. Two further settings are worth applying now. Set the default branch name to `main`, the modern convention:

```
git config --global init.defaultBranch main
```

And set the editor Git opens when it needs a message from you. If you are comfortable in a terminal editor, `nano` is the gentlest choice:

```
git config --global core.editor nano
```

Confirm your configuration at any time:

```
$ git config --global --list
user.name=Jane Investigator
user.email=jane@university.edu
init.defaultBranch=main
core.editor=nano
```


Draft, November 13, 2026. Content may change.

> **Tip**
>
> The `--global` flag writes these settings to your home directory (`~/.gitconfig`), so they apply to every repository on your machine. You configure your identity once, not once per project.

1.2.5. The three areas

Git's mental model has three areas. Understanding them is the single most important concept of Day 1, because every later command manipulates the boundaries between them.

```
working directory  --->  staging area  --->  repository
    (your files)         (the index)        (committed
                                               history)
     git add ---------------^                    ^
     git commit ----------------------------------
```

- The **working directory** is the folder of files you edit directly. It holds exactly one version of each file: the current one.
- The **staging area** (also called the **index**) is a holding zone. You place a snapshot of specific changes here with `git add`, declaring 'these changes belong in my next commit'.
- The **repository** is the permanent, versioned history. `git commit` takes whatever is in the staging area and records it as a new, immutable snapshot.

The staging area is what distinguishes Git from a simple 'save a version' button. It lets you commit some of your changes and not others, so that each commit is a single coherent unit rather than a jumble of unrelated edits. Day 2 takes up this feature deliberately.

1.2.6. Creating your first repository

Create a project folder, enter it, and turn it into a repository:

Draft, November 13, 2026. Content may change.

```
$ mkdir trial-analysis
$ cd trial-analysis
$ git init
Initialized empty Git repository in
/Users/jane/trial-analysis/.git/
```

`git init` creates a hidden subdirectory named `.git` that holds the entire history. The rest of the folder is your ordinary working directory. Deleting `.git` would revert the folder to an unversioned pile of files; everything Git knows lives inside it.

Check the state of the new repository:

```
$ git status
On branch main

No commits yet

nothing to commit (working tree clean)
```

Read `git status` output often; it is Git's way of telling you exactly where you are and what it will do next. Right now it reports an empty repository on the `main` branch.

1.2.7. Making your first commit

Create a small analysis script. You can use any editor; here we write a two-line R script:

```
$ echo 'dat <- read.csv("enrollment.csv")' > analysis.R
$ echo 'summary(dat$age)' >> analysis.R
```

Now `git status` notices the new file:

Draft, November 13, 2026. Content may change.

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be
   committed)
    analysis.R

nothing added to commit but untracked files present
```

The file is **untracked**: Git sees it but is not yet recording its history. Move it into the staging area with `git add`:

```
$ git add analysis.R
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   analysis.R
```

The file is now **staged**. Record it as a commit, with a message describing the change:

```
$ git commit -m 'Add baseline enrollment analysis'
[main (root-commit) 8f3a1a2] Add baseline enrollment analysis
 1 file changed, 2 insertions(+)
 create mode 100644 analysis.R
```

You have made your first commit. The short string `8f3a1a2` is the beginning of the commit's forty-character identifier, its **hash**, which names this snapshot uniquely and permanently. Confirm the clean state:

Draft, November 13, 2026. Content may change.

```
$ git status
On branch main
nothing to commit, working tree clean
```

'Working tree clean' means the working directory matches the latest commit exactly: there is nothing uncommitted. This is the state you want to be in whenever you stop work.

1.2.8. The add-commit cycle

Daily Git work is a loop: edit files, stage the changes you want, commit them with a message, repeat. Add a second line of analysis:

```
$ echo 'sd(dat$age, na.rm = TRUE)' >> analysis.R
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes)
    modified:   analysis.R

no changes added to commit but "git restore" present
```

Now the file is **modified** and **tracked** but the change is unstaged. Stage and commit it:

```
$ git add analysis.R
$ git commit -m 'Add age standard deviation'
[main 2b4c6d8] Add age standard deviation
 1 file changed, 1 insertion(+)
```

You can shorten the stage-and-commit of already-tracked files with `git commit -a`, which stages every tracked file that has changed:

Draft, November 13, 2026. Content may change.

```
$ echo 'table(dat$arm)' >> analysis.R
$ git commit -a -m 'Tabulate treatment arm'
[main 3c5d7e9] Tabulate treatment arm
 1 file changed, 1 insertion(+)
```

Note that `-a` stages only files Git already tracks; a brand-new file still needs an explicit `git add`.

1.2.9. Reading history with git log

`git log` shows the commits, newest first:

```
$ git log
commit 3c5d7e9... (HEAD -> main)
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 10:15:42 2026 -0700

    Tabulate treatment arm

commit 2b4c6d8...
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 10:12:03 2026 -0700

    Add age standard deviation

commit 8f3a1a2...
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 10:08:55 2026 -0700

    Add baseline enrollment analysis
```

The one-line-per-commit view is more useful day to day:

```
$ git log --oneline
3c5d7e9 Tabulate treatment arm
2b4c6d8 Add age standard deviation
8f3a1a2 Add baseline enrollment analysis
```


Draft, November 13, 2026. Content may change.

`HEAD` is Git's name for 'the commit you are currently on', almost always the tip of your current branch. Here `HEAD` points to `main`, which points to the newest commit. Day 2 reads this history in far more detail; for now, note that you have a complete, ordered, annotated record of how `analysis.R` came to be, which the four-file folder from the start of the lecture could never provide.

1.2.10. Ignoring files with .gitignore

Not everything in a project folder belongs under version control. Rendered output, cached files, and above all raw data with protected health information should be excluded. Git reads a file named `.gitignore` and refuses to track anything matching its patterns.

Suppose your analysis produces `report.html` and R writes a `.Rhistory` file, and your data lives in a `data-raw/` folder you must not commit. Create a `.gitignore`:

```
$ echo 'report.html' > .gitignore
$ echo '.Rhistory' >> .gitignore
$ echo 'data-raw/' >> .gitignore
```

Now those paths vanish from `git status`:

```
$ git status
On branch main
Untracked files:
    .gitignore

nothing added to commit but untracked files present
```

Git no longer mentions `report.html` or `data-raw/`. The `.gitignore` file itself, however, should be committed, so that everyone working on the project shares the same exclusions:

Draft, November 13, 2026. Content may change.

```
$ git add .gitignore
$ git commit -m 'Add gitignore for output and raw data'
[main 4d6e8fa] Add gitignore for output and raw data
 1 file changed, 3 insertions(+)
```

Warning

`.gitignore` prevents Git from tracking a file only if the file is not already tracked. If you have already committed a file, adding it to `.gitignore` does nothing; you must also stop tracking it with `git rm --cached <file>`. The lesson: write your `.gitignore` early, ideally as the first commit, before any data or output has been staged. Committing protected health information and only later discovering it in the history is a serious and common accident.

Check your understanding: staged versus committed

You run `git add results.csv` and then, before committing, edit `results.csv` again. What will `git commit` record: the version you staged, or the version now in your working directory?
Git records the version you **staged**. `git add` copies a snapshot of the file into the staging area at the moment you run it. Editing the file afterwards changes the working directory but not the staged snapshot, so `git status` will now show `results.csv` as both 'staged' (the first edit) and 'modified' (the second edit). To include the newer edit, run `git add results.csv` again before committing. This is a frequent source of 'but I changed that' confusion, and it follows directly from the three-area model.

1.3. Worked example: versioning a small analysis from scratch

You are handed a CSV of trial enrollment and asked to produce a baseline summary. We place the work under version control from the very first step, which is the habit to build.

Draft, November 13, 2026. Content may change.

Create and initialize the project:

```
$ mkdir baseline-summary
$ cd baseline-summary
$ git init
Initialized empty Git repository in
/Users/jane/baseline-summary/.git/
```

Before writing any analysis, protect the data and output by committing a `.gitignore` first:

```
$ printf 'data-raw/\n*.html\n.Rhistory\n.RData\n' > .gitignore
$ git add .gitignore
$ git commit -m 'Add gitignore before any data enters repo'
[main (root-commit) a1b2c3d] Add gitignore before any data
 1 file changed, 4 insertions(+)
```

Write the first version of the analysis and commit it:

```
$ cat > baseline.R <<'EOF'
dat <- read.csv("data-raw/enrollment.csv")
n_total <- nrow(dat)
mean_age <- mean(dat$age, na.rm = TRUE)
EOF
$ git add baseline.R
$ git commit -m 'Add total count and mean age'
[main 7e8f9a0] Add total count and mean age
 1 file changed, 3 insertions(+)
```

Extend the analysis with a sex breakdown, and commit that as a separate, self-contained change:

```
$ echo 'sex_table <- table(dat$sex)' >> baseline.R
$ git add baseline.R
$ git commit -m 'Add sex distribution table'
[main 9b0c1d2] Add sex distribution table
 1 file changed, 1 insertion(+)
```


Draft, November 13, 2026. Content may change.

Inspect the result:

```
$ git log --oneline
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
$ git status
On branch main
nothing to commit, working tree clean
```

In five commits' worth of habit you have built something the four-file folder never could. That is, a data file that was never at risk of being committed, an analysis whose every step is dated and explained, and a clean working tree you can walk away from. This is the baseline rhythm of every subsequent day.

1.4. Homework

Attempt each problem in your own terminal before checking the solution. Create a scratch directory to work in so that you can experiment freely.

1. **Confirm your setup.** Run the commands that print your Git version and your global configuration. Confirm that `user.name`, `user.email`, and `init.defaultBranch` are all set. If any is missing, set it.

2. **A repository from nothing.** Create a new directory `hw1-repo`, initialize it as a Git repository, and confirm with `git status` that you are on branch `main` with no commits yet. What does the `.git` directory contain? (List it with `ls -a` and `ls .git`.)

3. **Three commits.** In `hw1-repo`, create a file `notes.txt` containing the single line `Study protocol v1`. Stage and commit it. Then append a second line `Primary endpoint: 12-week change in HbA1c` and commit that. Then append a third line and commit it. Show the resulting history with `git log --oneline`. How many commits are there, and in what order are they listed?

Draft, November 13, 2026. Content may change.

4. **Staged versus modified.** In `hw1-repo`, edit `notes.txt` and run `git add notes.txt`. Before committing, edit `notes.txt` again. Run `git status` and read it carefully. Explain, in your own words, why the same file appears under two different headings.
5. **Write a .gitignore.** In a new directory, create a `.gitignore` that excludes: any file ending in `.html`, any file ending in `.pdf`, a directory named `data-raw/`, and the file `.RData`. Create some files matching those patterns and confirm with `git status` that Git ignores them. Which one file related to ignoring should you commit, and why?
6. **The already-tracked trap.** Create a file `secret.csv`, commit it, and then add `secret.csv` to your `.gitignore`. Run `git status`. Is `secret.csv` ignored now? Explain what you observe and state the command that would actually stop Git from tracking it.

1.5. Solutions

Problem 1.

```
git --version
git config --global --list
```

If, say, `user.email` is absent, set it with `git config --global user.email 'you@example.edu'`. The `--global --list` output should show `user.name`, `user.email`, and `init.defaultBranch=main` at a minimum.

Problem 2.

```
mkdir hw1-repo
cd hw1-repo
git init
git status
#> On branch main
#>
#> No commits yet
#>
```


Draft, November 13, 2026. Content may change.

```
#> nothing to commit (working tree clean)
ls -a
#> .  ..  .git
ls .git
#> HEAD  config  description  hooks  info  objects  refs
```

The `.git` directory holds the entire repository: the `objects` store (every version of every file and every commit), the `refs` (branch and tag pointers), `HEAD` (the pointer to your current position), and `config` (the repository's local settings). You will rarely look inside it, but it is worth seeing once so that 'the repository' is a concrete thing rather than an abstraction.

Problem 3.

```
echo 'Study protocol v1' > notes.txt
git add notes.txt
git commit -m 'Add protocol version line'

echo 'Primary endpoint: 12-week change in HbA1c' >> notes.txt
git commit -a -m 'Record primary endpoint'

echo 'Randomization: 1:1, stratified by site' >> notes.txt
git commit -a -m 'Record randomization scheme'

git log --oneline
#> c3d4e5f Record randomization scheme
#> b2c3d4e Record primary endpoint
#> a1b2c3d Add protocol version line
```

There are three commits. `git log` lists them **newest first**, so the most recent commit is at the top. (Your hashes will differ from those shown; hashes are computed from the content and metadata of each commit and are effectively unique.)

Problem 4. After the second edit, `git status` shows `notes.txt` under both 'Changes to be committed' (the staged first edit) and 'Changes not staged for commit' (the second edit, still only in the working directory). This happens because `git add` snapshots the file at the instant it runs. The

Draft, November 13, 2026. Content may change.

first edit was snapshotted into the staging area; the second edit changed the working directory but not that snapshot. A commit now would record only the first edit. Running `git add notes.txt` again would fold the second edit into the staging area as well.

Problem 5.

```
mkdir hw5-repo
cd hw5-repo
git init
cat > .gitignore <<'EOF'
*.html
*.pdf
data-raw/
.RData
EOF
touch report.html manuscript.pdf .RData
mkdir data-raw
touch data-raw/phi.csv
git status
#> On branch main
#>
#> No commits yet
#>
#> Untracked files:
#>   .gitignore
#>
#> nothing added to commit but untracked files present
```

Only `.gitignore` appears; the four ignored targets do not. You should commit `.gitignore` itself. Committing it shares the exclusion rules with every collaborator and with your future self, so that the same files are ignored everywhere the repository is cloned. An uncommitted `.gitignore` protects only your own machine.

Problem 6.

Draft, November 13, 2026. Content may change.

```
echo 'id,value' > secret.csv
git add secret.csv
git commit -m 'Add data file (mistake)'
echo 'secret.csv' >> .gitignore
git status
#> On branch main
#> nothing to commit, working tree clean
```

`secret.csv` is **not** ignored. `.gitignore` governs only **untracked** files; once a file has been committed, Git continues to track it regardless of `.gitignore`. To stop tracking it while keeping the local copy:

```
git rm --cached secret.csv
git commit -m 'Stop tracking secret.csv'
```

`git rm --cached` removes the file from Git's tracking (and from future commits) but leaves it in your working directory. Note that the file still exists in the earlier commit; truly purging sensitive data from history is a harder problem, covered in the *Practicum*. The practical lesson is the one from the lecture: write `.gitignore` first.

1.6. What's next

Day 2 goes inside the commit. You will compare versions with `git diff`, inspect any past snapshot with `git show`, navigate and search the history, learn what separates a useful commit message from a useless one, and, most importantly, learn to undo changes safely with `git restore`, `git revert`, and `git reset`. Keep the `baseline-summary` repository from the worked example; Day 2 builds on it.

2. Day 2: Recording History: Commits, Diffs, and Undoing

2.1. Learning objectives

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

- Compare versions of your files with `git diff`, both before and after staging.
- Compare any two commits, and inspect a single commit in full with `git show`.
- Read history in depth with `git log` and its most useful options, including the history of a single file.
- Refer to any commit by its hash or by a relative name such as `HEAD~1`.
- Write a commit that records a single conceptual change, with a message that explains why the change was made.
- Undo changes safely, distinguishing the operations that preserve your work from the few that destroy it.

2.2. Lecture

Day 1 built the add-commit habit: edit, stage, commit, repeat. Day 2 turns to the questions that the habit makes possible. What exactly changed between two versions? What did a past commit contain? How do we write a commit worth committing? And, above all, how do we undo a mistake without losing work? The last question is the heart of the day, and it is the reason version control repays the effort of learning it.

Draft, November 13, 2026. Content may change.

2.2.1. Comparing versions with git diff

A commit records a snapshot, but the useful unit of thought is usually the difference between two snapshots. `git diff` computes that difference and prints it as a set of lines added and removed.

With no arguments, `git diff` compares your working directory against the staging area, that is, it shows the changes you have made but not yet staged. Suppose `baseline.R` computes a mean age, and you edit the line to add a median:

```
$ git diff
diff --git a/baseline.R b/baseline.R
index 3a1b2c3..4d5e6f7 100644
--- a/baseline.R
+++ b/baseline.R
@@ -2,3 +2,4 @@ dat <- read.csv("data-raw/enrollment.csv")
 n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
+med_age <- median(dat$age, na.rm = TRUE)
```

Read this from the bottom. A line beginning with `+` was added; a line beginning with `-` was removed; a line with a leading space is unchanged context, shown so that you can see where the change sits. The `@@ -2,3 +2,4 @@` marker is a **hunk header**: it says the change begins near line 2 and reports how many lines the region spans before and after. The two lines beginning `---` and `+++` name the old and new versions of the file. For everyday reading you can ignore everything above the hunk header and focus on the `+` and `-` lines.

Once you stage a change, plain `git diff` no longer shows it, because there is no longer any difference between the working directory and the staging area. To see what you have staged, that is, what your next commit would record, use `git diff --staged`:

```
$ git add baseline.R
$ git diff
$ git diff --staged
```


Draft, November 13, 2026. Content may change.

