Using Unix Command Line Tools to Supercharge Your Development Workflow

Vijay Mathew

March 18, 2026

Table of contents

Home

Introduction: The Terminal Never Got Old

Every few years, a new tool promises to change how developers work. A new IDE with smarter autocomplete. A GUI that makes Git "finally make sense." A visual debugger that eliminates the need to ever touch a terminal again. Developers adopt them eagerly, and for good reason — good tools matter.

And yet, the terminal remains. Decades after its invention, the Unix command line is still open in a tab on the screens of some of the most productive developers in the world. Not out of nostalgia. Not because they haven't heard of the alternatives. But because, for a certain class of problems, nothing else comes close.

This book is about those tools — and how to use them to write better software, faster.

Why the Terminal Still Wins

The command line has a reputation for being cryptic, unforgiving, and steep to learn. That reputation isn't entirely undeserved. But it obscures something important: once you're fluent, the terminal is *faster* and *more expressive* than almost any graphical alternative.

Consider what happens when you want to find every file in a project that imports a deprecated module. In an IDE, you might open a search dialog,

configure some options, scroll through results, and manually check each one. In the terminal:

```
rg "from utils/legacy" --type ts
```

Done. One line. Results in milliseconds, even across a codebase with thousands of files.

Or suppose you want to monitor your application's error logs in real time, filtered to only show lines from a specific service:

```
tail -f app.log | grep "payment-service"
```

The terminal doesn't just give you access to individual tools. It gives you a way to *compose* them — to connect simple, focused utilities into pipelines that solve complex problems on the fly, without writing a dedicated script for each one.

That composability is the terminal's real superpower, and it's what this book is built around.

A Philosophy Built to Last

Unix was designed in the late 1960s around a set of principles that have aged remarkably well. The most important of these, for our purposes, is this: **each tool should do one thing, and do it well.**

`grep` searches text. It doesn't edit files. `sed` edits streams of text. It doesn't search for files. `find` locates files. It doesn't read them. Each tool is small, focused, and predictable — and because they all speak the same language (text flowing through standard input and output), they can be combined in ways their original authors never anticipated.

This is the opposite of how most modern software is built. Your IDE is a monolith — powerful, but opaque. When it does something unexpected, it's hard to know why. When you need it to do something it wasn't designed for, you're usually stuck.

The Unix toolkit is the opposite. Transparent, composable, and endlessly hackable. When you build a pipeline of five small tools, you understand every step of what's happening. And if one step needs to change, you change just that piece.

What This Book Will Teach You

This isn't a reference manual. You don't need to memorize every flag for every command — that's what `man` pages are for. What this book aims to give you is *fluency*: an intuition for which tool to reach for, how to compose tools together, and how to think in the Unix way when you hit a problem.

We'll start with the basics — navigating a codebase, reading files, searching for patterns — and build toward more powerful techniques: stream processing, automation, API interaction from the command line, and building lightweight developer workflows with nothing more than a shell and a few well-chosen tools.

Along the way, every concept will be grounded in real development scenarios. Not toy examples, but the kinds of problems you actually encounter: tracking down a bug in a log file, refactoring a function name across a large project, understanding what changed between two versions of a file, automating a deployment step you've been doing by hand.

How This Book Is Organized

Each chapter focuses on a specific category of work — searching, editing, automation, data wrangling — and follows the same structure. We start with the *why*: what problem does this class of tools solve, and why does solving it at the command line beat the alternatives? Then we move to the *how*: hands-on examples you can run immediately, building from simple usage to more powerful combinations. Each chapter closes with a set of **exercises** and a **cheat sheet** of the most useful commands covered.

You don't need to read this book cover to cover. If you're already comfortable navigating the filesystem and just want to get better at searching or automation, jump straight to those chapters. But if you're newer to the terminal, reading in order will give you a foundation that makes each subsequent chapter click faster.

All examples in this book were tested on macOS and Ubuntu. Where behavior differs between the two, or where a tool needs to be installed separately, that's called out explicitly. Windows users running WSL2 will find that almost everything works identically.

Who This Book Is For

If you're a developer who mostly lives in an IDE and only opens a terminal when you absolutely have to, this book will change how you work. You'll find that many tasks you currently consider tedious can be solved in seconds with the right command.

If you already use the terminal regularly but rely on a small set of familiar commands, this book will expand your toolkit and — more importantly — teach you how to combine what you know in more powerful ways.

If you're an experienced CLI user, you'll find advanced techniques, modern alternatives to classic tools, and a framework for thinking about command-line composition that might sharpen instincts you've already developed.

The only real prerequisite is that you're writing software and you want to get better at it.

A Note on Modern Tools

The Unix command line has also evolved. Alongside the classics — `grep`, `sed`, `awk`, `curl` — a new generation of tools has emerged that are faster, friendlier, and better suited to modern development workflows. `ripgrep` is faster than `grep` and respects your `.gitignore` by default. `jq` makes JSON — the lingua franca of modern APIs — a first-class citizen on the command line. `fzf` brings fuzzy search to everything. `bat` makes reading files in the terminal actually pleasant.

Where relevant, this book will introduce these modern tools alongside their classic counterparts — not to replace them, but to give you the full picture and let you choose the right instrument for the job.

A Word Before We Begin

Writing this book came from a simple frustration: watching talented developers spend ten minutes manually hunting through files for something a single `rg` command would have found in two seconds. Or copy-pasting JSON into an online formatter when `jq` was already installed on their machine. Or avoiding the terminal entirely because no one ever showed them that it doesn't have to be hostile.

The terminal has a learning curve. This book is designed to make that curve as short and as rewarding as possible. Every hour you invest in CLI

fluency pays back many times over — not just in raw speed, but in a deeper understanding of how your software and your system actually work.

The terminal has been around for over fifty years. The developers who know it well aren't holding on to the past. They're using the most expressive, composable, and battle-tested set of development tools ever built.

Let's get started.

Navigating and Understanding a Codebase Quickly

There's a moment every developer knows. You've just cloned a new repository — or been handed ownership of a project someone else wrote — and you're staring at a directory listing trying to figure out where anything is. Where's the entry point? Where do the tests live? What even *is* this `scripts/` folder?

Most developers handle this by opening the project in their IDE and clicking around. That works. But the terminal gives you something the IDE file tree doesn't: the ability to ask *precise questions* about a codebase and get immediate, scriptable answers.

This chapter is about building that orientation instinct at the command line — getting from "I have no idea what this project looks like" to "I understand its structure" as fast as possible.

Getting the Lay of the Land with `tree`

The first thing you want when encountering a new codebase is a structural overview. The `ls` command will show you the contents of the current directory, but it's `tree` that gives you the full picture at a glance.

```
tree -L 2
```

The `-L 2` flag limits the output to two levels deep — enough to understand the top-level organization without drowning in every nested file. A typical output might look like this:

```
.
|-- src
|   |-- components
|   |-- services
|   |-- utils
|   `-- index.ts
|-- tests
|   |-- unit
|   `-- integration
|-- scripts
|-- package.json
|-- tsconfig.json
`-- README.md
```

In ten lines you've learned more about this project's structure than five minutes of IDE clicking would have told you. You can see it's a TypeScript project, it separates unit and integration tests, and there's a `scripts/` directory worth investigating.

To go deeper on a specific directory:

```
tree src/services
```

To exclude directories that are usually noise — `node_modules`, build output, `.git`:

```
tree -L 3 -I "node_modules|dist|.git"
```

The `-I` flag takes a pipe-separated pattern of names to ignore. This is one you'll use almost every time in a JavaScript or Python project.

If `tree` isn't installed on your system, you can get it with `brew install tree` on macOS or `apt install tree` on Ubuntu. It's worth adding to your standard developer setup.

Targeted File Listing with `ls`

`tree` gives you the overview. `ls` gives you the details of a specific location — and with the right flags, it tells you quite a lot.

The version of `ls` most developers should be using day-to-day:

```
ls -lah
```

Breaking that down: - `-l` — long format, showing permissions, owner, size, and modification date - `-a` — show hidden files (dotfiles like `.env`, `.gitignore`, `.eslintrc`) - `-h` — human-readable file sizes (`4.2K` instead of `4302`)

The modification date column is particularly useful when you're trying to understand what changed recently in a project:

```
ls -laht
```

Adding `-t` sorts by modification time, newest first. Run this in a directory and you immediately see what's been touched most recently — useful for understanding where active development is happening, or for finding a file you edited an hour ago but can't remember the name of.

To list only directories:

```
ls -d */
```

And to see the contents of a directory without changing into it:

```
ls -lah src/services/
```

Finding Files with `find`

`tree` and `ls` are great for browsing. But when you know *what* you're looking for and just need to locate it, `find` is the right tool.

Finding by name

```
find . -name "*.config.js"
```

This searches from the current directory (`.`) recursively for any file matching the pattern `*.config.js`. The quotes are important — without them, the shell may expand the glob before `find` sees it.

Case-insensitive search:

```
find . -iname "readme*"
```

Finding by type

To find only directories named `tests`:

```
find . -type d -name "tests"
```

To find only files (not directories) with a `.env` extension:

```
find . -type f -name ".env*"
```

Finding by modification time

This is where `find` starts to pull away from anything an IDE file tree can do. To find files modified in the last 24 hours:

```
find . -mtime -1
```

Files modified more than 7 days ago:

```
find . -mtime +7
```

Files modified between 2 and 5 days ago:

```
find . -mtime +2 -mtime -5
```

This is invaluable when you're debugging a production issue and need to answer the question: *what changed recently?*

Excluding noisy directories

Like `tree`, `find` will happily descend into `node_modules` unless you tell it not to:

```
find . -name "*.ts" -not -path "*/node_modules/*" -not -path "*/dist/*"
```

Combining `find` with actions

`find` becomes dramatically more powerful when you pair it with `-exec` to run a command on each result:

```
find . -name "*.log" -exec rm {} \;
```

This finds every `.log` file and deletes it. The `{}` is a placeholder for the filename, and `\;` ends the `-exec` expression.

A safer pattern when you're not sure what you're about to delete — preview first:

```
find . -name "*.log"          # see what would be affected
find . -name "*.log" -exec rm {} \;   # then act
```

We'll return to `find` and `-exec` in later chapters when we cover batch operations. For now, the key mental model is: `find` locates; `-exec` acts.

Understanding Files Before Opening Them

Before you open a file, two tools can tell you a lot about what you're dealing with.

`file` — what kind of file is this?

```
file some-binary
```

```
some-binary: ELF 64-bit LSB executable, x86-64, dynamically linked
```

```
file mystery-data
```

```
mystery-data: gzip compressed data, was "backup.tar", last modified: Mon Feb 12 09:14:22 202
```

This is particularly useful when you encounter files without extensions, or when a file isn't behaving the way you expect. Before spending ten minutes trying to open something, a quick `file` check tells you exactly what you're dealing with.

`stat` — metadata about a file

```
stat src/index.ts
```

```
  File: src/index.ts
  Size: 3421        Blocks: 8          IO Block: 4096   regular file
Device: fd01h/64769d    Inode: 2894732     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   alice)   Gid: ( 1000/   alice)
Access: 2024-03-12 09:14:22.000000000 +0000
Modify: 2024-03-11 17:43:09.000000000 +0000
Change: 2024-03-11 17:43:09.000000000 +0000
```

The three timestamps at the bottom are often what you're after: when the file was last accessed, when its content was last modified, and when its metadata last changed. This is more precise than what `ls` shows, and it's scriptable — you can extract individual fields:

```
stat -c "%n %y" src/*.ts
```

This prints the name and last-modified time for every TypeScript file in `src/` — useful for a quick audit of recently touched files.

Putting It Together: A Real Workflow

Here's how these tools work together in practice. Imagine you've just been asked to investigate a bug in an unfamiliar service. Your first five minutes at the terminal might look like this:

```
# Get the overall structure
tree -L 2 -I "node_modules|dist|.git"

# Understand what's been touched recently
ls -laht src/

# Find the entry point
find . -name "index.*" -not -path "*/node_modules/*"

# Find any config files that might affect behavior
find . -name "*.config.*" -not -path "*/node_modules/*"

# Check if there are any files that changed in the last day
find . -mtime -1 -not -path "*/node_modules/*" -not -path "*/.git/*"
```

In under two minutes, without opening a single file, you have a clear picture of the project's structure, what it's built with, and what changed recently. That's a foundation you can investigate from.

Modern Alternatives Worth Knowing

The tools covered in this chapter are available on virtually every Unix system. But a few modern alternatives are worth knowing about, particularly if you're setting up a new development machine.

`eza` (formerly `exa`) is a modern replacement for `ls`, written in Rust. It has better defaults, color-coded output by file type, built-in Git status indicators next to each file, and a tree view mode that combines what `ls` and `tree` do separately:

```
eza --tree --level=2 --git-ignore
```

`broot` is an interactive tree navigator that lets you browse, search, and open files all from a single interface. It's particularly good for very large codebases where `tree` output becomes overwhelming.

`fd` is a modern alternative to `find` with a simpler syntax, faster performance, and `.gitignore` awareness by default:

```
fd "*.config.js"            # equivalent to: find . -name "*.config.js"
fd -t d tests               # find directories named "tests"
fd --changed-within 1d      # files modified in the last day
```

If you're working on a new machine, `fd` and `eza` are both worth installing. That said, `find` and `ls` are available everywhere — on production servers, in CI environments, in Docker containers — and fluency with them is non-negotiable. Learn the classics first, then layer in the modern tools where they help.

Chapter Summary

The tools in this chapter — `tree`, `ls`, `find`, `file`, and `stat` — are your orientation layer. They answer the question *what is this codebase?* before you ever open a file. Used together, they can give you a remarkably complete picture of an unfamiliar project in just a few minutes.

The key habits to build:

- Start every new project or investigation with `tree -L 2 -I "node_modules|dist|.git"`
- Use `ls -laht` to see what's been recently modified in a directory
- Use `find` with `-mtime` when you need to know what changed and when
- Check `file` before assuming you know what an unfamiliar file contains
- Build `find` commands incrementally — preview before you act

Exercises

1. Clone any open source project you haven't worked with before. Use `tree`, `ls`, and `find` to answer these questions without opening any file in an editor: What language is it written in? Where do the tests live? Are there any hidden configuration files in the root?

2. Find all files in a project that were modified in the last 3 days, excluding `node_modules` and `.git`.

3. Use `find` with `-exec` to list the line count (using `wc -l`) of every `.js` file in a project.

4. Install `eza` and compare its output to `ls -lah` on the same directory. Which information does each show that the other doesn't?

Quick Reference

Command | What it does |

||| | `tree -L 2 -I "node_modules\|dist"` | Project structure, 2 levels deep, ignoring noise | | `ls -laht` | Directory listing sorted by modification time | | `find . -name "*.ts"` | Find all TypeScript files recursively | | `find . -mtime -1` | Files modified in the last 24 hours | | `find . -type d -name "tests"` | Find directories named “tests” | | `find . -name "*.log" -exec rm {} \;` | Find and delete all log files | | `file mystery-file` | Identify what kind of file something is | | `stat src/index.ts` | Full metadata including timestamps |

Searching Code Like a Pro

If there's one skill that separates developers who are fast from developers who are slow, it's the ability to find things quickly. The right function definition. The one place a config value is set. Every file that calls a deprecated API. The line in a log file that explains why the server crashed at 3am.

IDEs have search. GitHub has search. But neither gives you the speed, precision, and composability of searching from the terminal. Once you've internalized the tools in this chapter, you'll find yourself reaching for them instinctively — not because you're a terminal purist, but because they're simply the fastest way to find what you're looking for.

`grep`: The Classic

`grep` has been part of Unix since 1974. The name stands for *globally search a regular expression and print* — which is exactly what it does. Despite its age, it remains one of the most useful tools in a developer's arsenal.

Basic usage

```
grep "function handleAuth" src/auth.js
```

This searches the file `src/auth.js` for the string `function handleAuth` and prints every matching line. Simple, fast, and often all you need.

To search recursively through a directory:

```
grep -r "handleAuth" src/
```

The `-r` flag tells grep to descend into subdirectories. This is the version you'll use most often.

Essential flags

A handful of flags transform `grep` from useful to indispensable:

```
grep -r "handleAuth" src/ -n        # show line numbers
grep -r "handleAuth" src/ -l        # show only filenames, not matching lines
grep -r "handleAuth" src/ -c        # count matches per file
grep -r "handleAuth" src/ -i        # case-insensitive search
grep -r "handleAuth" src/ -v        # invert: show lines that DON'T match
```

The `-l` flag deserves special mention. When you want to know *which files* contain a pattern — not necessarily where in those files — `-l` gives you a clean list you can pipe to other commands:

```
grep -rl "TODO" src/ | wc -l        # how many files have TODOs?
```

Context lines

One of the most useful but underused grep features is context. Instead of just showing the matching line, you can ask grep to show the lines around it:

```
grep -r "throw new Error" src/ -n -A 3 -B 3
```

- `-A 3` — show 3 lines **after** each match
- `-B 3` — show 3 lines **before** each match
- `-C 3` — show 3 lines on **both sides** (shorthand for `-A 3 -B 3`)

This is invaluable when you're tracking down a bug and the matching line alone doesn't give you enough context to understand what's happening.

Searching with regular expressions

`grep` supports full regular expressions, which is where its name comes from and where it earns its keep:

```
grep -r "import.*from ['\"]react['\"]" src/     # all React imports
grep -r "console\.\(log\|warn\|error\)" src/    # all console statements
grep -r "^export default" src/                  # files with default exports
```

Basic regex in grep uses older POSIX syntax. For modern regex syntax (the kind you're used to from JavaScript or Python), use `-E` for extended regex or `-P` for Perl-compatible regex:

```
grep -rE "import .+ from '.+'" src/             # cleaner extended syntax
grep -rP "(?<=userId: )\d+" logs/               # lookbehind (Perl regex)
```

`ripgrep`: The Modern Default

`grep` is powerful, but it has limitations that become painful on large modern codebases. It searches `node_modules`. It doesn't respect `.gitignore`. On a project with hundreds of thousands of files, it can be slow.

`ripgrep` — the `rg` command — solves all of these problems. It's written in Rust, built for modern development workflows, and in most real-world searches it's between 5x and 100x faster than `grep`. If you install only one tool from this entire book, make it `ripgrep`.

```
brew install ripgrep          # macOS
apt install ripgrep           # Ubuntu
```

Basic usage

The syntax is almost identical to `grep`, which makes switching effortless:

```
rg "handleAuth"               # search current directory recursively
rg "handleAuth" src/          # search specific directory
rg "handleAuth" src/auth.ts   # search specific file
```

By default, `rg` already does what you'd need several `grep` flags to achieve: it searches recursively, shows line numbers, respects `.gitignore`, and skips hidden files and binary files. A bare `rg "pattern"` in the root of your project is almost always what you want.

File type filtering

This is one of `rg`'s best features. Instead of wrestling with `find` to limit your search to specific file types, `rg` handles it natively:

```
rg "TODO" --type ts           # search only TypeScript files
rg "TODO" --type py           # search only Python files
rg "TODO" --type-not test     # exclude test files
```

To see what types `rg` knows about:

```
rg --type-list
```

You can also define custom types for your project:

```
rg "pattern" --type-add "vue:*.vue" --type vue
```

Searching for the right thing

A few patterns that come up constantly in real development work:

```
# Find all usages of a function
rg "getUserById"

# Find where a variable is defined (not just used)
rg "const getUserById"

# Find all files that import a specific module
rg "from ['\"]../api/users['\"]"

# Find all TODO and FIXME comments
rg "TODO|FIXME|HACK|XXX"

# Find hardcoded strings that look like API keys or secrets
rg "['\"][A-Za-z0-9]{32,}['\"]" --type ts
```

Multiline search

By default, `rg` searches line by line. For patterns that span multiple lines, use `-U`:

```
rg -U "useEffect\(\(\) => \{[^}]+\}, \[\]\)" src/
```

This would match `useEffect` calls with an empty dependency array, even if the callback spans multiple lines — the kind of pattern that's hard to search for any other way.

Word boundaries

To avoid matching a pattern as part of a longer word, use `-w`:

```
rg -w "id"          # matches "id" but not "userId" or "invalid"
```

Fixed string search

When your search term contains regex special characters — dots, parentheses, brackets — and you want to search for the literal string, use `-F`:

```
rg -F "user.profile.id"       # treats dots as literal dots
rg -F "arr[0]"                # treats brackets as literal brackets
```

Searching Git History with `git grep` and `git log`

Sometimes the pattern you're looking for isn't in the current code — it was deleted, or it exists in an older version. Git gives you two powerful tools for this.

`git grep`

`git grep` works like `rg` but searches within the files tracked by git, including the ability to search at a specific commit or across all branches:

```
git grep "handleAuth"                    # search current working tree
git grep "handleAuth" HEAD~5             # search as of 5 commits ago
git grep "handleAuth" main               # search the main branch
```

To search across *all* commits — useful when someone says "this function used to exist, find it":

```
git grep "handleAuth" $(git rev-list --all)
```

This is slow on large repositories, but it's thorough. It will find the pattern in any commit in the entire history of the repository.

`git log -S`: the pickaxe

The `-S` flag on `git log` is one of the most powerful and least-known tools in the Git arsenal. It searches commit history for commits that *added or removed* a specific string:

```
git log -S "handleAuth" --oneline
```

This gives you every commit where the string `handleAuth` was introduced or deleted — not just mentioned in a commit message, but actually changed in the diff. This answers the question: *when was this code added, and by whom?*

```
git log -S "handleAuth" --oneline -p
```

Adding `-p` shows the full diff for each matching commit, so you can see exactly what changed.

For regex-based history search, use `-G` instead of `-S`:

```
git log -G "handle[A-Z][a-z]+" --oneline
```

Searching Structured Data with `jq`

Not all searching is searching source code. Modern development means working with JSON constantly — API responses, config files, package manifests, log output. `grep` can search JSON, but it has no understanding of structure. `jq` does.

`jq` is a command-line JSON processor. You can think of it as `grep` + `awk` for JSON — it lets you filter, transform, and extract data from JSON with a compact query language.

