Data Science Team Training

Stephen D. Turner, Ph.D.

2026-02-22

Table of contents

Preface

Public health has always been a data-driven enterprise, but the tools available to practitioners have changed dramatically. Spreadsheets that once required weeks of manual work can now be analyzed in seconds. Reports that used to mean emailing static PDFs can be replaced by live, interactive dashboards. And workflows that depended on one person's institutional knowledge can be documented, versioned, and shared across teams. The goal of this book is to help public health professionals take advantage of these tools.

This book grew out of the CSTE Data Science Team Training (DSTT) program, where project coach Dr. Stephen D. Turner works with public health agencies across the country to build data science capacity in the public health workforce. The material here includes code, resources, workshop notes, and practical guidance accumulated and refined over several years of coaching teams at local and state health departments.

The chapters cover the foundational practices that make data science work sustainable and collaborative in a public health setting: organizing your data well, managing projects and workflows, using version control to track your work, building dashboards to communicate findings, and using AI tools to write and debug code more efficiently. These are not cutting-edge research topics. They are the practical skills that separate ad hoc analyses from reproducible, maintainable work.

While this material was created with DSTT participants in mind, it is intended to be broadly useful to anyone working at the intersection of data science and public health, whether you are a current or former DSTT participant, a public health practitioner looking to strengthen your analytical skills, or someone new to data science in a public health context.

No single book can cover everything, and this one does not try to. Appendix A points to additional reading and training for topics that go deeper than what is covered here.

Part I

Technical

1 Version Control

Git is a distributed version control system that tracks changes to files over time, allowing you to review history, revert mistakes, and collaborate without overwriting each other's work. For data science teams, version control is essential for reproducibility (knowing exactly which code produced which results), collaboration (multiple people working on the same project safely), and auditability (a clear record of what changed and why). GitHub is the most widely used platform for hosting Git repositories and adds features like pull requests, code review, and issue tracking on top of Git's core capabilities.

1.1 GitHub Setup

1.1.1 Creating a GitHub Account

Visit github.com and sign up for a free account. The free tier includes unlimited public and private repositories and is sufficient for most team workflows.

1.1.2 Adding an SSH Key

SSH keys let you authenticate with GitHub without entering your username and password on every push or pull. To generate a key:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

Accept the default file location (`~/.ssh/id_ed25519`) and optionally set a passphrase. Then copy the public key:

```
cat ~/.ssh/id_ed25519.pub
```

In GitHub, go to **Settings → SSH and GPG keys → New SSH key**, paste the output, give it a descriptive title (e.g., your machine name), and save.

Verify the connection works:

```
ssh -T git@github.com
```

You should see a message like `Hi username! You've successfully authenticated`.

1.1.3 Cloning a Repository via SSH

When cloning a repository, prefer the SSH URL over HTTPS. On any GitHub repository page, click **Code** and select the **SSH** tab to get the URL.

```
git clone git@github.com:org/repo.git
```

Using SSH avoids repeated password prompts and works with SSH agent forwarding for server-based workflows.

1.1.4 Pulling and Pushing

After cloning, your local repository is linked to the remote (called `origin` by default). To sync your local branch with the latest changes from the remote:

```
git pull
```

After committing changes locally, push them to the remote:

```
git push
```

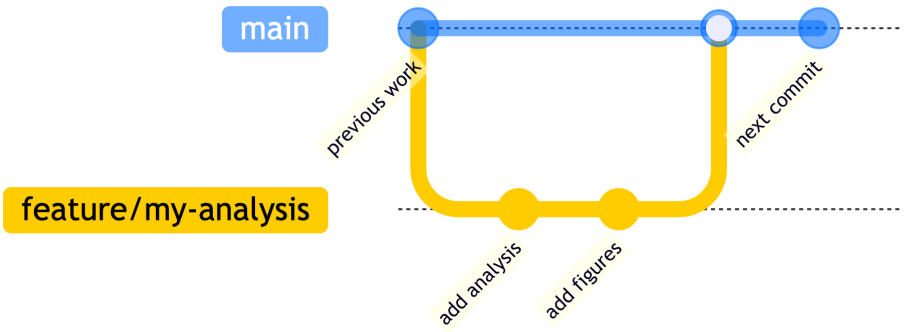
Git tracks the relationship between your local branch and the corresponding remote branch automatically, so these short-form commands will work once the tracking relationship is established.

1.2 Branching and Merging

Branches let multiple people work on different features or fixes simultaneously without interfering with each other. The `main` branch (or `master` in older repositories) typically represents the stable, production-ready state of the project.

1.2.1 Internal Organizational Workflow

In most team workflows, the `main` branch is protected — direct pushes are disabled and changes must go through a pull request (PR) with at least one reviewer. Developers create short-lived branches for each piece of work, then open a PR when ready.



Step 1: Create a branch from `main`

```
git checkout main
git pull
git checkout -b feature/my-analysis
```

Step 2: Make changes, stage, and commit

```
git add analysis.R writeup.qmd
git commit -m "Add initial exploratory analysis for Q1 data"
```

Step 3: Push the branch to GitHub

```
git push -u origin feature/my-analysis
```

The `-u` flag sets the upstream tracking branch so future `git push` and `git pull` commands work without specifying the remote and branch name.

Step 4: Open a pull request

On GitHub, navigate to the repository. A banner will appear prompting you to open a PR from your recently pushed branch. Click **Compare & pull request**, add a description, assign reviewers, and submit.