```
diff --git a/baseline.R b/baseline.R
index 3a1b2c3..4d5e6f7 100644
--- a/baseline.R
+++ b/baseline.R
@@ -2,3 +2,4 @@ dat <- read.csv("data-raw/enrollment.csv")
 n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
+med_age <- median(dat$age, na.rm = TRUE)
```

The two forms answer two different questions. `git diff` asks 'what have I changed but not yet staged?'. `git diff --staged` asks 'what am I about to commit?'. Reading the staged diff immediately before every commit is the single most effective habit for keeping mistakes out of your history. (`--cached` is an older synonym for `--staged`; the two are identical.)

> **Tip**
>
> Make `git diff --staged` a reflex. Run it in the moment between `git add` and `git commit`, every time. It is your last chance to notice that you staged a debugging line, a hard-coded path, or a stray data file before it becomes part of the permanent record.

2.2.2. Comparing commits

`git diff` also compares two commits when you name them. Give it two hashes and it reports what changed from the first to the second:

```
$ git diff 7e8f9a0 9b0c1d2
```

More often you compare against the present using relative names, described in full below. `git diff HEAD~1 HEAD` shows what the most recent commit changed:

```
$ git diff HEAD~1 HEAD
diff --git a/baseline.R b/baseline.R
index 4d5e6f7..5e6f7a8 100644
```

```
--- a/baseline.R
+++ b/baseline.R
@@ -3,3 +3,4 @@ n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
+sex_table <- table(dat$sex)
```

And `git diff HEAD` compares your working directory against the most recent commit, ignoring the staging area entirely, which answers 'what has changed since my last commit, staged or not?'.

2.2.3. Inspecting a single commit with git show

Where `git diff` compares two states, `git show` displays one commit in full: its metadata (author, date, message) followed by the diff it introduced. It is the natural way to answer 'what did this commit do?':

```
$ git show 9b0c1d2
commit 9b0c1d2f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 10:20:11 2026 -0700

    Add sex distribution table

diff --git a/baseline.R b/baseline.R
index 4d5e6f7..5e6f7a8 100644
--- a/baseline.R
+++ b/baseline.R
@@ -3,3 +3,4 @@ n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
+sex_table <- table(dat$sex)
```

`git show` with no argument shows `HEAD`, the most recent commit. Naming a commit shows that one. This is how you recover the reasoning behind a change months later: the message tells you why, and the diff tells you exactly what.

Draft, November 13, 2026. Content may change.

2.2.4. Reading history in depth

Day 1 introduced `git log` and `git log --oneline`. Several options turn the log from a list into an investigative tool.

`--stat` appends a per-file summary of how many lines each commit changed:

```
$ git log --oneline --stat
9b0c1d2 Add sex distribution table
 baseline.R | 1 +
 1 file changed, 1 insertion(+)
7e8f9a0 Add total count and mean age
 baseline.R | 3 +++
 1 file changed, 3 insertions(+)
```

`-p` (for 'patch') shows the full diff introduced by each commit, turning the log into a complete annotated history of every change. `-n <k>` limits the output to the most recent `k` commits, which pairs well with `-p`:

```
$ git log -p -n 1
commit 9b0c1d2f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 10:20:11 2026 -0700

    Add sex distribution table

diff --git a/baseline.R b/baseline.R
@@ -3,3 +3,4 @@ n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
+sex_table <- table(dat$sex)
```

`--graph` draws the branch structure to the left of the log. With a single line of history it is unremarkable, but once Day 3 introduces branches it becomes essential:

```
$ git log --oneline --graph
* 9b0c1d2 Add sex distribution table
* 7e8f9a0 Add total count and mean age
* a1b2c3d Add gitignore before any data enters repo
```

To see the history of a single file, and nothing else, pass the file after a `--` separator. The `--` tells Git that what follows is a path, not a commit name, which removes any ambiguity:

```
$ git log --oneline -- baseline.R
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
```

Note that the `.gitignore` commit does not appear, because it never touched `baseline.R`. Two further filters are worth knowing. `--author` restricts the log to one person, useful once you collaborate:

```
$ git log --oneline --author='Jane'
```

And `--since` (with its partner `--until`) restricts by date, accepting both calendar dates and human phrases:

```
$ git log --oneline --since='2026-07-01'
$ git log --oneline --since='2 weeks ago'
```

2.2.5. Referring to commits

Every command in this chapter takes a commit as an argument. There are three ways to name one.

The **full hash** is the forty-character identifier Git assigns, for example `9b0c1d2f0a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d`. A **short hash**, the first seven or so characters (`9b0c1d2`), is accepted anywhere the full hash is, as long as it is unambiguous within the repository. This is why `git log --oneline` prints short hashes: they are enough to name a commit.

Draft, November 13, 2026. Content may change.

`HEAD` names the commit you are currently on, almost always the newest on your branch. From `HEAD` you count backwards with a tilde: `HEAD~1` is the commit before `HEAD`, `HEAD~2` the one before that, and so on. These **relative references** save you from copying hashes for recent history:

```
HEAD~2    HEAD~1    HEAD
   |         |        |
   A ------- B ------ C
```

Here `C` is the newest commit, `HEAD` points to it, `HEAD~1` is `B`, and `HEAD~2` is `A`. `git show HEAD~1`, `git diff HEAD~2 HEAD`, and `git log HEAD~3` all use this. The relative form is the one you will reach for most, because undoing usually concerns the last commit or two.

> Check your understanding: HEAD~1 versus the short hash
>
> You want to see what your most recent commit changed. Is there any difference between `git show HEAD` and `git show <short-hash>` if the short hash is the one printed next to that commit in `git log --oneline`?
> No difference at all. Both name the same commit, so `git show` prints identical output. The distinction is convenience, not meaning: `HEAD` is a moving pointer that always refers to wherever you currently are, while the hash is a fixed name for one specific snapshot. If you make a new commit, `HEAD` moves to it and `git show HEAD` shows the new one, whereas `git show <hash>` still shows the original. Use relative names for 'the last commit' and hashes when you mean one particular commit regardless of where you stand.

2.2.6. The anatomy of a good commit

A commit is a unit of history, and its usefulness depends on two disciplines: what you put in it, and what you say about it.

One conceptual change per commit. A commit should record a single, self-contained change that can be described in one short sentence. If your commit message needs the word 'and', that is a signal you have bundled two changes that belong in two commits. 'Add median age and fix data

Draft, November 13, 2026. Content may change.

path' should be two commits: one that adds the median, one that fixes the path. The payoff is concrete. When a later diff reveals a bug, a history of small, single-purpose commits lets you find and undo exactly the change that caused it, without disturbing the unrelated work that happened to travel with it. The staging area (Day 1) is the tool that makes this possible: stage only the lines belonging to one change, commit them, then stage and commit the next.

A message that earns its place. The conventions for a commit message are stable across the profession:

- A **subject line** in the imperative mood, under about 50 characters: 'Add median age to baseline summary', not 'Added median age' or 'Adds median age'. The imperative reads as an instruction the commit carries out, which is the convention Git itself uses in its automatic messages.
- A **blank line** separating subject from body.
- A **body** that explains why the change was made, wrapped at about 72 characters. The diff already shows what changed; the body's job is the reasoning that the diff cannot express.

Contrast a poor message with a good one. Poor:

```
fixes
```

This tells a future reader nothing. Fixes what? Why? Good:

```
Use na.rm in mean age to exclude missing ages

Fourteen subjects have a missing age at baseline. Without
na.rm the mean was NA, which propagated into Table 1. This
restricts the mean to observed ages, consistent with the
handling of missingness in the statistical analysis plan.
```

A reader, including you in six months, can now reconstruct not just what changed but the analytical judgment behind it. This is what makes a commit history function as a lab notebook rather than a mechanical changelog.

To write a multi-line message, run `git commit` with no `-m` flag. Git opens the editor you configured on Day 1, with a template you fill in:

Draft, November 13, 2026. Content may change.

```
$ git commit
```

Type the subject on the first line, leave the second blank, write the body, then save and close the editor. Git records the whole message. Lines beginning with `#` are comments and are discarded. The `-m` form remains fine for short, subject-only commits; reach for the editor when the change deserves an explanation.

2.2.7. Undoing changes safely

The remainder of the lecture is the most important material of the week. Everyone makes mistakes in an analysis; the value of version control is that mistakes become recoverable. The essential skill is knowing which undo operations are safe (they preserve your work) and which are destructive (they discard it). Learn the distinction before you need it.

We proceed from the safest operations to the most dangerous.

Discarding uncommitted changes: `git restore <file>`. Suppose you have edited `baseline.R`, not yet staged or committed the edit, and you decide the edit was wrong. `git restore` overwrites the working-directory copy with the version from the last commit:

```
$ git restore baseline.R
```

This is destructive to *uncommitted* work. The edit you discarded was never committed, so Git has no record of it; it is gone. Use `git restore` only when you are certain the uncommitted changes should be thrown away.

> **Warning**
>
> `git restore <file>` permanently discards uncommitted changes to that file. There is no undo, because the discarded work was never recorded in any commit. Before running it, confirm with `git diff` that the changes it will erase are truly unwanted. This command is the most common cause of genuinely lost work among beginners, precisely because it looks innocuous.

Unstaging without losing work: `git restore --staged <file>`. If you have staged a change with `git add` but want to remove it from the staging area, `--staged` unstages it while leaving your working-directory edit untouched:

```
$ git restore --staged baseline.R
```

This is completely safe: it moves the change from 'staged' back to 'modified', and your edits remain in the file. It is the exact inverse of `git add`. (Older Git tutorials use `git reset HEAD <file>` for this; `git restore --staged` is the modern, clearer spelling.)

Fixing the most recent commit: `git commit --amend`. Perhaps you committed with a typo in the message, or you committed and then noticed you left a line out. `--amend` replaces the most recent commit with a new one:

```
$ git commit --amend -m 'Add median age to baseline summary'
```

To amend the *content* rather than the message, stage the additional change first, then amend; the staged change folds into the previous commit:

```
$ git add baseline.R
$ git commit --amend --no-edit
```

`--no-edit` keeps the existing message. Amending is convenient, but it does not edit the commit in place: it creates a replacement commit with a new hash and discards the old one. That rewriting is harmless while the commit lives only on your machine.

> **Warning**
>
> Never amend, or otherwise rewrite, a commit you have already shared, that is, pushed to GitHub or handed to a collaborator. Amending replaces the commit with one bearing a new hash; collaborators who based work on the original are left with a divergent history that is painful to reconcile. The rule is simple: amend freely before you push,

> never after. Pushing arrives on Day 4; until then every commit is private and safe to amend.

Undoing a past commit safely: `git revert <hash>`. When a commit is already shared, or simply is not the most recent one, the safe way to undo it is not to erase it but to add a new commit that reverses its effect. `git revert` does exactly this:

```
$ git revert 9b0c1d2
```

Git computes the inverse of commit `9b0c1d2` (every line it added is removed, every line it removed is added), applies that inverse as a new commit, and opens the editor for its message. The original commit remains in the history; the revert sits after it, canceling it. Because `git revert` only adds to history and never rewrites it, it is safe on shared branches, and it is the preferred way to undo anything others may have seen.

```
A -- B -- C -- C'
          ^     ^
        commit  revert of C
        to undo (a new commit)
```

Moving the branch pointer: `git reset`. `git reset` moves the current branch to point at an earlier commit, effectively removing the commits after it from the branch. It comes in three forms that differ in what they do to your staging area and working directory. Suppose your history is `A -- B -- C` and you reset to `B` (that is, to `HEAD~1`).

- `git reset --soft HEAD~1` moves the branch to `B` but leaves the staging area and working directory untouched. The changes from `C` are still staged, ready to recommit. This is how you 'uncommit' the last commit while keeping its changes staged, for example to split it or reword it.
- `git reset --mixed HEAD~1`, the default when no flag is given, moves the branch to `B` and unstages the changes from `C`, but leaves them in your working directory. The changes from `C` are now unstaged edits, as if you had never run `git add`.

Draft, November 13, 2026. Content may change.

- `git reset --hard HEAD~1` moves the branch to `B` and discards the changes from `C` from both the staging area and the working directory. The work in `C` is thrown away.

The three forms form a ladder from safe to dangerous. `--soft` and `--mixed` preserve your changes; they only move the boundaries between the three areas. `--hard` deletes uncommitted work, and it does so silently.

> ⚠ Warning
>
> `git reset --hard` discards all uncommitted changes in the files it touches, with no confirmation and no easy recovery. It is the most dangerous command in everyday Git. Reach for `--soft` or `--mixed` (or `git revert`) unless you are certain you want the work destroyed. When in doubt, commit your current state first: a commit is cheap, and a committed change is almost always recoverable even after a `--hard` reset, whereas an uncommitted one is not.

What is and is not recoverable. The reassuring generalization is that Git rarely loses committed work. A commit that seems to have vanished after an amend or a reset usually still exists in Git's internal records and can be retrieved; the tools for that recovery (notably the reflog) belong to the companion *Biostatistics Practicum* volume. The genuine exception is work that was never committed: uncommitted changes destroyed by `git restore` or `git reset --hard` leave no trace, because Git never recorded them. This asymmetry motivates the week's central habit. Commit early and often. A committed mistake is a recoverable mistake.

> Check your understanding: revert versus reset
>
> You pushed a commit to GitHub yesterday (Day 4 will teach pushing; assume for now it means others can see it). Today you realize that commit introduced a coding error. Should you use `git reset` to remove it or `git revert` to undo it?
> Use `git revert`. Because the commit has been shared, rewriting history with `git reset` would move your branch backwards and leave collaborators' copies pointing at a commit yours no longer contains, producing a divergent history that is troublesome to reconcile. `git`

Draft, November 13, 2026. Content may change.

> `revert` instead adds a new commit that cancels the faulty one, leaving the shared history intact and simply extending it with the correction. The governing principle: rewrite private history freely, but only ever add to shared history.

2.3. Worked example: catching and fixing an analytical mistake

We continue the `baseline-summary` repository from Day 1. Its history stands at three commits:

```
$ cd baseline-summary
$ git log --oneline
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
```

You are asked to add the mean baseline HbA1c to the summary. You edit `baseline.R` and add a line, but in haste you omit `na.rm = TRUE`, even though several subjects have a missing HbA1c. Before committing, you inspect the change:

```
$ git diff
diff --git a/baseline.R b/baseline.R
index 5e6f7a8..6f7a8b9 100644
--- a/baseline.R
+++ b/baseline.R
@@ -4,3 +4,4 @@ mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
 sex_table <- table(dat$sex)
+mean_hba1c <- mean(dat$hba1c)
```

The diff looks plausible at a glance, so you stage and commit it:

Draft, November 13, 2026. Content may change.