Basic extraction

```
cat package.json | jq '.dependencies'
```

This extracts the `dependencies` field from `package.json` and pretty-prints it. The `.` refers to the root of the JSON object, and `.dependencies` navigates to that key.

```
cat package.json | jq '.dependencies | keys'    # list dependency names
cat package.json | jq '.version'                 # just the version string
```

Filtering arrays

Where `jq` really shines is filtering arrays of objects — the shape that most API responses and log files take:

```
# From an array of users, find those with role "admin"
cat users.json | jq '.[] | select(.role == "admin")'

# Extract just the email field from each matching user
cat users.json | jq '.[] | select(.role == "admin") | .email'

# Find all log entries with status >= 400
cat access-log.json | jq '.[] | select(.status >= 400)'
```

Combining `curl` and `jq`

The most common real-world use of `jq` is parsing live API responses. Without `jq`:

```
curl https://api.example.com/users/42
# {"id":42,"name":"Alice","role":"admin","createdAt":"2024-01-15T...","permissions":[...]}
```

With `jq`:

```
curl -s https://api.example.com/users/42 | jq '.'             # pretty print
curl -s https://api.example.com/users/42 | jq '.name'         # just the name
curl -s https://api.example.com/users/42 | jq '.permissions[]' # list permissions
```

The `-s` flag on `curl` suppresses progress output, giving you clean JSON to pipe into `jq`.

Searching across multiple JSON files

Combining `find`, `cat`, and `jq` lets you search across an entire directory of JSON files:

```
find . -name "*.json" -not -path "*/node_modules/*" \
  -exec jq -r 'select(.version != null) | "\(input_filename): \(.version)"' {
```

This finds every JSON file in the project and prints its `version` field if it has one — useful for auditing the versions of all packages in a monorepo.

Building Search Pipelines

The real power of command-line search comes from combining tools. Each of the tools above produces text output that can be piped into the next tool in a chain.

Finding and counting

```
# How many TypeScript files have console.log statements?
rg "console\.log" --type ts -l | wc -l
```

Finding, filtering, and acting

```
# List all files with TODOs, sorted by number of TODOs
rg "TODO" -c --type ts | sort -t: -k2 -rn | head -20
```

Breaking this down: `rg "TODO" -c --type ts` counts TODO occurrences per file. `sort -t: -k2 -rn` sorts by the second colon-separated field (the count) in reverse numeric order. `head -20` shows only the top 20. The result is a ranked list of your most TODO-laden files.

Cross-referencing results

```
# Find all files that import 'lodash', then search those files for specific usage
rg "from 'lodash'" -l | xargs rg "\.cloneDeep"
```

Here, `rg "from 'lodash'" -l` gets the list of files that import lodash, and `xargs rg "\.cloneDeep"` runs a second `rg` search on just those files. This is a pattern you'll use often: narrow to a set of files with one search, then search within that set with another.

Searching and editing

```
# Find every file containing a pattern, then open them all in vim
rg "handleAuth" -l | xargs vim
```

Or more commonly, feed results into a command that makes a change — we'll cover this in depth in Chapter 5 when we get to `sed` and batch editing.

Practical Search Recipes

Here are some searches that come up repeatedly in real development work, ready to use or adapt:

```
# Find all environment variables referenced in code
rg "process\.env\.\w+" --type ts -o | sort | uniq

# Find potential SQL injection risks (string concatenation in queries)
rg "query\(.*\+" --type js

# Find all API endpoints defined in an Express app
rg "app\.(get|post|put|delete|patch)\(" --type js

# Find all files larger than expected (empty files - size 0)
find . -name "*.ts" -size 0

# Find recently modified config files
find . -name "*.config.*" -mtime -7 -not -path "*/node_modules/*"

# Count lines of code by file type (rough estimate)
find . -name "*.ts" -not -path "*/node_modules/*" | xargs wc -l | tail -1

# Find duplicate function names across files
rg "^function \w+" --type js -o | sed 's/.*://' | sort | uniq -d
```

Chapter Summary

Searching is one of the highest-leverage terminal skills you can develop. The tools in this chapter — `grep`, `rg`, `git grep`, `git log -S`, and `jq` — cover the full spectrum of what you'll need to find in a codebase, its history, and the data it produces.

The key habits to build:

- Default to `rg` over `grep` for any search in a development project — it's faster, smarter, and respects your `.gitignore`
- Use `-l` when you need a list of files to act on, not just lines to read
- Use `git log -S` when you need to know when and why something changed
- Reach for `jq` any time you're working with JSON — don't `grep` structured data
- Build pipelines: narrow with one search, then act on the results with another

Exercises

1. In any codebase you have access to, use `rg` to find every `TODO` comment. Then use `rg -c` and `sort` to rank the files by how many TODOs they contain.

2. Use `git log -S` to find the commit that introduced a specific function or variable in a git repository. Read the full diff to understand the context it was added in.

3. Fetch a JSON API response with `curl` and use `jq` to extract a specific nested field. Then filter an array within the response to show only items matching a condition.

4. Build a pipeline that finds all files importing a specific module and counts how many there are. Then extend it to list the files sorted alphabetically.

5. Use `rg` with `--type-list` to discover what file types it recognizes. Add a custom type for a file extension used in a project you work on.

Quick Reference

Command | What it does |

||| | `rg "pattern"` | Search current directory recursively (respects .gitignore) | | `rg "pattern" --type ts` | Search only TypeScript files | | `rg "pattern" -l` | List matching filenames only | | `rg "pattern" -c` | Count matches per file | | `rg "pattern" -w` | Match whole words only | | `rg "pattern" -F` | Treat pattern as literal string | | `rg "pattern" -C 3` | Show 3 lines of context around matches | | `grep -r "pattern" -n` | Recursive search with line numbers | | `grep -r "pattern" -A 3` | Show 3 lines after each match | | `git grep "pattern" HEAD~5` | Search codebase at a point in history | | `git log -S "pattern" --oneline` | Find commits that added/removed pattern | | `jq '.field'` | Extract a field from JSON | | `jq '.[] \| select(.x == "y")'` | Filter a JSON array | | `curl -s URL \| jq '.'` | Fetch and pretty-print a JSON API response |

Reading and Inspecting Files Without an Editor

Opening a file in an editor feels natural. It's what most of us learned first, and for writing code it's usually the right call. But for *reading* — for inspecting, skimming, monitoring, and understanding files — an editor is often the wrong tool. It's slow to open, it loads the entire file into memory, and it gives you no way to compose what you're seeing with other commands.

The terminal gives you a different set of primitives for reading files. Some are faster than opening an editor. Some can handle files so large that an editor would choke on them. Some let you watch a file change in real time. And all of them can be connected to the search and processing tools from the previous chapters, turning a simple file read into the first step of a more powerful pipeline.

This chapter covers the tools you'll use to read and inspect files at the command line — and, more importantly, when to reach for each one.

`cat`: Quick Reads and Concatenation

`cat` is the simplest file-reading tool. It reads a file and prints its contents to standard output — nothing more.

```
cat src/index.ts
```

For short files — configuration files, small scripts, environment templates — `cat` is the fastest way to see the contents without leaving the terminal. It's also the natural starting point for pipelines: read a file with `cat`, then pipe the output into `grep`, `jq`, `awk`, or whatever processing you need.

```
cat access.log | grep "ERROR" | wc -l     # count error lines in a log
cat package.json | jq '.scripts'           # extract npm scripts
```

Line numbers

When you need to reference specific lines — in a bug report, a code review comment, or a debugging session — the `-n` flag adds line numbers:

```
cat -n src/auth.ts
```

Showing invisible characters

Two flags that are surprisingly useful when debugging whitespace issues:

```
cat -A src/config.yaml      # show all non-printing characters
cat -e src/config.yaml      # show line endings ($ marks end of each line)
```

If a config file is behaving unexpectedly and you suspect Windows-style line endings (`\r\n` instead of `\n`), `cat -e` will show you the stray `^M` characters immediately. This is a small thing that saves a disproportionate amount of debugging time.

When not to use `cat`

`cat` reads the entire file before printing anything. For files that are hundreds of megabytes — large log files, database dumps, data exports — this is the wrong tool. Use `less` (covered next) or `tail` instead. A file that takes thirty seconds to `cat` takes milliseconds to inspect with `head`.

`less`: Navigating Large Files

`less` is an interactive file viewer. Unlike `cat`, it doesn't load the entire file at once — it reads and displays content on demand, which makes it fast even for files that are gigabytes in size.

```
less logs/application.log
```

Once inside `less`, navigation works like this:

Key | Action |

||| | `Space` or `f` | Page down | | `b` | Page up | | `G` | Jump to end of file | | `g` | Jump to beginning | | `/<pattern>` | Search forward | | `?<pattern>` | Search backward | | `n` | Next search match | | `N` | Previous search match | | `q` | Quit |

The search inside `less` supports regular expressions, which makes it genuinely useful for navigating large log files — you can jump directly to the next ERROR line, the next occurrence of a request ID, or the next stack trace.

Following a file

The `-F` flag makes `less` behave like `tail -f` — it keeps watching the file and shows new content as it's written:

```
less -F logs/application.log
```

The advantage over `tail -f` is that you can scroll back through the history while still receiving new lines — something `tail -f` alone can't do. For monitoring a live log file while also being able to investigate older entries, `less -F` is often the better choice.

Useful flags

```
less -N logs/application.log     # show line numbers
less -S logs/application.log     # don't wrap long lines (scroll horizontally
less -i logs/application.log     # case-insensitive search
less +G logs/application.log     # open at end of file (useful for logs)
```

The `-S` flag is particularly useful for log files or CSV data with very long lines, where wrapping makes the output unreadable.

Piping into `less`

Any command whose output is too long to read on one screen can be piped into `less`:

```
rg "ERROR" logs/ | less
git log --oneline | less
ls -lah /usr/lib | less
```

This is one of the most common terminal patterns: run a command, realize the output is long, pipe it into `less` so you can scroll through it at your own pace.

`head` and `tail`: Reading the Edges of Files

Most of the time, you don't need to read an entire file. You need the beginning, or the end. `head` and `tail` give you exactly that, without touching the rest of the file.

`head`

```
head src/index.ts              # first 10 lines (default)
head -n 50 src/index.ts        # first 50 lines
head -n -20 src/index.ts       # everything except the last 20 lines
```

`head` is most useful for checking file headers, understanding file format before processing it, or quickly confirming you have the right file. It's also the right tool when you want to preview the beginning of a large data file without reading the whole thing:

```
head -n 5 data/users.csv       # preview the first 5 rows of a CSV
```

`tail`

```
tail logs/app.log              # last 10 lines (default)
tail -n 100 logs/app.log       # last 100 lines
tail -n +50 logs/app.log       # everything from line 50 onwards
```

The `+` syntax on `-n` is subtle but powerful: `tail -n +50` means "start from line 50" rather than "show the last 50 lines." This lets you skip a file header and process everything after it:

```
tail -n +2 data/users.csv | wc -l     # count rows, excluding header
```

`tail -f`: Following a live file

The `-f` flag is one of the most-used developer commands in existence:

```
tail -f logs/application.log
```

This keeps the file open and prints new lines as they're written — essential for watching log output from a running service. Hit `Ctrl+C` to stop.

Following multiple files at once:

```
tail -f logs/app.log logs/error.log logs/access.log
```

`tail` will label each line with the filename it came from, so you can monitor several log streams simultaneously.

Combining `tail -f` with `grep` to filter the live stream:

```
tail -f logs/application.log | grep "payment-service"
tail -f logs/application.log | grep -E "ERROR|WARN"
```

This is the pattern you'll use constantly when debugging a running service: watch only the log lines you care about, filtered in real time.

Watching for a specific event

A more advanced pattern — wait until a specific string appears in a log file, then stop:

```
tail -f logs/app.log | grep -m 1 "Server started"
```

The `-m 1` flag on `grep` stops after the first match. This is useful in shell scripts that need to wait for a service to finish starting before proceeding.

`wc`: Counting Lines, Words, and Characters

`wc` (word count) is a small tool with a surprisingly wide range of uses in development workflows.

```
wc -l src/index.ts              # number of lines
wc -w src/index.ts              # number of words
wc -c src/index.ts              # number of bytes
wc -m src/index.ts              # number of characters
```

In practice, `-l` is the flag you'll use almost exclusively — line counts are the relevant metric for source files and logs.

Counting across multiple files

```
wc -l src/*.ts                  # line count for each TypeScript file, plus total
```

The output includes a per-file count and a total at the bottom — a quick way to get a rough sense of the size of different parts of a codebase.

Combining with `find` for a codebase summary

```
find . -name "*.ts" -not -path "*/node_modules/*" | xargs wc -l | sort -rn |
```

This finds all TypeScript files, counts their lines, sorts by line count in descending order, and shows the top 20 largest files. It's a useful way to identify the most complex parts of a codebase when you're getting oriented — the largest files are often the ones that have accumulated the most technical debt.

Counting search results

One of the most common uses of `wc -l` is counting the results of another command:

```
rg "TODO" -l | wc -l            # how many files have TODOs
git log --oneline | wc -l       # how many commits in this repo
cat access.log | grep "500" | wc -l    # how many 500 errors in the log
```

The pattern `command | wc -l` is so common it becomes muscle memory.

`bat`: A Better `cat`

`bat` is a modern replacement for `cat` that adds syntax highlighting, line numbers, and Git change indicators — all with sensible defaults that don't get in the way.

```
brew install bat                # macOS
apt install bat                 # Ubuntu (may be installed as batcat)
```

```
bat src/auth.ts                # syntax-highlighted file view
```

The output is color-coded by language, with line numbers on the left and a subtle header showing the filename. For reading source code at the terminal, it's dramatically more readable than plain `cat`.

Git integration

One of `bat`'s best features is that it shows Git change indicators in the gutter — a `+` for added lines, `~` for modified lines — so you can see what's changed in the current working tree without running `git diff`:

```
bat src/auth.ts                # modified lines are highlighted in the gutter
```

Useful flags

```
bat -n src/auth.ts             # line numbers only, no other decorations
bat -p src/auth.ts             # plain output (no decorations, like cat)
bat -A src/auth.ts             # show non-printing characters (like cat -A)
bat --line-range 50:100 src/auth.ts    # show only lines 50-100
```

The `--line-range` flag is particularly useful — instead of opening a file in an editor just to look at a specific section, you can read exactly the lines you need.

Using `bat` in pipelines

When used in a pipeline, `bat` automatically disables its decorations and behaves like `cat`:

```
bat src/auth.ts | grep "export"        # works exactly like cat | grep
```

This means you can alias `cat` to `bat` without breaking any pipelines — something many developers do in their shell configuration.

Reading Compressed and Binary Files

Not every file you need to inspect is a plain text file. A few tools handle the common cases.

Compressed files

Log files and data exports are often compressed with `gzip`. Rather than decompressing the file to read it, you can use the `z`-prefixed variants of common tools:

```
zcat logs/archive.log.gz               # like cat, for gzip files
zless logs/archive.log.gz              # like less, for gzip files
zgrep "ERROR" logs/archive.log.gz      # like grep, for gzip files
```

This means you can grep through months of archived logs without decompressing them first — a significant time and disk space saving when you're searching historical data.

Binary files

For binary files — compiled executables, data files, unknown blobs — `xxd` produces a hex dump that lets you inspect the raw bytes:

```
xxd some-binary | head -20
```

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 1011 0000 0000 0000  ..>.............
```

The left column is the byte offset, the middle is the hex representation, and the right is the ASCII interpretation (with `.` for non-printable bytes). For ELF binaries, the `ELF` signature is visible immediately — confirming what `file` told you in Chapter 1.

For a more readable inspection of what strings are embedded in a binary:

```
strings some-binary | grep -i "version"
strings some-binary | grep -i "error"
```

`strings` extracts printable character sequences from a binary file. It's useful for getting a quick sense of what a compiled binary does, what error messages it might produce, or what version it is.

Comparing Files with `diff`

Reading files often leads to a related question: how does this file differ from another version? `diff` is the classic tool for this.

```
diff file-a.ts file-b.ts
```

The output shows lines that differ between the two files, with `<` for lines only in the first file and `>` for lines only in the second.

Unified diff format

The default output is functional but hard to read quickly. The unified format — which is what Git uses — is much clearer:

```
diff -u file-a.ts file-b.ts
```

```
 file-a.ts   2024-03-11 17:43:09
+++ file-b.ts   2024-03-12 09:14:22
@@ -12,7 +12,7 @@
 function handleAuth(req, res) {
-  const token = req.headers.authorization;
+  const token = req.headers['authorization'];
   if (!token) {
```

Lines prefixed with `-` were removed, lines with `+` were added, and lines with neither are context.

Comparing directories

```
diff -r src/ src-backup/              # recursive directory comparison
diff -rq src/ src-backup/             # just show which files differ
```

The `-q` flag (quiet) only reports which files differ, not the actual differences — useful when you just need a list of changed files between two directory trees.

A practical pattern: comparing config environments

```
diff -u config/development.yaml config/production.yaml
```

A quick way to audit the differences between environment configs — something that's useful before a deployment and surprisingly easy to overlook.

Putting It Together: A Real Workflow

Here's how these tools work together in a realistic debugging scenario. Imagine a service is throwing errors intermittently and you need to investigate.

```
# How big is the log file? Is it worth opening in less?
wc -l logs/application.log

# Get a quick look at the most recent entries
tail -n 50 logs/application.log

# Watch the live log stream, filtered to errors only
tail -f logs/application.log | grep -E "ERROR|WARN"

# How many errors happened in the last hour?
# (assuming logs have timestamps in ISO format)
grep "$(date -u +%Y-%m-%dT%H)" logs/application.log | grep "ERROR" | wc -l

# Find the first occurrence of the error to understand when it started
grep -n "NullPointerException" logs/application.log | head -1

# Read the surrounding context at that line number
# (if the first occurrence was at line 4821)
tail -n +4810 logs/application.log | head -n 30
```

```
# Check if the error is also in yesterday's archived log
zgrep "NullPointerException" logs/application.log.1.gz | wc -l
```

Each of these commands takes a second or two to run. Together, they give you a detailed picture of the error — when it started, how often it occurs, what the surrounding context looks like — without ever opening a log file in an editor or downloading it to your local machine.

Chapter Summary

The tools in this chapter — `cat`, `less`, `head`, `tail`, `wc`, `bat`, `diff`, and the `z`-prefixed variants — give you a complete toolkit for reading and inspecting files at the terminal. The key is knowing which tool fits which situation.

The key habits to build:

- Use `cat` for short files and as the start of pipelines; use `less` for anything you need to scroll through
- Keep `tail -f | grep "pattern"` in your muscle memory — it's the fastest way to monitor a live service
- Use `wc -l` as a quick sanity check on results before acting on them
- Install `bat` and consider aliasing `cat` to it — the syntax highlighting pays for itself immediately
- Use `tail -n +2` to skip headers when processing structured files like CSVs
- Reach for `zcat` and `zgrep` before decompressing archived logs

Exercises

1. Find the largest source file in a codebase using `find`, `xargs`, and `wc -l`. Then use `bat --line-range` to read just the first 30 lines of it.

2. Start a process that writes to a log file (any web server or background process will do). Use `tail -f | grep` to watch its output filtered to a specific pattern in real time.

3. Take two config files that differ slightly — for example, a development and production environment config. Use `diff -u` to produce a clean summary of the differences.

4. Find a gzip-compressed file on your system (check `/var/log` on Linux, or create one with `gzip`). Use `zless` and `zgrep` to inspect and search it without decompressing it.

5. Use the `tail -n +2` pattern to skip the header row of a CSV file and pipe the result into `wc -l` to get an accurate row count.

Quick Reference

Command | What it does |

||| | `cat file` | Print file contents to stdout | | `cat -n file` | Print with line numbers | | `cat -A file` | Show non-printing characters | | `less file` | Interactively browse a file | | `less -F file` | Follow file (like tail -f, but scrollable) | | `less +G file` | Open at end of file | | `head -n 50 file` | First 50 lines | | `tail -n 100 file` | Last 100 lines | | `tail -n +2 file` | Everything from line 2 onwards | | `tail -f file` | Follow file in real time | | `tail -f file \| grep "pattern"` | Follow and filter live output | | `wc -l file` | Count lines | | `bat file` | Syntax-highlighted file view | | `bat --line-range 50:100 file` | View specific line range | | `diff -u file-a file-b` | Unified diff between two files | | `diff -rq dir-a/`

`dir-b/` | Which files differ between two directories | | `zcat file.gz` | cat for gzip-compressed files | | `zgrep "pattern" file.gz` | grep for gzip-compressed files | | `xxd file \| head` | Hex dump of a binary file | | `strings file` | Extract printable strings from a binary |

Editing Files from the Terminal

Reading files is passive. At some point you need to change them — rename a variable across an entire codebase, strip headers from a batch of CSV files, reformat log output, update a configuration value in a dozen places at once. This is where most developers reach for their IDE's find-and-replace, or write a throwaway Python script, or do it manually and hate every second of it.

The terminal has better answers. `sed` and `awk` are stream editors — tools designed to transform text as it flows through a pipeline. They're not interactive editors. You don't open a file, move a cursor, and save. Instead, you describe a transformation, apply it to a stream of text, and the result comes out the other side. That difference in mental model is what makes them so powerful: the same transformation you apply to one file can be applied to ten thousand files just as easily.

This chapter covers `sed`, `awk`, and a handful of supporting tools that together give you everything you need to edit files from the terminal — from simple substitutions to complex structural transformations.

`sed`: Stream Editing

`sed` stands for *stream editor*. It reads input line by line, applies a transformation to each line, and writes the result to standard output. The most common transformation — the one you'll use in probably 80% of your `sed` invocations — is substitution.

Basic substitution

```
sed 's/old/new/' file.txt
```

The `s` command means substitute. The syntax is `s/pattern/replacement/`. By default, this replaces the *first* occurrence of the pattern on each line and prints the result to stdout. The original file is untouched.

To replace *all* occurrences on each line, add the `g` (global) flag:

```
sed 's/old/new/g' file.txt
```

To make the substitution case-insensitive:

```
sed 's/old/new/gi' file.txt
```

In-place editing

Printing to stdout is useful in pipelines, but most of the time you want to edit the file itself. The `-i` flag enables in-place editing:

```
sed -i 's/old/new/g' file.txt
```

On macOS, `-i` requires an extension argument for the backup file. Use an empty string to skip the backup:

```
sed -i '' 's/old/new/g' file.txt    # macOS
sed -i 's/old/new/g' file.txt       # Linux
```

This is a cross-platform difference worth memorizing — it's the most common reason `sed` commands copied from the internet fail on macOS.

Always preview before editing in place. Run the command without `-i` first to confirm the output looks right, then add `-i` to commit the change:

```
sed 's/apiUrl/API_URL/g' src/config.ts        # preview
sed -i '' 's/apiUrl/API_URL/g' src/config.ts  # commit
```

Renaming across an entire codebase

Combined with `find` and `xargs`, `sed` becomes a codebase-wide refactoring tool:

```
find . -name "*.ts" -not -path "*/node_modules/*" \
  | xargs sed -i '' 's/getUserById/fetchUserById/g'
```

This renames `getUserById` to `fetchUserById` in every TypeScript file in the project. A task that would take ten minutes of IDE find-and-replace across multiple files takes one command.

Be careful with names that appear as substrings of other names — `sed` doesn't understand code, only text. The pattern `getUserById` would also match inside `getAdminUserById` if that existed. Use word boundaries to be safe:

```
# macOS (uses \b for word boundary in extended regex)
find . -name "*.ts" | xargs sed -i '' 's/\bgetUserById\b/fetchUserById/g'

# Linux (GNU sed)
find . -name "*.ts" | xargs sed -i 's/\bgetUserById\b/fetchUserById/g'
```

Deleting lines

`sed` can delete lines matching a pattern:

```
sed '/console\.log/d' src/index.ts
```

The `d` command deletes lines matching the pattern. This removes every `console.log` line from the output. Combined with `-i`:

```
find . -name "*.ts" | xargs sed -i '' '/console\.log/d'
```

Strips all `console.log` statements from your TypeScript codebase before a commit. A blunt instrument — use with care — but occasionally exactly what you need.

To delete blank lines:

```
sed '/^$/d' file.txt
```

`^$` matches lines that are empty from start (`^`) to end (`$`).

Printing specific lines

The `-n` flag suppresses default output, and the `p` command prints matching lines — together they let you extract specific lines from a file:

```
sed -n '10,20p' file.ts          # print lines 10 through 20
sed -n '/function handleAuth/p' file.ts   # print lines matching pattern
```

This is an alternative to `head`/`tail` for extracting specific line ranges, and more flexible because it supports pattern matching.

Inserting and appending text

`sed` can insert a line before a match (`i`) or append a line after it (`a`):

```
# Insert a comment before every function declaration
sed '/^function /i\// Auto-generated' src/utils.ts

# Append a blank line after every closing brace
sed '/^}/a\\' src/utils.ts
```

These are less commonly used in day-to-day development, but they're invaluable for code generation and template processing scripts.

Multiple expressions

Run multiple `sed` expressions in one pass with `-e`:

```
sed -e 's/foo/bar/g' -e 's/baz/qux/g' file.txt
```

Or equivalently, separate expressions with semicolons:

```
sed 's/foo/bar/g; s/baz/qux/g' file.txt
```

`awk`: Structured Text Processing

If `sed` is a scalpel — precise, focused, best for specific line-by-line transformations — `awk` is a Swiss Army knife. It's a full programming language built around the idea of processing structured text: files where data is organized into rows and columns, like CSV files, log files, or command output.

The mental model for `awk` is simple: it reads input line by line, splits each line into fields, and executes a program for each line. That program can filter, transform, calculate, and reformat the data any way you need.

Basic field extraction

The most common use of `awk` is extracting specific columns from structured output:

```
ls -lah | awk '{print $5, $9}'
```

`$1`, `$2`, `$3`... refer to the first, second, third fields of each line (split on whitespace by default). `$0` refers to the entire line. `$NF` refers to the last field (NF stands for Number of Fields).

This extracts the file size (`$5`) and filename (`$9`) from `ls -lah` output — a clean two-column view instead of the full listing.

```
ps aux | awk '{print $1, $2, $11}'    # user, PID, and command name
```

Changing the field separator

For CSV files or other delimited formats, use `-F` to set the field separator:

```
awk -F',' '{print $1, $3}' data.csv          # extract columns 1 and 3
awk -F':' '{print $1}' /etc/passwd           # extract usernames
awk -F'\t' '{print $2}' data.tsv             # tab-delimited files
```

Filtering with conditions

`awk` programs consist of `condition { action }` pairs. The action runs only when the condition is true:

```
# Print lines where the third field is greater than 1000
awk '$3 > 1000 {print $0}' data.csv

# Print lines where the second field matches a pattern
awk '$2 ~ /ERROR/ {print $0}' logs/app.log

# Print lines where the fifth field equals "admin"
awk '$5 == "admin" {print $1, $2}' users.csv
```

The `~` operator is a regex match. `!~` is its inverse (does not match).

Built-in patterns: BEGIN and END

`awk` has two special patterns: `BEGIN` runs before any input is processed, and `END` runs after all input is processed. These are used for headers, footers, and aggregations:

```
# Print a header, then process data, then print a summary
awk 'BEGIN {print "Name, Size"} {print $9, $5} END {print "Done"}' <(ls -lah)
```

Calculating totals

`awk`'s ability to accumulate values across lines makes it the right tool for quick aggregations:

```
# Sum the third column of a CSV
awk -F',' '{sum += $3} END {print "Total:", sum}' sales.csv

# Count lines matching a pattern
awk '/ERROR/ {count++} END {print count, "errors"}' app.log

# Average response time from a log file
awk '{sum += $NF; count++} END {print "Average:", sum/count, "ms"}' access.lo
```

Reformatting output

`awk`'s `printf` gives you full control over output formatting:

```
awk -F',' '{printf "%-20s %10.2f\n", $1, $3}' sales.csv
```

`%-20s` left-aligns a string in a 20-character field. `%10.2f` right-aligns a floating-point number with two decimal places in a 10-character field. This produces clean columnar output from messy CSV data.

A practical example: parsing access logs

Apache and Nginx access logs are structured text — a perfect target for `awk`. A typical log line looks like:

```
192.168.1.1 - - [12/Mar/2024:09:14:22 +0000] "GET /api/users HTTP/1.1" 200 14
```

Extracting useful information:

```
# Count requests by status code
awk '{print $9}' access.log | sort | uniq -c | sort -rn

# Find the top 10 most requested URLs
awk '{print $7}' access.log | sort | uniq -c | sort -rn | head -10

# Calculate total bytes transferred
awk '{sum += $10} END {print sum / 1024 / 1024, "MB"}' access.log

# Find all IPs that generated 500 errors
awk '$9 == 500 {print $1}' access.log | sort | uniq
```

These one-liners replace what would otherwise require a dedicated log analysis script. They run in seconds even on log files with millions of lines.

`tr`: Translating Characters

`tr` (translate) is a smaller, more focused tool that transforms individual characters. It doesn't understand lines or fields — it operates character by character on a stream.

Basic character translation

```
echo "hello world" | tr 'a-z' 'A-Z'     # lowercase to uppercase
echo "Hello World" | tr 'A-Z' 'a-z'     # uppercase to lowercase
```

Deleting characters

```
echo "hello, world!" | tr -d '[:punct:]'   # remove punctuation
echo "phone: (555) 123-4567" | tr -d '()-. '  # extract digits
cat file.txt | tr -d '\r'                  # remove Windows carriage returns
```

The last example — removing `\r` characters — is one of the most practical uses of `tr` in development. When a file created on Windows is used on Linux, the `\r` characters cause all kinds of mysterious behavior. `tr -d '\r'` fixes it instantly.

Squeezing repeated characters

The `-s` flag replaces sequences of repeated characters with a single instance:

```
echo "too    many    spaces" | tr -s ' '   # squeeze multiple spaces to one
cat file.txt | tr -s '\n'                  # remove blank lines
```

`cut`: Extracting Columns

`cut` is simpler than `awk` for the specific task of extracting columns from delimited text:

```
cut -d',' -f1,3 data.csv          # extract columns 1 and 3 from a CSV
cut -d':' -f1 /etc/passwd         # extract usernames
cut -f2 data.tsv                   # extract column 2 from tab-delimited file
```

`-d` sets the delimiter (comma, colon, tab, etc.) and `-f` specifies which fields to extract. For simple column extraction, `cut` is cleaner than `awk`. For

anything more complex — filtering, calculating, reformatting — reach for `awk`.

Extracting character ranges

`cut` can also extract by character position rather than field:

```
cut -c1-10 file.txt              # first 10 characters of each line
cut -c-5 file.txt                # first 5 characters
cut -c10- file.txt               # everything from character 10 onwards
```

`sort` and `uniq`: Organizing and Deduplicating

These two tools are almost always used together, and they appear constantly in the pipelines built around `sed` and `awk`.

`sort`

```
sort file.txt                    # alphabetical sort
sort -r file.txt                 # reverse order
sort -n file.txt                 # numeric sort
sort -rn file.txt                # reverse numeric sort
sort -k2 file.txt                # sort by second field
sort -t',' -k3 -n data.csv       # sort CSV by third column numerically
sort -u file.txt                 # sort and remove duplicates
```

The `-k` flag specifies the sort key — which field to sort by. Combined with `-t` to set the delimiter, it works on any structured text:

```
# Sort a CSV by the third column (numerically, descending)
sort -t',' -k3 -rn data.csv
```

`uniq`

`uniq` removes consecutive duplicate lines. Because it only compares adjacent lines, it almost always needs to be preceded by `sort`:

```
sort file.txt | uniq               # remove duplicates
sort file.txt | uniq -c            # count occurrences of each line
sort file.txt | uniq -d            # show only duplicate lines
sort file.txt | uniq -u            # show only unique lines (no duplicates)
```

The `sort | uniq -c | sort -rn` pipeline is one of the most useful in existence for frequency analysis:

```
# What are the most common words in a file?
cat README.md | tr ' ' '\n' | tr -d '[:punct:]' | sort | uniq -c | sort -rn

# What are the most common HTTP status codes in a log?
awk '{print $9}' access.log | sort | uniq -c | sort -rn

# What IP addresses appear most often?
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10
```

`tee`: Writing and Passing Through

`tee` reads from stdin and writes to both stdout *and* a file simultaneously. It's named after the T-shaped plumbing fitting that splits a pipe into two directions.

```
command | tee output.txt            # write to file AND see on screen
command | tee -a output.txt         # append to file (don't overwrite)
```

The most common use case: you're running a long build or test command and you want to see the output in real time *and* save it to a file for later:

```
npm test | tee test-results.txt
make build 2>&1 | tee build.log
```

The `2>&1` redirects stderr to stdout, so both streams are captured.

Splitting a pipeline

`tee` can also feed output into multiple subsequent pipeline branches:

```
cat access.log | tee >(grep "ERROR" > errors.log) >(grep "WARN" > warnings.log) | wc -l
```

This reads the log once, writes ERROR lines to `errors.log`, WARN lines to `warnings.log`, and counts total lines — all in a single pass. This is a more advanced pattern, but it's the right solution when you need to process a large file in multiple ways without reading it multiple times.

Putting It Together: Real Editing Workflows

Refactoring a module name across a codebase

```
OLD="UserService"
NEW="AccountService"

# Preview the changes first
grep -r "$OLD" src/ --include="*.ts" -l

# Apply the rename
find src/ -name "*.ts" | xargs sed -i '' "s/\b$OLD\b/$NEW/g"

# Verify the old name is gone
grep -r "$OLD" src/ --include="*.ts"
```

Cleaning and normalizing a CSV file

```
cat raw-data.csv \
  | tr -d '\r' \                          # remove Windows line endings
  | sed '1d' \                            # remove header row
  | awk -F',' '$3 > 0 {print $0}' \       # filter rows where column 3 > 0
  | sort -t',' -k2 \                      # sort by column 2
  | tee cleaned-data.csv \                # save to file
  | wc -l                                 # print row count
```

Generating a report from log data

```
echo "=== Error Report ===" > report.txt
echo "Generated: $(date)" >> report.txt
echo "" >> report.txt

echo "Top 10 Error Messages:" >> report.txt
```

```
grep "ERROR" app.log \
  | awk '{$1=$2=$3=""; print $0}' \      # strip timestamp fields
  | sort | uniq -c | sort -rn \          # count and rank
  | head -10 >> report.txt

echo "" >> report.txt
echo "Errors by Hour:" >> report.txt
grep "ERROR" app.log \
  | awk '{print substr($2, 1, 2)}' \     # extract hour from timestamp
  | sort | uniq -c >> report.txt
```

Chapter Summary

The tools in this chapter — `sed`, `awk`, `tr`, `cut`, `sort`, `uniq`, and `tee` — form the core of the Unix text processing toolkit. Used individually, each one is useful. Used together in pipelines, they can replace entire scripts for data transformation, log analysis, and codebase-wide refactoring.

The key habits to build:

- Always preview `sed` substitutions before using `-i` to edit in place
- Remember the macOS vs Linux difference for `sed -i` — it catches everyone eventually
- Use `sort | uniq -c | sort -rn` as your go-to frequency analysis pipeline
- Reach for `awk` when you need to work with columns or do arithmetic; use `cut` for simple column extraction
- Use `tee` whenever you want to both see and save the output of a long-running command
- Build transformations incrementally in a pipeline — add one tool at a time and check the output before adding the next

Exercises

1. Use `sed` to rename a function across all files of a given type in a project. Preview the changes first, then apply them. Verify the old name no longer appears.

2. Take an Apache or Nginx access log (or generate a sample one) and use `awk` to produce a frequency table of HTTP status codes, sorted from most to least common.

3. Use `tr`, `sort`, and `uniq -c` to find the ten most common words in a text file, excluding punctuation.

4. Build a pipeline that reads a CSV file, filters rows based on a numeric condition in one column, sorts by another column, and writes the result to a new file using `tee`.

5. Use `sed -n` with a line range to extract a specific function from a source file — just the lines from the function definition to its closing brace.

Quick Reference

Command | What it does |

||| | `sed 's/old/new/g' file` | Replace all occurrences in output | | `sed -i '' 's/old/new/g' file` | Replace in place (macOS) | | `sed -i 's/old/new/g' file` | Replace in place (Linux) | | `sed '/pattern/d' file` | Delete lines matching pattern | | `sed -n '10,20p' file` | Print lines 10–20 only | | `awk '{print $1, $3}' file` | Print fields 1 and 3 | | `awk -F',' '{print $2}' file` | Use comma as field separator | | `awk '$3 > 100 {print}' file` | Print lines where field 3 > 100 | | `awk '{sum+=$3} END{print sum}' file` | Sum a column | | `tr 'a-z' 'A-Z'` | Convert to uppercase | | `tr -d '\r'` | Remove Windows carriage returns | | `tr -s ' '` | Squeeze multiple spaces to one | | `cut -d',' -f1,3 file`

| Extract CSV columns 1 and 3 | | `sort -t',' -k3 -rn file` | Sort CSV by column 3, descending | | `sort \| uniq -c \| sort -rn` | Frequency count pipeline | | `command \| tee file.txt` | Write to file and stdout simultaneously |

Git Workflows from the Command Line

Most developers use Git every day. Most developers also use only a fraction of what Git can do.

The typical workflow — `git add`, `git commit`, `git push`, `git pull` — covers the basics of moving code around. But Git is also a database. Every version of every file your project has ever contained is stored in it. Every change, who made it, when, and — if they wrote a good commit message — why. Every branch, every merge, every rebase.

The command line is where that database becomes queryable. GUI Git clients are fine for basic operations, but they put a layer of abstraction between you and the underlying data that makes the more powerful operations either impossible or difficult to discover. The terminal gives you direct access to everything Git knows — and Git knows a lot.

This chapter covers the Git workflows and commands that most developers either don't know exist or have heard of but never learned properly. The assumption is that you're already comfortable with the basics. What follows is everything beyond them.

Understanding Your Repository at a Glance

Before making changes, it helps to understand exactly where things stand. These commands give you that picture quickly.

`git status` — but more useful

Most developers know `git status`. Fewer know its short form:

```
git status -s
```

The `-s` flag produces compact output — one line per file, with a two-character status code on the left:

```
M  src/auth.ts          # modified, staged
 M src/config.ts        # modified, not staged
?? src/new-file.ts      # untracked
A  src/added.ts         # new file, staged
D  src/deleted.ts       # deleted, staged
```

The first character is the staging area status; the second is the working tree status. Once you're used to reading this format, it's faster than the verbose default output.

`git log` — actually readable

The default `git log` output is verbose and hard to scan. These formats are far more useful:

```
git log --oneline                           # one line per commit
git log --oneline --graph                   # with branch/merge graph
git log --oneline --graph --all             # include all branches
git log --oneline --graph --all --decorate # with branch and tag names
```

The `--graph --all --decorate` combination gives you a visual representation of your entire branch history — merges, divergences, tags, and all — in the terminal. It's not as pretty as a GUI, but it's always available and it's composable:

```
git log --oneline --graph --all --decorate | head -30    # recent history only
git log --oneline --graph --all | grep "feature/"        # filter to feature branches
```

Filtering log output

```
git log --oneline --author="Alice"              # commits by a specific author
git log --oneline --since="2 weeks ago"         # recent commits
git log --oneline --since="2024-01-01" --until="2024-03-01"  # date range
git log --oneline --grep="fix"                  # commits with "fix" in message
git log --oneline src/auth.ts                   # commits that touched a specific file
```

The last one is especially useful: `git log <filename>` shows the complete history of a specific file — every commit that changed it, in chronological order. Pair it with `-p` to see the actual diff for each change:

```
git log -p src/auth.ts                  # full diff history of a file
git log -p --follow src/auth.ts         # follow file across renames
```

`--follow` is important for files that have been renamed — without it, git log stops at the rename boundary.

`git log --stat` and `git log --shortstat`

When you want to understand the *scope* of recent changes without reading every diff:

```
git log --stat --oneline                # files changed per commit
git log --shortstat --oneline           # summary stats per commit
```

```
a3f2c1b Fix authentication token handling
 src/auth.ts | 12 ++++++
 src/config.ts |  2 +-
 2 files changed, 7 insertions(+), 7 deletions(-)
```

Investigating Changes

Understanding *what* changed is one thing. Understanding *why* it changed — and *when* — requires a different set of tools.

`git diff` — beyond the basics

```
git diff                          # unstaged changes
git diff --staged                 # staged changes (what will be committed)
git diff HEAD                     # all changes since last commit
git diff main..feature-branch     # changes between two branches
git diff HEAD~3..HEAD             # changes in the last 3 commits
git diff HEAD~1 src/auth.ts       # changes to a specific file since last co
```

For a high-level summary of what changed without the full diff:

```
git diff --stat main..feature-branch      # files changed and line counts
git diff --name-only main..feature-branch # just the filenames
git diff --name-status main..feature-branch # filenames with change type (A/
```

The `--name-only` output is useful when you want to feed changed files into another command:

```
git diff --name-only main..HEAD | xargs rg "TODO"    # TODOs in changed files only
```

`git blame` — who wrote this?

`git blame` annotates every line of a file with the commit and author that last modified it:

```
git blame src/auth.ts
```

```
a3f2c1b (Alice  2024-03-10 14:22:18 +0000  42) function handleAuth(req, res) {
b7d9e2a (Bob    2024-02-28 09:15:44 +0000  43)   const token = req.headers['authorization'];
a3f2c1b (Alice  2024-03-10 14:22:18 +0000  44)   if (!token) {
```

The output shows commit hash, author, date, line number, and line content. This is the command you reach for when you're staring at a confusing piece of code and asking "why does this work this way?" — `git blame` tells you who to ask and when it was written.

Useful flags:

```
git blame -L 40,60 src/auth.ts        # blame only lines 40-60
git blame -w src/auth.ts              # ignore whitespace changes
git blame -M src/auth.ts              # detect lines moved within the file
git blame -C src/auth.ts              # detect lines copied from other files
```

`git show` — inspecting a specific commit

```
git show a3f2c1b                      # full diff for a commit
git show a3f2c1b --stat               # summary of files changed
git show a3f2c1b:src/auth.ts          # file contents at that commit
git show HEAD~3:src/config.ts         # file contents 3 commits ago
```

The `git show <commit>:<file>` form is particularly useful — it lets you read the old version of a file without checking it out. When someone says "this worked last week," you can compare the current version to the version from a week ago:

```
git show HEAD~7:src/auth.ts | diff - src/auth.ts      # diff old vs current
```

`git log -S` — the pickaxe (revisited)

We covered this in Chapter 2, but it's worth emphasizing again in context. The `-S` flag finds commits that *added or removed* a specific string:

```
git log -S "handleAuth" --oneline           # when was this function introduc
git log -S "API_KEY" --oneline              # when was this constant added?
git log -S "DROP TABLE" --oneline           # find dangerous migrations
```

This is one of the most powerful debugging tools in Git. When you find an unexpected piece of code and need to understand its history, `git log -S` will find the exact moment it was introduced.

Branching and Merging Efficiently

Creating and switching branches

```
git switch -c feature/new-auth          # create and switch (modern syntax)
git switch main                         # switch to existing branch
git switch -                            # switch to previous branch
```

The `git switch` command was introduced in Git 2.23 as a cleaner alternative to `git checkout` for branch operations. `git switch -` (switch to previous branch) is particularly handy — it works like `cd -` for navigating between the two branches you're currently working across.

Keeping a branch up to date

```
git fetch origin                          # fetch remote changes without merging
git rebase origin/main                    # rebase current branch onto main
git pull --rebase origin main             # fetch and rebase in one step
```

The `--rebase` flag on `git pull` is worth making a habit. Instead of creating a merge commit every time you pull, it replays your local commits on top of the updated remote branch — keeping history linear and readable.

To make this the default:

```
git config --global pull.rebase true
```

Interactive rebase — cleaning up before merging

Before merging a feature branch, it's good practice to clean up your commits — squashing work-in-progress commits, rewriting unclear messages, reordering changes for clarity. Interactive rebase is the tool for this:

```
git rebase -i HEAD~5                      # interactively edit the last 5 commits
git rebase -i origin/main                 # edit all commits not yet in main
```

This opens an editor with a list of commits and instructions:

```
pick a3f2c1b Add authentication middleware
pick b7d9e2a WIP
pick c8e3f4a fix tests
pick d9f4g5h more fixes
pick e0g5h6i cleanup

# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# s, squash = use commit, but meld into previous commit
# f, fixup = like squash, but discard this commit's log message
# d, drop = remove commit
```

Changing `pick` to `squash` (or `s`) merges a commit into the one above it. `fixup` does the same but discards the commit message. `reword` lets you edit the message. `drop` removes the commit entirely.

A common pattern before a pull request:

```
git rebase -i origin/main
# squash all WIP commits into their parent commits
# reword any unclear commit messages
# drop any debugging commits
```

Stashing work in progress

When you need to switch contexts mid-work without committing:

```
git stash                                  # stash all changes
git stash push -m "auth refactor WIP"  # stash with a description
git stash list                              # see all stashes
git stash pop                               # apply and remove most recent stash
```

```
git stash apply stash@{2}              # apply a specific stash without removing
git stash drop stash@{0}               # delete a specific stash
```

A stash is a stack — `git stash pop` takes the most recent entry. If you have multiple stashes, `git stash list` shows them all, and you can apply them by index.

Cherry-picking commits

To apply a specific commit from one branch to another without merging the whole branch:

```
git cherry-pick a3f2c1b                # apply a single commit
git cherry-pick a3f2c1b..e0g5h6i      # apply a range of commits
git cherry-pick a3f2c1b --no-commit    # apply changes without committing
```

`--no-commit` applies the changes to the working tree and staging area but doesn't create a commit — useful when you want to cherry-pick changes and combine them with other modifications before committing.

Undoing Things

Git's undo model is one of the most important things to understand deeply. There are several different tools, each appropriate for different situations.

`git restore` — discarding working tree changes

```
git restore src/auth.ts                # discard changes to a file
git restore .                          # discard all unstaged changes
git restore --staged src/auth.ts       # unstage a file (keep changes)
```

`git restore` is the modern replacement for `git checkout -- <file>`. It's clearer about intent and harder to misuse.

`git reset` — moving the branch pointer

```
git reset HEAD~1                        # undo last commit, keep changes stage
git reset --soft HEAD~1                # undo last commit, keep changes staged
git reset --mixed HEAD~1               # undo last commit, unstage changes (de
git reset --hard HEAD~1                # undo last commit, discard all changes
```

The three modes of `git reset` are important to understand:

- `--soft`: the commit is undone but all its changes remain staged. Useful for re-committing with a different message or squashing commits manually.
- `--mixed` (default): the commit is undone and changes are unstaged but preserved in the working tree. Useful for re-organizing what goes into the next commit.
- `--hard`: the commit is undone and all changes are discarded permanently. Use with caution — this is one of the few Git operations that can lose work.

`git revert` — undoing without rewriting history

When you need to undo a commit that has already been pushed to a shared branch, `git reset` is the wrong tool — it rewrites history, which causes

problems for anyone else working on the branch. `git revert` is the right tool:

```
git revert a3f2c1b                    # create a new commit that undoes a3f2c1b
git revert HEAD~3..HEAD               # revert the last 3 commits
git revert a3f2c1b --no-commit        # stage the revert without committing
```

`git revert` creates a new commit that is the inverse of the specified commit — it adds lines that were removed and removes lines that were added. History is preserved, and the undo is visible to everyone.

`git reflog` — the safety net

The reflog is a local log of every position HEAD has been at, including commits that have been reset, amended, or rebased away. It's your safety net when you've done something destructive:

```
git reflog                             # show HEAD movement history
git reflog show feature/auth           # show reflog for a specific branch
```

```
a3f2c1b HEAD@{0}: commit: Add auth middleware
b7d9e2a HEAD@{1}: reset: moving to HEAD~1
c8e3f4a HEAD@{2}: commit: WIP
d9f4g5h HEAD@{3}: checkout: moving from main to feature/auth
```

If you accidentally ran `git reset --hard` and lost commits, `git reflog` shows you where HEAD was before the reset. You can recover those commits:

```
git reset --hard HEAD@{2}             # go back to where HEAD was 2 moves ago
git checkout -b recovery HEAD@{2}     # create a branch at an earlier HEAD position
```

The reflog only exists locally and expires after 90 days by default, but within that window it has saved countless developers from genuine data loss.

Working with Remotes

Fetching and inspecting before merging

A good habit before merging remote changes is to fetch and inspect them first:

```
git fetch origin                                # fetch without merging
git log --oneline HEAD..origin/main             # what's new on remote main?
git diff HEAD origin/main                       # what changed?
git diff --stat HEAD origin/main                # summary of changes
```

This gives you a chance to understand what you're about to merge before you merge it — and to resolve any obvious conflicts mentally before they appear in your working tree.

Tracking branches

```
git branch -vv                                  # show all branches with tracki
git branch -r                                   # show all remote branches
git branch -a                                   # show local and remote branche
```

`git branch -vv` is a useful overview — it shows each local branch, the commit it's at, its tracking remote (if any), and whether it's ahead, behind, or diverged from the remote:

```
  feature/auth    a3f2c1b [origin/feature/auth: ahead 2] Add auth middleware
* main            b7d9e2a [origin/main] Update README
  old-feature     c8e3f4a [origin/old-feature: gone] Old work
```

Branches marked `gone` have had their remote deleted — usually after a merged pull request. You can clean them up:

```
git fetch --prune                                  # remove tracking refs for deleted remotes
git branch -vv | grep ': gone]' | awk '{print $1}' | xargs git branch -d
```

This one-liner finds all local branches whose remote has been deleted and removes them — a useful cleanup after a sprint or release.

Pushing efficiently

```
git push -u origin feature/auth                # push and set upstream
git push --force-with-lease                    # safer force push
git push origin --delete feature/old           # delete a remote branch
```

`--force-with-lease` is a safer alternative to `git push --force`. It refuses to push if the remote branch has been updated since you last fetched — protecting you from accidentally overwriting someone else's commits.

Git Aliases — Building Your Own Commands

Git's built-in commands are verbose. Aliases let you define shorter, more memorable versions:

```
git config --global alias.st "status -s"
git config --global alias.lg "log --oneline --graph --all --decorate"
git config --global alias.co "switch"
git config --global alias.last "log -1 HEAD --stat"
git config --global alias.undo "reset --soft HEAD~1"
git config --global alias.unstage "restore --staged"
```

After setting these, `git lg` gives you the decorated graph log, `git undo` undoes your last commit softly, and `git unstage <file>` removes a file from staging.

Aliases can also run shell commands by prefixing with `!`:

```
git config --global alias.root "rev-parse --show-toplevel"    # print repo ro
git config --global alias.whoami "config user.email"          # print current
```

Your aliases are stored in `~/.gitconfig`. Viewing and editing them directly is often easier than using `git config`:

```
cat ~/.gitconfig          # view all git configuration
```

Searching the Repository

Beyond what we covered in Chapter 2, Git offers several powerful ways to search the repository itself.

`git grep`

```
git grep "handleAuth"                          # search tracked files
git grep -n "handleAuth"                       # with line numbers
git grep -l "handleAuth"                       # filenames only
git grep "handleAuth" -- "*.ts"                # search only TypeScript files
git grep "handleAuth" HEAD~10                  # search at a point in history
```

`git grep` is faster than `rg` for repository searches because it reads directly from Git's object store rather than the filesystem.

Finding which commit broke something with `git bisect`

`git bisect` is one of the most powerful and least-used tools in Git. It performs a binary search through commit history to find the exact commit that introduced a bug.

```
git bisect start
git bisect bad                                 # current commit has the bug
git bisect good v1.2.0                         # this version was fine
```

Git will check out a commit halfway between the good and bad versions. You test whether the bug exists, then tell Git:

```
git bisect good                                # this commit is fine
# or
git bisect bad                                 # this commit has the bug
```

Git narrows the range by half each time. For a range of 1000 commits, it takes only 10 steps to find the exact offending commit. When it's done:

```
git bisect reset                               # return to original HEAD
```

`git bisect` can also be automated with a test script:

```
git bisect start HEAD v1.2.0
git bisect run npm test                    # automatically test each commit
```

This runs `npm test` at each commit and uses the exit code to determine good or bad automatically — finding the breaking commit with no manual intervention.

Shell Aliases and Scripts for Git Workflows

Beyond Git's own alias system, shell-level aliases and functions can further streamline your workflow:

```
# In ~/.zshrc or ~/.bashrc

# Common shortcuts
alias gs="git status -s"
alias ga="git add"
alias gc="git commit -m"
alias gp="git push"
alias gl="git log --oneline --graph --all --decorate"

# Switch to main and pull latest
alias gomain="git switch main && git pull --rebase"

# Delete merged branches
alias gclean="git branch --merged main | grep -v 'main\|*' | xargs git branch

# Open the repository in GitHub (macOS)
alias ghopen="open \$(git remote get-url origin | sed 's/git@github.com:/http
```

The `gclean` alias is worth highlighting: it deletes all local branches that have already been merged into main — the kind of housekeeping that's easy to forget and builds up over time.

Putting It Together: Real Git Workflows

Investigating a production bug

```
# When did this start happening?
git log -S "NullPointerException" --oneline

# Who changed the relevant file recently?
git log --oneline --since="1 week ago" src/auth.ts

# What exactly changed in that commit?
git show a3f2c1b

# Who wrote the specific line that's failing?
git blame -L 87,92 src/auth.ts

# Is this bug in the current release tag?
git log v2.1.0..HEAD --oneline src/auth.ts
```

Preparing a clean pull request

```
# Start from updated main
git switch main && git pull --rebase

# Create feature branch
```

```
git switch -c feature/auth-refactor

# ... do work, make commits ...

# Before opening PR: clean up commits
git rebase -i origin/main

# Check what will be merged
git diff --stat origin/main..HEAD
git log --oneline origin/main..HEAD

# Push
git push -u origin feature/auth-refactor
```

Recovering from a mistake

```
# Accidentally committed to main instead of a branch
git log --oneline -3                          # find the commit hash
git switch -c feature/accidental-commit       # create branch here
git switch main
git reset --hard HEAD~1                       # remove commit from main
git switch feature/accidental-commit          # continue work on the branch

# Accidentally deleted a branch
git reflog                                    # find the last commit on that
git checkout -b recovered-branch HEAD@{4}    # recreate it
```

Chapter Summary

Git's command line interface is the most complete way to interact with a repository. The tools in this chapter — `git log` with its many flags, `git diff`, `git blame`, `git bisect`, interactive rebase, the reflog — give you a level of visibility and control over your codebase's history that no GUI can fully replicate.

The key habits to build:

- Use `git log --oneline --graph --all --decorate` (or an alias for it) as your default history view
- Reach for `git blame` whenever you encounter code that doesn't make obvious sense — the history is usually the explanation
- Always use `--force-with-lease` instead of `--force` when pushing
- Learn `git bisect` — it's the fastest way to find a breaking commit in a large history
- Use interactive rebase to clean up commits before opening a pull request
- Keep `git reflog` in the back of your mind as your safety net — almost nothing in Git is truly permanent

Exercises

1. Find a repository you've worked on for a while. Use `git log --oneline --graph --all --decorate` to visualize its branch history. Identify any branches that have been merged and not cleaned up.

2. Use `git blame` to investigate a piece of code you've always found confusing. Find the commit that introduced it and read the full diff with `git show`. Does the commit message explain the decision?

3. Practice `git bisect` on any repository: introduce a known bug, commit a few more times, then use `git bisect` to find the exact commit that introduced it.

4. Use interactive rebase on a feature branch to squash all commits into a single well-described commit. Then use `git log -p` to verify the result looks correct.

5. Set up at least three Git aliases — one for your preferred `git log` format, one for a common operation you do repeatedly, and one shell alias for a multi-step workflow.

Quick Reference

Command | What it does |

||| | `git status -s` | Compact status view | | `git log --oneline --graph --all --decorate` | Visual branch history | | `git log --oneline src/file.ts` | History of a specific file | | `git log -p --follow src/file.ts` | Full diff history, following renames | | `git log -S "pattern" --oneline` | Find commits that added/removed a string | | `git diff --name-only main..HEAD` | Files changed since branching from main | | `git blame -L 40,60 src/file.ts` | Annotate specific lines | | `git show <commit>:src/file.ts` | View file at a specific commit | | `git bisect start / good / bad` | Binary search for a breaking commit | | `git rebase -i origin/main` | Interactive rebase for cleanup | | `git stash push -m "description"` | Stash with a label | | `git restore --staged src/file.ts` | Unstage a file | | `git reset --soft HEAD~1` | Undo last commit, keep changes staged | | `git revert <commit>` | Undo a commit without rewriting history | | `git reflog` | View full HEAD movement history | | `git push --force-with-lease` | Safe force push | | `git fetch --prune` | Remove stale remote tracking branches | | `git branch -vv` | Show branches with tracking status |

Working with Data and APIs from the Command Line

Modern software development is largely about data in motion. APIs returning JSON. Webhooks delivering payloads. Databases exporting CSVs. Log pipelines emitting structured events. Configuration files in YAML and TOML. Data files in formats ranging from clean CSV to deeply nested JSON to XML that looks like it was designed to cause suffering.

Most developers handle this data through GUI tools — Postman for API testing, TablePlus for database inspection, Excel for CSV manipulation. These tools are fine for what they are. But the terminal gives you something none of them can: the ability to compose data operations into pipelines, automate them, and apply the same transformation to one file or ten thousand files with equal ease.

This chapter covers the terminal toolkit for working with data and APIs: `curl` for HTTP, `jq` for JSON, and a collection of tools for CSV, YAML, and other formats you'll encounter in the wild. By the end of it, you'll be able to explore, transform, and manipulate data from the command line as fluently as you manipulate code.

`curl`: The Universal HTTP Client

`curl` is the command-line tool for making HTTP requests. It supports every HTTP method, every authentication scheme, file uploads, cookies, redirects, and dozens of other features. It's available on virtually every

system, which means a `curl` command you write today will work on any server you SSH into for the foreseeable future.

Basic requests

```
curl https://api.example.com/users           # GET request
curl -s https://api.example.com/users        # silent mode (no progress bar)
curl -sS https://api.example.com/users       # silent but show errors
```

The `-s` flag suppresses the progress bar that `curl` shows by default — essential when piping output to `jq` or other tools. `-sS` keeps errors visible while suppressing progress.

HTTP methods

```
# POST with JSON body
curl -X POST https://api.example.com/users \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com"}'

# PUT
curl -X PUT https://api.example.com/users/42 \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Updated"}'

# PATCH
curl -X PATCH https://api.example.com/users/42 \
  -H "Content-Type: application/json" \
  -d '{"email": "newemail@example.com"}'
```

```
# DELETE
curl -X DELETE https://api.example.com/users/42
```

Headers and authentication

```
# Custom header
curl -H "X-API-Key: your-key-here" https://api.example.com/data

# Bearer token authentication
curl -H "Authorization: Bearer eyJhbGc..." https://api.example.com/data

# Basic authentication
curl -u username:password https://api.example.com/data

# Multiple headers
curl -H "Content-Type: application/json" \
     -H "Authorization: Bearer token" \
     -H "X-Request-ID: abc123" \
     https://api.example.com/data
```

Reading request body from a file

For large or complex request bodies, putting the JSON inline gets unwieldy. Read it from a file instead:

```
curl -X POST https://api.example.com/users \
  -H "Content-Type: application/json" \
  -d @payload.json
```

The `@` prefix tells `curl` to read the body from a file. This is also useful for sending binary data or form submissions.

Response inspection

By default `curl` only shows the response body. To see more:

```
curl -i https://api.example.com/users        # include response headers
curl -I https://api.example.com/users        # HEAD request (headers only)
curl -v https://api.example.com/users        # verbose: full request and resp
curl -w "\nStatus: %{http_code}\nTime: %{time_total}s\n" \
     -s https://api.example.com/users        # custom output format
```

The `-w` flag (write-out) is particularly powerful. It lets you print specific response metadata after the request:

```
curl -s -o /dev/null \
     -w "Status: %{http_code} | Time: %{time_total}s | Size: %{size_download
     https://api.example.com/users
```

`-o /dev/null` discards the body; `-w` prints just the metrics you care about. This is the pattern for quick performance checks and health monitoring scripts.

Following redirects and handling cookies

```
curl -L https://short.url/abc               # follow redirects
curl -c cookies.txt https://example.com     # save cookies to file
curl -b cookies.txt https://example.com     # send cookies from file
curl -b cookies.txt -c cookies.txt \
     https://example.com/login              # maintain a session
```

Downloading files

```
curl -O https://example.com/file.zip        # save with original filename
curl -o custom-name.zip https://example.com/file.zip  # save with custom name
curl -C - -O https://example.com/large.zip  # resume an interrupted download
```

Using a config file for repeated requests

When you're working repeatedly with the same API during development, a `curl` config file saves you from typing the same headers every time:

```
# ~/.curlrc or project-specific curl.config
header = "Authorization: Bearer your-token"
header = "Content-Type: application/json"
silent
```

```
curl -K curl.config https://api.example.com/users
```

`jq`: JSON as a First-Class Citizen

We introduced `jq` briefly in Chapter 2. Here we go deep. `jq` is a complete query and transformation language for JSON — think of it as `sed` and `awk` combined, but designed specifically for structured data.

Installation

```
brew install jq           # macOS
apt install jq            # Ubuntu
```

The basics

```
echo '{"name":"Alice","age":30}' | jq '.'           # pretty-print
echo '{"name":"Alice","age":30}' | jq '.name'       # extract field
echo '{"name":"Alice","age":30}' | jq '.age + 1'    # compute value
```

The `.` identity filter passes input through unchanged but pretty-prints it. Field access uses dot notation: `.name`, `.user.profile.email`, `.config.database.host`.

Navigating nested structures

```
# Nested object access
echo '{"user":{"profile":{"email":"alice@example.com"}}}' | jq '.user.profile

# Array access
echo '{"items":[1,2,3]}' | jq '.items[0]'           # first element
echo '{"items":[1,2,3]}' | jq '.items[-1]'          # last element
echo '{"items":[1,2,3]}' | jq '.items[1:3]'         # slice

# Iterate over array
echo '[{"name":"Alice"},{"name":"Bob"}]' | jq '.[]'
echo '[{"name":"Alice"},{"name":"Bob"}]' | jq '.[].name'
```

Filtering arrays with `select`

```
# Users with role "admin"
cat users.json | jq '.[] | select(.role == "admin")'
```

```
# Items with price over 100
cat products.json | jq '.[] | select(.price > 100)'

# Multiple conditions
cat users.json | jq '.[] | select(.role == "admin" and .active == true)'

# Using regex in select
cat users.json | jq '.[] | select(.email | test("@example\\.com$"))'
```

Transforming and reshaping

This is where `jq` becomes genuinely powerful. You can reshape JSON into entirely different structures:

```
# Extract specific fields into a new object
cat users.json | jq '.[] | {id: .id, email: .email}'

# Rename fields
cat users.json | jq '.[] | {userId: .id, emailAddress: .email}'

# Add computed fields
cat orders.json | jq '.[] | . + {total: (.price * .quantity)}'

# Build a flat array from a nested structure
cat users.json | jq '[.[] | .email]'

# Convert array to object keyed by id
cat users.json | jq 'map({(.id | tostring): .}) | add'
```

Aggregation

```
# Count items
cat users.json | jq '[.[] | select(.active)] | length'

# Sum a field
cat orders.json | jq '[.[].amount] | add'

# Average
cat metrics.json | jq '[.[].response_time] | add / length'

# Min and max
cat metrics.json | jq '[.[].response_time] | min, max'

# Group by a field
cat orders.json | jq 'group_by(.status)'

# Count by status
cat orders.json | jq 'group_by(.status) | map({status: .[0].status, count: le
```

Practical patterns with `curl` and `jq`

The real power of `jq` in development is as the downstream processor of API responses:

```
# Fetch users and list their emails
curl -s https://api.example.com/users | jq '.[].email'

# Find users created in the last 7 days
curl -s https://api.example.com/users | \
  jq --arg date "$(date -u -d '7 days ago' +%Y-%m-%dT%H:%M:%SZ)" \
  '[.[] | select(.createdAt > $date)]'
```

```
# Get a specific user and format output
curl -s https://api.example.com/users/42 | \
  jq '{name: .name, email: .email, roles: .roles | join(", ")}'

# Extract pagination info alongside data
curl -s "https://api.example.com/users?page=1" | \
  jq '{total: .meta.total, page: .meta.page, users: [.data[].email]}'
```

Output formatting

```
jq -r '.name'              # raw output (no quotes around strings)
jq -c '.'                  # compact output (single line)
jq -r '.[] | .name' users.json | sort    # raw output suitable for piping
```

The `-r` flag is important when piping `jq` output into other commands — without it, strings are quoted, which causes issues with tools that expect plain text.

Building JSON with `jq`

`jq` isn't just for reading JSON — it can create it:

```
# Build a JSON object from shell variables
NAME="Alice"
EMAIL="alice@example.com"
jq -n --arg name "$NAME" --arg email "$EMAIL" \
  '{name: $name, email: $email}'

# Build from command output
jq -n \
```

```
  --arg hostname "$(hostname)" \
  --arg date "$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
  --argjson uptime "$(cat /proc/uptime | awk '{print $1}')" \
  '{hostname: $hostname, timestamp: $date, uptime: $uptime}'
```

This pattern — building JSON payloads with `jq -n` — is useful in deployment scripts and monitoring tools where you need to send structured data to an API.

Working with CSV Data

JSON is the format of APIs. CSV is the format of everything else — spreadsheet exports, database dumps, analytics data, billing records. The terminal handles CSV well, but with some caveats.

Basic CSV processing with `awk` and `cut`

For simple cases, `awk` and `cut` from Chapter 4 are sufficient:

```
cut -d',' -f1,3 data.csv                    # extract columns 1 and 3
awk -F',' '$3 > 1000 {print $1, $3}' data.csv   # filter and extract
```

The limitation is that basic `awk` and `cut` don't handle quoted fields — a CSV value like `"Smith, John"` (with a comma inside quotes) will break naive field splitting. For well-formed, quote-free CSVs, they work fine. For real-world CSVs, you need something better.

`csvkit`: the CSV toolkit

`csvkit` is a suite of command-line tools specifically designed for CSV files. It handles quoted fields, type inference, and encoding issues correctly.