Step 5: Review, merge, and delete

A reviewer approves the PR and merges it into `main` via the GitHub UI. After merging, delete the branch on GitHub (there is a button on the merged PR page). Locally, clean up with:

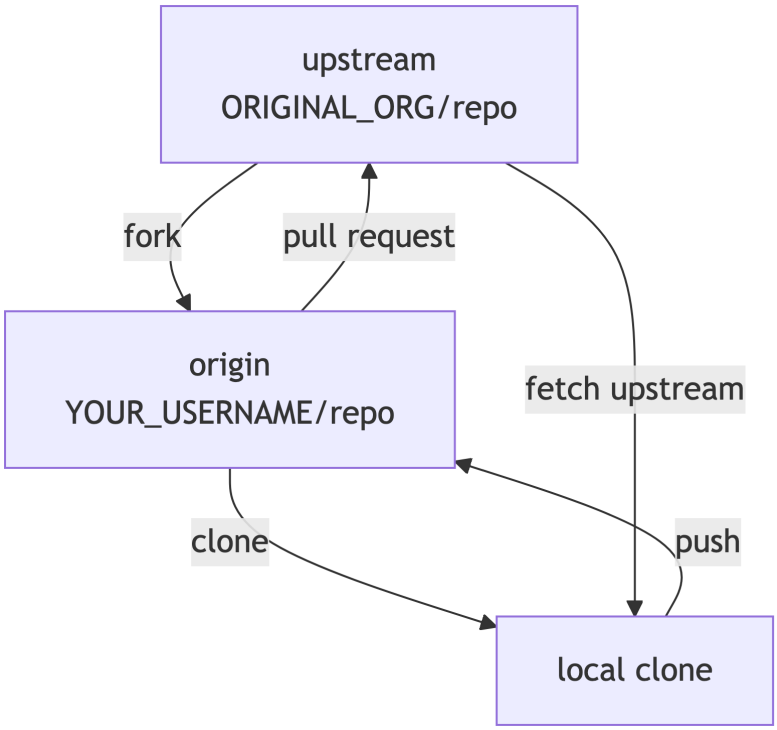
```
git checkout main
git pull
git branch -d feature/my-analysis
```

> **Tip**
>
> **Branch naming conventions** help the team understand what a branch is for at a glance. Common prefixes:
>
> - `feature/` — new functionality (e.g., `feature/survival-analysis`)
> - `fix/` — bug fixes (e.g., `fix/date-parsing-error`)
> - `analysis/` — exploratory or one-off analyses (e.g., `analysis/q2-cohort`)
> - `docs/` — documentation updates

> **Note**
>
> Keep branches short-lived — ideally merged within a few days. Long-running branches diverge significantly from `main`, making merges painful and increasing the likelihood of conflicts.

1.2.2 Contributing to External Repositories

When contributing to a repository you don't have write access to — such as an open-source tool or a public project from another team — the fork workflow is used instead. A fork is a personal copy of the repository under your GitHub account.



Step 1: Fork the repository

On the GitHub repository page, click **Fork** (top right) and choose your account as the destination.

Step 2: Clone your fork

```
git clone git@github.com:YOUR_USERNAME/repo.git
cd repo
```

Step 3: Add the original repository as `upstream`

```
git remote add upstream git@github.com:ORIGINAL_ORG/repo.git
```

Step 4: Branch, work, commit, and push to your fork

```
git checkout -b fix/typo-in-readme
# ... make changes ...
git add README.md
git commit -m "Fix typo in installation section"
git push -u origin fix/typo-in-readme
```

Step 5: Open a pull request to the upstream repository

On your fork's GitHub page, click **Contribute → Open pull request**. This creates a PR from your fork's branch into the original repository's `main` branch.

Step 6: Keep your fork in sync with upstream

As the original repository receives new commits, your fork will fall behind. Sync it with:

```
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

> **Note**
>
> In the fork workflow, `origin` refers to **your fork** and `upstream` refers to the **original repository**. Keeping these straight avoids accidentally pushing to or pulling from the wrong remote.

1.3 Git in Positron

Positron includes a built-in Source Control panel (the branching icon in the left sidebar, or `Ctrl+Shift+G` / `Cmd+Shift+G`) that provides a graphical interface for the most common Git operations.

Viewing changes

The Source Control panel lists all files with uncommitted changes. Files are grouped into **Staged Changes** and **Changes** (unstaged). Click any file to open a diff view showing exactly what was added or removed.

Staging files

Hover over a file and click the `+` icon to stage it, or click the `+` next to the **Changes** heading to stage all modified files at once. To unstage, click the `-` icon next to a staged file.

Committing

Type a commit message in the text box at the top of the Source Control panel and click the **Commit** button (or press `Ctrl+Enter` / `Cmd+Enter`). This is equivalent to `git commit -m "your message"`.

Branching

The current branch name appears in the status bar at the bottom of the window. Click it to open the branch picker, where you can switch to an existing branch or create a new one. This is equivalent to `git checkout` or `git checkout -b`.

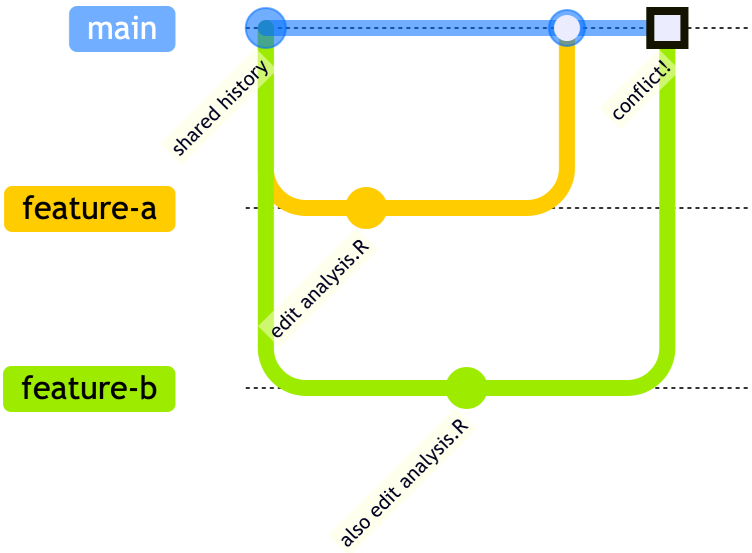
Pushing and pulling

The Source Control panel has **Push** and **Pull** buttons in its toolbar (the `...` menu or the sync icon in the status bar). The sync button performs a pull followed by a push in one action.

> **Tip**
>
> The integrated terminal in Positron (**Terminal → New Terminal**) is always available for Git operations that the UI doesn't expose — such as adding a remote, rebasing, cherry-picking, or any command requiring flags. The UI and terminal work on the same repository state, so you can mix both freely.

1.4 Merge Conflicts

A merge conflict occurs when two branches have made changes to the same lines of the same file, and Git cannot automatically determine which version to keep. Conflicts most commonly arise during `git merge`, `git rebase`, or when accepting a pull request that conflicts with recent changes to `main`.



1.4.1 What conflict markers look like

When a conflict occurs, Git edits the affected file to mark the conflicting regions:

```
<<<<<<< HEAD
result <- model |> predict(new_data)
=======
result <- model %>% predict(new_data)
>>>>>>> feature/update-pipe-syntax
```

- Everything between `<<<<<<< HEAD` and `=======` is the version from your current branch.
- Everything between `=======` and `>>>>>>>` is the version from the branch being merged in.

1.4.2 Resolving conflicts

1. Run `git merge <branch>` (or let a PR trigger the conflict). Git will report which files have conflicts.
2. Open each conflicted file. Positron highlights conflict regions with inline buttons: **Accept Current Change**, **Accept Incoming Change**, **Accept Both Changes**, or **Compare Changes**. Click the appropriate option or edit the file manually to produce the correct final version.
3. After resolving all conflicts in a file, save it and stage it:

```
git add path/to/resolved-file.R
```

4. Once all conflicted files are resolved and staged, complete the merge:

```
git commit
```

Git will pre-populate a commit message describing the merge; you can accept it as-is.

> **Warning**
>
> Never leave conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) in committed code. The file will be syntactically broken and the code will not run. Always verify the conflict is fully resolved before staging.

Tip

`git status` is your best friend during a conflict. It lists which files are in conflict (`both modified`), which are already resolved and staged, and what step to take next (commit or continue a rebase).

```
git status
```

2 Data Organization

Good data organization is foundational to reproducible, efficient research. Decisions made at the moment of data collection and storage ripple through every downstream step—analysis, visualization, sharing, and archiving. This chapter covers practical principles for organizing data in spreadsheets, naming files consistently, structuring project directories, and storing data in formats that work well with code.

2.1 Data Organization in Spreadsheets

Broman and Woo's "Data Organization in Spreadsheets" (Broman and Woo 2018) offers twelve practical recommendations for anyone who stores data in Excel or similar tools. The core philosophy: structure your data so that it can be read directly into R or Python without manual cleanup.

2.1.1 Be Consistent

Consistency is the single most important principle. Pick conventions and stick to them within a project:

- Use the same codes for categorical variables (e.g., always `"M"` and `"F"`, not sometimes `"Male"`)
- Use the same missing value code throughout (e.g., `NA`)
- Use the same variable names across files and time points
- Use the same date format everywhere

2.1.2 Make It a Rectangle

Raw data should be a single, clean rectangle:

- **Rows are observations** (subjects, samples, time points)
- **Columns are variables**
- **One header row** at the top with short, meaningful column names
- No merged cells, no blank rows used for visual separation, no summary rows mixed in

This is consistent with the principles of **tidy data** (Wickham 2014), where each variable forms a column, each observation forms a row, and each type of observational unit forms a table.

2.1.3 One Thing Per Cell

Each cell should contain exactly one value. Don't cram multiple pieces of information into a single cell (e.g., `"120/80"` for blood pressure—use two columns instead). Don't use a cell to store a note alongside a value.

2.1.4 Fill in All Cells

Don't leave cells blank to imply "same as above." Every cell should contain an explicit value. Use a consistent missing data code (e.g., `NA`) rather than leaving cells empty—blank cells are ambiguous: was the value missing, not collected, or zero?

2.1.5 Write Dates as YYYY-MM-DD

Use ISO 8601 format for all dates: `2024-03-15`. This format:

- Sorts correctly alphabetically
- Is unambiguous across locales (`03/04/2024` means different things in the US vs. Europe)
- Is not mangled by Excel (unlike many other date formats)

> **Warning**
>
> Excel silently converts many text strings to dates (e.g., the human gene name `SEPT1` becomes a date). Consider storing year, month, and day in separate columns if you're collecting data in Excel, or prefix dates with an apostrophe to force text storage.

2.1.6 Choose Good Names for Things

Apply consistent naming rules to columns, sheets, and files:

- No spaces—use underscores (`_`) or hyphens (`-`)
- No special characters (`@`, `#`, `%`, `(`, `)`)
- Short but meaningful (`participant_id`, not `p` or `the_participant_identifier_number`)
- All lowercase is a safe default
- Avoid starting names with numbers

2.1.7 No Calculations in Raw Data Files

Keep raw data raw. Don't add sum rows, averages, or derived columns to the data file—do that in your analysis script. The raw data file is a permanent record of what was collected; calculations belong in code that can be inspected, corrected, and rerun.

2.1.8 Don't Use Formatting as Data

Color-coding, bold text, or cell highlighting to indicate something (e.g., red = outlier, bold = reviewed) is invisible to code. Add an explicit column instead: `is_outlier` (`TRUE`/`FALSE`) or `review_status` ("reviewed", "pending").

2.1.9 Create a Data Dictionary

Every dataset should be accompanied by a data dictionary (also called a codebook) that documents:

- Variable name
- Description of what it measures
- Units
- Allowable values or range
- How missing values are coded

A simple spreadsheet or CSV with one row per variable works fine.

2.1.10 Save Data in Plain Text

Save the analysis-ready version of your data as CSV (comma-separated values), not as `.xlsx`. CSV files:

- Can be opened by any software
- Are readable in version control diffs
- Don't have hidden formatting or macros
- Are stable long-term

Keep the original Excel file if needed, but always export a clean CSV as the file you hand off to analysis.

2.1.11 Make Backups

Raw data should be treated as read-only and backed up in at least two places (e.g., local drive + cloud storage). Consider these tiers:

- **Primary:** Your working copy
- **Local backup:** External drive or NAS
- **Off-site/cloud:** Institutional storage, OneDrive, or similar

Version control (see Chapter 1) is excellent for code and small text files but is not ideal for large binary data files.

2.2 File Naming

Jenny Bryan's "How to Name Files" (Bryan 2015) lays out three principles that make file names work well for both humans and computers. See also her talk from NormConf 2022:

https://www.youtube.com/watch?v=ES1LTlnpLMk

2.2.1 Three Principles

1. Machine readable

File names should be parseable by code without heroics:

- No spaces (use `_` or `-` instead)
- No special characters (`&`, `*`, `#`, `(`, `)`, accented letters)
- No case sensitivity issues—stick to lowercase
- Use delimiters consistently so you can split on them: underscores between metadata fields, hyphens within a field

2. Human readable

The name should tell you what's in the file without opening it:

- Include a descriptive "slug" that summarizes the content
- `2024-03-15-enrollment-summary.csv` is better than `data_v3_FINAL.csv`

3. Plays well with default ordering

Files should sort into a useful order automatically:

- Use ISO 8601 dates (`YYYY-MM-DD`) at the start of the name so files sort chronologically
- Left-pad numbers with zeros: `fig-01.png`, `fig-02.png`, …, `fig-10.png` (not `fig-1.png`, `fig-2.png`, `fig-10.png`)

2.2.2 Recommended Pattern: `YYYY-MM-DD-slug`

For dated outputs (reports, snapshots, exports), the pattern `YYYY-MM-DD-descriptive-slug.ext` works well:

```
2024-03-15-enrollment-summary.csv
2024-06-01-adverse-events-cleaned.csv
2024-09-30-q3-interim-analysis.html
```

For numbered sequences (figures, scripts with a natural order), use zero-padded numbers:

```
01-import-data.R
02-clean-data.R
03-fit-models.R
04-make-figures.R
```

For files that don't change often and aren't dated, a simple descriptive slug is fine:

```
protocol.pdf
data-dictionary.xlsx
consent-form-v2.pdf
```

> **Tip**
>
> A good file name is essentially metadata. When you can glob a directory and parse the file names with code—extracting dates, conditions, or sample IDs—you've done it right.

2.3 Project Directory Structure

2.3.1 Simple project-oriented workflow

A consistent folder structure makes projects navigable by collaborators (and your future self). One convention is shown below:

```
project-name/
   data/
       .gitignore    # Don't commit data to version control.
       raw/          # original data -- never edited directly
```

```
    processed/    # cleaned, analysis-ready files
R/                # scripts and functions
output/
    .gitignore    # Don't commit large output files.
    figures/
    tables/
report.qmd
```

Key rules:

- **Raw data is sacred.** The `data/raw/` folder is read-only. Never overwrite or edit raw data files. Also, you don't want to commit large files to version control, so be careful with your `.gitignore`.
- **All data manipulation is scripted.** Cleaned data in `data/processed/` is generated by a script, not by hand.
- **One project = one directory.** Keep all files related to a project together.

The ProjectTemplate (R) and Cookiecutter Data Science (Python) tools can scaffold these structures automatically.

2.3.2 Version controlling code with data on a shared drive

Finally, it's not uncommon in public health settings to have a setup where you're working with collaborators version controlling code as described in Chapter 1, but the data lives on a secure shared drive that cannot be moved or copied into the repository. This creates a tension between reproducibility and security that can be resolved with careful project organization and path management.

Imagine that you have a distributed research team working with sensitive data that:

- Lives on a secure shared drive that cannot be moved
- Is organized in nested folders, for example, by state and year (e.g., `Lab/StateA/2023/`, `Lab/StateB/2024/`, etc.)
- Requires collaborative code development across Windows, Mac, and Linux machines
- Needs version control for reproducibility and collaboration

Storing data in a *database* would be preferable to a shared drive, but many public health agencies don't have the resources or infrastructure to set up and maintain databases. So how can we manage this?

2.3.2.1 The Conflict

Traditional approaches create impossible trade-offs:

Option 1: One code repository without proper path management

- Forces hardcoded paths: `setwd("Z:/Lab/StateA/2024")`
- Breaks on different operating systems (Z:/ vs /Volumes/)
- Breaks when different team members have different drive mappings
- Violatesproject-oriented workflow principles
- Code is not reproducible

Option 2: Code repositories inside each data folder

- Creates dozens of scattered git repositories
- Makes code reuse nearly impossible
- Version control nightmare (which repo has what function?)
- Accidental data commits to version control
- Multiple people doing git operations in the same shared folder causes conflicts. Not merge conflicts, but file locking and permission issues that can break everyone's workflow.

Option 3: Move data into code repository

- Violates data security requirements
- Data cannot leave the secure shared drive
- Makes git repository enormous and slow
- Not feasible

2.3.2.2 A Better Way: Centralized Code Repository with Configurable Paths

One way to manage this is with a **single centralized code repository with configurable paths**:

- One repository for all analysis code (easy to find, maintain, collaborate)
- Reproducible across team members with different operating systems
- Secure: data never leaves the shared drive or enters version control
- Flexible: works with nested folder structures
- Collaborative: proper git workflow without conflicts
- Project-oriented: no `setwd()`, paths are configurable not hardcoded

See stephenturner/demo-project-path-config for an example of one way to do this using configuration files. This can also be achieved with environment variables. **How it works:**

- Code lives in a git repository on each person's **local machine**
- Each team member has a personal `config.yml` file (not version controlled) with their specific paths
- Code uses helper functions to construct paths dynamically
- Data stays on the shared drive (read-only for most operations)
- Outputs go to each person's local directories
- Everyone commits code changes, never data

2.4 File Formats

Format	Best for	Avoid when
`.csv`	Tabular data, analysis input/output	Binary data, very wide datasets
`.tsv`	Tabular data with commas in values	—
`.json`	Nested/hierarchical data, configs	Large flat tables
`.parquet`	Large tabular data, columnar access	Simple small datasets
`.xlsx`	Data entry, sharing with non-coders	Final analysis-ready data

Plain text formats (CSV, TSV, JSON) are preferred for long-term storage and reproducibility. Proprietary binary formats (`.xlsx`, `.sav`, `.sas7bdat`) may become unreadable as software evolves.

2.5 Additional Best Practices

Use version control for code, not data. Git is designed for text files. Large or binary data files should live in dedicated storage (cloud drives, data repositories, databases) and be referenced in your code, not committed to the repository. Use a good `.gitignore`.

Document your cleaning steps. The gap between raw and processed data should be bridged entirely by a script that anyone can run. Comments in that script explaining *why* you made certain decisions are invaluable.

Plan for sharing. Before a project ends, ask: could someone else reproduce this analysis from the raw data? Good file organization, consistent naming, and a clear README make the answer yes.

Avoid "final" in file names. `analysis-final-FINAL-v3-USE-THIS-ONE.R` is a sign that version control was not used. Use Git instead, and name the file simply `analysis.R`.

3 Coding with AI

AI tools for coding have matured rapidly, covering a spectrum from simple inline completions to fully autonomous agents that can read, write, and execute code across an entire project. This chapter provides a brief orientation to the tools most relevant to data scientists working in Positron.

3.1 Code Completion with GitHub Copilot

GitHub Copilot is a code completion service that suggests code as you type. Install the **GitHub Copilot** extension in Positron, and suggestions will appear inline in the editor and in notebooks — press Tab to accept a suggestion or Escape to dismiss it.

Copilot draws on the surrounding code and comments to infer what you're trying to write. It works well for boilerplate, common patterns, and filling in function arguments you'd otherwise have to look up.

A GitHub Copilot subscription is required; a free tier is available for individual developers.

3.2 Positron Assistant

Positron has a built-in AI assistant (available since version 2025.07.0-204) accessible via the chat panel in the left sidebar. Unlike Copilot, which only completes code, the assistant understands your full session context — loaded data frames, console history, plots, and project structure — making it well-suited to data analysis workflows.

The assistant operates in three modes:

- **Ask** — conversational Q&A. Use it to explain error messages, get help with a function, or talk through an analysis approach without modifying any files.
- **Edit** — makes targeted changes to selected code based on a natural-language instruction. Useful for refactoring, renaming, or reformatting a block of code.
- **Agent** — more autonomous. The agent can create and modify files, run terminal commands, and work across multiple files to complete a larger task.

The assistant uses a bring-your-own-key model: you connect it to an AI provider (Anthropic, OpenAI, GitHub Copilot, or AWS Bedrock) using your own API key. Posit does not see your prompts or code.

> **Positron Assistant documentation**
>
> The Positron Assistant documentation covers setup, model configuration, and mode details.

3.3 Databot

Databot is an experimental Positron extension purpose-built for exploratory data analysis. Where Positron Assistant helps you write code, Databot *writes and executes* short analysis snippets on your behalf, treating insights as the goal rather than the code. You describe what you want to understand about your data, and Databot iterates through code execution to get there.

Databot is currently in research preview and requires an Anthropic API key. It is designed for experienced R and Python users who can critically evaluate the code it produces — AI models can and do make mistakes in open-ended data exploration.

Open Databot via the Command Palette (`Ctrl+Shift+P` / `Cmd+Shift+P`) and search for "Databot".

> **Tip**
>
> See the Databot documentation for installation instructions and warnings that are worth reading.

3.4 Claude Code

Claude Code is a terminal-based AI coding agent from Anthropic. Unlike the Positron Assistant chat panel, Claude Code operates directly in a shell session with full access to the filesystem: it can read and write files, search the codebase, run commands, and coordinate changes across many files in a single session.

Install it (instructions below for MacOS, Linux, WSL, see docs for Windows).