```
$ git add baseline.R
$ git commit -m 'Add mean baseline HbA1c'
[main 1a2b3c4] Add mean baseline HbA1c
 1 file changed, 1 insertion(+)
```

Running the script, you find `mean_hba1c` is `NA`. You inspect the offending commit to confirm the cause:

```
$ git show HEAD
commit 1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b
Author: Jane Investigator <jane@university.edu>
Date:   Sat Jul 25 11:02:44 2026 -0700

    Add mean baseline HbA1c

diff --git a/baseline.R b/baseline.R
@@ -4,3 +4,4 @@ mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
 sex_table <- table(dat$sex)
+mean_hba1c <- mean(dat$hba1c)
```

The diff confirms it: the new line lacks `na.rm = TRUE`, so a single missing HbA1c forces the mean to `NA`. There are two ways to fix this, and which you choose depends on whether the faulty commit has been shared.

Fix one: amend, because the commit is still private. You have not yet pushed anything (GitHub is Day 4), so the commit lives only on your machine and you may rewrite it. Correct the line, stage it, and fold it into the previous commit:

```
$ git restore baseline.R
```

Wait: that would discard the whole line, since the line was committed but you want to keep an edited version. Instead, edit the file to add `na.rm = TRUE`, confirm the fix, then amend:

Draft, November 13, 2026. Content may change.

```
$ git diff
diff --git a/baseline.R b/baseline.R
@@ -4,3 +4,4 @@ mean_age <- mean(dat$age, na.rm = TRUE)
 med_age <- median(dat$age, na.rm = TRUE)
 sex_table <- table(dat$sex)
-mean_hba1c <- mean(dat$hba1c)
+mean_hba1c <- mean(dat$hba1c, na.rm = TRUE)
$ git add baseline.R
$ git commit --amend --no-edit
[main 2b3c4d5] Add mean baseline HbA1c
 1 file changed, 1 insertion(+)
```

The history now contains a single, correct commit for the HbA1c change; the flawed version is gone, replaced by `2b3c4d5`:

```
$ git log --oneline
2b3c4d5 Add mean baseline HbA1c
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
```

Fix two: revert, if the commit had already been shared. Suppose instead that you had pushed commit `1a2b3c4` to a shared repository before noticing the error. Amending is now off the table. You instead add a new commit that reverses the faulty one:

```
$ git revert 1a2b3c4
[main 3c4d5e6] Revert "Add mean baseline HbA1c"
 1 file changed, 1 deletion(-)
```

This removes the faulty line by adding a commit, then you add the corrected line in a further commit:

```
$ echo 'mean_hba1c <- mean(dat$hba1c, na.rm = TRUE)' >> baseline.R
$ git add baseline.R
$ git commit -m 'Add mean baseline HbA1c with na.rm'
[main 4d5e6f7] Add mean baseline HbA1c with na.rm
```

The history now preserves the full record, including the mistake and its correction, which is exactly what you want on a branch others rely on:

```
$ git log --oneline
4d5e6f7 Add mean baseline HbA1c with na.rm
3c4d5e6 Revert "Add mean baseline HbA1c"
1a2b3c4 Add mean baseline HbA1c
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
```

The two fixes embody the chapter's governing rule. Rewrite private history to keep it clean; add to shared history to keep it honest. The mistake, a forgotten `na.rm`, is the kind of small analytical slip that version control turns from a silent corruption of your results into a documented, correctable event.

2.4. Homework

Attempt each problem in your own terminal before checking the solution. Continue in a scratch copy of the `baseline-summary` repository, or create a fresh repository with a few commits to experiment on. Nothing here needs to be shared, so you may amend and reset freely.

1. **Staged versus unstaged diffs.** In a repository with at least one commit, edit a tracked file. Run `git diff`, then `git add` the file, then run `git diff` again, followed by `git diff --staged`. Explain what each of the three `git diff` commands shows and why the second one printed nothing.

2. **Inspect a commit.** Using `git log --oneline` to find a hash, run `git show <hash>` on one of your commits. Identify in the output the author, the date, the message, and the diff. Then run `git show HEAD` and confirm it shows your most recent commit.

3. **History of one file.** In a repository where at least two different files have been committed, run `git log --oneline` and then `git log --oneline -- <one-file>`. Explain the difference between the two

Draft, November 13, 2026. Content may change.

outputs and why a commit might appear in the first but not the second.

4. **Relative references.** Without copying any hash, use relative references to (a) show the diff introduced by the most recent commit, and (b) show the diff between the commit two before `HEAD` and `HEAD`. Write the two commands you used.

5. **Unstage safely.** Edit a tracked file and stage it with `git add`. Now unstage it without losing your edit. Which command did you use, and how did you confirm with `git status` that the edit survived in the working directory?

6. **Amend a message.** Make a commit with a deliberately poor message such as `stuff`. Then, before doing anything else, replace that message with a proper imperative subject line. Which command did you use, and what happened to the commit's hash? Confirm the new message with `git log --oneline`.

7. **Revert versus amend.** Make a commit that introduces an obvious error into a file, such as a line that would not run. Undo it two different ways in two separate experiments: first by amending, then (after recreating the bad commit) by reverting. Compare the resulting `git log --oneline` in each case. Which one leaves a record that the error ever existed?

8. **The three resets.** Starting from a clean repository with history `A -- B -- C`, predict, before running anything, what `git status` will show after each of `git reset --soft HEAD~1`, `git reset --mixed HEAD~1`, and `git reset --hard HEAD~1`. Then test each on a fresh copy and reconcile your prediction with the result. State plainly which of the three you should never run with uncommitted work you care about, and why.

2.5. Solutions

Problem 1.

Draft, November 13, 2026. Content may change.

```
echo 'extra_line <- 1' >> baseline.R
git diff
#> ... shows the added line as a '+' line ...
git add baseline.R
git diff
#> (prints nothing)
git diff --staged
#> ... shows the added line as a '+' line ...
```

The first `git diff` compares the working directory against the staging area and shows your unstaged edit. After `git add`, the edit is in the staging area, so the working directory and staging area now match and plain `git diff` prints nothing. `git diff --staged` compares the staging area against the last commit and shows the change you are about to commit. The three commands answer, respectively, 'what is unstaged?', 'what is unstaged?' (now nothing), and 'what is staged?'.

Problem 2.

```
git log --oneline
#> 9b0c1d2 Add sex distribution table
#> 7e8f9a0 Add total count and mean age
git show 7e8f9a0
#> commit 7e8f9a0...
#> Author: Jane Investigator <jane@university.edu>
#> Date:   ...
#>
#>     Add total count and mean age
#>
#>     ... diff follows ...
git show HEAD
```

The `git show <hash>` output has four parts: the `commit` line with the full hash, the `Author` line, the `Date` line, the indented message, and then the diff the commit introduced. `git show HEAD` shows whichever commit is newest on the current branch, which here is `9b0c1d2`.

Problem 3.

Draft, November 13, 2026. Content may change.

```
git log --oneline
#> 9b0c1d2 Add sex distribution table
#> 7e8f9a0 Add total count and mean age
#> a1b2c3d Add gitignore before any data enters repo
git log --oneline -- baseline.R
#> 9b0c1d2 Add sex distribution table
#> 7e8f9a0 Add total count and mean age
```

The unfiltered log lists every commit. The filtered log lists only commits that changed `baseline.R`. The commit `a1b2c3d` created `.gitignore` and never touched `baseline.R`, so it is absent from the filtered output. The `--` separates the path from any commit names, telling Git that `baseline.R` is a file to filter on, not a revision.

Problem 4.

```
git diff HEAD~1 HEAD
git diff HEAD~2 HEAD
```

(a) `git diff HEAD~1 HEAD` compares the commit before the most recent against the most recent, which is precisely the change the most recent commit introduced. (You could also use `git show HEAD`.) (b) `git diff HEAD~2 HEAD` compares the commit two back against the current one, showing the combined effect of the last two commits. No hashes were typed; relative references sufficed.

Problem 5.

```
echo 'x <- 2' >> baseline.R
git add baseline.R
git restore --staged baseline.R
git status
#> On branch main
#> Changes not staged for commit:
#>   (use "git add <file>..." to update what will be committed)
#>   (use "git restore <file>..." to discard changes)
#>  modified:   baseline.R
```


Draft, November 13, 2026. Content may change.

`git restore --staged baseline.R` unstaged the file. The edit survived: `git status` reports `baseline.R` under 'Changes not staged for commit', that is, as a modified but unstaged file, exactly the state it was in before `git add`. The command is the inverse of staging and touches only the staging area, never the working directory, so it is safe.

Problem 6.

```
echo 'y <- 3' >> baseline.R
git add baseline.R
git commit -m 'stuff'
#> [main aaa1111] stuff
git commit --amend -m 'Add helper constant y'
#> [main bbb2222] Add helper constant y
git log --oneline
#> bbb2222 Add helper constant y
#> ...
```

`git commit --amend -m '<message>'` replaced the most recent commit. The hash changed, from `aaa1111` to `bbb2222`, because amending does not edit the old commit in place; it creates a new commit with the corrected message and discards the original. The old message `stuff` is gone from the history. This rewriting is safe only because the commit was never shared.

Problem 7.

```
echo 'this is not valid R $$$' >> baseline.R
git add baseline.R
git commit -m 'Add broken line'

# Experiment A: amend it away
git restore --source=HEAD~1 baseline.R
git add baseline.R
git commit --amend -m 'Add baseline note'
git log --oneline
#> ccc3333 Add baseline note        <- the broken commit is gone

# Experiment B: recreate the bad commit, then revert
echo 'this is not valid R $$$' >> baseline.R
```


Draft, November 13, 2026. Content may change.

```
git commit -a -m 'Add broken line'
git revert --no-edit HEAD
git log --oneline
#> ddd4444 Revert "Add broken line" <- both commits recorded
#> eee5555 Add broken line
```

Amending leaves no trace that the error ever existed: the faulty commit is replaced and vanishes from the log. Reverting preserves the full record: both the faulty commit and the commit that undoes it remain in the history. Amending is right for private mistakes you want to erase; reverting is right for shared mistakes, where the honest, non-destructive record is what collaborators need. (The `git restore --source=HEAD~1 <file>` form used in experiment A restores the file to its state one commit back, discarding the broken line before amending.)

Problem 8. Starting from `A -- B -- C`:

- `git reset --soft HEAD~1` moves the branch to `B` but leaves everything else alone. `git status` shows the changes from `C` as staged ('Changes to be committed'), ready to recommit.
- `git reset --mixed HEAD~1` (the default) moves the branch to `B` and unstages the changes from `C`. `git status` shows them as modified but unstaged ('Changes not staged for commit'). The work is still in your files.
- `git reset --hard HEAD~1` moves the branch to `B` and discards the changes from `C` entirely. `git status` reports 'nothing to commit, working tree clean'; the work from `C` is gone from both the staging area and the working directory.

The one never to run with uncommitted work you care about is `git reset --hard`. It discards changes with no confirmation and, for work that was never committed, no means of recovery. The committed changes from `C` in this exercise are in fact recoverable through Git's internal records, but any *uncommitted* edits sitting alongside them would be destroyed without recourse. `--soft` and `--mixed` only move the boundaries between Git's three areas and never delete your edits.

Draft, November 13, 2026. Content may change.

2.6. What's next

You can now read history, compare any two versions, and undo almost any mistake. So far, however, the history has been a single straight line of commits. Day 3 introduces **branching**, the mechanism that lets you develop a change in isolation, without disturbing a working analysis, and then **merge** it back. Branching is what turns Git from a personal undo system into a tool for parallel work, and, from Day 4 onward, into a tool for collaboration. Keep the `baseline-summary` repository, since Day 3 branches from it.

Draft, November 13, 2026. Content may change.

3. Day 3: Branching and Merging

3.1. Learning objectives

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

- Explain what a branch is: a lightweight, movable pointer to a commit, not a copy of your files.
- List branches with `git branch` and create and switch between them with `git switch`.
- Describe how `HEAD` moves and what happens to the working directory when you switch branches.
- Delete a branch with `git branch -d`, and explain the danger of the `-D` force variant.
- Combine work with `git merge`, and distinguish a fast-forward merge from a three-way merge.
- Adopt the feature-branch workflow and explain why it keeps `main` in a known-good state.
- Recognize, understand, and resolve a merge conflict, and back out of one with `git merge --abort`.

3.2. Lecture

3.2.1. What a branch is

The word ‘branch’ invites a misleading picture, that of a copy of your project folder that you edit separately and later paste back. That is not what a branch is, and holding the wrong picture makes every later command confusing.

Draft, November 13, 2026. Content may change.

A **branch is a pointer to a commit**. Nothing more. Recall from Day 1 that each commit is a snapshot with a forty-character hash, and that each commit records the hash of the commit before it. The commits therefore form a chain running back through history. A branch is simply a small, named label that points at one commit in that chain, and that moves forward automatically each time you commit.

Consider a repository with three commits on `main`:

```
              main
               |
A -- B -- C
```

`main` is not the three commits; it is the label pointing at `C`, the most recent one. When you make a fourth commit, Git creates commit `D`, sets its parent to `C`, and slides the `main` label forward to `D`:

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

This is why a branch is described as **lightweight** and **movable**. Creating one writes a single line to a file under `.git`; it does not copy `A`, `B`, or `C`. A branch is one of the cheapest things in Git, which is precisely why branching is used so freely.

Alongside the branch labels sits one more pointer, `HEAD`, which you met briefly on Day 1. `HEAD` marks 'the commit you are currently on'. Almost always it points not at a commit directly but at a branch, and that branch points at a commit. Written out, the arrangement above is:

```
HEAD -> main -> D
```

Reading this chain, 'you are on branch `main`, which is currently at commit `D`', is the mental model that makes the rest of the day straightforward.

Draft, November 13, 2026. Content may change.

3.2.2. Listing, creating, and switching branches

List the branches in a repository with `git branch`:

```
$ git branch
* main
```

The asterisk marks the branch you are currently on. A new repository has exactly one branch, `main`, so there is little to see yet.

Create a new branch and switch onto it in a single step with `git switch -c`, where `-c` stands for 'create':

```
$ git switch -c sensitivity-analysis
Switched to a new branch 'sensitivity-analysis'
```

The new branch is created pointing at the same commit you were on, and `HEAD` now follows it. List the branches again to confirm:

```
$ git branch
  main
* sensitivity-analysis
```

Both branches point at the same commit for the moment; the only thing that has changed is which one `HEAD` follows. To move back to an existing branch, use `git switch` without the `-c`:

```
$ git switch main
Switched to branch 'main'
```

> **Tip**
>
> If you have used Git before, or read older tutorials, you will have seen `git checkout` used for this: `git checkout -b <name>` to create and `git checkout <name>` to switch. That command still works and does exactly the same thing. The `git switch` command was introduced in 2019 to give branch switching its own dedicated, less

overloaded verb. This book teaches `git switch`, but you should recognize `git checkout` when you meet it in the wild.

3.2.3. How HEAD moves and what happens to your files

When you run `git switch <branch>`, two things happen. First, `HEAD` is repointed to the named branch. Second, and this is the part that surprises newcomers, Git rewrites the files in your working directory to match the commit that branch points at. Switching branches is not a passive change of label; it changes the files you see on disk.

An example makes this concrete. Suppose on branch `main` the file `baseline.R` ends at a mean-age calculation, and on branch `sensitivity-analysis` you have added twenty lines of subgroup code and committed them. Switching between the two branches swaps the on-disk contents of `baseline.R` back and forth: the extra twenty lines appear when you switch to the feature branch and vanish when you switch back to `main`. They are not lost when they vanish; they are safe in the commit on the feature branch, and the branch label remembers where that commit is.

⚠ Warning

Git will refuse to switch branches if you have uncommitted changes that switching would overwrite, printing 'Your local changes to the following files would be overwritten by checkout'. This is a protection, not an obstacle. It is telling you that you have work in the working directory that belongs to neither branch cleanly. The remedy is almost always to commit your changes on the current branch first, so that they are safely recorded, and then switch. Committing before switching is a habit worth building early.

Check your understanding: where did my code go?

You are on branch `sensitivity-analysis`, where `baseline.R` contains a subgroup analysis you committed. You run `git switch main` and open `baseline.R`; the subgroup code is gone. Have you lost it?

Draft, November 13, 2026. Content may change.

No. The subgroup code lives in a commit on the `sensitivity-analysis` branch, and that branch label still points at that commit. Switching to `main` rewrote your working directory to match `main`, which never had the subgroup code, so it is absent from the files on disk right now. Running `git switch sensitivity-analysis` repoints `HEAD` to the feature branch and rewrites the working directory again, and the subgroup code reappears exactly as you left it. Nothing committed to a branch is lost by switching away from it.

3.2.4. Deleting a branch

Once a branch has served its purpose, typically because its work has been merged into `main`, delete the label to keep the repository tidy. Deleting a branch removes only the pointer, not the commits it pointed at, provided those commits are reachable from another branch.