```
pip install csvkit
```

The toolkit includes several commands:

```
csvstat data.csv              # summary statistics for each column
csvcut -c 1,3 data.csv        # extract columns by index or name
csvgrep -c email -m "@example.com" data.csv    # filter rows by column value
csvsort -c amount -r data.csv # sort by column, descending
csvjoin -c id users.csv orders.csv    # join two CSVs on a column
csv2json data.csv             # convert CSV to JSON
```

`csvstat` for quick data profiling

```
csvstat data.csv
```

```
  1. id
        Type of data:          Number
        Contains null values:  False
        Unique values:         1000
        Sum:                   500500
        Min:                   1
        Max:                   1000

  2. email
        Type of data:          Text
        Contains null values:  True (excluded from calculations)
```

```
Unique values:          998
Max length:             42
```

This gives you a complete statistical profile of every column in a CSV — types, null counts, unique values, min/max — in seconds. It's the first thing to run when you receive an unfamiliar data file.

`csvgrep` for filtering

```
# Find rows where status is "error"
csvgrep -c status -m "error" events.csv

# Find rows where email matches a pattern
csvgrep -c email -r "@example\.com$" users.csv

# Find rows where amount is above threshold (pipe through awk after)
csvgrep -c status -m "completed" orders.csv | awk -F',' '$3 > 1000'
```

Converting between formats

```
csv2json data.csv | jq '.[0]'                # CSV to JSON, inspect first r
csv2json data.csv > data.json                # convert entire file
json2csv -k id,name,email data.json          # JSON array to CSV
```

The ability to convert between formats unlocks cross-tool processing: convert CSV to JSON, process with `jq`, convert back to CSV, or feed directly into an API with `curl`.

Working with YAML

YAML appears constantly in modern development — Kubernetes manifests, Docker Compose files, CI/CD configuration, Ansible playbooks, application config. Unlike JSON, it's designed to be human-readable, which also makes it harder to process programmatically.

`yq`: jq for YAML

`yq` is a YAML processor modeled closely on `jq`. Its syntax is nearly identical, making it easy to learn if you already know `jq`.

```
brew install yq           # macOS
```

```
# Read a field
yq '.database.host' config.yaml

# Update a field
yq '.database.host = "new-host"' config.yaml

# In-place edit
yq -i '.database.host = "new-host"' config.yaml

# Extract a nested array
yq '.services[].name' docker-compose.yaml

# Filter array items
yq '.jobs[] | select(.name == "deploy")' .github/workflows/deploy.yaml
```

Practical YAML workflows

```
# Find all Kubernetes deployments with fewer than 2 replicas
cat deployment.yaml | yq 'select(.kind == "Deployment" and .spec.replicas < 2

# Update the image tag in a Kubernetes manifest
yq -i '.spec.template.spec.containers[0].image = "myapp:v2.1.0"' deployment.y

# Extract all environment variable names from a Docker Compose file
yq '.services[].environment | keys' docker-compose.yaml

# Compare two config files field by field
diff <(yq -o=json 'sort_keys(..)' config-dev.yaml) \
     <(yq -o=json 'sort_keys(..)' config-prod.yaml)
```

The last example is particularly useful for auditing differences between environment configs: convert both YAML files to sorted JSON, then diff them. The conversion to JSON normalizes formatting differences so the diff shows only actual value changes.

Converting YAML to JSON and back

```
yq -o=json config.yaml                    # YAML to JSON
yq -o=json config.yaml | jq '.database'   # extract JSON field from YAML sour
cat data.json | yq -P '.'                 # JSON to pretty YAML
```

Being able to convert between YAML and JSON means you can use `jq`'s richer query language on YAML files — convert to JSON, process with `jq`, convert back if needed.

Working with Other Formats

XML with `xmllint`

XML appears less frequently in modern development than it used to, but it's still common in enterprise systems, Maven build files, Android resources, and SOAP APIs. `xmllint` is available on most systems without installation:

```
# Pretty-print XML
xmllint --format messy.xml

# Validate against a schema
xmllint --schema schema.xsd data.xml

# XPath queries
xmllint --xpath "//user[@role='admin']/email/text()" users.xml

# Extract elements
xmllint --xpath "string(//version)" pom.xml       # Maven version
```

For heavier XML processing, `xsltproc` applies XSLT transformations, but that's beyond the scope of this book.

SQLite from the command line

SQLite databases are surprisingly common in development — mobile apps, local caches, test databases, embedded systems. The `sqlite3` command-line client lets you query them directly:

```
sqlite3 database.db                          # open interactive shell
sqlite3 database.db "SELECT * FROM users LIMIT 10;"   # one-off query
sqlite3 -csv database.db "SELECT * FROM users;" > users.csv   # export to CSV
sqlite3 -json database.db "SELECT * FROM users;" | jq '.[0]'  # export to JSON
```

The `-csv` and `-json` output flags make `sqlite3` a natural part of data pipelines — query a database, convert to JSON, process with `jq`, post to an API with `curl`.

Environment variables and `.env` files

```
# Load a .env file into the current shell
export $(cat .env | grep -v '#' | xargs)

# Check which variables are set
env | grep "API_"

# Run a command with specific environment variables
env API_KEY=test123 node scripts/migrate.js

# Compare .env files across environments
diff <(cat .env.development | sort) <(cat .env.production | sort)
```

Building API Workflows

Individual `curl` and `jq` commands are useful. Combined into workflows, they become small but complete data processing systems.

Paginating through an API

Many APIs return data in pages. This script fetches all pages automatically:

```
#!/bin/bash
page=1
all_users="[]"

while true; do
  response=$(curl -s "https://api.example.com/users?page=$page&per_page=100" \
    -H "Authorization: Bearer $API_TOKEN")

  users=$(echo "$response" | jq '.data')
  count=$(echo "$users" | jq 'length')

  if [ "$count" -eq 0 ]; then
    break
  fi

  all_users=$(echo "$all_users $users" | jq -s '.[0] + .[1]')
  ((page++))
done

echo "$all_users" | jq 'length'
echo "$all_users" > all-users.json
```

Transforming and posting data

A common ETL (extract, transform, load) pattern — read data from one API, transform it, post to another:

```
#!/bin/bash
# Migrate users from old API to new API

curl -s https://old-api.example.com/users \
  -H "Authorization: Bearer $OLD_API_TOKEN" | \
jq -c '.[] | {
  name: .full_name,
  email: .email_address,
  role: (if .admin then "admin" else "user" end)
}' | \
while read -r user; do
  curl -s -X POST https://new-api.example.com/users \
    -H "Authorization: Bearer $NEW_API_TOKEN" \
    -H "Content-Type: application/json" \
    -d "$user"
  echo "Migrated: $(echo $user | jq -r '.email')"
done
```

Testing an API endpoint

A quick smoke test script for an API:

```
#!/bin/bash
BASE_URL="https://api.example.com"
TOKEN="$API_TOKEN"
PASS=0
FAIL=0

check() {
  local description=$1
  local expected=$2
  local actual=$3
```

```
  if [ "$actual" = "$expected" ]; then
    echo "[PASS] $description"
    ((PASS++))
  else
    echo "[FAIL] $description (expected: $expected, got: $actual)"
    ((FAIL++))
  fi
}

# Test: list users returns 200
status=$(curl -s -o /dev/null -w "%{http_code}" \
  -H "Authorization: Bearer $TOKEN" \
  "$BASE_URL/users")
check "GET /users returns 200" "200" "$status"

# Test: create user returns 201
status=$(curl -s -o /dev/null -w "%{http_code}" -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name":"Test User","email":"test@example.com"}' \
  "$BASE_URL/users")
check "POST /users returns 201" "201" "$status"

echo ""
echo "Results: $PASS passed, $FAIL failed"
```

Monitoring an endpoint

```
#!/bin/bash
# Monitor an endpoint and alert on non-200 responses
```

```
URL="https://api.example.com/health"
INTERVAL=30

while true; do
  response=$(curl -s -w "\n%{http_code}" "$URL")
  body=$(echo "$response" | head -n -1)
  status=$(echo "$response" | tail -n 1)
  timestamp=$(date -u +%Y-%m-%dT%H:%M:%SZ)

  if [ "$status" != "200" ]; then
    echo "[$timestamp] ALERT: $URL returned $status"
    echo "Body: $body" | head -5
  else
    echo "[$timestamp] OK: $URL returned $status"
  fi

  sleep $INTERVAL
done
```

`httpie`: A More Ergonomic Alternative

`httpie` (the `http` command) is a modern alternative to `curl` with a more readable syntax, automatic JSON formatting, and sensible defaults for API development.

```
pip install httpie
# or
brew install httpie
```

```
# GET request
http https://api.example.com/users
```

```
# POST with JSON (automatically sets Content-Type)
http POST https://api.example.com/users name=Alice email=alice@example.com

# With headers
http https://api.example.com/users Authorization:"Bearer token"

# With query parameters
http https://api.example.com/users page==2 per_page==50
```

The `=` operator sets JSON string fields, `:=` sets non-string JSON values, `==` sets query parameters, and `:` sets headers. It's more readable than `curl` for interactive API exploration, though `curl` is still the right choice for scripts — it's available everywhere and its behavior is more predictable.

Putting It Together: A Complete Data Pipeline

Here's a realistic end-to-end pipeline: fetch sales data from an API, enrich it with user data from a CSV, generate a summary report, and post the results to a Slack webhook.

```
#!/bin/bash

# 1. Fetch orders from API
echo "Fetching orders..."
curl -s "https://api.example.com/orders?status=completed" \
  -H "Authorization: Bearer $API_TOKEN" \
  > /tmp/orders.json

# 2. Load user data from CSV and convert to JSON for joining
echo "Loading user data..."
csv2json users.csv > /tmp/users.json
```

```
# 3. Join orders with user data, compute metrics
echo "Computing metrics..."
jq -s '
  .[0] as $orders |
  .[1] as $users |
  ($users | map({(.id | tostring): .}) | add) as $user_map |
  {
    total_orders: ($orders | length),
    total_revenue: ([.$orders[].amount] | add),
    average_order: ([.$orders[].amount] | add / length),
    top_customers: (
      $orders |
      group_by(.user_id) |
      map({
        user: $user_map[.[0].user_id | tostring].name,
        orders: length,
        revenue: ([.[].amount] | add)
      }) |
      sort_by(-.revenue) |
      .[0:5]
    )
  }
' /tmp/orders.json /tmp/users.json > /tmp/report.json

# 4. Format as readable text
echo "Formatting report..."
REPORT=$(jq -r '
  "Sales Report - \(now | strftime("%Y-%m-%d"))\n" +
  "Total Orders: \(.total_orders)\n" +
  "Total Revenue: $\(.total_revenue | . * 100 | round / 100)\n" +
  "Average Order: $\(.average_order | . * 100 | round / 100)\n\n" +
  "Top Customers:\n" +
  (.top_customers[] | "  \(.user): \(.orders) orders, $\(.revenue)")
```

```
' /tmp/report.json)

echo "$REPORT"

# 5. Post to Slack
curl -s -X POST "$SLACK_WEBHOOK_URL" \
  -H "Content-Type: application/json" \
  -d "$(jq -n --arg text "$REPORT" '{text: $text}')"

echo "Report posted to Slack."
```

This pipeline — fetch, transform, aggregate, report, notify — is the kind of thing that might take an hour to build in a proper programming language and a few minutes to assemble from command-line tools. It's also easy to debug: you can run each step independently, inspect the intermediate files in `/tmp`, and modify one stage without touching the others.

Chapter Summary

The terminal is a complete environment for working with data and APIs. `curl` handles any HTTP interaction. `jq` handles any JSON transformation. `csvkit` and `yq` cover CSV and YAML. And the Unix pipeline model means all of these tools compose naturally — the output of one becomes the input of the next.

The key habits to build:

- Always use `-s` with `curl` in pipelines to suppress progress output
- Use `jq -r` when piping `jq` output into other commands — raw strings, not quoted JSON
- Use `jq -n` with `--arg` and `--argjson` for building JSON payloads from shell variables

- Run `csvstat` first on any unfamiliar CSV to understand its structure before processing
- Convert YAML to JSON with `yq -o=json` when you need `jq`'s richer query capabilities
- Build pipelines incrementally — get each step working before adding the next

Exercises

1. Pick any public REST API (GitHub, Open-Meteo, JSONPlaceholder). Use `curl` to fetch a collection endpoint and `jq` to extract a specific field from every item in the response. Then filter the results to items matching a condition.

2. Find a CSV file with at least five columns. Run `csvstat` on it to understand its structure. Then use `csvgrep` to filter it, `csvsort` to sort the result, and `csv2json` to convert it to JSON for further processing with `jq`.

3. Write a shell script that fetches data from an API endpoint, checks that the HTTP status code is 200, extracts a count from the response body with `jq`, and prints a pass/fail result.

4. Take a Kubernetes YAML manifest or Docker Compose file and use `yq` to extract all image names. If you don't have one handy, find one in any open source repository.

5. Build a pipeline that fetches JSON from an API, extracts a list of email addresses, sorts them, deduplicates them, and writes the result to a file — one email per line.

Quick Reference

Command | What it does |

||| | `curl -s URL` | GET request, silent mode | | `curl -X POST -d @file.json URL` | POST with JSON body from file | | `curl -H "Authorization: Bearer token" URL` | Request with auth header | | `curl -s -o /dev/null -w "%{http_code}" URL` | Get HTTP status code only | | `curl -L URL` | Follow redirects | | `jq '.'` | Pretty-print JSON | | `jq '.field'` | Extract a field | | `jq '.[].field'` | Extract field from each array item | | `jq 'select(.x == "y")'` | Filter by condition | | `jq '{a: .x, b: .y}'` | Reshape into new object | | `jq '[.[].field] \| add'` | Sum a field across array | | `jq -r '.field'` | Raw string output (no quotes) | | `jq -n --arg x "$VAR" '{key: $x}'` | Build JSON from shell variables | | `csvstat file.csv` | Profile a CSV file | | `csvcut -c 1,3 file.csv` | Extract CSV columns | | `csvgrep -c col -m value file.csv` | Filter CSV by column value | | `csv2json file.csv` | Convert CSV to JSON | | `yq '.field' file.yaml` | Extract YAML field | | `yq -i '.field = "value"' file.yaml` | Edit YAML in place | | `yq -o=json file.yaml` | Convert YAML to JSON | | `sqlite3 -json db.sqlite "SELECT..."` | Query SQLite, output JSON |

Automating Repetitive Dev Tasks

There's a rule of thumb in software development: if you do something twice, you'll do it a third time. If you do it a third time, you'll do it a hundred times. The question isn't whether a task is worth automating — it's whether you'll recognize the moment when it becomes worth it.

That moment usually arrives earlier than you think. The deployment sequence you run every afternoon. The database reset you perform before every test run. The five-command sequence to set up a new feature branch. The log-scraping you do every time something breaks in production. Each of these feels small in isolation. Together, they account for a surprising fraction of a developer's day — and more importantly, they account for a surprising fraction of the errors that make days longer than they need to be.

This chapter is about eliminating that friction. We'll cover shell scripts, aliases and functions, `make` as a task runner, environment management, and the patterns that separate automation that holds up over time from automation that creates more problems than it solves.

Shell Scripts: The Foundation

A shell script is a text file containing shell commands. That's it. There's no compilation step, no runtime to install, no dependency to manage. If a sequence of commands works in your terminal, it works in a shell script.

Anatomy of a shell script

```
#!/usr/bin/env bash
# Setup script for the development environment

set -euo pipefail

echo "Setting up development environment..."

# Install dependencies
npm install

# Copy environment template if .env doesn't exist
if [ ! -f .env ]; then
  cp .env.example .env
  echo "Created .env from template - please fill in your values"
fi

# Run database migrations
npm run db:migrate

echo "Setup complete."
```

Three things every shell script should have:

The shebang line (`#!/usr/bin/env bash`) tells the operating system which interpreter to use. `/usr/bin/env bash` is more portable than `/bin/bash` because it finds `bash` wherever it's installed on the system.

`set -euo pipefail` is a safety net: - `-e` exits immediately if any command fails - `-u` treats unset variables as errors - `-o pipefail` causes a pipeline to fail if any command in it fails (not just the last one)

Without these, a script will cheerfully continue running after a failure, often making things worse. With them, it stops at the first sign of trouble.

Comments explaining what each section does. Shell scripts have a reputation for being inscrutable — good comments are the antidote.

Making a script executable

```
chmod +x scripts/setup.sh        # make executable
./scripts/setup.sh               # run it
```

Or without making it executable:

```
bash scripts/setup.sh
```

Variables

```
#!/usr/bin/env bash
set -euo pipefail

# Constants
readonly DEPLOY_DIR="/var/www/app"
readonly LOG_FILE="/var/log/deploy.log"

# Variables
environment=${1:-development}    # first argument, default to "development"
timestamp=$(date +%Y%m%d_%H%M%S)
branch=$(git rev-parse --abbrev-ref HEAD)

echo "Deploying branch $branch to $environment at $timestamp"
```

`readonly` declares a constant — attempting to reassign it will cause an error. `${1:-development}` uses the first command-line argument if provided, falling back to `development` if not. `$(command)` runs a command and captures its output.

Conditionals

```
# File existence checks
if [ -f config.yaml ]; then echo "Config exists"; fi
if [ ! -f config.yaml ]; then echo "Config missing"; fi
if [ -d logs/ ]; then echo "Logs directory exists"; fi

# String comparisons
if [ "$environment" = "production" ]; then
  echo "Deploying to production - double-checking..."
fi

if [ -z "$API_TOKEN" ]; then
  echo "Error: API_TOKEN is not set" >&2
  exit 1
fi

if [ -n "$DEBUG" ]; then
  echo "Debug mode enabled"
fi

# Numeric comparisons
if [ "$count" -gt 0 ]; then echo "Found $count items"; fi
if [ "$exit_code" -ne 0 ]; then echo "Command failed"; fi
```

The `[ ]` syntax is POSIX test syntax. Note the spaces inside the brackets — they're required. `[[ ]]` is bash-specific but more powerful, supporting

regex matching and logical operators without escaping:

```
if [[ "$branch" =~ ^feature/ ]]; then
  echo "On a feature branch"
fi

if [[ "$status" == "200" && "$body" != "" ]]; then
  echo "Success"
fi
```

Loops

```
# Loop over files
for file in src/*.ts; do
  echo "Processing $file"
  npx tsc --noEmit "$file"
done

# Loop over an array
environments=("development" "staging" "production")
for env in "${environments[@]}"; do
  echo "Checking $env config..."
  diff "config/$env.yaml" "config/production.yaml" || true
done

# Loop over command output
git diff --name-only HEAD~1 | while read -r file; do
  echo "Changed: $file"
done

# Numeric loop
for i in $(seq 1 10); do
```

```
  echo "Attempt $i..."
  curl -s https://api.example.com/health && break
  sleep 5
done
```

Functions

Functions let you organize scripts into reusable, named pieces:

```
#!/usr/bin/env bash
set -euo pipefail

# Logging helpers
log()   { echo "[$(date +%H:%M:%S)] $*"; }
error() { echo "[ERROR] $*" >&2; }
die()   { error "$*"; exit 1; }

# Check required tools are installed
require() {
  command -v "$1" &>/dev/null || die "$1 is required but not installed"
}

# Wait for a service to become available
wait_for() {
  local url=$1
  local max_attempts=${2:-30}
  local attempt=1

  log "Waiting for $url..."
  while ! curl -sf "$url" &>/dev/null; do
    if [ $attempt -ge $max_attempts ]; then
      die "Timed out waiting for $url"
```

```
    fi
    sleep 2
    ((attempt++))
  done
  log "$url is ready"
}

# Main script
require curl
require jq
require node

wait_for "http://localhost:3000/health"
log "Starting deployment..."
```

The `log`, `error`, and `die` helpers are worth having in every script. They make output consistent and ensure errors go to stderr (`>&2`) rather than stdout, which matters when the script's output is piped to other commands.

Error handling

```
# Run a command, capture exit code without triggering set -e
result=$(some_command) || exit_code=$?
if [ "${exit_code:-0}" -ne 0 ]; then
  echo "Command failed with code $exit_code"
fi

# Cleanup on exit
cleanup() {
  log "Cleaning up..."
```

```
  rm -f /tmp/deploy_lock
  # kill any background processes
  jobs -p | xargs -r kill 2>/dev/null || true
}
trap cleanup EXIT

# Create lock file to prevent concurrent runs
LOCK_FILE="/tmp/deploy_lock"
if [ -f "$LOCK_FILE" ]; then
  die "Deployment already in progress (lock file exists: $LOCK_FILE)"
fi
touch "$LOCK_FILE"
```

The `trap` command runs a function when the script exits — whether normally, due to an error, or from a signal. It's the shell equivalent of a `finally` block, and it's essential for cleaning up temporary files, releasing locks, and terminating background processes.

Aliases and Functions: Instant Shortcuts

Shell scripts live in files and need to be called explicitly. Aliases and shell functions live in your shell configuration and are available instantly in any terminal session.

Aliases

An alias is a simple text substitution — a short name that expands to a longer command:

```
# In ~/.zshrc or ~/.bashrc

# Navigation
alias ..="cd .."
alias ...="cd ../.."
alias ll="ls -lah"
alias lt="ls -laht"          # sorted by modification time

# Git
alias gs="git status -s"
alias ga="git add"
alias gc="git commit -m"
alias gp="git push"
alias gpl="git pull --rebase"
alias gl="git log --oneline --graph --all --decorate"
alias gd="git diff"
alias gds="git diff --staged"

# Docker
alias dps="docker ps --format 'table {{.Names}}\t{{.Status}}\t{{.Ports}}'"
alias dlog="docker logs -f"

# Development
alias serve="python3 -m http.server 8080"
alias myip="curl -s https://ipinfo.io/ip"
alias week="date +%V"        # current week number

# Safety nets
alias rm="rm -i"             # confirm before deleting
alias cp="cp -i"             # confirm before overwriting
alias mv="mv -i"             # confirm before overwriting
```

After editing your shell config, reload it:

```
source ~/.zshrc              # or source ~/.bashrc
```

Shell functions

When you need more than simple substitution — arguments, conditionals, multiple commands — use a function:

```
# Create a directory and cd into it
mkcd() {
  mkdir -p "$1" && cd "$1"
}

# Find and kill a process by name
killport() {
  local port=$1
  local pid=$(lsof -ti tcp:"$port")
  if [ -z "$pid" ]; then
    echo "No process on port $port"
  else
    kill -9 $pid
    echo "Killed process $pid on port $port"
  fi
}

# Git: create a branch, push it, and set upstream in one step
gbc() {
  git switch -c "$1" && git push -u origin "$1"
}

# Quick git commit with message
gcm() {
  git add -A && git commit -m "$*"
```

```
}

# Extract any archive format
extract() {
  case "$1" in
    *.tar.gz|*.tgz)    tar -xzf "$1"  ;;
    *.tar.bz2|*.tbz2)  tar -xjf "$1"  ;;
    *.tar.xz)          tar -xJf "$1"  ;;
    *.zip)             unzip "$1"      ;;
    *.gz)              gunzip "$1"     ;;
    *.bz2)             bunzip2 "$1"    ;;
    *.7z)              7z x "$1"       ;;
    *)  echo "Unknown archive format: $1" ;;
  esac
}