```
curl -fsSL https://claude.ai/install.sh | bash
```

Then launch it from Positron's integrated terminal (**Terminal → New Terminal**):

```
claude
```

Claude Code is particularly effective for multi-file refactors, complex debugging sessions, and tasks that are tedious to coordinate manually.

3.5 Cost, Privacy, and Model Choice

3.5.1 Capability

Frontier models from OpenAI and Anthropic are substantially more capable than open-weight alternatives for complex reasoning and coding tasks. For most everyday coding work (fixing errors, writing functions, refactoring) a mid-tier model like Anthropic's Claude Sonnet is fast, inexpensive, and more than capable enough. Save the most powerful (and expensive) frontier models for genuinely hard problems.

3.5.2 Cost

API pricing is metered by token (roughly, by word). Typical coding assistance sessions cost cents to a few dollars; heavy agent sessions that read and write many files may cost a bit more. GitHub Copilot charges a flat monthly subscription instead, which can be more predictable for high-volume users.

3.5.3 Privacy

A common concern is whether proprietary code sent to a cloud AI service will be used to train future models. The short answer is: **not if you're paying for API access**.

Both Anthropic and OpenAI state explicitly that inputs and outputs from paid API usage are not used for model training. This is distinct from their free consumer products, which may use interactions for improvement. If your team is using Positron Assistant, Claude Code, or the OpenAI API with your own key, your code is not becoming training data.

> **Note**
>
> See the authoritative statements directly:
>
> - Anthropic's privacy policy
> - OpenAI's privacy policy

Open-weight models (such as Meta's Llama or Google's Gemma) can be run entirely on local hardware, which provides the strongest possible privacy guarantee and meets strict data-residency or air-gap requirements. The trade-off is lower capability and the infrastructure overhead of running the model yourself. For most teams using cloud APIs with a commercial agreement, the privacy risk is lower than it is commonly assumed to be.

4 Dashboards

Quarto dashboards use `format: dashboard` to create multi-panel HTML data displays — no server required. They're a middle ground between a static report (a single linear document) and a Shiny app (interactive but requires a running R process). Dashboards are ideal for sharing summaries, KPIs, and plots that need to look polished without the overhead of maintaining a server.

The examples in this chapter are drawn from a real dashboard tracking U.S. drug overdose mortality data from the National Vital Statistics System (2015–2023).

> **Live demo and code**
>
> - **Live demo:** stephenturner.github.io/quarto-dashboard-demo
> - **Source code:** github.com/stephenturner/quarto-dashboard-demo

> **Quarto dashboard documentation**
>
> The Quarto documentation is the definitive resource for dashboard features, options, and examples. This chapter provides a practical walkthrough of building a dashboard, but the Quarto docs are the place to go for comprehensive reference material.

4.1 Dashboard Anatomy

4.1.1 YAML Front Matter

A minimal Quarto dashboard needs only two things in the front matter: a title and `format: dashboard`.