```
$ git branch -d sensitivity-analysis
Deleted branch sensitivity-analysis (was e4f5a6b).
```

The `-d` flag ('delete') is deliberately cautious. It refuses to delete a branch whose commits have not been merged anywhere else, because deleting such a branch would strand those commits with no label pointing at them, making them very hard to recover. In that situation Git prints an error and declines:

```
$ git branch -d draft-analysis
error: the branch 'draft-analysis' is not fully merged.
```

The capital `-D` flag forces the deletion regardless.

> ⚠ Warning
>
> `git branch -D <name>` deletes a branch even when its commits are unmerged, stranding that work. The commits are not immediately erased, and an advanced tool called the reflog can sometimes recover them, but treat `-D` as discarding the branch's unique commits. Use the

> lower-case `-d` by default and reach for `-D` only when you are certain the branch holds nothing you want to keep.

3.2.5. Merging: fast-forward and three-way

Branching would be pointless if work could not be brought back together. **Merging** combines the commits of one branch into another. You run `git merge <branch>` while on the branch you want to receive the work; the named branch is merged into your current branch.

There are two ways a merge can play out, and understanding which one occurred is the key to reading Git's output.

A **fast-forward merge** happens when the receiving branch has not moved since the other branch was created. Suppose `main` points at `C`, and a feature branch added `D` and `E` on top of it:

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

Because `main` still points at `C`, and `C` is directly in the ancestry of `E`, Git can incorporate the feature simply by sliding the `main` label forward to `E`. No new commit is needed; the history is already a straight line. This is a fast-forward:

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


Draft, November 13, 2026. Content may change.

A **three-way merge** happens when both branches have moved since they diverged, so that neither is an ancestor of the other. Suppose after creating the feature branch you also committed `F` on `main`:

```
                  main
                   |
A -- B -- C -- F
               \
                D -- E
                     |
                  feature
```

Now `main` cannot simply slide forward, because `F` and `E` are on divergent lines. Git instead creates a new **merge commit**, `M`, whose distinctive feature is that it has **two parents**, `F` and `E`, tying the two lines back together:

```
                        main
                         |
A -- B -- C -- F ------- M
               \        /
                D -- E -
```

It is called a three-way merge because Git computes the result by comparing three commits: the two branch tips (`F` and `E`) and their common ancestor (`C`). The merge commit records the combined state and closes the loop in the graph.

You can see the whole shape at once with the graph option to `git log`:

```
$ git log --graph --oneline --all
*   a7b8c9d (HEAD -> main) Merge branch 'feature'
|\
| * e4f5a6b (feature) Add subgroup analysis output
| * d3e4f5a Add subgroup analysis
* | f5a6b7c Round mean age to one decimal
|/
* 9b0c1d2 Add sex distribution table
```


Draft, November 13, 2026. Content may change.

```
* 7e8f9a0 Add total count and mean age
* a1b2c3d Add gitignore before any data enters repo
```

The `--graph` option draws the branch structure to the left, `--oneline` compresses each commit to one line, and `--all` shows every branch rather than only the current one. This command is the single most useful way to understand what your history looks like, and it is worth running often.

3.2.6. The feature-branch workflow

Branches earn their keep through a simple discipline called the **feature-branch workflow**. The rule is: do not commit work-in-progress directly on `main`. Instead, for each self-contained piece of work, create a branch, do the work and commit it there, and merge the branch back into `main` only once the work is complete and correct.

For a biostatistician a 'self-contained piece of work' is usually one analytical task: a sensitivity analysis, a new table, a requested subgroup, a change to the modeling approach. Each gets its own branch, named for what it does:

```
git switch -c subgroup-by-site
git switch -c add-sensitivity-analysis
git switch -c fix-age-eligibility
```

The reason this is safer than working on `main` directly is worth stating plainly. The `main` branch should always hold a version of the analysis that runs and is trustworthy, the version you would be willing to hand to a collaborator or use for a report. If you edit `main` directly, then halfway through a large change your analysis is in a broken, half-finished state, and `main` no longer means anything reliable. A branch quarantines the mess. While the feature branch is broken and half-written, `main` continues to point at the last known-good commit, untouched. Only when the feature is finished and working does it rejoin `main` in a single merge. This same discipline becomes the foundation of collaboration on Day 5, where each collaborator's branch is reviewed before it joins `main`.

Draft, November 13, 2026. Content may change.

3.2.7. Merge conflicts

Most merges succeed without your intervention. Git is adept at combining changes that touch different parts of a file, or different files entirely; it simply applies both sets of changes. A **merge conflict** arises only in one specific situation: when the two branches being merged have changed **the same lines of the same file** in different ways. Git cannot know which version you intended, so rather than guess, it stops and asks you to decide.

Suppose `baseline.R` on both branches contains this line defining the analytic cohort:

```
analytic <- subset(dat, age >= 18 & age <= 65)
```

On a branch named `raise-lower-age` you tightened the lower bound to 21:

```
analytic <- subset(dat, age >= 21 & age <= 65)
```

Meanwhile on `main` a colleague widened the upper bound to 75:

```
analytic <- subset(dat, age >= 18 & age <= 75)
```

Both edits touch the same line. Merging `raise-lower-age` into `main` produces a conflict:

```
$ git merge raise-lower-age
Auto-merging baseline.R
CONFLICT (content): Merge conflict in baseline.R
Automatic merge failed; fix conflicts and then commit the
result.
```

Open `baseline.R` and you will find that Git has rewritten the conflicting region with **conflict markers**:

Draft, November 13, 2026. Content may change.

```
<<<<<<< HEAD
analytic <- subset(dat, age >= 18 & age <= 75)
=======
analytic <- subset(dat, age >= 21 & age <= 65)
>>>>>>> raise-lower-age
```

Read the markers precisely. The block between `<<<<<<< HEAD` and `=======` is the version on your current branch, that is, `main`, since `HEAD` points at `main`. The block between `=======` and `>>>>>>> raise-lower-age` is the version from the branch you are merging in. Git has placed both in front of you and is asking which should survive, or how they should be combined.

You resolve the conflict by **editing the file** so that it contains exactly the code you want, and **deleting all three marker lines**. Here both edits are correct and compatible: the intended cohort is ages 21 through 75. So you edit the region to read:

```
analytic <- subset(dat, age >= 21 & age <= 75)
```

with no markers remaining. Then you stage the resolved file and commit, which completes the merge:

```
$ git add baseline.R
$ git commit -m 'Merge raise-lower-age; combine age bounds'
[main b8c9d0e] Merge raise-lower-age; combine age bounds
```

Two points deserve emphasis. First, resolving a conflict is an ordinary edit followed by an ordinary `git add`; there is no special 'resolve' command. Staging the file is how you tell Git 'this file is settled'. Second, you must actually read the code and choose the correct result. Git guarantees only that the conflict markers are gone; it cannot check that the surviving code is statistically correct. Resolving a conflict carelessly, by keeping one side at random, is a real way to introduce a silent analytical error.

If a conflict appears and you would rather not deal with it now, back out of the merge entirely:

Draft, November 13, 2026. Content may change.

```
$ git merge --abort
```

This returns the repository to exactly the state it was in before you ran `git merge`, as though the merge had never been attempted. Nothing is lost, and you can try again later.

> Check your understanding: what do the markers mean?
>
> During a merge you see `<<<<<<< HEAD` above one block of code and `>>>>>>> add-subgroup` below another, separated by `=======`. Which block is your current work, and what must you do before you can commit?
> The block just below `<<<<<<< HEAD` is the version on your current branch, the one `HEAD` points at. The block just above `>>>>>>> add-subgroup` is the version coming from the `add-subgroup` branch you are merging in. To finish, edit the file so it contains the code you actually want, which may be one side, the other, or a combination of both, and delete all three marker lines (`<<<<<<<`, `=======`, `>>>>>>>`). Then run `git add` on the file and `git commit`. Leaving even one marker line in the file is a common mistake; the markers are not valid R and will cause a syntax error if you try to run the script.

3.3. Worked example: a subgroup analysis on a feature branch

We continue the `baseline-summary` repository from Days 1 and 2. Move into it and confirm the starting state:

```
$ cd baseline-summary
$ git switch main
$ git log --oneline
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
```


Draft, November 13, 2026. Content may change.

You are asked to add an age-stratified subgroup analysis. This is a self-contained piece of work, so it belongs on its own branch rather than on `main`. Create and switch to one:

```
$ git switch -c subgroup-analysis
Switched to a new branch 'subgroup-analysis'
```

Add the subgroup code to `baseline.R` and commit it on the feature branch:

```
$ cat >> baseline.R <<'EOF'
dat$age_group <- ifelse(dat$age < 65, "under65", "65plus")
age_group_n <- table(dat$age_group)
EOF
$ git add baseline.R
$ git commit -m 'Add age-group stratification'
[subgroup-analysis d3e4f5a] Add age-group stratification
 1 file changed, 2 insertions(+)
```

Extend it with a per-group mean age, and commit that as a second step on the same branch:

```
$ echo 'age_group_means <- tapply(dat$age, dat$age_group, mean)' >> bas
$ git commit -a -m 'Add per-group mean age'
[subgroup-analysis e4f5a6b] Add per-group mean age
 1 file changed, 1 insertion(+)
```

The feature branch now has two commits that `main` does not. Switch back to `main` and confirm that the subgroup code is absent there, exactly as the lecture described:

```
$ git switch main
Switched to branch 'main'
$ tail -n 1 baseline.R
sex_table <- table(dat$sex)
```


Draft, November 13, 2026. Content may change.

The last line of `baseline.R` on `main` is the sex table from Day 1; the subgroup lines are not present, because they live only on the feature branch. Now make a small, independent change on `main` itself, so that when we merge, both branches will have moved and the merge will be a genuine three-way merge. Round the mean age for reporting:

```
$ echo 'mean_age <- round(mean_age, 1)' >> baseline.R
$ git commit -a -m 'Round mean age to one decimal'
[main f5a6b7c] Round mean age to one decimal
 1 file changed, 1 insertion(+)
```

Both branches have now advanced since they diverged at `9b0c1d2`. Merge the feature branch into `main`:

```
$ git merge subgroup-analysis
Merge made by the 'recursive' strategy.
 baseline.R | 3 +++
 1 file changed, 3 insertions(+)
```

Because the two branches changed different lines of `baseline.R`, the change on `main` at the top of the file and the subgroup additions at the bottom, Git combined them automatically with no conflict, and recorded a merge commit. The message 'Merge made by the recursive strategy' confirms this was a three-way merge rather than a fast-forward.

Inspect the resulting history:

```
$ git log --graph --oneline --all
*   a7b8c9d (HEAD -> main) Merge branch 'subgroup-analysis'
|\
| * e4f5a6b (subgroup-analysis) Add per-group mean age
| * d3e4f5a Add age-group stratification
* | f5a6b7c Round mean age to one decimal
|/
* 9b0c1d2 Add sex distribution table
* 7e8f9a0 Add total count and mean age
* a1b2c3d Add gitignore before any data enters repo
```


Draft, November 13, 2026. Content may change.

The graph shows the story exactly: the two lines diverging at `9b0c1d2`, the rounding commit on one side and the two subgroup commits on the other, and the merge commit `a7b8c9d` tying them back together. The feature branch has done its job, so delete its label:

```
$ git branch -d subgroup-analysis
Deleted branch subgroup-analysis (was e4f5a6b).
```

The subgroup commits remain in the history, reachable from `main` through the merge commit; only the redundant label is gone. Throughout this exercise `main` never held a broken, half-written analysis: the subgroup work was assembled in isolation and joined `main` only when complete. That is the entire point of the feature-branch workflow, and it is the rhythm every subsequent day assumes.

3.4. Homework

Attempt each problem in your own terminal before checking the solution. Work in a scratch repository so that you can experiment freely; create one with `mkdir hw3 && cd hw3 && git init` and make an initial commit to start from.

1. **List and create.** In a fresh repository with at least one commit, run `git branch` and note the single branch and the asterisk. Create and switch to a branch named `experiment` with one command. Run `git branch` again and describe what changed.

2. **Switching swaps files.** On `main`, create a file `model.R` containing the single line `fit <- lm(y ~ x, data = dat)` and commit it. Create a branch `add-covariate`, and on it change the line to `fit <- lm(y ~ x + age, data = dat)` and commit. Switch back to `main` and open `model.R`. Which version do you see? Switch to `add-covariate` and look again. Explain.

3. **A fast-forward merge.** Starting from `main`, create a branch `add-readme`, add a file `README.md` with one line and commit it, then switch back to `main` and immediately merge `add-readme`. Read Git's

Draft, November 13, 2026. Content may change.

output. Does it say 'fast-forward'? Explain why no merge commit was created.

4. **A three-way merge.** Reset to a scenario where both branches move: on `main` create `analysis.R` with two lines and commit. Branch to `feature-a`, append a line, and commit. Switch to `main`, append a *different* line (at a different position), and commit. Merge `feature-a` into `main`. Did a conflict occur? Show the history with `git log --graph --oneline --all` and identify the merge commit.

5. **Delete a branch safely.** After completing problem 4, delete the `feature-a` branch with `git branch -d`. Does Git allow it? Now create a branch `throwaway`, commit something on it that you never merge, switch back to `main`, and try `git branch -d throwaway`. What happens, and why? What flag would force it, and what is the risk?

6. **Create a conflict.** On `main`, create `threshold.R` containing `alpha <- 0.05` and commit. Branch to `strict-alpha` and change the line to `alpha <- 0.01`, then commit. Switch to `main` and change the same line to `alpha <- 0.025`, then commit. Merge `strict-alpha` into `main`. Show the conflict markers Git inserts and identify which block is `main` and which is `strict-alpha`.

7. **Resolve the conflict.** Continue from problem 6. Decide that the correct value is `alpha <- 0.01`, edit `threshold.R` to that single line with no markers, and complete the merge. Show the commands and confirm with `git log --oneline` that a merge commit exists.

8. **Abort a merge.** Recreate a conflict as in problem 6 (use fresh branches and a fresh conflicting line). This time, instead of resolving it, run `git merge --abort`. Run `git status` and open the file. What state is the repository in? Explain what `--abort` undid.

3.5. Solutions

Problem 1.

Draft, November 13, 2026. Content may change.

```
git branch
#> * main
git switch -c experiment
#> Switched to a new branch 'experiment'
git branch
#>   main
#> * experiment
```

Before, one branch `main` existed and the asterisk marked it as current. After, a second branch `experiment` exists and the asterisk has moved to it, because `git switch -c` both created the branch and moved `HEAD` onto it. Both branches still point at the same commit; only the current branch has changed.

Problem 2.

```
echo 'fit <- lm(y ~ x, data = dat)' > model.R
git add model.R
git commit -m 'Add base model'

git switch -c add-covariate
echo 'fit <- lm(y ~ x + age, data = dat)' > model.R
git commit -a -m 'Add age covariate'

git switch main
cat model.R
#> fit <- lm(y ~ x, data = dat)
git switch add-covariate
cat model.R
#> fit <- lm(y ~ x + age, data = dat)
```

On `main` you see the base model without `age`; on `add-covariate` you see the model with `age`. Switching branches rewrites the working directory to match the commit the branch points at, so the on-disk contents of `model.R` change as you switch. Neither version is lost; each is preserved in a commit on its respective branch.

Problem 3.

Draft, November 13, 2026. Content may change.

```
git switch -c add-readme
echo '# Analysis project' > README.md
git add README.md
git commit -m 'Add README'

git switch main
git merge add-readme
#> Updating 9b0c1d2..c1d2e3f
#> Fast-forward
#>  README.md | 1 +
#>  1 file changed, 1 insertion(+)
```

Git reports 'Fast-forward'. No merge commit was created because `main` had not moved since `add-readme` was created: `main` still pointed at the commit that is the direct parent of the new commit. Git therefore combined the work simply by sliding the `main` label forward to the branch's commit. A straight-line history needs no merge commit.

Problem 4.