# Search for a process
psg() {
  ps aux | grep -v grep | grep "$1"
}

# HTTP status code lookup
httpcode() {
  curl -s -o /dev/null -w "%{http_code}" "$1"
}
```

Organizing your shell configuration

As your aliases and functions grow, keeping them all in `.zshrc` or `.bashrc` becomes unwieldy. A cleaner pattern:

```
# In ~/.zshrc
for file in ~/.config/shell/*.sh; do
  [ -r "$file" ] && source "$file"
done
```

Then organize into separate files:

```
~/.config/shell/
|-- aliases.sh
|-- functions.sh
|-- git.sh
|-- docker.sh
`-- work.sh          # work-specific shortcuts (not in dotfiles repo)
```

This approach also makes it easy to share your shell config as a dotfiles repository — a common practice among developers that lets you replicate your environment on a new machine in minutes.

`make`: The Underrated Task Runner

`make` was created in 1976 to automate C compilation. Fifty years later, it's one of the best task runners available for any language or project type — not because of its build features, but because of its interface.

Every `make` target is a named task you can run with `make <target>`. The targets are defined in a `Makefile` in the project root. Any developer can run `make help` (if you write a help target) and see every available task. The interface is always the same, regardless of whether the project is Node, Python, Go, or something else.

A practical Makefile for a modern project

```
# Makefile
.PHONY: help install dev build test lint format clean deploy

# Default target: show help
help:
    @echo "Available targets:"
    @echo "  install     Install dependencies"
    @echo "  dev         Start development server"
    @echo "  build       Build for production"
    @echo "  test        Run test suite"
    @echo "  lint        Run linter"
    @echo "  format      Format code"
    @echo "  clean       Remove build artifacts"
    @echo "  deploy      Deploy to staging"

install:
    npm install

dev:
    npm run dev

build:
    npm run build

test:
    npm test

lint:
    npx eslint src/ --ext .ts,.tsx

format:
```

```
    npx prettier --write src/

clean:
    rm -rf dist/ node_modules/.cache

# Run checks before committing
check: lint test
    @echo "All checks passed"

# Deploy with confirmation for production
deploy:
    @read -p "Deploy to staging? [y/N] " confirm && \
    [ "$$confirm" = "y" ] && npm run deploy:staging || echo "Aborted"
```

Key Makefile concepts

`.PHONY` declares targets that aren't actual files. Without it, `make` checks whether a file named `test` exists and skips running the target if it does. Any target that doesn't produce a file of the same name should be listed in `.PHONY`.

Indentation with tabs — this is a notorious Make gotcha. Recipe lines *must* be indented with a tab character, not spaces. Many editors convert tabs to spaces automatically; if you're getting "missing separator" errors, this is usually why.

`@` prefix suppresses command echoing. Without `@`, `make` prints each command before running it. Prefixing with `@` runs it silently — useful for `echo` statements and prompts that would look redundant alongside the command itself.

`$$` is how you write a literal `$` in a Makefile recipe — the first `$` escapes the second.

Variables in Makefiles

```
# Variables
APP_NAME := myapp
VERSION := $(shell git describe --tags --always)
BUILD_DIR := dist
DOCKER_IMAGE := $(APP_NAME):$(VERSION)

build:
    @echo "Building $(APP_NAME) version $(VERSION)"
    npm run build
    @echo "Output in $(BUILD_DIR)"

docker-build:
    docker build -t $(DOCKER_IMAGE) .
    @echo "Built $(DOCKER_IMAGE)"

docker-push: docker-build
    docker push $(DOCKER_IMAGE)
```

Target dependencies

Targets can depend on other targets, which run first:

```
# docker-push depends on docker-build, which runs automatically
docker-push: docker-build
    docker push $(DOCKER_IMAGE)

# release runs lint, test, build in sequence
release: lint test build
    @echo "Ready to release version $(VERSION)"
```

Environment-specific targets

```
ENV ?= development      # default, overridable with ENV=staging make deploy

deploy:
    @echo "Deploying to $(ENV)"
    ./scripts/deploy.sh $(ENV)
```

```
make deploy                         # deploys to development
ENV=staging make deploy             # deploys to staging
ENV=production make deploy          # deploys to production
```

A self-documenting Makefile

A neat trick for auto-generating help output from comments:

```
.PHONY: help

help: ## Show this help
    @grep -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) | \
    awk 'BEGIN {FS = ":.*?## "}; {printf "\033[36m%-20s\033[0m %s\n", $$1, $$

install: ## Install dependencies
    npm install

test: ## Run test suite
    npm test

build: ## Build for production
    npm run build
```

```
deploy: ## Deploy to staging
    ./scripts/deploy.sh staging
```

Any target with a `##` comment will appear in `make help` with its description, automatically formatted in a two-column layout. As the project grows, the help output grows with it — with no manual maintenance.

Environment Management

Automation breaks in subtle ways when environment variables are wrong — missing API keys, wrong database URLs, development credentials used in production. Good environment management prevents this class of problem.

Loading environment variables

```
# Simple: export from .env file
export $(cat .env | grep -v '^#' | xargs)

# Safer: handle spaces in values
set -a; source .env; set +a

# With dotenv tools
# direnv (https://direnv.net/) - loads .envrc automatically when you cd into a directory
# autoenv - similar to direnv
```

`direnv` deserves special mention. You define environment variables in a `.envrc` file in your project directory, and `direnv` automatically loads them when you enter the directory and unloads them when you leave. No manual `source .env` step, no risk of forgetting:

```
# .envrc
export DATABASE_URL="postgres://localhost:5432/myapp_dev"
export API_KEY="dev-key-not-for-production"
export NODE_ENV="development"
```

```
brew install direnv
echo 'eval "$(direnv hook zsh)"' >> ~/.zshrc   # add to shell
direnv allow .                                  # approve the .envrc file
```

Validating required variables

A function to check that required environment variables are set before running a script:

```
require_env() {
  local missing=0
  for var in "$@"; do
    if [ -z "${!var:-}" ]; then
      echo "Error: required environment variable $var is not set" >&2
      missing=1
    fi
  done
  [ $missing -eq 0 ] || exit 1
}

# Usage at the top of scripts
require_env DATABASE_URL API_TOKEN SLACK_WEBHOOK_URL
```

Managing multiple environments

```
# scripts/env.sh - central environment loader
load_env() {
  local env=${1:-development}
  local env_file=".env.$env"

  if [ ! -f "$env_file" ]; then
    echo "Error: $env_file not found" >&2
    exit 1
  fi

  set -a
  source "$env_file"
  set +a
  echo "Loaded environment: $env"
}

# Usage
load_env production
load_env staging
```

Secrets management

For real applications, secrets shouldn't live in `.env` files — they should live in a secrets manager. But you can still access them from the command line:

```
# AWS Secrets Manager
secret=$(aws secretsmanager get-secret-value \
  --secret-id "myapp/production/database" \
  --query SecretString \
```

```
  --output text | jq -r '.password')

# HashiCorp Vault
export DATABASE_PASSWORD=$(vault kv get -field=password secret/myapp/databas

# GitHub CLI (for CI/CD secrets during development)
gh secret set API_TOKEN < api_token.txt
```

Scheduling and Background Tasks

`cron`: scheduled execution

`cron` runs commands on a schedule. Edit your crontab with:

```
crontab -e
```

The format is five time fields followed by the command:

```
# Minute  Hour  Day  Month  Weekday  Command
  0       9     *    *      1-5      /home/alice/scripts/daily-report.sh
  */15    *     *    *      *        /home/alice/scripts/health-check.sh
  0       0     1    *      *        /home/alice/scripts/monthly-cleanup.sh
  0       2     *    *      *        /home/alice/scripts/backup.sh >> /var/l
```

`*/15` means "every 15 minutes." `1-5` for the weekday field means Monday through Friday. `*` means "every."

Always use absolute paths in crontab entries — cron runs with a minimal environment and `PATH` may not include your usual directories. Always redirect output to a log file, otherwise cron will try to email it to you (which usually means it disappears silently).

A quick cron expression reference:

```
# Every minute
* * * * *

# Every hour at :30
30 * * * *

# Every day at 2am
0 2 * * *

# Every Monday at 9am
0 9 * * 1

# Every 5 minutes
*/5 * * * *
```

Running scripts in the background

```
# Run in background, continue even after terminal closes
nohup ./scripts/long-running.sh &

# Run with output captured
nohup ./scripts/long-running.sh > logs/output.log 2>&1 &

# Check what's running in background
jobs -l

# Bring background job to foreground
fg %1

# Disown a job (detach from terminal)
./scripts/long-running.sh &
disown $!
```

`watch`: repeated execution

`watch` runs a command repeatedly and displays the output, refreshing every two seconds by default:

```
watch -n 5 "curl -s https://api.example.com/health | jq '.status'"
watch -n 1 "git log --oneline -5"
watch -n 2 "docker ps --format 'table {{.Names}}\t{{.Status}}'"
```

This is useful for monitoring live output without writing a full polling loop. `Ctrl+C` stops it.

A Practical Automation Toolkit

Here's a collection of scripts that solve common development automation problems, ready to adapt:

Pre-commit checks

```
#!/usr/bin/env bash
# scripts/pre-commit.sh - run before every commit
set -euo pipefail

log() { echo "> $*"; }

log "Running pre-commit checks..."

# Lint
log "Linting..."
npx eslint src/ --ext .ts,.tsx --quiet
```

```
# Type check
log "Type checking..."
npx tsc --noEmit

# Tests
log "Running tests..."
npm test -- --passWithNoTests

# Check for secrets (basic)
log "Checking for secrets..."
if git diff --cached | grep -E "(api_key|password|secret)\s*=\s*['\"][^'\"]{8,}" -i; then
  echo "[WARN] Possible secret detected in staged changes" >&2
  exit 1
fi

log "All checks passed [OK]"
```

Wire this to Git's pre-commit hook:

```
cp scripts/pre-commit.sh .git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
```

Or use `husky` for a more robust hook management solution if you're working on a team.

Database reset script

```
#!/usr/bin/env bash
# scripts/db-reset.sh - reset database to clean state
set -euo pipefail
```

```
log() { echo "[db-reset] $*"; }

: "${DATABASE_URL:?DATABASE_URL must be set}"

# Confirm if running against a non-development database
if [[ "$DATABASE_URL" != *"localhost"* ]]; then
  read -rp "WARNING: This doesn't look like a local database. Reset $DATABASE
  [[ "$confirm" =~ ^[Yy]$ ]] || { log "Aborted"; exit 0; }
fi

log "Dropping database..."
npm run db:drop 2>/dev/null || true

log "Creating database..."
npm run db:create

log "Running migrations..."
npm run db:migrate

log "Seeding..."
npm run db:seed

log "Done - database reset complete"
```

New feature branch setup

```
#!/usr/bin/env bash
# scripts/new-feature.sh - set up a new feature branch
set -euo pipefail

if [ -z "${1:-}" ]; then
```

```
  echo "Usage: $0 <feature-name>"
  exit 1
fi

FEATURE_NAME=$1
BRANCH_NAME="feature/$FEATURE_NAME"

# Ensure main is up to date
git switch main
git pull --rebase origin main

# Create and push the branch
git switch -c "$BRANCH_NAME"
git push -u origin "$BRANCH_NAME"

echo "Created and pushed branch: $BRANCH_NAME"
echo "You're now on $BRANCH_NAME, ready to work"
```

Deployment script with rollback

```
#!/usr/bin/env bash
# scripts/deploy.sh - deploy with automatic rollback on failure
set -euo pipefail

ENV=${1:-staging}
log() { echo "[deploy:$ENV] $*"; }
die() { log "ERROR: $*" >&2; exit 1; }

require_env DATABASE_URL API_TOKEN

PREVIOUS_VERSION=$(git describe --tags --abbrev=0 2>/dev/null || echo "unknown")
```

```
CURRENT_VERSION=$(git describe --tags --always)

log "Deploying $CURRENT_VERSION (previous: $PREVIOUS_VERSION)"

# Run health check to verify current state
log "Pre-deploy health check..."
curl -sf "https://$ENV.example.com/health" || die "Pre-deploy health check fa

# Deploy
log "Running migrations..."
npm run db:migrate

log "Deploying application..."
npm run deploy:"$ENV"

# Post-deploy health check
log "Post-deploy health check..."
sleep 10
if ! curl -sf "https://$ENV.example.com/health"; then
  log "Health check failed - initiating rollback"
  git revert HEAD --no-edit
  npm run deploy:"$ENV"
  die "Deployment failed - rolled back to previous version"
fi

log "Deployment successful [OK]"
log "Version $CURRENT_VERSION is live on $ENV"
```

Putting It Together: A Complete Developer Automation Setup

Here's how all these pieces fit together into a complete project automation setup:

```
project/
|-- Makefile                    # top-level task runner
|-- scripts/
|   |-- setup.sh                # first-run environment setup
|   |-- dev.sh                  # start full development stack
|   |-- pre-commit.sh           # pre-commit quality checks
|   |-- db-reset.sh             # database reset
|   |-- new-feature.sh          # branch creation workflow
|   `-- deploy.sh               # deployment with health checks
|-- .env.example                # template for environment variables
|-- .envrc                      # direnv config (not committed if contains secrets)
`-- .git/hooks/
    `-- pre-commit -> ../../scripts/pre-commit.sh
```

The `Makefile` ties it all together as the single entry point:

```
.PHONY: help setup dev test lint build deploy reset

help: ## Show available commands
    @grep -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) | \
    awk 'BEGIN {FS = ":.*?## "}; {printf "\033[36m%-20s\033[0m %s\n", $$1, $$2}'

setup: ## Set up development environment from scratch
    ./scripts/setup.sh

dev: ## Start development server
```

```
    ./scripts/dev.sh

test: ## Run test suite
    npm test

lint: ## Run linter and type checker
    npx eslint src/ && npx tsc --noEmit

build: ## Build for production
    npm run build

reset: ## Reset database to clean state
    ./scripts/db-reset.sh

feature: ## Create a new feature branch (usage: make feature NAME=my-feature)
    ./scripts/new-feature.sh $(NAME)

deploy: ## Deploy to staging
    ./scripts/deploy.sh staging

deploy-prod: ## Deploy to production
    ./scripts/deploy.sh production
```

Any developer on the team runs `make setup` to get started, `make dev` to start working, and `make deploy` to ship. The underlying complexity — environment loading, health checks, rollback logic — is hidden behind a consistent interface that never changes.

Chapter Summary

Automation is one of the highest-leverage investments you can make in your development workflow. A shell script that takes an hour to write

but saves five minutes a day pays for itself in less than two weeks — and unlike manual processes, it doesn't vary, doesn't forget steps, and can be reviewed and improved over time.

The key habits to build:

- Start every shell script with `#!/usr/bin/env bash` and `set -euo pipefail`
- Use `trap cleanup EXIT` to ensure cleanup happens even when scripts fail
- Add a `Makefile` to every project as the standard entry point for common tasks
- Use `direnv` to manage project-specific environment variables automatically
- Validate required environment variables at the top of scripts before doing anything
- Write `make help` or a help target in every `Makefile` — future team members will thank you
- Use `trap`, locks, and health checks in deployment scripts to make failures recoverable

Exercises

1. Write a `setup.sh` script for a project you currently work on. It should install dependencies, copy `.env.example` to `.env` if it doesn't exist, run database migrations, and print a success message. Make it idempotent — safe to run multiple times.

2. Add a `Makefile` to a project you work on with at least five targets: `install`, `dev`, `test`, `lint`, and `clean`. Add `##` comments to each target and implement a self-documenting `help` target.

3. Write a shell function called `newpr` that: switches to main, pulls the latest changes, creates a new branch based on a name argument, and pushes it to origin with upstream tracking set. Add it to your shell config.

4. Set up `direnv` for a project. Move your `.env` contents to `.envrc` and verify that variables load automatically when you enter the directory and unload when you leave.

5. Write a deployment script for any project that includes: a pre-deploy health check, the deployment step itself, a post-deploy health check, and a rollback step that runs if the post-deploy check fails.

Quick Reference

Command / Pattern | What it does |

||| | `#!/usr/bin/env bash` | Portable shebang line | | `set -euo pipefail` | Exit on error, undefined vars, pipe failures | | `${VAR:-default}` | Variable with fallback default | | `${VAR:?error message}` | Variable or die with message | | `$(command)` | Command substitution | | `trap cleanup EXIT` | Run cleanup function on exit | | `[ -f file ]` | Test if file exists | | `[ -z "$VAR" ]` | Test if variable is empty | | `[[ "$str" =~ regex ]]` | Regex match in bash | | `source .env` | Load environment file | | `set -a; source .env; set +a` | Load .env and export all variables | | `make <target>` | Run a Makefile target | | `.PHONY: target` | Declare target as not a file | | `## comment` | Self-documenting Makefile target | | `watch -n 5 "command"` | Repeat command every 5 seconds | | `nohup cmd > log.txt 2>&1 &` | Run in background, capture output | | `crontab -e` | Edit scheduled tasks | | `0 2 * * *` | Cron: daily at 2am | | `*/15 * * * *` | Cron: every 15 minutes |

Terminal Quality of Life

There's a difference between using the terminal and *living* in the terminal. The developers in the first group open it when they have to, do what they need to do, and close it as soon as possible. The developers in the second group have it open all day, navigate it fluidly, and find that many tasks they once did in GUIs feel slower and more cumbersome by comparison.

The difference usually isn't knowledge of more commands. It's the accumulated effect of dozens of small improvements to the terminal environment itself — a faster prompt, smarter history, better search, tools that reduce friction at every turn. Individually, each improvement saves a few seconds. Together, they change the character of working in the terminal from something you endure to something you prefer.

This chapter covers those improvements. Some are configuration changes. Some are tools worth installing. Some are habits and keyboard shortcuts that take a week to internalize and then become automatic. None of them are complicated. All of them are worth the investment.

Choosing and Configuring Your Shell

Most systems default to `bash`. It's reliable, ubiquitous, and fine. But two alternatives are worth knowing about: `zsh` and `fish`.

Zsh

`zsh` has been the default shell on macOS since Catalina (2019). It's largely compatible with `bash` but adds several quality-of-life improvements out of the box:

- Better tab completion (interactive menus instead of flat lists)
- Spelling correction (`cd porjects` -> "Did you mean `projects`?")
- Shared history across terminal sessions
- More powerful globbing (`**/*.ts` matches recursively without `find`)
- Better array handling and string manipulation

If you're on macOS, you're likely already using `zsh`. On Linux, switch with:

```
chsh -s $(which zsh)
```

Oh My Zsh

Oh My Zsh is a framework for managing your zsh configuration. It provides a plugin system, a theme system, and a large collection of pre-built aliases and functions.

```
sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/tools/insta
```

Useful plugins to enable in `~/.zshrc`:

```
plugins=(
  git                  # git aliases and completion
  z                    # jump to frecent directories
  docker               # docker completion
  node                 # node version info
  history-substring-search   # better history search
```

```
  zsh-autosuggestions        # fish-style suggestions (install separately)
  zsh-syntax-highlighting    # command highlighting (install separately)
)
```

Fish

`fish` (Friendly Interactive Shell) takes a different approach — it's designed from the ground up for interactive use, with autosuggestions, syntax highlighting, and web-based configuration built in. Its scripting syntax is intentionally different from `bash`, which means bash scripts don't run directly in `fish` (you call them explicitly with `bash script.sh`).

For interactive daily use, fish is arguably the most polished shell available. For scripting, stick with bash. The two coexist without issues.

```
brew install fish              # macOS
chsh -s $(which fish)          # set as default
fish_config                    # web-based configuration UI
```

A Better Prompt with Starship

Your prompt is the text that appears before every command. The default prompt tells you very little — usually just your username, hostname, and current directory. A well-configured prompt can show your current git branch, whether you have uncommitted changes, the active Node or Python version, the last command's exit code, and more — all without you having to run a single command to find out.

Starship is a cross-shell prompt written in Rust. It's fast (adds less than 5ms to prompt rendering), works in any shell, and is configured with a single TOML file.