```yaml
---
title: "My Dashboard"
format: dashboard
---
```

Additional options control the theme, layout behavior, and navigation elements:

```yaml
---
title: "Drug Overdose Mortality"
subtitle: "National Vital Statistics System, 2015-2023"
format:
  dashboard:
    scrolling: false
    theme: cosmo
date: today
---
```

Setting `scrolling: false` (the default) sizes each row to fill the viewport height, producing a fixed-height layout. Setting `scrolling: true` allows the page to scroll freely, which is useful when the content is too tall to fit on one screen.

4.1.2 Rows and Columns

Quarto dashboards lay out content in rows by default. Level-2 headings (`##`) define rows, and each code cell within a row becomes a card that fills its share of the available width.

```
## Row {height=25%}

[cards here - share the row equally]

## Row {height=75%}

[cards here - taller row for plots]
```

The `{height=}` attribute (as a percentage or pixel value) controls how much vertical space each row takes. To switch to a column-based layout, use `orientation: columns` in the YAML and `{width=}` attributes on headings instead.

4.1.3 Cards

Every executable code cell in a dashboard automatically becomes a card. The cell's output (a plot, a table, text) fills the card. Use the `#| title:` cell option to give the card a header:

```
#| title: "Monthly Admissions"

ggplot(admissions, aes(x = month, y = n)) +
  geom_col()
```

4.1.4 Value Boxes

Value boxes are compact summary cards ideal for KPIs and headline numbers. They are created with the `#| content: valuebox` cell option. The cell should return a named list with `icon`, `color`, and `value` keys.

```
#| content: valuebox
#| title: "Total Patients"

list(
  icon  = "person-fill",
  color = "primary",
  value = nrow(patients)
)
```

Icons are Bootstrap Icons names (without the `bi-` prefix). Colors can be Bootstrap theme colors (`"primary"`, `"success"`, `"danger"`, `"warning"`, `"info"`) or any CSS color string.

4.1.5 Tabsets

Adding `.tabset` to a row makes each card within that row a tab instead of a side-by-side panel. Each cell's `#| title:` becomes the tab label.

````
## Row {.tabset}

```r
#| title: "Plot"

# plot code
```

```r
#| title: "Data"

# table code
```
````

Tabsets are useful when you have multiple views of the same data (e.g., a plot and the underlying table) and don't want them to compete for screen space.

4.2 Building a Dashboard

This section walks through building the overdose mortality dashboard step by step. All code blocks here are display-only; the executable versions live in the demo repository.

4.2.1 Project Structure

A Quarto dashboard lives in its own project directory with a `_quarto.yml` file and an `index.qmd`:

```
quarto-dashboard-demo/
  _quarto.yml
  index.qmd
  od-data-clean.csv
```

The `_quarto.yml` sets the project type and default format options:

```yaml
project:
  type: default

format:
  dashboard:
    theme: cosmo
    scrolling: false
```

4.2.2 Loading Data

The setup chunk loads packages and reads the data. Compute any summary values you'll reuse across multiple cells here so they're available throughout the document.

```
#| label: setup
#| message: false
#| warning: false

library(readr)
library(dplyr)
library(ggplot2)
library(plotly)
library(DT)

od <- read_csv("od-data-clean.csv")

# Computed once, reused in value boxes and plots
total_state_years <- nrow(od)
year_min          <- min(od$year)
year_max          <- max(od$year)
```

```r
top_state_recent <- od |>
  filter(year == max(year)) |>
  arrange(desc(pod)) |>
  slice(1)
```

The `pod` column is the proportion of all deaths attributable to overdose — the primary metric throughout the dashboard.

4.2.3 Value Boxes

Three value boxes occupy the first row. Each is a separate code cell with `#| content: valuebox` and a `#| title:`. Place them all under the same `## Row` heading so they share the row equally.

```r
## Row {height=20%}
```

```r
#| content: valuebox
#| title: "State-Year Observations"

list(
  icon  = "table",
  color = "primary",
  value = scales::comma(total_state_years)
)
```

```r
#| content: valuebox
#| title: "Years Covered"

list(
  icon  = "calendar-range",
  color = "info",
  value = paste0(year_min, "-", year_max)
)
```

```r
#| content: valuebox
#| title: "Highest OD % (Most Recent Year)"

list(
  icon  = "graph-up-arrow",
  color = "danger",
```

```
  value = paste0(
    top_state_recent$state, " - ",
    scales::percent(top_state_recent$pod, accuracy = 0.1)
  )
)
```

4.2.4 Static plots (ggplot2)

A static ggplot2 plot renders as a PNG image inside a card. It works well for PDF exports and environments without JavaScript, but users can't hover for values or zoom in.

```
#| title: "Overdose % Over Time - Selected States"

highlight_states <- c("WV", "DC", "OH", "KY")

od |>
  filter(state %in% highlight_states) |>
  ggplot(aes(x = year, y = pod, color = state, group = state)) +
  geom_line(linewidth = 1) +
  geom_point(size = 2) +
  scale_y_continuous(labels = scales::percent_format(accuracy = 0.1)) +
  scale_x_continuous(breaks = year_min:year_max) +
  labs(
    x     = NULL,
    y     = "Overdose deaths (% of all deaths)",
    color = "State"
  ) +
  theme_minimal(base_size = 13)
```

4.2.5 Interactive plots (plotly)

Wrapping a ggplot in `plotly::ggplotly()` converts it to an interactive plot with hover tooltips, zoom, and pan — no layout changes needed.

```
#| title: "Overdose % Over Time - Selected States"

p <- od |>
  filter(state %in% highlight_states) |>
  ggplot(aes(x = year, y = pod, color = state, group = state)) +
  geom_line(linewidth = 1) +
```

```
  geom_point(size = 2) +
  scale_y_continuous(labels = scales::percent_format(accuracy = 0.1)) +
  scale_x_continuous(breaks = year_min:year_max) +
  labs(
    x     = NULL,
    y     = "Overdose deaths (% of all deaths)",
    color = "State"
  ) +
  theme_minimal(base_size = 13)

ggplotly(p)
```

> **Tip**
>
> Swapping a static ggplot2 plot for `ggplotly()` is a one-line change that requires no modifications to the dashboard layout. Add interactivity where it helps users explore data; keep static plots where simplicity or export quality matters more.

For finer control over tooltips, use the `text` aesthetic and pass `tooltip = "text"` to `ggplotly()`:

```
bar_data |>
  ggplot(aes(
    x    = reorder(state, total_od),
    y    = total_od,
    fill = highlighted,
    text = paste0(state, ": ", scales::comma(total_od))
  )) +
  geom_col() +
  coord_flip()

ggplotly(p2, tooltip = "text")
```

4.2.6 Tables (DT)

`DT::datatable()` produces an interactive HTML table with built-in search, sorting, and pagination.