```
echo 'dat <- read.csv("data.csv")' > analysis.R
echo 'n <- nrow(dat)' >> analysis.R
git add analysis.R
git commit -m 'Add data load and count'

git switch -c feature-a
echo 'mean_x <- mean(dat$x)' >> analysis.R
git commit -a -m 'Add mean of x'

git switch main
cat > analysis.R <<'EOF'
dat <- read.csv("data.csv")
summary(dat)
n <- nrow(dat)
EOF
git commit -a -m 'Add summary call'

git merge feature-a
```


Draft, November 13, 2026. Content may change.

```
#> Merge made by the 'recursive' strategy.
git log --graph --oneline --all
#> *   9f0a1b2 (HEAD -> main) Merge branch 'feature-a'
#> |\
#> | * 7d8e9f0 (feature-a) Add mean of x
#> * | 6c7d8e9 Add summary call
#> |/
#> * 5b6c7d8 Add data load and count
```

No conflict occurred, because the two branches changed different lines of `analysis.R`: `feature-a` appended a line at the end, while `main` inserted a `summary(dat)` line in the middle. The merge commit `9f0a1b2` is the node with two parents, shown by the `|\` fork in the graph, joining the `feature-a` line and the `main` line back together.

Problem 5.

```
git branch -d feature-a
#> Deleted branch feature-a (was 7d8e9f0).

git switch -c throwaway
echo 'x <- 1' > scratch.R
git add scratch.R
git commit -m 'Scratch work'
git switch main
git branch -d throwaway
#> error: the branch 'throwaway' is not fully merged.
```

Deleting `feature-a` is allowed because its commits were merged into `main` in problem 4, so they remain reachable. Deleting `throwaway` is refused because its commit was never merged; deleting the label would strand that commit with nothing pointing at it. The `-D` flag (capital) forces the deletion, but the risk is exactly that stranding: the unique work on the branch becomes very hard to recover. Use `-D` only when you are certain the branch holds nothing you want.

Problem 6.

Draft, November 13, 2026. Content may change.

```
echo 'alpha <- 0.05' > threshold.R
git add threshold.R
git commit -m 'Set alpha to 0.05'

git switch -c strict-alpha
echo 'alpha <- 0.01' > threshold.R
git commit -a -m 'Tighten alpha to 0.01'

git switch main
echo 'alpha <- 0.025' > threshold.R
git commit -a -m 'Set alpha to 0.025'

git merge strict-alpha
#> CONFLICT (content): Merge conflict in threshold.R
#> Automatic merge failed; fix conflicts and then commit
```

Opening `threshold.R` shows:

```
<<<<<<< HEAD
alpha <- 0.025
=======
alpha <- 0.01
>>>>>>> strict-alpha
```

The block below `<<<<<<< HEAD` (`alpha <- 0.025`) is the version on `main`, the current branch. The block above `>>>>>>> strict-alpha` (`alpha <- 0.01`) is the version from the branch being merged in. Both branches changed the same line, which is what forced the conflict.

Problem 7.

```
echo 'alpha <- 0.01' > threshold.R   # edited: no markers left
git add threshold.R
git commit -m 'Merge strict-alpha; adopt alpha = 0.01'
git log --oneline
#> a1b2c3d Merge strict-alpha; adopt alpha = 0.01
#> ...
```


Draft, November 13, 2026. Content may change.

Editing the file down to the single intended line, with all three marker lines removed, resolves the conflict. `git add` stages the resolved file, which is how you signal to Git that the conflict is settled, and `git commit` records the merge commit. The merge commit appears at the top of `git log --oneline`, confirming the merge completed.

Problem 8.

```
echo 'seed <- 100' > config.R
git add config.R
git commit -m 'Set seed to 100'

git switch -c other-seed
echo 'seed <- 200' > config.R
git commit -a -m 'Change seed to 200'

git switch main
echo 'seed <- 300' > config.R
git commit -a -m 'Change seed to 300'

git merge other-seed
#> CONFLICT (content): Merge conflict in config.R
git merge --abort
git status
#> On branch main
#> nothing to commit, working tree clean
cat config.R
#> seed <- 300
```

After `git merge --abort`, `git status` reports a clean working tree and `config.R` holds `seed <- 300`, the version on `main` before the merge. The command undid the entire merge attempt, removed the conflict markers, and restored the repository to precisely its pre-merge state, as though `git merge` had never been run. This is the safe escape hatch when a conflict appears at an inconvenient moment.

Draft, November 13, 2026. Content may change.

3.6. What's next

Everything so far has been local. The repository, its branches, and its history live only on your own machine. Day 4 connects your work to the wider world. You will learn what GitHub is, authenticate with the `gh` command-line tool, create a remote repository, and use `git push`, `git pull`, `git fetch`, and `git clone` to synchronize your local history with a copy hosted online. The branching and merging skills from today carry over directly. That is, a remote is, in large part, just another place branches live. Keep the `baseline-summary` repository, since Day 4 publishes it.

Draft, November 13, 2026. Content may change.

4. Day 4: GitHub: Remotes and Publishing

4.1. Learning objectives

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

- Explain what GitHub is, how it differs from Git, and what a 'remote' is.
- Install the GitHub command-line tool `gh` and authenticate it with your GitHub account.
- Publish an existing local repository to GitHub in a single command with `gh repo create`.
- Describe the remote mechanics that `gh` sets up: `origin`, upstream tracking, and the difference between `git push` and `git push -u`.
- Send commits to GitHub with `git push` and retrieve changes with `git fetch` and `git pull`.
- Copy a repository, with its full history, using `git clone` and `gh repo clone`.
- Browse a repository from the terminal with `gh repo view` and `gh browse`.
- Write a `README.md` as the front page of a repository.
- Track tasks and defects as issues with `gh issue`.

4.2. Lecture

4.2.1. What GitHub is, and what it is not

Days 1 through 3 used Git entirely offline. Every commit you made lives in the `.git` directory inside your project folder, on your own machine, and

Draft, November 13, 2026. Content may change.

nowhere else. That is a complete version-control system, but it has two limitations. If your laptop is lost, so is your work, and no collaborator can see the history, because it exists only on your disk.

GitHub removes both limitations. It is a website that **hosts** Git repositories: it stores a copy of your repository on its servers, from which you can retrieve it anywhere and to which collaborators can contribute. It is important to be precise about the relationship. GitHub does not replace Git and adds no new version-control concepts. The commits, branches, and history are exactly the ones Git already made on your machine; GitHub simply holds a synchronized copy and layers collaboration tools (issues, pull requests, code review) on top.

GitHub is one host among several. GitLab and Bitbucket offer the same core service, and an organization can run its own private server. The Git commands you learned this week work identically against any of them. This book uses GitHub because it is the de facto standard in academic biostatistics, and it uses GitHub's official command-line tool, `gh`, throughout.

Check your understanding: does GitHub store new history?

You have made ten commits locally. You publish the repository to GitHub. Has GitHub created any new commits, or changed any of your commit hashes?
No. Publishing copies your existing commits, unchanged, to GitHub's servers. The ten commits keep their exact hashes, authors, timestamps, and messages, because a commit's hash is computed from its content and would change if anything about it changed. GitHub holds a faithful copy of the history Git already built. This is why the distinction matters: Git does the version control; GitHub does the hosting. Nothing about your local history depends on GitHub existing.

4.2.2. What a remote is

A **remote** is a named reference to another copy of your repository, almost always one hosted elsewhere, such as on GitHub. It is a bookmark: a short name paired with a URL, so that instead of typing the full address every

Draft, November 13, 2026. Content may change.

time you synchronize, you refer to it by name. By universal convention the primary remote is named `origin`.

```
   your laptop                       GitHub
  +-------------+                 +-------------+
  | local repo  |  --- push -->   | origin      |
  | (.git)      |  <-- fetch ---  | (remote copy)|
  +-------------+                 +-------------+
```

You **push** commits from your local repository up to the remote, and you **fetch** or **pull** commits from the remote down to your local repository. The remote is just another repository; the same objects (commits, trees, blobs) move between the two. Everything in this chapter is machinery for keeping a local repository and its `origin` remote synchronized.

4.2.3. Installing the GitHub CLI

The GitHub CLI is a program named `gh`. It is separate from Git: `git` manages your local repository, while `gh` talks to GitHub's servers on your behalf, so that repository creation, issues, and later pull requests can be driven from the terminal rather than a web browser.

macOS. With Homebrew, `brew install gh`.

Windows. Download the installer from https://cli.github.com, or use `winget install --id GitHub.cli`. Run subsequent commands in Git Bash.

Linux. Follow the package-manager instructions at https://github.com/cli/cli#installation; on Debian or Ubuntu the tool is installed from GitHub's package repository.

Confirm the installation:

```
$ gh --version
gh version 2.55.0 (2026-06-18)
https://github.com/cli/cli/releases/latest
```

Any recent version is suitable. You also need a free GitHub account, created at https://github.com, before the next step.

Draft, November 13, 2026. Content may change.

4.2.4. Authenticating with gh auth login

Before `gh` can act on your behalf it must prove to GitHub that you are you. Run the interactive login once per machine:

```
$ gh auth login
? Where do you use GitHub? GitHub.com
? What is your preferred protocol for Git operations
  on this host? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI?
  Login with a web browser

! First copy your one-time code: A1B2-C3D4
Press Enter to open github.com in your browser...
```

Work through the prompts as shown. Choose **GitHub.com** (not an enterprise server), **HTTPS** as the protocol (the simplest option, requiring no SSH-key setup), and **Yes** to let `gh` configure Git to use your GitHub credentials. Finally choose **Login with a web browser**. The tool prints a one-time code, opens your browser, and asks you to paste the code and confirm. On success:

```
✓ Authentication complete.
- gh config set -h github.com git_protocol https
✓ Configured git protocol
✓ Logged in as rgt47
```

You authenticate once. The credential is stored securely by your operating system, and every later `gh` command, and every `git push` to GitHub, uses it silently. Confirm the state at any time:

```
$ gh auth status
github.com
  ✓ Logged in to github.com account rgt47 (keyring)
  - Active account: true
  - Git operations protocol: https
  - Token scopes: 'gist', 'read:org', 'repo'
```


Draft, November 13, 2026. Content may change.

> **Tip**
>
> Choosing HTTPS at the protocol prompt is the right default for this book. It lets `gh` manage your credentials for you, so `git push` never asks for a password. SSH keys are a fine alternative once you are comfortable, but they require generating and registering a key pair first, which HTTPS avoids entirely.

4.2.5. Publishing a local repository in one step

You already have a local repository from Days 1 through 3. Publishing it to GitHub is a single command run from inside the repository:

```
$ gh repo create baseline-summary \
    --public --source=. --remote=origin --push
```

Each flag does one thing:

- `baseline-summary` is the name the repository will have on GitHub. Omit it to reuse the current folder's name.
- `--public` makes the repository visible to anyone. Use `--private` instead to restrict it to you and people you explicitly grant access (discussed below).
- `--source=.` tells `gh` to publish the existing repository in the current directory, rather than create an empty new one.
- `--remote=origin` names the new GitHub repository as your `origin` remote locally, wiring the two together.
- `--push` immediately pushes your existing commits up, so the GitHub copy is populated at once.

The output confirms each step:

```
✓ Created repository rgt47/baseline-summary on GitHub
  https://github.com/rgt47/baseline-summary
✓ Added remote https://github.com/rgt47/baseline-summary.git
✓ Pushed commits to
  https://github.com/rgt47/baseline-summary.git
```

Your local history is now mirrored on GitHub, and the local repository knows where its remote copy lives. That is the entire publishing workflow.

4.2.6. Public, private, and protected health information

The choice between `--public` and `--private` is not cosmetic; for a biostatistician it is a data-governance decision. A public repository is readable by the entire internet. A private repository is visible only to you and collaborators you name.

```
$ gh repo create trial-2026-interim \
    --private --source=. --remote=origin --push
✓ Created repository rgt47/trial-2026-interim on GitHub
```

> **Warning**
>
> Protected health information (PHI) must never be pushed to a public repository, and ideally never to GitHub at all, public or private. Pushing publishes not only the current files but the entire history: a data file committed once and deleted later still lives in an earlier commit and travels to GitHub with everything else. Making a repository private reduces but does not eliminate the exposure, because access is still granted through a third-party commercial service outside your institution's control. The safe practice established on Day 1 is the safeguard here: keep raw data in a `data-raw/` folder that `.gitignore` excludes, so it is never committed and therefore never pushed. Verify what you are about to publish with `git log --stat` before running `gh repo create`.

4.2.7. The remote mechanics gh sets up

`gh repo create` is a convenience wrapper. It is worth seeing the plain Git commands underneath, both to understand what happened and because you will meet them in repositories set up by others. Inspect your remotes:

```
$ git remote -v
origin  https://github.com/rgt47/baseline-summary.git (fetch)
origin  https://github.com/rgt47/baseline-summary.git (push)
```

The name `origin` maps to a URL, listed twice because Git allows a different address for fetching and for pushing; here they are the same. Had you created the GitHub repository through the website instead, you would connect it manually with the command `gh` ran for you:

```
git remote add origin \
  https://github.com/rgt47/baseline-summary.git
```

And you would send your first commits with:

```
$ git push -u origin main
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Writing objects: 100% (12/12), 1.34 KiB, done.
To https://github.com/rgt47/baseline-summary.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

The `-u` flag (short for `--set-upstream`) is worth understanding. It records that your local `main` branch **tracks** `origin/main`, that is, establishes a default pairing between the two. Once that link exists, you no longer name the remote and branch each time; a bare `git push` knows to send `main` to `origin`, and a bare `git pull` knows where to retrieve from. You set the upstream once per branch; thereafter:

```
$ git push
Everything up-to-date
```

4.2.8. Pushing later commits

After the initial publish, your daily rhythm gains one step. You still edit, stage, and commit locally as on Days 1 through 3. When you want the GitHub copy to catch up, you push:

Draft, November 13, 2026. Content may change.

```
$ echo 'quantile(dat$age, na.rm = TRUE)' >> baseline.R
$ git commit -a -m 'Add age quantiles'
[main 5f6a7b8] Add age quantiles
 1 file changed, 1 insertion(+)
$ git push
Enumerating objects: 5, done.
Writing objects: 100% (3/3), 312 bytes, done.
To https://github.com/rgt47/baseline-summary.git
   9b0c1d2..5f6a7b8  main -> main
```

The last line reports that `origin/main` moved from commit `9b0c1d2` to `5f6a7b8`. Committing and pushing are separate acts: commits accumulate locally, and a push sends whatever has accumulated since the last push. It is common and correct to make several commits and push them together.

4.2.9. Getting changes: fetch and pull

Synchronization runs both ways. When the GitHub copy has commits your local repository lacks, because a collaborator pushed, or because you pushed from another computer, you bring them down. There are two commands, and the difference between them matters.

`git fetch` downloads new commits from the remote into your local repository but does **not** change your working files. It updates `origin/main`, Git's record of where the remote branch is, and leaves your own `main` untouched:

```
$ git fetch
remote: Enumerating objects: 4, done.
From https://github.com/rgt47/baseline-summary
   5f6a7b8..a1c2e3f  main -> origin/main
```

After a fetch you can inspect what arrived before letting it touch your work. `git pull` goes one step further: it fetches and then immediately **merges** the fetched commits into your current branch. It is exactly `git fetch` followed by `git merge origin/main`:

Draft, November 13, 2026. Content may change.

```
$ git pull
Updating 5f6a7b8..a1c2e3f
Fast-forward
 baseline.R | 2 ++
 1 file changed, 2 insertions(+)
```

For everyday solo work, `git pull` is what you want: it is the one command that makes your local branch match the remote. When collaboration introduces the possibility that both sides have changed, a pull can produce a merge conflict, handled exactly as on Day 3. The remote dimension of conflicts is the subject of Day 5; for now, note only that fetch inspects and pull incorporates.

Check your understanding: fetch versus pull

You run `git fetch` and see that `origin/main` has advanced by three commits. You then run `git log --oneline` and your own `main` still points where it did before. Is something broken?
No; this is exactly what `git fetch` does. Fetch updates only the remote-tracking branch `origin/main`, which records where GitHub's `main` is. It deliberately leaves your local `main` and your working files unchanged, so that you can review the incoming commits before adopting them. To move your own `main` up to match, run `git merge origin/main`, or run `git pull`, which performs the fetch and the merge together. Fetch is the cautious half of pull.