```
curl -sS https://starship.rs/install.sh | sh
```

Add to your shell config:

```
# ~/.zshrc
eval "$(starship init zsh)"

# ~/.bashrc
eval "$(starship init bash)"

# ~/.config/fish/config.fish
starship init fish | source
```

Configuring Starship

Starship is configured in `~/.config/starship.toml`:

```
# ~/.config/starship.toml

# Prompt format - what appears and in what order
format = """
$directory\
$git_branch\
$git_status\
$nodejs\
$python\
$cmd_duration\
$line_break\
$character"""

[directory]
truncation_length = 3
```

```
truncate_to_repo = true      # show path relative to git root

[git_branch]
symbol = " "
style = "bold purple"

[git_status]
conflicted = "!"
ahead = "up:${count}"
behind = "down:${count}"
diverged = "div:up:${ahead_count}:down:${behind_count}"
modified = "!"
untracked = "?"
staged = "+"
renamed = ">>"
deleted = "x"

[nodejs]
format = "[$symbol($version)]($style) "
symbol = " "

[python]
format = "[$symbol$version]($style) "

[cmd_duration]
min_time = 2_000             # show duration for commands taking over 2 seconds
format = "took [$duration]($style) "

[character]
success_symbol = "[>](bold green)"
error_symbol = "[>](bold red)"    # turns red after a failed command
```

The `$git_status` module alone is worth the setup — at a glance you can

see whether your working tree is clean, how many commits ahead or behind you are, and whether you have staged changes, all without running `git status`.

Smarter History

The default shell history is a flat list of commands you've run. With the right configuration and tools, it becomes a searchable, context-aware database of everything you've ever done at the terminal.

Increasing history size

The default history size on most systems is embarrassingly small — 500 or 1000 commands. Increase it dramatically:

```
# ~/.zshrc or ~/.bashrc
HISTSIZE=100000               # commands kept in memory
HISTFILESIZE=200000           # commands kept on disk
SAVEHIST=100000               # zsh: commands saved to file
```

Deduplication and timestamps

```
# ~/.zshrc
setopt HIST_IGNORE_DUPS       # don't store duplicate consecutive commands
setopt HIST_IGNORE_ALL_DUPS   # remove older duplicate entries
setopt HIST_IGNORE_SPACE      # don't store commands starting with a space
setopt HIST_VERIFY            # show expanded history command before running
setopt SHARE_HISTORY          # share history across all terminal sessions
setopt EXTENDED_HISTORY       # save timestamps with each command
```

```
# ~/.bashrc
HISTCONTROL=ignoreboth       # ignore duplicates and commands starting with space
HISTTIMEFORMAT="%Y-%m-%d %T "  # add timestamps
```

The `HIST_IGNORE_SPACE` / `ignorespace` option is a useful security feature — any command you prefix with a space won't be saved to history. Useful for commands that contain credentials you don't want persisted.

Reverse history search

`Ctrl+R` is built into both bash and zsh. Press it and start typing to search backward through your history:

```
(reverse-i-search)`curl': curl -s https://api.example.com/users | jq '.[].email'
```

Keep pressing `Ctrl+R` to cycle through older matches. Press `Enter` to run the matched command, or the right arrow to edit it first.

`fzf`: fuzzy history search

`fzf` is a general-purpose fuzzy finder. When integrated with your shell, it replaces the default `Ctrl+R` history search with an interactive full-screen interface that searches your entire history with fuzzy matching:

```
brew install fzf
$(brew --prefix)/opt/fzf/install    # install shell integrations
```

Once installed, `Ctrl+R` opens a scrollable, searchable list of your entire command history. Type any substring to filter it — you don't need to remember the exact command, just a fragment of it. Arrow keys navigate, `Enter` selects, `Ctrl+C` cancels.

`fzf` also integrates with `Ctrl+T` (fuzzy file finder — paste a file path into your command) and `Alt+C` (fuzzy directory navigation). These three key bindings alone make `fzf` one of the highest-value tools in this entire book.

Smarter Directory Navigation

`cd` is the most-typed command in the terminal. Most developers type it far more than necessary.

z: jump to frecent directories

`z` tracks the directories you visit most frequently and most recently (hence "frecent") and lets you jump to them by typing a fragment of the path:

```
# Install via oh-my-zsh plugin, or:
brew install zoxide            # modern alternative (recommended)
```

After visiting a directory a few times:

```
z proj                         # jumps to ~/work/projects/myproject
z api                          # jumps to ~/work/projects/myproject/src/api
z doc                          # jumps to ~/work/projects/myproject/docs
```

`z` learns your patterns over time. After a few days of normal use, you rarely need to type full paths — a two or three character fragment is usually enough to get to where you want to go.

`zoxide`: the modern replacement

`zoxide` is a faster, more featureful replacement for `z`, also written in Rust:

```
brew install zoxide
echo 'eval "$(zoxide init zsh)"' >> ~/.zshrc    # or bash, fish
```

Usage is identical to `z`, but `zoxide` also provides `zi` — an interactive mode that uses `fzf` to let you choose between multiple matching directories:

```
zi                              # fuzzy search all visited directories
zi proj                         # fuzzy search directories matching "proj"
```

Better `cd` with `AUTO_CD`

In zsh, you can navigate to a directory by just typing its name — no `cd` required:

```
# ~/.zshrc
setopt AUTO_CD
```

With this set, typing `src` in your terminal changes to the `src` directory if it exists. Combined with `zoxide`, directory navigation becomes extremely fast.

`pushd` and `popd`: directory stack

Less well-known than `cd`, but useful when you're working across multiple directories:

```
pushd /tmp                      # go to /tmp, remember current location
pushd ~/projects/myapp          # go to myapp, stack now has two entries
dirs -v                         # view the directory stack
popd                            # return to ~/projects/myapp
popd                            # return to original directory
```

Think of it as a browser's back button for the terminal. For quick context switches where you know you'll need to return, `pushd`/`popd` is cleaner than typing paths twice.

`bat`, `eza`, and `fd`: Modernizing the Classics

We mentioned these tools in earlier chapters. Here's the full picture on setup and configuration.

`bat`: a better `cat`

```
brew install bat
```

`bat` adds syntax highlighting, line numbers, and git change markers to file viewing. To use it as a drop-in `cat` replacement:

```
# ~/.zshrc
alias cat="bat --paging=never"    # no paging (behaves like cat)
alias less="bat"                  # use bat's pager with syntax highlighting
```

Configure `bat` in `~/.config/bat/config`:

```
--theme="Dracula"
--style="numbers,changes,header"
--paging=never
```

`bat` is also used by `fzf` to show syntax-highlighted previews in its interface — a combination we'll cover shortly.

`eza`: a better `ls`

```
brew install eza
```

```
# ~/.zshrc
alias ls="eza"
alias ll="eza -lah"
alias lt="eza -lah --sort=modified"
alias tree="eza --tree"
```

`eza` adds color coding by file type, git status indicators in the file listing, and a built-in tree mode. Its `--git` flag shows the git status of each file alongside the listing — the most common reason to run `git status` and `ls` in the same breath:

```
eza -lah --git                  # show git status for each file
eza --tree --git-ignore         # tree view, respecting .gitignore
```

`fd`: a better `find`

```
brew install fd
```

`fd` has a simpler syntax than `find`, respects `.gitignore` by default, and is significantly faster:

```
fd "*.ts"                       # find TypeScript files
fd -t d tests                   # find directories named "tests"
fd -e ts -e tsx                 # find by extension
fd --changed-within 1d          # files modified in the last day
fd "config" --exec bat {}       # find and view with bat
```

The `--exec` flag works like `find`'s `-exec` but with a cleaner syntax. `{}` is replaced with the matching filename.

`fzf` Beyond History Search

`fzf` is more than a history search tool. It's a general-purpose fuzzy selector that can be composed with any command that produces a list of items.

Fuzzy file opening

```
# Open a file in your editor using fuzzy search
vim $(fzf)
code $(fzf)

# With bat preview
fzf --preview 'bat --color=always {}'
```

Git with `fzf`

```
# Interactively checkout a branch
git switch $(git branch | fzf | tr -d '[:space:]')

# Interactively add files to staging
git add $(git status -s | fzf -m | awk '{print $2}')

# Interactively view commit history
git log --oneline | fzf --preview 'git show {1} --stat --color=always'
```

Kill a process interactively

```
# Select a process to kill from an interactive list
kill $(ps aux | fzf | awk '{print $2}')
```

`fzf` functions worth adding to your shell config

```
# Fuzzy cd: navigate to any subdirectory with fzf
fcd() {
  local dir
  dir=$(find . -type d -not -path '*/node_modules/*' -not -path '*/.git/*' | \
    fzf --preview 'eza --tree --level=1 {}') && cd "$dir"
}

# Fuzzy edit: find and open a file in $EDITOR
fe() {
  local file
  file=$(fd --type f --hidden --exclude .git | \
    fzf --preview 'bat --color=always {}') && ${EDITOR:-vim} "$file"
}
```

```
# Fuzzy grep: search file contents and open at matching line
fg() {
  local result file line
  result=$(rg --line-number --color=always "$1" | \
    fzf --ansi --preview 'bat --color=always {1} --highlight-line {2}')
  file=$(echo "$result" | cut -d: -f1)
  line=$(echo "$result" | cut -d: -f2)
  ${EDITOR:-vim} "$file" +"$line"
}
```

The `fg` function is particularly powerful: it uses `rg` to search for a pattern, `fzf` to let you choose the match, and opens the file at the matching line in your editor. It replaces the common workflow of "search for something, note the file and line, open the file, navigate to the line."

Multiplexers: `tmux`

A terminal multiplexer lets you run multiple terminal sessions within a single window, split panes side by side, and keep sessions running after you disconnect. `tmux` is the most widely used.

```
brew install tmux              # macOS
apt install tmux               # Ubuntu
```

Core concepts

`tmux` has three levels of organization:

- **Sessions**: independent workspaces, each with their own windows and panes

- **Windows**: tabs within a session (like browser tabs)
- **Panes**: splits within a window (side by side or top/bottom)

The default prefix key is `Ctrl+B` — hold `Ctrl`, press `B`, release both, then press the next key.

Essential key bindings

```
Ctrl+B c       New window
Ctrl+B n       Next window
Ctrl+B p       Previous window
Ctrl+B 0-9     Switch to window by number
Ctrl+B ,       Rename current window

Ctrl+B %       Split pane vertically (side by side)
Ctrl+B "       Split pane horizontally (top/bottom)
Ctrl+B arrows  Navigate between panes
Ctrl+B z       Zoom current pane (toggle fullscreen)
Ctrl+B x       Close current pane

Ctrl+B d       Detach from session (session keeps running)
tmux attach    Reattach to last session
tmux ls        List running sessions
tmux new -s work   New session named "work"
tmux attach -t work  Attach to session named "work"
```

A practical tmux layout for development

A common layout for web development:

```
+-----------------+------------------+
|                 |                  |
```

```
|    Editor       |   Dev Server     |
|                 |                  |
|                 +------------------+
|                 |                  |
|                 |   Test Runner    |
|                 |                  |
+-----------------+------------------+
```

Set this up with a script:

```
#!/usr/bin/env bash
# scripts/dev-session.sh - start development tmux session

SESSION="dev"

# Don't create if already exists
tmux has-session -t $SESSION 2>/dev/null && tmux attach -t $SESSION && exit

tmux new-session -d -s $SESSION -n "editor"

# Main editor pane
tmux send-keys -t $SESSION "cd ~/projects/myapp && $EDITOR ." Enter

# Split right: dev server
tmux split-window -h -t $SESSION
tmux send-keys -t $SESSION "cd ~/projects/myapp && make dev" Enter

# Split bottom right: tests
tmux split-window -v -t $SESSION
tmux send-keys -t $SESSION "cd ~/projects/myapp" Enter

# Focus back on editor
tmux select-pane -t $SESSION:0.0
```

```
tmux attach -t $SESSION
```

Now `./scripts/dev-session.sh` opens your full development environment in one command — editor, dev server, and test runner all running, each in its own pane. If your connection drops or you close your laptop, `tmux attach` brings everything back exactly as you left it.

`.tmux.conf`: essential configuration

The defaults in `tmux` are serviceable but not great. A minimal `~/.tmux.conf`:

```
# Remap prefix from Ctrl+B to Ctrl+A (easier to reach)
unbind C-b
set-option -g prefix C-a
bind-key C-a send-prefix

# Split panes with | and - (more intuitive)
bind | split-window -h
bind - split-window -v
unbind '"'
unbind %

# Reload config with Ctrl+A r
bind r source-file ~/.tmux.conf \; display "Config reloaded"

# Navigate panes with vim keys
bind h select-pane -L
bind j select-pane -D
bind k select-pane -U
bind l select-pane -R
```

```
# Enable mouse support
set -g mouse on

# Increase scrollback buffer
set -g history-limit 50000

# Start windows and panes at 1 (not 0)
set -g base-index 1
setw -g pane-base-index 1

# Enable true color support
set -g default-terminal "screen-256color"
set -ga terminal-overrides ",*256col*:Tc"

# Status bar
set -g status-bg black
set -g status-fg white
set -g status-left "#[bold]#S "
set -g status-right "%H:%M %d-%b"
```

Keyboard Shortcuts Worth Memorizing

Terminal keyboard shortcuts are multiplicative — they work in any command, in any context, without any setup. Spending thirty minutes learning them saves hours every week.

Line editing (readline shortcuts)

These work in bash, zsh, and any readline-enabled program:

```
Ctrl+A          Move to beginning of line
```

```
Ctrl+E          Move to end of line
Alt+B           Move back one word
Alt+F           Move forward one word
Ctrl+W          Delete word before cursor
Alt+D           Delete word after cursor
Ctrl+U          Delete from cursor to beginning of line
Ctrl+K          Delete from cursor to end of line
Ctrl+Y          Paste what was last deleted (yank)
Ctrl+L          Clear screen (like running 'clear')
Ctrl+C          Cancel current command
Ctrl+D          Send EOF (exit shell if line is empty)
Ctrl+Z          Suspend current process (resume with 'fg')
```

`Ctrl+W` and `Ctrl+Y` together form a cut/paste pair — delete a word, move to where you want it, paste it back. `Ctrl+U` and `Ctrl+Y` do the same for an entire line.

History navigation

```
Ctrl+R          Reverse search through history
Ctrl+P          Previous command (like up arrow)
Ctrl+N          Next command (like down arrow)
!!              Repeat last command
!$              Last argument of last command
!^              First argument of last command
!*              All arguments of last command
!git            Last command starting with "git"
```

`!$` is particularly useful: `mkdir new-directory && cd !$` — create a directory and immediately cd into it using the last argument from the previous command.

Process control

```
Ctrl+C          Interrupt (kill) current process
Ctrl+Z          Suspend current process
fg              Resume suspended process in foreground
bg              Resume suspended process in background
jobs            List background and suspended jobs
```

Terminal Emulators Worth Considering

The terminal emulator is the application that runs your shell. The default options — Terminal.app on macOS, gnome-terminal on Linux — work fine, but several alternatives offer meaningful improvements for developers.

iTerm2 (macOS)

iTerm2 is the most popular terminal emulator for macOS developers. Notable features:

- **Split panes** without needing tmux
- **Shell integration** — shows command history, marks command boundaries, tracks working directory
- **Semantic history** — `Cmd+click` on a filename or URL opens it
- **Inline images** — display images directly in the terminal
- **Tmux integration** — use tmux sessions with iTerm2's native tab/pane UI

```
brew install --cask iterm2
```

Warp (macOS and Linux)

Warp is a newer terminal emulator that rethinks the interface more aggressively. Commands and their output are grouped into blocks you can select, copy, and share. It has built-in AI assistance, a command palette, and a modern text editing experience in the input area.

For developers who find the traditional terminal interface frustrating, Warp is worth trying. For developers who are already comfortable in the terminal, the changes can feel disorienting at first.

Alacritty

Alacritty is a GPU-accelerated terminal emulator focused entirely on performance. It has minimal features by design — no tabs, no splits — but renders faster than any other option. If you use tmux for session management, Alacritty is a fast, lightweight host for it.

Ghostty

Ghostty is a newer terminal emulator that balances performance with features, with native platform integration on macOS and Linux. It's worth watching if you want a fast emulator that doesn't sacrifice all convenience features.

A Recommended Setup

Here's a complete recommended setup for a developer who wants to invest in their terminal environment once and benefit for years:

Shell: zsh with Oh My Zsh, or fish for a more opinionated experience

Prompt: Starship — fast, informative, works with any shell

Tools to install:

```
brew install \
  fzf \          # fuzzy finder
  zoxide \       # smart directory navigation
  bat \          # better cat
  eza \          # better ls
  fd \           # better find
  ripgrep \      # better grep
  jq \           # JSON processing
  tmux \         # terminal multiplexer
  starship       # prompt
```

Shell config additions (`~/.zshrc`):

```
# Tool integrations
eval "$(starship init zsh)"
eval "$(zoxide init zsh)"
$(brew --prefix)/opt/fzf/install --all --no-update-rc

# Aliases
alias cat="bat --paging=never"
alias ls="eza"
alias ll="eza -lah --git"
alias lt="eza -lah --sort=modified"
alias find="fd"
alias grep="rg"

# History
HISTSIZE=100000
SAVEHIST=100000
setopt SHARE_HISTORY
```

```
setopt HIST_IGNORE_DUPS
setopt HIST_IGNORE_SPACE
setopt EXTENDED_HISTORY

# Navigation
setopt AUTO_CD
alias ..="cd .."
alias ...="cd ../.."

# fzf functions
fe() {
  local file
  file=$(fd --type f | fzf --preview 'bat --color=always {}') && \
    ${EDITOR:-vim} "$file"
}
```

Tmux: configured with `~/.tmux.conf` from section 8.7, with a per-project dev session script.

This setup takes about an hour to install and configure. The payoff — in raw speed, reduced friction, and the genuine pleasure of a well-tuned environment — is felt every working day for years afterward.

Chapter Summary

A well-configured terminal environment is an investment that compounds over time. The tools and settings in this chapter don't just make individual tasks faster — they change the quality of the experience of working at the command line, from something that requires deliberate effort to something that feels fluid and natural.

The key habits to build:

- Install and configure Starship — the information density in a good prompt pays for itself immediately
- Spend one session learning and practicing the readline shortcuts — `Ctrl+A/E`, `Alt+B/F`, `Ctrl+W/K/U/Y` — until they're automatic
- Install `fzf` and use `Ctrl+R` for history search every day until it's instinctive
- Set up `zoxide` and let it learn your navigation patterns for a week before judging it
- Install `bat`, `eza`, and `fd` and alias them over their classic equivalents
- Give `tmux` a genuine two-week trial — the learning curve is real, but so is the payoff

Exercises

1. Install Starship and configure it to show at minimum: current directory, git branch, git status, and the exit code indicator (green/red prompt character). Use it for a full work day and note what information you find yourself checking that the prompt now gives you for free.

2. Install `fzf` and its shell integrations. Spend one day using `Ctrl+R` for every history search instead of pressing the up arrow. Notice how often you find the command you want faster than you expected.

3. Install `zoxide` and use it exclusively for directory navigation for one week. Note how quickly it learns your patterns and how rarely you need to type full paths.

4. Install `tmux` and create a dev session script for a project you work on. The script should create a named session with at least two panes — one for editing and one for running the dev server. Practice detaching and reattaching with `Ctrl+A d` and `tmux attach`.

5. Audit your shell configuration. Add timestamps and deduplication to your history. Set `HISTSIZE` to at least 50,000. Add at least five aliases for

commands you type frequently. Source the config and verify everything works.

Quick Reference

Essential tools

Tool | Install | What it does |

|||| | `starship` | `brew install starship` | Fast, informative cross-shell prompt | | `fzf` | `brew install fzf` | Fuzzy finder for history, files, anything | | `zoxide` | `brew install zoxide` | Smart frecent directory navigation | | `bat` | `brew install bat` | Syntax-highlighted cat replacement | | `eza` | `brew install eza` | Modern ls with git integration | | `fd` | `brew install fd` | Fast, gitignore-aware find replacement | | `tmux` | `brew install tmux` | Terminal multiplexer |

Readline shortcuts

Shortcut | What it does |

||| | `Ctrl+A / Ctrl+E` | Beginning / end of line | | `Alt+B / Alt+F` | Back / forward one word | | `Ctrl+W` | Delete word before cursor | | `Ctrl+U / Ctrl+K` | Delete to beginning / end of line | | `Ctrl+Y` | Paste last deleted text | | `Ctrl+R` | Reverse history search | | `Ctrl+L` | Clear screen | | `Ctrl+Z` | Suspend process | | `!!` | Repeat last command | | `!$` | Last argument of last command |

tmux shortcuts (with Ctrl+A prefix)

Shortcut | What it does |

||| | `Ctrl+A c` | New window | | `Ctrl+A \|` | Split vertically | | `Ctrl+A -` | Split horizontally | | `Ctrl+A arrows` | Navigate panes | | `Ctrl+A z` | Zoom pane | | `Ctrl+A d` | Detach session | | `tmux attach` | Reattach to session | | `tmux new -s name` | New named session |

Process Management and Debugging

Every developer has been there. A development server that stopped responding but didn't exit. A test runner that's been "running" for forty minutes. A port that's supposedly in use by a process you can't identify. A script that behaves differently on your machine than on CI, and you can't figure out why the environment is different.

These situations have something in common: they're not code problems. They're *process* problems — questions about what's running, what it's doing, what environment it's operating in, and how to control it. Most developers handle them by restarting their machine, killing their terminal session, or spending twenty minutes guessing. The terminal gives you precise tools to diagnose and resolve them in seconds.

This chapter covers the tools for understanding and managing processes, debugging environment issues, and wrapping commands defensively so they fail gracefully rather than hanging forever. These aren't glamorous tools — you won't use them every day — but when you need them, nothing else will do.

Understanding What's Running with `ps`

`ps` (process status) shows information about running processes. It's been part of Unix since the beginning, and its flags are notoriously inconsistent across platforms — a legacy of competing Unix standards that was never fully resolved. The version you'll use most often sidesteps this by using BSD-style flags that work on both macOS and Linux:

```
ps aux
```

Breaking down the flags: - `a` — show processes from all users, not just your own - `u` — user-oriented format (shows username, CPU, memory) - `x` — include processes not attached to a terminal

The output looks like this:

```
USER       PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAN
alice     1234   0.0  0.1  4286548  12345   ??  S    10:22AM   0:00.12 node s
alice     5678  98.3  2.4  8432156  98765   ??  R    10:45AM  12:34.56 pythor
root       123   0.0  0.0  4198572   1234   ??  Ss   9:01AM    0:01.23 /usr/s
```

The columns you'll look at most often:

- **PID** — process ID, the number you use to send signals to a process
- **%CPU** — CPU usage percentage
- **%MEM** — memory usage percentage
- **STAT** — process state: `R` (running), `S` (sleeping), `Z` (zombie), `T` (stopped)
- **COMMAND** — the command that started the process

Filtering `ps` output

`ps aux` on a busy system produces hundreds of lines. Combine with `grep` to find what you're looking for:

```
ps aux | grep node                    # find Node.js processes
ps aux | grep -v grep | grep python   # find Python processes (exclude grep i
ps aux | grep "port 3000"             # find process mentioning port 3000
```

The `grep -v grep` idiom is a classic — without it, the `grep` command itself shows up in the results because its command line contains the search term.

Finding the most resource-hungry processes

```
ps aux --sort=-%cpu | head -10        # top 10 by CPU (Linux)
ps aux --sort=-%mem | head -10        # top 10 by memory (Linux)
ps aux | sort -rk3 | head -10         # top 10 by CPU (macOS/portable)
ps aux | sort -rk4 | head -10         # top 10 by memory (macOS/portable)
```

`pgrep`: finding process IDs cleanly

When you just need the PID of a named process — not the full listing — `pgrep` is cleaner than `ps aux | grep`:

```
pgrep node                            # PID(s) of node processes
pgrep -l node                         # PIDs with process names
pgrep -a node                         # PIDs with full command lines
pgrep -u alice node                   # node processes owned by alice
```

`pgrep` returns just the PIDs, which makes it easy to compose with other commands:

```
pgrep -a node | wc -l                 # how many node processes are running?
```

Controlling Processes with `kill`

`kill` sends a signal to a process. Despite the name, it doesn't only kill — signals can pause, resume, reload, or terminate a process depending on which signal you send.

Common signals