```
#| title: "Data"

od |>
```

```r
  mutate(pod = scales::percent(pod, accuracy = 0.01)) |>
  rename(
    State          = state,
    Year           = year,
    `Total Deaths` = ndeath,
    `OD Deaths`    = nod,
    `OD %`         = pod
  ) |>
  DT::datatable(
    filter   = "top",
    options  = list(pageLength = 10, scrollX = TRUE),
    rownames = FALSE
  )
```

`filter = "top"` adds a per-column search box above each column header, allowing users to filter by state or year without any server-side code.

4.3 Previewing and Rendering

From inside the project directory:

```
quarto preview index.qmd   # live-reload preview in browser
quarto render index.qmd    # render to index.html
```

> **Note**
>
> `quarto preview` on a dashboard opens a standalone browser window showing the dashboard itself. If you also have the book open in another preview, they run independently — the dashboard preview is not embedded in the book.

4.4 Hosting on GitHub Pages

Because `format: dashboard` is incompatible with `format: html` (the book format used here), a dashboard cannot be a page in a Quarto book. The cleanest solution is to keep the dashboard in its own GitHub repository and deploy it separately.

4.4.1 Repository Setup

Create a new GitHub repository (e.g., `quarto-dashboard-demo`) and push the project files:

```
git init
git add .
git commit -m "Initial dashboard"
git remote add origin git@github.com:stephenturner/quarto-dashboard-demo.git
git push -u origin main
```

4.4.2 Publishing

`quarto publish gh-pages` renders the project and pushes the output to the `gh-pages` branch. GitHub Pages serves that branch automatically.

```
quarto publish gh-pages
```

After the first publish, GitHub Pages will be available at `https://stephenturner.github.io/quarto-dashboard-demo`. Subsequent publishes update the live site.

> Tip
>
> **Automate with GitHub Actions.** Instead of running `quarto publish` manually on your local machine, you can set up a GitHub Actions workflow that re-renders and deploys on every push to `main`. The Quarto documentation provides a ready-to-use workflow YAML — copy it into `.github/workflows/publish.yml` in your repository and GitHub will handle the rest.

Part II

Nontechnical

5 Project Management

Data science projects in public health fail more often from unclear scope, misaligned expectations, and coordination breakdowns than from analytical errors. A rigorous analysis of the wrong question, delivered six months late to an audience that can't act on it, is a failed project regardless of its technical quality. This chapter covers the non-technical practices that help projects succeed: defining scope before work begins, collaborating as a team, working with IT and data governance partners, and communicating findings to people who will actually use them.

For a broader treatment of managing data science work, Roger Peng, Brian Caffo, and Jeff Leek's *Executive Data Science* (Peng, Caffo, and Leek 2015) is a concise, practical resource for both team leads and the analysts working alongside them.

5.1 Defining the Project

5.1.1 Start with the question

The most important work in a data science project often happens before any data is touched. Peng and Matsui's *The Art of Data Science* (Peng and Matsui 2015) frames the entire analysis process around one deceptively simple task: state the question precisely. Vague questions produce vague analyses. "Can you look at overdose trends?" is not a question that will lead anywhere useful. "Has the age distribution of overdose deaths in our state shifted since 2019, and does the pattern differ by rural versus urban counties?" is a question that can be answered.

Before agreeing to take on a project, work with the requestor to articulate:

- What specific question are we answering?
- Who will use the findings, and what decision or action do they enable?
- What would a satisfying answer look like – a number, a chart, a report, a recommendation?
- What does success look like, and how will we know when we've reached it?

Getting these questions answered upfront is the difference between a project that delivers value and one that drifts.

5.1.2 Project charters

For anything beyond a quick turnaround request, a brief project charter documents the shared understanding between the data science team and its stakeholders before work begins. A charter doesn't need to be a formal document – a shared notes page or a few paragraphs in a project tracking tool is sufficient. What matters is that the key parties have read it and agreed to it before anyone opens a dataset.

A useful project charter for public health data science covers:

- **The question**: stated precisely, as described above.
- **Data sources and access**: what data will be used, where it lives, and what approvals are needed (data use agreements, IRB review, IT provisioning).
- **Deliverables**: what will actually be produced and in what format (report, dashboard, dataset, presentation slides)?
- **Timeline**: key milestones and a realistic completion date, with explicit acknowledgment of dependencies.
- **Stakeholders**: who the primary requestor is, who the audience is, who has final say on scope.
- **Out of scope**: what the project will explicitly *not* address; this is often as important as what it will.

The charter is also the right place to surface risks early. If the analysis depends on data that hasn't been collected yet, or on linking two systems that have never been joined, that should be visible before commitments are made.

5.2 Managing Scope

Scope creep – the gradual expansion of a project beyond its original definition – is endemic to data science work and is usually well-intentioned. Stakeholders see early results and ask "can you also look at…?", and analysts, wanting to be helpful, say yes. Over time, a focused project becomes an unfocused one.

Make scope changes explicit. When a new request comes in mid-project, treat it as a scope change. Acknowledge it, discuss whether it belongs in the current project or a future one, and adjust the timeline if it's added. The charter is the reference point for these conversations.

Protect the last mile. The gap between "the analysis is done" and "the findings are in front of the people who need them" is consistently larger than anticipated. Time for writing, review, revision, and presentation needs to be built into project plans – not squeezed in at the end.

Know when to say "not yet." Not every analytic question can be answered well with available data, methods, or time. It is better to scope a project to what can be answered defensibly than to produce a shaky analysis of a broader question. A clear, well-supported

answer to a narrow question is more useful to a program than a hedged non-answer to a large one.

5.3 Collaborating as a Team

5.3.1 Version control

The foundation of collaborative data science work is version control. Chapter 1 covers Git and GitHub in depth – the mechanics of branches, pull requests, and conflict resolution. The project management implication is straightforward: a shared repository is the canonical record of a project's code and analysis. Work that lives only on one person's laptop is not team work; it is a liability.

For collaborative projects, adopt a simple branching convention (see Section 1.2) where changes are reviewed before being merged into the main branch. Even lightweight review (e.g. a colleague reading a pull request before it's merged) catches errors, shares knowledge, and prevents the "only one person understands this code" problem that becomes acute when someone leaves.

5.3.2 Reproducibility as a management goal

A reproducible analysis is one that can be re-run by a different person, on a different machine, at a future date, and produce the same result. In public health, this matters for accountability (can we explain exactly how this number was produced?), for updating results when new data arrives, and for knowledge transfer when team composition changes.

Reproducibility is a project management commitment. Building it in from the start costs far less than retrofitting it after a number is questioned.

5.3.3 Project structure

Consistent project organization makes it easier for team members to find things, onboard to a project quickly, and hand off work. A minimal structure that works well for most public health data science projects is shown below, but Chapter 2 covers project structure and data organization in more detail.