4.2.10. Cloning an existing repository

Publishing sends a local repository up to GitHub. **Cloning** does the reverse: it copies a repository from GitHub down to a new folder on your machine. This is how you obtain a collaborator's project, or your own project on a second computer. With plain Git you pass the URL:

```
$ git clone https://github.com/rgt47/baseline-summary.git
Cloning into 'baseline-summary'...
remote: Enumerating objects: 15, done.
```


Draft, November 13, 2026. Content may change.

```
remote: Total 15 (delta 3), reused 15 (delta 3)
Receiving objects: 100% (15/15), 2.1 KiB, done.
Resolving deltas: 100% (3/3), done.
```

`gh` offers a shorter form that needs only the `owner/name`, and that authenticates automatically for private repositories:

```
$ gh repo clone rgt47/baseline-summary
Cloning into 'baseline-summary'...
```

A clone is not a download of the current files alone. It copies the **entire history**: every commit, branch, and tag, the complete `.git` directory. The clone is a fully independent repository, already wired to the source as its `origin` remote, so you can immediately `git push` and `git pull` against it. Cloning is the standard way any project reaches a new machine.

4.2.11. The README as the front page

A file named `README.md` in the top of a repository is special to GitHub: it is rendered, as formatted text, on the repository's landing page. It is the first thing any visitor sees, and for a research repository it should state what the analysis is, what the files are, and how to reproduce the results. The `.md` extension denotes **Markdown**, a plain- text formatting syntax; a leading `#` marks a heading, `-` marks a list item, and backticks mark code.

```
$ cat > README.md <<'EOF'
# Baseline Summary

Baseline characteristic summaries for the 2026 trial
enrollment cohort.

## Files

- `baseline.R` -- computes counts, mean age, and the sex
  distribution from the enrollment data.
```


Draft, November 13, 2026. Content may change.

```
## Reproducing

Place the enrollment CSV at `data-raw/enrollment.csv`, then
run `Rscript baseline.R`.
EOF
$ git add README.md
$ git commit -m 'Add README describing the analysis'
[main 6b7c8d9] Add README describing the analysis
 1 file changed, 14 insertions(+)
$ git push
```

Because a README is an ordinary tracked file, it is versioned like any other; edits to it are commits, and it stays synchronized through push and pull. On GitHub, the rendered README now heads the repository page.

4.2.12. Browsing a repository from the terminal

You need not open a browser to inspect a published repository. `gh repo view` prints its metadata and rendered README to the terminal:

```
$ gh repo view
rgt47/baseline-summary
Baseline characteristic summaries for the 2026 trial cohort.

# Baseline Summary
Baseline characteristic summaries for the 2026 trial
enrollment cohort.
...
View this repository on GitHub:
https://github.com/rgt47/baseline-summary
```

To open the same page in your web browser, add `--web`:

```
$ gh repo view --web
Opening github.com/rgt47/baseline-summary in your browser.
```

And `gh browse` opens the repository, or a specific file or line, in the browser directly, which is convenient for sending a colleague a link to exactly the code under discussion:

```
$ gh browse baseline.R
Opening github.com/rgt47/baseline-summary/blob/main/baseline.R
```

4.2.13. Issues as a tracked to-do list

An **issue** on GitHub is a titled, numbered, discussable note attached to a repository. Issues are how a project records what remains to be done and what has gone wrong: a planned analysis, a bug in a script, a reviewer's request. Unlike a private to-do list, issues are shared, searchable, and permanently linked to the repository. Create one from the terminal:

```
$ gh issue create \
    --title 'Add sensitivity analysis excluding site 4' \
    --body 'Reviewer 2 flagged that site 4 enrolled
late and may bias the baseline age summary. Add a
sensitivity analysis rerunning baseline.R with site 4
excluded.'
Creating issue in rgt47/baseline-summary

https://github.com/rgt47/baseline-summary/issues/1
```

The issue is now number 1. List the open issues:

```
$ gh issue list
Showing 1 of 1 open issue in rgt47/baseline-summary

ID  TITLE                                    LABELS  UPDATED
#1  Add sensitivity analysis excluding site 4         now
```

View the full text and discussion of one issue:

Draft, November 13, 2026. Content may change.

```
$ gh issue view 1
Add sensitivity analysis excluding site 4 #1
Open - rgt47 opened about 1 minute ago - 0 comments

  Reviewer 2 flagged that site 4 enrolled late and may
  bias the baseline age summary. Add a sensitivity
  analysis rerunning baseline.R with site 4 excluded.

View this issue on GitHub:
https://github.com/rgt47/baseline-summary/issues/1
```

When the work is done, close the issue, optionally with a comment recording how:

```
$ gh issue close 1 \
    --comment 'Added sensitivity analysis in commit 8d9e0f1.'
✓ Closed issue #1 (Add sensitivity analysis excluding site 4)
```

Framed this way, issues are a research log of open questions and their resolutions, kept beside the code rather than in a separate document that drifts out of date. Day 5 connects issues to pull requests, so that a commit can close an issue automatically.

4.3. Worked example: publishing the baseline-summary repository

We continue the `baseline-summary` repository from Days 1 through 3. It has several commits recording the growth of `baseline.R`, a `.gitignore` protecting the raw data, and a clean working tree. The task today is to publish it, add to it, and record a follow-up task as an issue.

First confirm you are authenticated. If this is your first time, run `gh auth login` and complete the browser flow described in the lecture; otherwise check the existing state:

Draft, November 13, 2026. Content may change.

```
$ gh auth status
github.com
  ✓ Logged in to github.com account rgt47 (keyring)
  - Active account: true
  - Git operations protocol: https
```

Before publishing, confirm that nothing sensitive is about to travel to GitHub. Review which files the history contains:

```
$ git log --oneline
9b0c1d2 Add sex distribution table
7e8f9a0 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
$ git ls-files
.gitignore
baseline.R
```

Only `.gitignore` and `baseline.R` are tracked; the raw data under `data-raw/` was never committed, exactly as the Day 1 discipline intended. It is safe to publish. Do so in one command:

```
$ gh repo create baseline-summary \
    --public --source=. --remote=origin --push
✓ Created repository rgt47/baseline-summary on GitHub
  https://github.com/rgt47/baseline-summary
✓ Added remote https://github.com/rgt47/baseline-summary.git
✓ Pushed commits to
  https://github.com/rgt47/baseline-summary.git
```

Verify that the remote was wired up and the history landed:

```
$ git remote -v
origin  https://github.com/rgt47/baseline-summary.git (fetch)
origin  https://github.com/rgt47/baseline-summary.git (push)
$ gh repo view
rgt47/baseline-summary
```


Draft, November 13, 2026. Content may change.

```
No description provided.

This repository does not have a README

View this repository on GitHub:
https://github.com/rgt47/baseline-summary
```

The repository is live but has no front page. Add a README, commit it, and push the new commit:

```
$ cat > README.md <<'EOF'
# Baseline Summary

Baseline characteristic summaries for the 2026 trial
enrollment cohort.

## Files

- `baseline.R` -- total count, mean age, sex distribution.

## Reproducing

Place the enrollment CSV at `data-raw/enrollment.csv`, then
run `Rscript baseline.R`.
EOF
$ git add README.md
$ git commit -m 'Add README describing the analysis'
[main 6b7c8d9] Add README describing the analysis
 1 file changed, 12 insertions(+)
$ git push
Enumerating objects: 4, done.
Writing objects: 100% (3/3), 421 bytes, done.
To https://github.com/rgt47/baseline-summary.git
   9b0c1d2..6b7c8d9  main -> main
```

Notice the push named neither remote nor branch: the upstream link that `gh repo create` established lets a bare `git push` do the right thing. Finally, record a follow-up analysis task as an issue, so it is not forgotten:

```
$ gh issue create \
    --title 'Report age as median (IQR) for skewed cohort' \
    --body 'Age is right-skewed; a mean may mislead. Add
median and interquartile range to baseline.R alongside the
existing mean.'
Creating issue in rgt47/baseline-summary

https://github.com/rgt47/baseline-summary/issues/1
```

The repository is now published, documented, and carrying a tracked task, all from the terminal. In a handful of commands the local work of Days 1 through 3 has become a shareable, backed-up, self-documenting research artifact.

4.4. Homework

Attempt each problem in your own terminal before checking the solution. These problems assume you have created a free GitHub account and authenticated `gh` with `gh auth login`. Delete any practice repositories afterwards with `gh repo delete <name>` if you wish to keep your account tidy.

1. **Confirm your tools.** Run the commands that print your `gh` version and your authentication status. Confirm that you are logged in to `github.com` and that the Git operations protocol is `https`.

2. **Publish a repository.** Create a local repository named `hw4-repo`, make at least two commits (for example, a `README.md` and a short R script), and publish it to GitHub as a public repository in a single `gh repo create` command that also pushes. Confirm with `git remote -v` that `origin` points to your new GitHub repository.

3. **Push a later commit.** In the repository from Problem 2, add a new line to your R script, commit it, and push it with a bare `git push` (no remote or branch named). Explain why naming the remote and branch was unnecessary.

Draft, November 13, 2026. Content may change.

4. **Public versus private.** Create a second repository `hw4-private` and publish it with `--private`. Using `gh repo view`, confirm its visibility. State one reason a biostatistician would choose `--private`, and explain why even a private GitHub repository is an inappropriate home for protected health information.

5. **Clone your own work.** In a different directory (for example, a temporary folder simulating a second computer), clone the repository from Problem 2 using `gh repo clone`. Run `git log --oneline` in the clone. Does the clone contain the full history or only the latest files? Confirm that the clone already has `origin` configured.

6. **Open and close an issue.** In the repository from Problem 2, open an issue titled `Add unit tests for the summary function` with a one-sentence body. List your open issues, view the issue by its number, then close it with a closing comment. Confirm with `gh issue list` that no open issues remain.

7. **Fetch versus pull.** In the clone from Problem 5, do nothing. In the original repository from Problem 2, make a new commit and push it. Return to the clone and run `git fetch`. Does `git log --oneline` in the clone show the new commit yet? Now run `git merge origin/main` (or `git pull`) and check again. Explain the difference in one or two sentences.

8. **Reconstruct what gh automated.** Create a local repository `hw4-manual` with one commit, and create an empty repository of the same name on GitHub with `gh repo create hw4-manual --public` (no `--source`). Then connect and publish it using only plain Git commands: `git remote add`, and `git push -u`. Confirm the result with `git remote -v` and `gh repo view`. What did the `-u` flag record, and how would you verify it?

4.5. Solutions

Problem 1.

Draft, November 13, 2026. Content may change.

```
gh --version
gh auth status
#> github.com
#>   ✓ Logged in to github.com account rgt47 (keyring)
#>   - Active account: true
#>   - Git operations protocol: https
```

If `gh auth status` reports that you are not logged in, run `gh auth login` and complete the browser flow, choosing GitHub.com, HTTPS, and login via web browser.

Problem 2.

```
mkdir hw4-repo
cd hw4-repo
git init
echo '# HW4 Repo' > README.md
git add README.md
git commit -m 'Add README'
echo 'x <- rnorm(100)' > sim.R
git add sim.R
git commit -m 'Add simulation script'
gh repo create hw4-repo --public --source=. --remote=origin --push
#> ✓ Created repository rgt47/hw4-repo on GitHub
#> ✓ Added remote https://github.com/rgt47/hw4-repo.git
#> ✓ Pushed commits to https://github.com/rgt47/hw4-repo.git
git remote -v
#> origin  https://github.com/rgt47/hw4-repo.git (fetch)
#> origin  https://github.com/rgt47/hw4-repo.git (push)
```

The single `gh repo create` created the GitHub repository, added it as `origin`, and pushed both commits.

Problem 3.

```
echo 'y <- rnorm(100)' >> sim.R
git commit -a -m 'Add second simulated variable'
git push
```


Draft, November 13, 2026. Content may change.

```
#> To https://github.com/rgt47/hw4-repo.git
#>    a1b2c3d..b2c3d4e  main -> main
```

Naming the remote and branch was unnecessary because `gh repo create --push` set the upstream of local `main` to `origin/main`. Once a branch has an upstream, a bare `git push` sends the current branch to its tracked remote branch by default. You would only name them explicitly to push somewhere other than the recorded upstream.

Problem 4.

```
mkdir hw4-private
cd hw4-private
git init
echo '# Private' > README.md
git add README.md
git commit -m 'Add README'
gh repo create hw4-private --private --source=. --remote=origin --pus
gh repo view
#> rgt47/hw4-private
#> ...
#> Visibility: private
```

A biostatistician chooses `--private` to keep unpublished analyses, draft manuscripts, or study code restricted to the research team before results are public. Even so, a private GitHub repository is an inappropriate home for protected health information: the data would reside on a third-party commercial server outside institutional control, access is governed by GitHub accounts rather than by the study's data- use agreement, and the entire commit history, including any data committed and later deleted, travels with the repository. Keep raw data out of the repository entirely, as established on Day 1.

Problem 5.

```
cd /tmp
gh repo clone rgt47/hw4-repo hw4-clone
cd hw4-clone
```

```
git log --oneline
#> b2c3d4e Add second simulated variable
#> a1b2c3d Add simulation script
#> 9f8e7d6 Add README
git remote -v
#> origin  https://github.com/rgt47/hw4-repo.git (fetch)
#> origin  https://github.com/rgt47/hw4-repo.git (push)
```

The clone contains the **full history**, all three commits, not merely the latest files. A clone copies the entire `.git` directory, so it is a complete, independent repository. It also comes with `origin` already configured to point at the repository it was cloned from, so `git push` and `git pull` work immediately.

Problem 6.

```
cd ~/hw4-repo
gh issue create \
  --title 'Add unit tests for the summary function' \
  --body 'The summary function has no tests; add coverage.'
#> https://github.com/rgt47/hw4-repo/issues/1
gh issue list
#> #1  Add unit tests for the summary function   now
gh issue view 1
#> Add unit tests for the summary function #1
#> Open - rgt47 opened about 1 minute ago
gh issue close 1 --comment 'Tests added in a later commit.'
#> ✓ Closed issue #1 (Add unit tests for the summary function)
gh issue list
#> No open issues in rgt47/hw4-repo
```

Creating an issue returns its URL and number; listing, viewing, and closing all reference that number. After closing, `gh issue list` shows no open issues, because by default it lists only open ones. (Add `--state closed` to see closed issues.)

Problem 7.

Draft, November 13, 2026. Content may change.

```
# In the original repository:
cd ~/hw4-repo
echo 'z <- rnorm(100)' >> sim.R
git commit -a -m 'Add third simulated variable'
git push

# In the clone:
cd /tmp/hw4-clone
git fetch
git log --oneline
#> b2c3d4e Add second simulated variable
#> a1b2c3d Add simulation script
#> 9f8e7d6 Add README
```

After `git fetch`, the clone's own `main` does **not** yet show the new commit; fetch downloaded it into `origin/main` but left the local branch untouched. Advancing the local branch requires a merge:

```
git merge origin/main
#> Updating b2c3d4e..c3d4e5f
#> Fast-forward
git log --oneline
#> c3d4e5f Add third simulated variable
#> b2c3d4e Add second simulated variable
#> ...
```

`git pull` would have done the fetch and the merge in one step. In short, fetch retrieves commits without changing your branch; pull retrieves and then merges them in.

Problem 8.

```
mkdir hw4-manual
cd hw4-manual
git init
echo '# Manual' > README.md
git add README.md
```


Draft, November 13, 2026. Content may change.

```
git commit -m 'Add README'
gh repo create hw4-manual --public
#> ✓ Created repository rgt47/hw4-manual on GitHub
git remote add origin https://github.com/rgt47/hw4-manual.git
git push -u origin main
#> branch 'main' set up to track 'origin/main'.
#>  * [new branch]      main -> main
git remote -v
#> origin  https://github.com/rgt47/hw4-manual.git (fetch)
#> origin  https://github.com/rgt47/hw4-manual.git (push)
```

Without `--source`, `gh repo create` made an empty GitHub repository and did not touch your local one, so you supplied the two steps `gh` normally automates: `git remote add` registered the URL under the name `origin`, and `git push -u` sent `main` and recorded `origin/main` as its upstream. The `-u` flag established the tracking relationship; verify it with `git branch -vv`, whose output annotates `main` with `[origin/main]`, or with `git status`, which now reports your branch as up to date with `origin/main`.

4.6. What's next

Day 5 turns publishing into collaboration. You will learn the fork-and-pull-request workflow that lets several people contribute to one repository without overwriting each other, opening and reviewing pull requests with `gh pr`, and resolving conflicts that arise when two contributors change the same lines remotely. You will also see Git through the RStudio IDE, and revisit `.gitignore` for R projects, `renv`, and data. Keep the `baseline-summary` repository you published today, since Day 5 collaborates on it.

Draft, November 13, 2026. Content may change.

5. Day 5: Collaboration and Reproducible Research

5.1. Learning objectives

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

- Explain why sharing analysis scripts by email fails, and how a shared Git remote solves the same problem.
- Distinguish the two collaboration models, namely the shared-repository model and the fork-and-pull-request model, and say when each is appropriate.
- Run the complete pull-request workflow from the terminal with the `gh` command-line tool: branch, push, open, review, and merge.
- Fork a repository with `gh repo fork` and keep the fork current with `gh repo sync`.
- Recognize a rejected (non-fast-forward) push and integrate remote work with `git pull`, resolving any conflict that results.
- Perform the same operations through the RStudio Git pane and the `usethis` helper functions.
- Write a `.gitignore` appropriate to an R analysis project, and state without hesitation why data with protected health information must never be committed.

5.2. Lecture

5.2.1. The collaboration problem

Two biostatisticians are assigned to the same trial. One writes the baseline table, the other writes the efficacy analysis, and both edit `baseline.R`.

Draft, November 13, 2026. Content may change.

Worked by email, the sequence is familiar, and it is doomed. The first analyst emails `baseline.R`. The second edits it and emails it back. The first, who has meanwhile made her own edits, now holds two divergent files with the same name and no way to combine them except by hand. Whoever saves last wins, and the other's work is silently lost. This is the third failure named in the preface, and it is a version-control failure.

A shared Git remote dissolves the problem. Both analysts clone the same GitHub repository. Each records commits locally, exactly as in the previous four days, and then pushes them to the shared remote. Git integrates concurrent work rather than overwriting it: when two people change different lines, Git merges the changes automatically; when they change the same lines, Git stops and asks a human to choose, using precisely the conflict-resolution skills from Day 3. Nothing is lost, and the history records who did what and when. The remaining question is one of process: how do collaborators propose, review, and integrate one another's changes in an orderly way? That process is the subject of today.

5.2.2. Two collaboration models

There are two established ways to collaborate through a Git host, and they differ in who is permitted to write to the shared repository.

The **shared-repository model** grants a small group of trusted collaborators direct push access to one repository. Each collaborator clones it, works on a branch, pushes that branch, and opens a pull request for review before the branch is merged into `main`. This is the model for a research team whose members are known to one another, for example the biostatistics group on a single trial. It is simple, and it keeps all work in one place.

The **fork-and-pull-request model** does not require the contributor to have write access. The contributor creates a personal copy of the repository, called a **fork**, under their own account, pushes branches to that fork, and opens a pull request asking the maintainers of the original repository to pull in the change. This is the model of open source, where anyone may propose a change but only maintainers may accept it. In a biostatistics setting it fits contributing to an external package, or to a consortium repository where you are not a core member.

Draft, November 13, 2026. Content may change.

The two models share the same unit of proposal, the pull request, and differ only in where the contributor's branch lives: in the shared repository itself, or in a fork. Learn the pull-request workflow once and both models follow.

Check your understanding: which model?

Your three-person biostatistics team maintains a private GitHub repository for an ongoing trial. All three of you are named collaborators with push access. Should you fork the repository to add an interim-analysis script, or work on a branch within the shared repository?
Work on a branch within the shared repository. Forking is for contributors who lack write access, typically outside contributors to a project they do not maintain. Because all three of you have push access to one repository, the shared-repository model is simpler and keeps every branch, issue, and pull request in a single place. You would create a branch such as `interim-analysis`, push it, and open a pull request for a colleague to review before it merges into `main`. Reserve forking for the case where you are proposing a change to a repository you cannot push to.

5.2.3. The pull-request workflow with gh

A **pull request** (PR) is a request to merge one branch into another, wrapped in a page for discussion and review. It is a GitHub feature, not a Git feature, so we drive it with the official GitHub command-line tool, `gh`, introduced on Day 4. Assume you have cloned the shared repository and authenticated with `gh auth login`. The workflow has four steps: branch and push, open, review, merge.

Step one: branch, commit, and push. Create a feature branch with the Day 3 commands, make your commits, then push the branch to the remote. The first push of a new branch needs `-u` to set its upstream:

```
$ git switch -c add-sex-summary
Switched to a new branch 'add-sex-summary'
$ echo 'sex_table <- table(dat$sex)' >> baseline.R
$ git commit -a -m 'Add sex distribution to baseline summary'
```


Draft, November 13, 2026. Content may change.

```
[add-sex-summary 1a2b3c4] Add sex distribution to baseline
 1 file changed, 1 insertion(+)
$ git push -u origin add-sex-summary
Enumerating objects: 5, done.
...
To github.com:rgt47/baseline-summary.git
 * [new branch]      add-sex-summary -> add-sex-summary
branch 'add-sex-summary' set up to track 'origin/add-sex-summary'.
```

The `-u` flag (short for `--set-upstream`) links your local branch to the remote branch, so that later `git push` and `git pull` on this branch need no arguments.

Step two: open the pull request. With the branch on the remote, `gh pr create` opens a PR proposing to merge it into `main`:

```
$ gh pr create --title 'Add sex distribution to baseline' \
    --body 'Adds a sex frequency table to baseline.R for
the Table 1 summary. No change to existing outputs.'

Creating pull request for add-sex-summary into main in
rgt47/baseline-summary

https://github.com/rgt47/baseline-summary/pull/7
```

`gh` prints the URL of the new pull request; here it is number 7. If you would rather have `gh` fill the title and body from your commit messages, use `--fill`:

```
$ gh pr create --fill
Creating pull request for add-sex-summary into main in
rgt47/baseline-summary

https://github.com/rgt47/baseline-summary/pull/7
```

Step three: inspect and review. A reviewer, typically a colleague, examines the proposed change before it merges. List the open pull requests:

Draft, November 13, 2026. Content may change.

```
$ gh pr list
Showing 1 of 1 open pull request in rgt47/baseline-summary

ID   TITLE                                BRANCH           CREATED
#7   Add sex distribution to baseline   add-sex-summary  about 2 min
```

View a pull request's description and discussion in the terminal, or open it in a browser:

```
$ gh pr view 7
Add sex distribution to baseline #7
Open  jane-investigator wants to merge 1 commit into main from add-s

  Adds a sex frequency table to baseline.R for the Table 1
  summary. No change to existing outputs.
```

```
$ gh pr view 7 --web
Opening github.com/rgt47/baseline-summary/pull/7 in your browser.
```

Read the actual code change with `gh pr diff`:

```
$ gh pr diff 7
diff --git a/baseline.R b/baseline.R
index 8f3a1a2..1a2b3c4 100644
--- a/baseline.R
+++ b/baseline.R
@@ -1,3 +1,4 @@
 dat <- read.csv("data-raw/enrollment.csv")
 n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
+sex_table <- table(dat$sex)
```

To run or test the proposed code before approving it, check the branch out locally. `gh pr checkout` fetches the pull request's branch and switches to it, even for a pull request that originates from someone else's fork:

```
$ gh pr checkout 7
remote: Enumerating objects: 5, done.
...
Switched to branch 'add-sex-summary'
```

Having read and, if appropriate, run the change, record a review verdict. `gh pr review` offers three verdicts:

```
$ gh pr review 7 --approve \
    -b 'Confirmed the table renders; approving.'
```

```
$ gh pr review 7 --comment \
    -b 'Consider na.rm handling for missing sex.'
```

```
$ gh pr review 7 --request-changes \
    -b 'Please add a comment naming the source column.'
```

An `--approve` clears the pull request to merge; a `--comment` leaves feedback without a verdict; a `--request-changes` blocks the merge until the author pushes a fix. If changes are requested, the author simply commits and pushes again to the same branch, and the pull request updates in place; there is no need to open a new one.

Step four: merge. Once approved, merge the pull request and delete its now-redundant branch:

```
$ gh pr merge 7 --squash --delete-branch
Squashed and merged pull request #7 (Add sex distribution to baseline)
Deleted branch add-sex-summary and switched to branch main
```

GitHub offers three merge strategies, and `gh pr merge` takes one of `--merge`, `--squash`, or `--rebase`. A plain `--merge` creates a merge commit that preserves every commit on the branch. `--rebase` replays the branch's commits onto `main` individually, with no merge commit. `--squash` combines all of the branch's commits into a single commit on `main`. For most analysis work, prefer `--squash`: a feature branch often accumulates small 'fix typo'

Draft, November 13, 2026. Content may change.

and 'try again' commits whose individual history is noise, and squashing collapses them into one clean, reviewable entry on `main`. The `--delete-branch` flag tidies up the merged branch on the remote, which is good hygiene once its work has landed.

> **Tip**
>
> You do not have to remember the pull-request number. Run any `gh pr` command from a checked-out branch and `gh` infers the pull request associated with that branch, so `gh pr view`, `gh pr diff`, and `gh pr merge` all work with no number when you are on the branch in question.

5.2.4. Keeping your local main current

After a pull request merges on GitHub, your collaborators' `main` and the remote `main` have moved ahead of the copy on your machine. Before starting new work, update your local `main`:

```
$ git switch main
Switched to branch 'main'
$ git pull
Updating 8f3a1a2..9c8b7a6
Fast-forward
 baseline.R | 1 +
 1 file changed, 1 insertion(+)
```

`git pull` fetches the new commits from the remote and integrates them. Here the integration is a **fast-forward**: your local `main` had no commits of its own, so Git simply advances your `main` pointer to match the remote. Make `git switch main` followed by `git pull` the first thing you do each working session, so that you always branch from current work.

5.2.5. Forking and syncing a fork

When you lack write access to a repository, fork it. One command creates the fork under your account and clones it locally:

Draft, November 13, 2026. Content may change.

```
$ gh repo fork biostat-consortium/trial-templates --clone
✓ Created fork rgt47/trial-templates
✓ Cloned fork
✓ Added remote origin pointing to your fork
✓ Added remote upstream pointing to biostat-consortium/trial-templates
```

`gh` configures two remotes for you: `origin` is your fork, which you can push to, and `upstream` is the original repository, which you cannot. You work on branches, push them to `origin`, and open pull requests against `upstream`; `gh pr create` detects the fork relationship and targets the upstream repository automatically.

The original repository keeps moving while your fork sits still, so a fork drifts out of date. Bring it current with `gh repo sync`, which pulls the upstream changes into your fork:

```
$ gh repo sync rgt47/trial-templates
✓ Synced the "main" branch from biostat-consortium/trial-templates to r
```

Sync your fork before starting each new contribution, for the same reason you pull `main` before starting new work: so that your change is built on the current state and merges cleanly.

5.2.6. When your push is rejected

Suppose you and a colleague both start from the same `main`, both commit locally, and your colleague pushes first. When you then push, Git refuses:

```
$ git push
To github.com:rgt47/baseline-summary.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to
'github.com:rgt47/baseline-summary.git'
hint: Updates were rejected because the remote contains work
hint: that you do not have locally. This is usually caused by
hint: another repository pushing to the same ref.
```


Draft, November 13, 2026. Content may change.

This is a **non-fast-forward** rejection. It is not an error in the sense of something broken; it is Git protecting your colleague's commit from being overwritten. The remote `main` has a commit you do not have, so Git cannot simply advance the pointer. You must first integrate the remote work into your local branch, then push the combined result:

```
$ git pull
```

If your change and your colleague's change touch different lines, `git pull` merges them automatically and you may push the result. If they touch the same lines, `git pull` stops with a merge conflict, exactly as a local `git merge` does on Day 3:

```
Auto-merging baseline.R
CONFLICT (content): Merge conflict in baseline.R
Automatic merge failed; fix conflicts and then commit the result.
```

Resolve it with the same procedure you already know. Open the conflicted file, find the marked region, and edit it to the correct combined content:

```
<<<<<<< HEAD
mean_age <- mean(dat$age, na.rm = TRUE)
=======
median_age <- median(dat$age, na.rm = TRUE)
>>>>>>> 9c8b7a6...
```

Delete the markers and keep the lines that belong, here both of them:

```
mean_age <- mean(dat$age, na.rm = TRUE)
median_age <- median(dat$age, na.rm = TRUE)
```

Then stage the resolved file and complete the merge with a commit, after which the push succeeds:

```
$ git add baseline.R
$ git commit -m 'Merge remote main; keep mean and median age'
$ git push
```


Draft, November 13, 2026. Content may change.

The only new element compared with Day 3 is the remote context; the conflict itself is identical, and so is its resolution.

> **Tip**
>
> Some teams prefer a linear history without merge commits. In that case pull with `git pull --rebase`, which replays your local commits on top of the fetched remote commits instead of creating a merge commit. Conflicts are resolved the same way, staged with `git add`, and the rebase continued with `git rebase --continue`. The default `git pull` is sufficient for this book; `git pull --rebase` is a refinement, and rebasing is treated properly in the companion *Biostatistics Practicum* volume.

> **Warning**
>
> When a push is rejected, do not reach for `git push --force`. Forcing overwrites the remote branch with your local one and discards whatever your collaborator pushed, which is the very data loss version control exists to prevent. On a shared branch, force-pushing is destructive and can erase a colleague's committed work without warning. The correct response to a rejected push is always to `git pull` first, integrate, and push the combined result. Reserve any form of forced push for branches only you use, and even then prefer the safer `git push --force-with-lease`, discussed in the *Practicum*.

5.2.7. Git through the RStudio IDE

Many biostatisticians live in RStudio rather than a terminal, and RStudio exposes the same Git operations through a graphical pane. When a project is an RStudio Project (an `.Rproj` file) and a Git repository, a **Git** tab appears in the top-right pane. Everything it offers maps directly onto commands you already know:

- Each changed file has a checkbox. Ticking it is `git add`; the box is the staging area made visible.

Draft, November 13, 2026. Content may change.

- The **Commit** button opens a window showing the diff of staged changes and a box for the commit message. Clicking **Commit** runs `git commit`.
- **Push** and **Pull** buttons run `git push` and `git pull` against the configured remote.
- A **New Branch** button runs `git branch` and `git switch`, and a branch dropdown switches between branches.
- The **History** button (the clock icon) shows the commit log with per-commit diffs, the graphical form of `git log`.
- Diff markers and merge-conflict markers appear in the same `<<<<<<<`, `=======`, `>>>>>>>` form, edited directly in the source editor.

The pane is a convenience layer over the identical Git machinery; nothing it does is unavailable from the terminal, and understanding the terminal commands is what makes the buttons legible.

The `usethis` package supplies R functions that automate the setup and pull-request steps from within an R session. These are convenience wrappers over the same Git and GitHub operations taught all week, callable at the R console:

```
usethis::use_git()
usethis::use_github()
usethis::create_github_token()
usethis::pr_init('add-sex-summary')
usethis::pr_push()
```

`use_git()` initializes a repository and makes the first commit, the equivalent of `git init` plus an initial `git add` and `git commit`. `use_github()` creates a GitHub repository and connects it as the remote, the equivalent of `gh repo create` followed by setting the remote and pushing. `create_github_token()` opens the GitHub page for minting a personal access token, which `usethis` then stores for authenticated operations. `pr_init()` creates and switches to a new branch, and `pr_push()` pushes that branch and opens the pull-request page in your browser. They are a smooth on-ramp for analysts who prefer to stay in R, and they do nothing you could not do with `git` and `gh`.

Draft, November 13, 2026. Content may change.