```
project-name/
  data/
      .gitignore    # Don't commit data to version control.
      raw/          # original data -- never edited directly
      processed/    # cleaned, analysis-ready files
```

```
R/                # scripts and functions
output/
    .gitignore    # Don't commit large output files.
    figures/
    tables/
report.qmd
```

The most important convention: raw data is read-only. All transformations happen in code, so there is always a reproducible path from the original files to any result.

5.4 Working with Your IT Team

IT is a necessary partner in any institutional public health settings. Server access, database credentials, software installation, network permissions, and data transfer approvals all flow through IT. Projects that treat IT as an afterthought routinely stall at the moment they are otherwise ready to move.

5.4.1 Start early

IT requests take time, and often more time than anticipated. Getting new software approved and installed on a shared server may take days or weeks. Database access for a sensitive dataset may require formal requests, security reviews, and management sign-off at multiple levels. If the analysis depends on accessing a new data source or running on infrastructure you haven't used before, initiate those conversations at project kickoff.

5.4.2 Be specific about what you need

A clear, specific request is far easier to fulfill than a vague one. When approaching IT, be prepared to specify:

- What software or tools are needed, including version numbers if relevant.
- What data needs to be accessed, where it lives, and in what format.
- What level of compute and storage the analysis requires.
- Whether the work involves sensitive or regulated data (HIPAA, PII, restricted-use files) that requires special handling.
- Any hard deadlines driven by reporting requirements or funding timelines.

Framing requests in terms of the program need (e.g., "this analysis supports the quarterly overdose surveillance report due to the state health officer in March") helps IT understand the stakes and prioritize accordingly.

5.4.3 Secure data environments

Much of the data used in public health analytics is sensitive: vital records, case surveillance data, insurance claims, linked longitudinal datasets. Familiarize yourself with your organization's data governance policies and the specific terms of any data use agreements governing the datasets you work with. Some analyses must be conducted within designated secure environments (managed servers, data enclaves, air-gapped systems) and outputs may need to pass a disclosure review before leaving.

These constraints shape what tools and workflows are actually feasible. It is better to understand them at project start than to build an analysis pipeline that cannot be used where the data actually lives.

5.5 Communicating Findings

A finding that is not communicated effectively is a finding that does not exist, for practical purposes. Public health data science ultimately serves program and policy decisions made by people who did not run the analysis and may not read a methods section.

5.5.1 Know your audience

Different stakeholders need different things:

- **Program staff** doing day-to-day work need actionable findings at the right level of detail, often a summary with supporting data available for those who want to dig in.
- **Program directors and health officers** need the bottom line and its implications, typically in a brief format (one page, one slide) with uncertainty communicated plainly.
- **Methodologically sophisticated collaborators** (epidemiologists, biostatisticians, peer reviewers) need full methods, sensitivity analyses, and honest discussion of limitations.

One analysis rarely serves all three audiences from a single document. Plan to produce multiple outputs from the same underlying work.

5.5.2 Communicate uncertainty honestly

Public health decisions are made under uncertainty, and analysis should convey what is known, what is estimated, and with what confidence – not project false precision. An administrator told "overdose deaths increased by exactly 12.3%" who later learns the estimate carried substantial uncertainty will trust the team less the next time. One given "we estimate a 10–15% increase, with the uncertainty driven primarily by changes in reporting completeness" has something they can actually act on.

That said, excessive hedging is its own failure mode. When the data clearly shows something, say so clearly. Uncertainty does not mean ambiguity about everything.

5.5.3 Match the output format to the need

A written report, an interactive dashboard, a two-page brief, and a slide deck are all appropriate in different contexts. The analysis determines what can be said; the output format determines whether it is heard. See Chapter 4 for guidance on building dashboards for data that needs to be explored or updated regularly.

5.6 Further Reading

- *Executive Data Science* (Peng, Caffo, and Leek 2015). A short, practical book on managing data science teams and working with executive stakeholders. Essential reading for team leads; useful context for analysts.
- *The Art of Data Science* (Peng and Matsui 2015). Covers the full data analysis process from question formulation through communication. The epicycles-of-analysis framework is particularly useful for understanding why projects rarely proceed in a straight line, and why that is normal.

References

Broman, Karl W., and Kara H. Woo. 2018. "Data Organization in Spreadsheets." *The American Statistician* 72 (1): 2–10. https://doi.org/10.1080/00031305.2017.1375989.

Bryan, Jennifer. 2015. "How to Name Files." https://github.com/jennybc/how-to-name-files.

Peng, Roger D., Brian Caffo, and Jeffrey T. Leek. 2015. *Executive Data Science*. Leanpub. https://leanpub.com/eds.

Peng, Roger D., and Elizabeth Matsui. 2015. *The Art of Data Science*. Leanpub. https://leanpub.com/artofdatascience.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* 59 (10): 1–23. https://doi.org/10.18637/jss.v059.i10.

A Other resources

There isn't much to see here yet...