5.2.8. A .gitignore for R projects

Day 1 introduced `.gitignore`; an R analysis project has a specific set of paths that should never be committed. A realistic `.gitignore` for such a project reads:

```
# RStudio and R session files
.Rproj.user/
.Rhistory
.RData
.Rapp.history

# Rendered output
*.html
*.pdf

# renv: track the lockfile, ignore the library
renv/library/
renv/staging/

# Data with protected health information: never commit
data-raw/
*.csv
```

A few of these deserve comment. The `.Rproj.user/` directory holds machine-specific RStudio state and is never shared; the `.Rproj` file itself, by contrast, is small and useful and is normally committed, which is why it is absent from the list. The `renv` lines encode an important distinction: you commit `renv.lock`, the manifest that records exact package versions, but you ignore `renv/library/`, the large installed library that `renv` can rebuild from the lockfile on any machine. This is how a project pins its dependencies without committing thousands of package files; `renv` in depth is a topic of the *Practicum*.

The final block is the one that matters most. Raw data in a clinical trial routinely contains protected health information, and committing it, even once, even to a private repository, places it permanently in the history where a later `git clone` will reproduce it in full. The Day 1 warning bears

Draft, November 13, 2026. Content may change.

repeating with emphasis: write these ignore rules before any data file is ever staged. A `.gitignore` committed as the first commit protects the repository from its first moment; one written after a `data-raw/` folder has already been committed does nothing to remove what is already in the history.

> **Warning**
>
> Committing protected health information to Git is a reportable data breach, not a tidiness problem, and adding the file to `.gitignore` afterwards does not undo it: the data remains in every past commit and in every clone. Removing it from history requires rewriting the repository with tools covered in the *Practicum*, and if the repository was ever pushed, you must assume the data is compromised and follow your institution's breach procedure. The only reliable defense is prevention: a `.gitignore` that excludes data, committed before any data enters the project.

5.2.9. Best-practices recap

The week reduces to a small set of habits that, held consistently, deliver the reproducibility and collaboration that motivated it:

- Commit early and often. A commit is cheap; a lost afternoon of unrecorded work is not.
- Make each commit one conceptual change, so that history is a sequence of legible steps rather than a pile of mixed edits.
- Write commit messages that state what changed and why, in the imperative mood, as Day 2 taught.
- Never commit data with protected health information, and never commit credentials or tokens. Guard both with `.gitignore` written first.
- Branch for every feature or analysis, keeping `main` always in a working state.
- Open a pull request and have a colleague review before merging into `main`. Review is where errors are caught before they reach the shared history.

- Pull before you start and pull before you push, so that your work is always built on current shared state.

> Check your understanding: squash versus merge
>
> Your feature branch has five commits, three of which are `fix typo`, `oops`, and `try again`. You are about to merge it into `main`. Which merge strategy keeps the history of `main` cleanest, and what becomes of those five commits?
> Use `gh pr merge --squash`. The squash strategy combines all five commits on the branch into a single new commit on `main`, so the noise of `fix typo`, `oops`, and `try again` disappears from the shared history and `main` gains one clean, well-described entry for the whole feature. A plain `--merge` would instead preserve all five commits plus a merge commit, carrying the noise onto `main`; `--rebase` would replay all five individually with no merge commit, which also keeps the noise. For analysis work, where branch commits are often exploratory, squash gives the most readable `main`. Add `--delete-branch` to remove the merged branch afterwards.

5.3. Worked example: a full collaborative cycle

We return one last time to the `baseline-summary` repository, which Day 4 published to GitHub at `github.com/rgt47/baseline-summary`. A colleague has asked for a median-age summary to accompany the mean. We deliver it through a complete pull-request cycle: branch, push, open, review, merge, and sync.

Begin from a current `main`. The first act of any session is to pull, so that we branch from the latest shared state:

```
$ git switch main
Already on 'main'
$ git pull
Already up to date.
```

Create a feature branch and make the change on it:

Draft, November 13, 2026. Content may change.

```
$ git switch -c add-median-age
Switched to a new branch 'add-median-age'
$ echo 'median_age <- median(dat$age, na.rm = TRUE)' >> baseline.R
$ git commit -a -m 'Add median age to baseline summary'
[add-median-age 4d5e6f7] Add median age to baseline summary
 1 file changed, 1 insertion(+)
```

Push the branch to the remote, setting its upstream:

```
$ git push -u origin add-median-age
Enumerating objects: 5, done.
...
To github.com:rgt47/baseline-summary.git
 * [new branch]      add-median-age -> add-median-age
branch 'add-median-age' set up to track 'origin/add-median-age'.
```

Open the pull request, letting `gh` fill the title and body from the commit message:

```
$ gh pr create --fill
Creating pull request for add-median-age into main in
rgt47/baseline-summary

https://github.com/rgt47/baseline-summary/pull/8
```

Confirm it exists and read the proposed change:

```
$ gh pr list
Showing 1 of 1 open pull request in rgt47/baseline-summary

ID   TITLE                            BRANCH          CREATED
#8   Add median age to baseline sum.  add-median-age  about 1 minute
```

```
$ gh pr diff 8
diff --git a/baseline.R b/baseline.R
index 1a2b3c4..4d5e6f7 100644
```

```
--- a/baseline.R
+++ b/baseline.R
@@ -2,3 +2,4 @@ dat <- read.csv("data-raw/enrollment.csv")
 n_total <- nrow(dat)
 mean_age <- mean(dat$age, na.rm = TRUE)
 sex_table <- table(dat$sex)
+median_age <- median(dat$age, na.rm = TRUE)
```

A reviewer checks the change out, confirms it runs, and records an approval:

```
$ gh pr checkout 8
Switched to branch 'add-median-age'
$ gh pr review 8 --approve \
    -b 'Runs cleanly; median complements the mean. Approving.'
```

With the approval recorded, merge the pull request, squashing its commits into one and deleting the branch:

```
$ gh pr merge 8 --squash --delete-branch
Squashed and merged pull request #8 (Add median age to baseline summary
Deleted branch add-median-age and switched to branch main
```

Finally, update the local `main` so that it reflects the merge, and confirm the clean state:

```
$ git pull
Updating 9c8b7a6..2f3a4b5
Fast-forward
 baseline.R | 1 +
 1 file changed, 1 insertion(+)
$ git log --oneline
2f3a4b5 Add median age to baseline summary (#8)
9c8b7a6 Add sex distribution to baseline summary (#7)
8f3a1a2 Add total count and mean age
a1b2c3d Add gitignore before any data enters repo
```


Draft, November 13, 2026. Content may change.

The history on `main` is a clean sequence of reviewed, squashed changes, each tied to the pull request that introduced it by the `(#8)` and `(#7)` references GitHub appends automatically. Nothing was emailed, nothing was overwritten, and every change passed a colleague's review before it entered the shared record. This is the workflow a practicing biostatistics team runs every day.

5.4. Homework

Attempt each problem before reading its solution. For problems requiring a remote, use a private test repository on your own GitHub account so that you can experiment freely. Several problems ask you to simulate a second collaborator; you can do this convincingly with a second clone of the same repository in a separate directory.

1. **Confirm your tooling.** Run the commands that report your `gh` version and your authentication status. Confirm that you are logged in to `github.com` and that your token has the scopes `gh` needs. If you are not authenticated, authenticate.

2. **A branch and a pull request.** In a clone of a test repository, create a branch `add-readme-note`, append a line to `README.md`, commit it, and push the branch with its upstream set. Then open a pull request with a title and body of your choosing. What number does `gh` assign it, and what URL does it print?

3. **Inspect a pull request.** Using the pull request from problem 2, run the commands that list open pull requests, view the pull request's description in the terminal, and print its diff. What does each command show that the others do not?

4. **Review and merge.** Approve the pull request from problem 2 with a review comment, then merge it with the squash strategy and delete its branch. Afterwards, switch to `main` and update it. What integration does `git pull` report, and why is it a fast-forward?

5. **A rejected push.** Simulate two collaborators with two clones of the same repository in two directories. In the first clone, commit a change to `main` and push it. In the second clone, commit a different

Draft, November 13, 2026. Content may change.

change to the same line of the same file and try to push. Record the exact message Git prints. Why is the push rejected, and what is the correct next command?

6. **Resolve a pulled conflict.** Continuing from problem 5, run `git pull` in the second clone, resolve the resulting conflict so that both collaborators' intent is preserved, commit the merge, and push. Show the conflict markers you resolved and the final content you kept.

7. **Fork and sync.** Fork any public repository with `gh repo fork --clone`. Identify the two remotes `gh` configured and state which one you may push to. Then run `gh repo sync` on your fork and explain what it did.

8. **An R .gitignore.** Write a `.gitignore` for an R analysis project that excludes RStudio session files, the `renv` library but not its lockfile, rendered HTML and PDF, and a `data-raw/` directory of protected data. Create files matching each pattern, confirm with `git status` that all are ignored, and explain in one sentence why the `data-raw/` rule is the most important line in the file.

5.5. Solutions

Problem 1.

```
gh --version
gh auth status
```

`gh auth status` reports the host, the account you are logged in as, and the token's scopes, which should include `repo` and `read:org` for the operations in this chapter. If it reports that you are not logged in, run `gh auth login` and follow the prompts, as on Day 4.

Problem 2.

Draft, November 13, 2026. Content may change.

```
git switch -c add-readme-note
echo '' >> README.md
echo 'Maintained by the biostatistics group.' >> README.md
git commit -a -m 'Note maintaining group in README'
git push -u origin add-readme-note
gh pr create --title 'Note maintaining group' \
  --body 'Adds a maintainer line to the README.'
#> https://github.com/<you>/<repo>/pull/1
```

`gh` assigns the next available pull-request number, here 1 in a fresh repository, and prints the full URL to the pull request, which you can open in a browser or pass to `gh pr view`.

Problem 3.

```
gh pr list
#> shows the number, title, branch, and age of each open PR
gh pr view 1
#> shows this PR's title, state, author, and body text
gh pr diff 1
#> shows the actual line-by-line code change
```

`gh pr list` gives the inventory of open pull requests without detail; `gh pr view` gives one pull request's description and discussion but not its code; `gh pr diff` gives the code change itself. Together they answer 'what is open?', 'what does this one propose, in prose?', and 'what exactly does it change?'.

Problem 4.

```
gh pr review 1 --approve -b 'Looks good; approving.'
gh pr merge 1 --squash --delete-branch
git switch main
git pull
#> Updating a1b2c3d..d4e5f6a
#> Fast-forward
#>  README.md | 2 +-
#>  1 file changed, 1 insertion(+), 1 deletion(-)
```


Draft, November 13, 2026. Content may change.

`git pull` reports a **fast-forward**. Your local `main` had no commits of its own since the branch was created, so integrating the merged change requires only advancing the `main` pointer to the remote's commit; no merge commit is needed. A fast-forward is the common, frictionless case that results from pulling other people's already-merged work into an unchanged local branch.

Problem 5.

```
# In clone A
echo 'a_line <- 1' >> shared.R
git commit -a -m 'A: set a_line'
git push
# In clone B, having edited the same line differently
echo 'a_line <- 2' >> shared.R
git commit -a -m 'B: set a_line'
git push
#> ! [rejected]        main -> main (fetch first)
#> error: failed to push some refs to '...'
#> hint: Updates were rejected because the remote contains
#> hint: work that you do not have locally.
```

The push from clone B is rejected as a non-fast-forward because the remote `main` now holds clone A's commit, which clone B does not have; advancing the remote to B's commit would discard A's. Git will not do that silently. The correct next command is `git pull`, to integrate A's commit into B before pushing the combined result. It is never `git push --force`, which would destroy A's commit.

Problem 6.

```
# In clone B
git pull
#> CONFLICT (content): Merge conflict in shared.R
```

Open `shared.R` and find the conflict:

Draft, November 13, 2026. Content may change.

```
<<<<<<< HEAD
a_line <- 2
=======
a_line <- 1
>>>>>>> a1b2c3d...
```

Resolve it to preserve both collaborators' intent, here by keeping both assignments under distinct names, then remove the markers:

```
a_line_b <- 2
a_line_a <- 1
```

Complete the merge and push:

```
git add shared.R
git commit -m 'Merge: keep both a_line values'
git push
```

The push now succeeds because clone B's `main` contains both commits plus the merge that reconciles them, so advancing the remote no longer discards anything.

Problem 7.

```
gh repo fork <owner>/<repo> --clone
cd <repo>
git remote -v
#> origin    git@github.com:<you>/<repo>.git (push)
#> upstream  git@github.com:<owner>/<repo>.git (fetch)
gh repo sync <you>/<repo>
#> ✓ Synced the "main" branch from <owner>/<repo> to <you>/<repo>
```

`gh` configures `origin`, your fork, and `upstream`, the original repository. You may push only to `origin`; you propose changes to `upstream` through pull requests. `gh repo sync` pulled the latest commits from the upstream `main` into your fork's `main`, bringing a fork that had drifted out of date back into agreement with the original.

Draft, November 13, 2026. Content may change.

Problem 8.

```
cat > .gitignore <<'EOF'
.Rproj.user/
.Rhistory
.RData
*.html
*.pdf
renv/library/
data-raw/
EOF
mkdir -p .Rproj.user renv/library data-raw
touch .RData report.html manuscript.pdf data-raw/phi.csv
touch renv/library/placeholder
git init
git add .gitignore
git status
#> On branch main
#> Changes to be committed:
#>   new file:   .gitignore
#> (no other files listed)
```

Only `.gitignore` is stageable; every other created path matches an ignore rule and is suppressed. Note that `renv.lock`, were it present, would not match `renv/library/` and so would remain trackable, which is the intended behavior: pin versions by committing the lockfile, ignore the rebuildable library. The `data-raw/` line is the most important because that directory holds raw clinical data with protected health information, which committing would turn into a permanent, cloneable breach of confidentiality.

5.6. What's next

There is no Day 6, because you have finished the boot camp. In five days you moved from never having made a commit to running the full collaborative workflow of a practicing biostatistics team: recording history, navigating and undoing it, branching and merging, publishing to GitHub,

Draft, November 13, 2026. Content may change.

and proposing, reviewing, and integrating changes through pull requests. The `baseline-summary` repository that began as an empty folder on Day 1 is now a reviewed, collaborative, reproducible record of an analysis. That is the whole point: your commit history is a lab notebook, and you now keep it.

What remains is the infrastructure that surrounds this everyday core, and it is the subject of the companion *Biostatistics Practicum* volume. There you will meet Git's internals and the reflog, interactive rebasing and history rewriting, dependency-pinned and containerized reproducibility with `renv` in depth and Docker, Quarto pipelines that render an analysis from data to manuscript in one command, continuous integration with GitHub Actions, and the version-control practices that regulated clinical-trial work and CDISC deliverables demand. If any of the language of R analysis in these worked examples was unfamiliar, the sibling boot camp *R for Biostatistics* covers the R language over an equivalent week, and the two are designed to be taken together.

For now, congratulations on completing the week. Keep the `baseline-summary` repository; it is the seed of your first reproducible project, and the habits you built on it are the ones that carry into every analysis you will publish.

Draft, November 13, 2026. Content may change.

References

Draft, November 13, 2026. Content may change.

Credits

The chapter list was constructed by surveying publicly available version-control teaching materials from major data-science and biostatistics courses, including Jenny Bryan's *Happy Git and GitHub for the useR*, the Software Carpentry version-control lessons, and university course modules at Cornell (INFO 5940), Carnegie Mellon (Statistical Computing), and American University (DATA 413/613), and adopting the topics common to the majority of them.

Cover artwork generated procedurally by the R script at `images/build-cover.R`. Watercolor palette: deep rust through terracotta and amber to pale sand, anchored on a brand rust `#8f3a1a` chosen to echo the Git project's signature orange. 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-bootcamp.rgtlab.org on Netlify, with continuous deployment via GitHub Actions from the `rgt47/git-bootcamp` 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 watercolor gradient anchored on `#8f3a1a` (the volume's brand rust) and overlaid with Avenir typography. Re-running the script regenerates the cover; the procedural approach makes the cover reproducible and editable in version control rather than dependent on a one-shot image-generation step. That the cover of a version-control textbook is itself a versioned, scripted artifact is deliberate.

Last rendered: see the build timestamp on the homepage.