```
kill -15 <PID>          # SIGTERM: polite termination request (default)
kill <PID>              # same as kill -15
kill -9 <PID>           # SIGKILL: immediate, forceful termination
kill -1 <PID>           # SIGHUP: reload config (for daemons)
kill -19 <PID>          # SIGSTOP: pause process
kill -18 <PID>          # SIGCONT: resume paused process
```

The difference between SIGTERM (`-15`) and SIGKILL (`-9`) matters:

SIGTERM is a request. The process receives it and can handle it gracefully — flushing buffers, closing connections, cleaning up temporary files — before exiting. Well-behaved processes respect SIGTERM.

SIGKILL is not a request. The kernel terminates the process immediately, without giving it a chance to clean up. Use it only when SIGTERM doesn't work — a hung process, an unresponsive daemon, or a process that has explicitly ignored SIGTERM.

The right approach is always to try SIGTERM first:

```
kill 1234               # try polite termination first
sleep 5
kill -9 1234            # force kill if still running
```

`pkill`: kill by name

`pkill` is to `kill` what `pgrep` is to `ps` — it finds processes by name and sends a signal to them:

```
pkill node                          # SIGTERM all node processes
pkill -9 node                       # SIGKILL all node processes
pkill -u alice node                 # kill alice's node processes only
pkill -f "node server.js"           # match against full command line
```

The `-f` flag matches against the full command line rather than just the process name — useful when multiple processes share a binary name but have different arguments:

```
pkill -f "node src/api-server.js"     # kill only the API server, not other node processes
```

Killing a process on a specific port

One of the most common process management tasks in web development:

```
# Find what's on port 3000
lsof -ti tcp:3000

# Kill it
kill $(lsof -ti tcp:3000)

# One-liner to kill whatever is on a port
kill -9 $(lsof -ti tcp:3000)
```

`lsof` (list open files) with `-ti tcp:<port>` returns just the PID of the process listening on that port. The `$()` wraps it as a subshell, feeding the PID directly to `kill`. This is worth turning into a shell function:

```
# In ~/.zshrc or ~/.bashrc
killport() {
  local port=$1
```

```
  local pid=$(lsof -ti tcp:"$port")
  if [ -z "$pid" ]; then
    echo "Nothing running on port $port"
  else
    kill -9 $pid
    echo "Killed process $pid on port $port"
  fi
}
```

Killing a process tree

When a process has spawned child processes, killing the parent doesn't always kill the children. To kill an entire process tree:

```
# Kill process and all its children (Linux)
kill -- -$(pgrep -o node)            # send signal to process group

# More explicit
pkill -P <parent-pid>                # kill children of a specific PID
```

`top` and `htop`: Live Process Monitoring

`ps` gives you a snapshot. `top` gives you a live view, refreshing every few seconds:

```
top
```

Inside `top`, useful keys: - `q` — quit - `k` — kill a process (prompts for PID) - `u` — filter by user - `o` — sort by a column - `1` — toggle showing individual CPU cores

`top` is available everywhere. `htop` is a significantly improved version with color, mouse support, and a more intuitive interface:

```
brew install htop              # macOS
apt install htop               # Ubuntu
```

`htop` adds visual CPU and memory bars, easier process selection, and the ability to kill processes without needing to type their PID. If `htop` is available, use it. If not, `top` is always there.

`lsof`: what files does a process have open?

`lsof` (list open files) is a versatile diagnostic tool. In Unix, everything is a file — including network connections, pipes, and devices. `lsof` shows all of them:

```
lsof -p <PID>                          # all files open by a process
lsof -p <PID> | grep -i "\.log"        # log files a process has open
lsof -i                                # all network connections
lsof -i tcp                            # TCP connections only
lsof -i tcp:3000                       # what's using port 3000
lsof -i :3000                          # same, shorter form
lsof +D /var/log                       # all processes with files open in a directory
lsof /path/to/file                     # which process has this file open
```

The last form is invaluable when you're trying to delete or modify a file and getting "resource busy" errors — `lsof /path/to/file` tells you exactly which process is holding it open.

`timeout`: Wrapping Commands Defensively

A command that hangs is worse than a command that fails. A failed command gives you an error and exits. A hanging command blocks your pipeline, your script, or your CI job indefinitely — until something external intervenes.

`timeout` wraps a command and kills it if it runs longer than a specified duration:

```
timeout 30 curl https://api.example.com/slow-endpoint
timeout 5m npm test
timeout 1h ./scripts/long-migration.sh
```

Time units: `s` (seconds, default), `m` (minutes), `h` (hours), `d` (days).

`timeout` exits with code `124` if the time limit was reached. This lets you handle the timeout case explicitly in scripts:

```
timeout 30 curl -s https://api.example.com/health
exit_code=$?

if [ $exit_code -eq 124 ]; then
  echo "Health check timed out after 30 seconds" >&2
  exit 1
elif [ $exit_code -ne 0 ]; then
  echo "Health check failed with code $exit_code" >&2
  exit 1
else
  echo "Health check passed"
fi
```

The signal sent on timeout

By default, `timeout` sends SIGTERM when the time limit is reached. If the process doesn't respond to SIGTERM within a grace period, you can configure it to follow up with SIGKILL:

```
timeout --kill-after=5s 30s curl https://api.example.com/slow-endpoint
```

This sends SIGTERM after 30 seconds, then SIGKILL 5 seconds later if the process is still running. For processes that might ignore SIGTERM, this guarantees termination.

Using `timeout` in scripts

Any operation that involves a network call, an external service, or an operation of unbounded duration should be wrapped in `timeout` in production scripts:

```
#!/usr/bin/env bash
set -euo pipefail

# Wait for database to be ready, with a timeout
DB_READY=false
for i in $(seq 1 30); do
  if timeout 2 bash -c "echo > /dev/tcp/localhost/5432" 2>/dev/null; then
    DB_READY=true
    break
  fi
  echo "Waiting for database... ($i/30)"
  sleep 2
done
```

```
$DB_READY || { echo "Database did not become ready in time" >&2; exit 1; }

# Run migrations with a timeout
echo "Running migrations..."
timeout 5m npm run db:migrate || {
  echo "Migrations timed out or failed" >&2
  exit 1
}
```

Environment Debugging with `env`, `printenv`, and `export`

A significant class of bugs in development — scripts that work locally but fail in CI, applications that behave differently across environments, commands that can't find other commands — trace back to environment variables. These tools help you see exactly what environment a process is operating in.

`env` and `printenv`: inspecting the environment

```
env                              # print all environment variables
printenv                         # same output, slightly different format
env | sort                       # sorted for easier reading
env | grep PATH                  # find PATH and related variables
env | grep -i "api\|token\|key"  # find potential credentials
```

`printenv` can also look up specific variables:

```
printenv PATH                       # print PATH
printenv HOME SHELL EDITOR          # print multiple variables
printenv NODE_ENV                   # check current Node environment
```

Understanding `PATH`

Most "command not found" errors come down to `PATH` — the list of directories the shell searches when you type a command. When something works interactively but not in a script, or works for one user but not another, check `PATH` first:

```
echo $PATH
echo $PATH | tr ':' '\n'            # one directory per line, easier to read
```

A common issue: a tool installed by `brew`, `nvm`, or `pyenv` is available in your interactive shell but not in scripts or cron jobs, because those run with a minimal `PATH` that doesn't include the custom directories.

To see what PATH a cron job will have:

```
env -i bash -c 'echo $PATH'         # simulate a minimal environment
```

`env -i` starts with a completely empty environment — it shows you exactly what `PATH` looks like with none of your shell customizations applied.

Running a command with a modified environment

`env` can also set or override variables for a single command without affecting the current shell:

```
env NODE_ENV=production node server.js        # run with specific env var
env -u HOME node server.js                    # run with HOME unset
env NODE_ENV=test DATABASE_URL=sqlite://memory npm test    # multiple overri
```

This is cleaner than `export`-ing a variable, running a command, and then unsetting it — and it doesn't affect other commands running in the same shell.

`export`: making variables available to child processes

A subtle but important distinction: setting a variable in your shell doesn't automatically make it available to programs you run from that shell:

```
MY_VAR="hello"
bash -c 'echo $MY_VAR'           # prints nothing - MY_VAR wasn't exported

export MY_VAR="hello"
bash -c 'echo $MY_VAR'           # prints "hello"
```

`export` marks a variable to be inherited by child processes. This is why environment setup scripts use `export` for every variable — and why `.env` file loaders use `set -a` (which automatically exports everything) or `export $(cat .env | xargs)`.

Debugging environment differences

When a script works interactively but fails in CI, the issue is almost always environment. This diagnostic pattern helps isolate it:

```
# Add near the top of a failing script to dump the full environment
echo "=== Environment ===" >&2
env | sort >&2
echo "=== PATH ===" >&2
echo $PATH | tr ':' '\n' >&2
echo "=== Which ===" >&2
which node npm python 2>&1 >&2
echo "==================" >&2
```

Redirecting to stderr (`>&2`) ensures this diagnostic output doesn't interfere with the script's normal stdout output if it's being piped or captured.

`which`, `type`, and `command`: Resolving Command Mysteries

"Command not found" and "this isn't the version I expected" are the two most common environment debugging scenarios. These tools resolve them.

`which`: finding a command's location

```
which node                          # /usr/local/bin/node
which python                        # /usr/bin/python
which python3                       # /opt/homebrew/bin/python3
which git                           # /usr/bin/git
```

`which` searches your `PATH` and returns the full path of the first matching executable. When you have multiple versions of a tool installed — multiple

Pythons, multiple Nodes — `which` shows you which one will be used when you type the bare command name.

To see *all* matching executables in PATH (not just the first):

```
which -a python                    # shows all python executables in PATH orde
which -a node
```

This is useful when you suspect a version manager like `nvm`, `pyenv`, or `rbenv` isn't working correctly — `which -a` shows you every version on your PATH and their order of precedence.

`type`: more information than `which`

`type` is a shell builtin that goes further than `which` — it distinguishes between external executables, shell builtins, shell functions, and aliases:

```
type ls                            # ls is an alias for eza
type cd                            # cd is a shell builtin
type grep                          # grep is /usr/bin/grep
type ll                            # ll is a shell function
```

This is important because `which` only finds external executables — it won't find aliases or shell functions. When a command behaves differently than expected, `type` tells you exactly what it is:

```
type git                           # is it the git you think it is?
type python                        # which python is this?
type make                          # is this the system make or a custom one?
```

`command -v`: the portable alternative

In scripts, `which` isn't always available and its behavior varies across systems. `command -v` is the POSIX-portable way to check whether a command exists:

```
command -v node                    # prints path if found, nothing if not
command -v node &>/dev/null && echo "node is installed" || echo "node not found"
```

This is the pattern to use in setup scripts when checking for required tools:

```
for tool in node npm git curl jq; do
  if ! command -v "$tool" &>/dev/null; then
    echo "Error: $tool is required but not installed" >&2
    exit 1
  fi
done
```

`hash`: clearing the command cache

Shells cache the location of commands after the first lookup. If you install a new version of a tool and the shell still finds the old one, the cache is the culprit:

```
hash -r                            # clear entire command cache
hash -d node                       # clear cache for a specific command
```

After running `hash -r` or starting a new shell session, the next invocation of any command will do a fresh PATH lookup.

Debugging with `strace` and `dtrace`

For the hardest process debugging problems — a process that's doing something unexpected and you can't figure out what — system call tracing tools let you see exactly what a process is doing at the operating system level.

`strace` (Linux)

`strace` intercepts and records system calls made by a process — every file it opens, every network connection it makes, every signal it receives:

```
strace ls                                    # trace system calls made by l
strace -p <PID>                              # attach to a running process
strace -e trace=file ls                      # trace only file-related cal
strace -e trace=network curl example.com     # trace only network calls
strace -o output.txt node server.js          # write trace to file
```

The output is verbose — `ls` makes dozens of system calls — but it's the ground truth of what a process is doing. When a process claims it can't find a file, `strace -e trace=file` shows you exactly which paths it's searching and why each one fails.

`dtruss` (macOS)

macOS uses `dtruss` as the equivalent of `strace`:

```
sudo dtruss ls                               # requires sudo on macOS
sudo dtruss -p <PID>                         # attach to running process
```

System call tracing is a power tool — reach for it when all other debugging approaches have failed. But when you need it, it provides an unambiguous view of exactly what a process is doing that no other tool can match.

Background Jobs and Process Groups

We covered background jobs briefly in Chapter 7. Here's the fuller picture from a process management perspective.

Job control

```
./long-running-script.sh &          # run in background immediately
Ctrl+Z                              # suspend the foreground process
bg                                  # resume suspended process in background
fg                                  # bring background process to foreground
fg %2                               # bring specific job to foreground
jobs                                # list all background and suspended jobs
jobs -l                             # list with PIDs
```

`wait`: waiting for background jobs

In scripts that launch multiple background processes, `wait` blocks until they complete:

```
#!/usr/bin/env bash
set -euo pipefail

# Run three tasks in parallel
./scripts/run-tests.sh &
```

```
PID_TESTS=$!

./scripts/build-assets.sh &
PID_BUILD=$!

./scripts/generate-docs.sh &
PID_DOCS=$!

# Wait for all three and check exit codes
wait $PID_TESTS || { echo "Tests failed" >&2; exit 1; }
wait $PID_BUILD || { echo "Build failed" >&2; exit 1; }
wait $PID_DOCS  || { echo "Docs generation failed" >&2; exit 1; }

echo "All tasks completed successfully"
```

This pattern — launch multiple background jobs, capture their PIDs with `$!`, then `wait` for each one — is how you parallelize work in a shell script without reaching for a more complex tool.

Process substitution

A more advanced technique: process substitution treats the output of a command as a file:

```
diff <(sort file1.txt) <(sort file2.txt)     # diff sorted versions without t
comm <(sort list1.txt) <(sort list2.txt)     # find common and unique lines
wc -l <(find . -name "*.ts")                 # count matching files
```

`<(command)` creates a temporary named pipe and passes it as a file argument. This avoids the need for intermediate temporary files in many situations.

Practical Debugging Workflows

Scenario 1: Something is using my port

```
# Find what's on port 3000
lsof -i :3000

# Output:
# COMMAND   PID   USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
# node     1234  alice   23u  IPv6 0x...            0t0  TCP *:3000 (LISTEN)

# Kill it
kill $(lsof -ti :3000)

# Verify the port is free
lsof -i :3000        # should return nothing
```

Scenario 2: A script works locally but fails in CI

```
# Check what environment CI is running in
# Add to the failing script temporarily:
echo "Node: $(node --version)"
echo "npm: $(npm --version)"
echo "PATH: $PATH"
echo "Working directory: $(pwd)"
echo "User: $(whoami)"
env | sort | grep -E "NODE|NPM|CI|PATH"

# Check if the command even exists in CI's PATH
command -v npx || echo "npx not found"
which node || echo "node not in PATH"
```

Scenario 3: A process is consuming too much CPU

```
# Find the offender
ps aux --sort=-%cpu | head -5

# Get more detail on the process
ps -p <PID> -o pid,ppid,cmd,%cpu,%mem,etime

# Find all files it has open (might indicate what it's stuck on)
lsof -p <PID> | tail -20

# Check if it's in a tight loop or waiting
# Look at STAT column: R = running, S = sleeping, D = waiting on IO
ps -p <PID> -o stat
```

Scenario 4: A script hangs occasionally in CI

```
#!/usr/bin/env bash
# Wrap every potentially hanging operation with timeout

# Database health check - shouldn't take more than 5 seconds
timeout 5 bash -c 'until pg_isready -h localhost; do sleep 0.5; done' || {
  echo "Database not ready after 5 seconds" >&2
  exit 1
}

# External API call - 30 second limit
timeout 30 curl -sf https://api.example.com/health || {
  echo "API health check failed or timed out" >&2
  exit 1
}
```

```
# Test suite - should complete within 10 minutes
timeout 10m npm test || {
  exit_code=$?
  if [ $exit_code -eq 124 ]; then
    echo "Test suite timed out after 10 minutes" >&2
  else
    echo "Test suite failed with code $exit_code" >&2
  fi
  exit 1
}
```

Scenario 5: "Command not found" that makes no sense

```
# The full diagnostic sequence
which mycommand                    # is it in PATH?
type mycommand                     # is it an alias, function, or builtin?
echo $PATH | tr ':' '\n'           # what directories are in PATH?
ls /usr/local/bin | grep my        # is it installed but not in PATH?
hash -r && mycommand               # clear cache and retry

# Check if it exists but isn't executable
ls -la $(which mycommand 2>/dev/null || echo "/usr/local/bin/mycommand")

# Check if it's a different architecture (Apple Silicon issue)
file $(which mycommand)            # is it x86_64 on an ARM machine?
```

Chapter Summary

Process management and environment debugging feel like niche skills until the moment you need them — and then they feel like the most important

skills in the book. The tools in this chapter give you complete visibility into what's running on your system, what environment it's operating in, and how to control or terminate it precisely.

The key habits to build:

- Always try SIGTERM before SIGKILL — give processes a chance to clean up
- Use `lsof -i :<port>` as your first move when a port appears to be in use
- Wrap any network call or unbounded operation in `timeout` in scripts — hanging is worse than failing
- Use `type` rather than `which` when debugging command resolution — it finds aliases and functions too
- Add `env | sort` and `which` diagnostics to scripts that fail in CI but work locally
- Use `command -v` in scripts to check for required tools — it's portable and reliable
- Keep `kill -9` as a last resort, not a first instinct

Exercises

1. Find the process using the most CPU on your system right now using `ps aux`. Get its PID, then use `lsof -p <PID>` to see what files it has open. What do those open files tell you about what the process is doing?

2. Start a simple HTTP server (`python3 -m http.server 8080`), then use `lsof -i :8080` to find its PID. Kill it using `kill $(lsof -ti :8080)` and verify the port is free.

3. Write a shell function called `killport` that takes a port number as an argument, checks whether anything is running on that port, and kills it if so — or prints a message if the port is free. Add it to your shell config.

4. Write a script that launches three commands in parallel using `&`, captures their PIDs with `$!`, and uses `wait` to check whether each one succeeded. If any fail, the script should report which ones and exit with a non-zero code.

5. Find a command you use regularly and run `type`, `which -a`, and `command -v` on it. If you have version managers like `nvm` or `pyenv` installed, check how they affect the output. Make sure you understand exactly which binary runs when you type that command.

Quick Reference

Command | What it does |

||| | `ps aux` | Show all running processes | | `ps aux \| grep node` | Find processes by name | | `ps aux --sort=-%cpu \| head` | Top processes by CPU | | `pgrep -a node` | Find PIDs by process name | | `pkill node` | Kill processes by name | | `pkill -f "node server.js"` | Kill by full command line | | `kill <PID>` | Send SIGTERM (polite) | | `kill -9 <PID>` | Send SIGKILL (forceful) | | `lsof -i :<port>` | What process is on this port? | | `kill $(lsof -ti :<port>)` | Kill process on a port | | `lsof -p <PID>` | Files open by a process | | `top` / `htop` | Live process monitor | | `timeout 30 <command>` | Kill command after 30 seconds | | `timeout --kill-after=5s 30s <cmd>` | SIGTERM then SIGKILL | | `env` | Print all environment variables | | `env \| sort \| grep PATH` | Find PATH-related variables | | `env NODE_ENV=prod node app.js` | Run with modified environment | | `env -i bash -c 'echo $PATH'` | Simulate minimal environment | | `which node` | Find command location in PATH | | `which -a python` | Find all versions in PATH | | `type ls` | Is it alias, builtin, or binary? | | `command -v node` | Portable command existence check | | `hash -r` | Clear shell command cache | | `jobs -l` | List background jobs with PIDs | | `wait $PID` | Wait for background job | | `strace -p <PID>` | Trace system calls (Linux) |

Composing Tools with Pipes and Redirection

Every tool in this book has been useful on its own. `rg` finds patterns. `jq` transforms JSON. `sed` edits streams. `awk` processes columns. `curl` makes HTTP requests. But if you've been paying attention, you've noticed that the most powerful examples in every chapter weren't single commands — they were chains. `rg` feeding into `sort`, feeding into `uniq`, feeding into `head`. `curl` feeding into `jq`, feeding into another `curl`. `find` feeding into `xargs`, feeding into `sed`.

That chaining is not a coincidence or a stylistic choice. It's the central design principle of Unix, articulated by Doug McIlroy in 1964 and built into every tool ever written for the platform since: *write programs that do one thing well, write programs that work together, and write programs that handle text streams, because that is a universal interface.*

The pipe — the `|` character — is the mechanism that makes "work together" possible. Redirection operators extend it to files. `xargs` bridges the gap between tools that weren't designed to connect. And process substitution handles the cases where the model needs to bend.

This chapter is about understanding these mechanisms deeply enough that composing tools becomes instinctive — not something you look up, but something you reach for automatically whenever a problem involves more than one step.

The Unix Philosophy in Practice

Before the mechanics, the mental model. Unix tools are designed around three standard streams:

- **stdin** (standard input, file descriptor 0) — where a program reads input from
- **stdout** (standard output, file descriptor 1) — where a program writes normal output
- **stderr** (standard error, file descriptor 2) — where a program writes error messages

By default, stdin comes from the keyboard, and stdout and stderr go to the terminal. The pipe operator redirects stdout from one program into stdin of the next — connecting them without either program knowing or caring that the connection exists.

This is the key insight: `grep` doesn't know whether it's reading from a file, from a pipe, or from a network socket. It just reads stdin and writes stdout. That ignorance is what makes composition possible — any tool that reads stdin and writes stdout can be connected to any other tool that does the same.

```
command1 | command2 | command3
```

`command1` writes to stdout. The pipe redirects that into `command2`'s stdin. `command2` processes it and writes to its own stdout. The pipe redirects that into `command3`'s stdin. `command3` produces the final output.

Each step is independent. Each step can be tested alone. And the whole pipeline does something none of the steps could do individually.

The Pipe Operator |

The pipe is the most important character in the Unix terminal. Here's how to think about it and use it effectively.

Basic piping

```
# Count lines in a file
cat access.log | wc -l

# Find and count unique values
cat access.log | awk '{print $9}' | sort | uniq -c | sort -rn

# Search and view with paging
rg "ERROR" logs/ | less

# Filter and format
ps aux | grep node | awk '{print $1, $2, $11}'
```

Building pipelines incrementally

The right way to build a pipeline is one step at a time. Start with the first command and verify its output. Add the second command and verify. Continue until the pipeline does what you want.

Suppose you want to find the ten most common IP addresses in an access log that returned 500 errors:

```
# Step 1: look at the raw data
head -5 access.log
```

```
# Step 2: extract the status code and IP
awk '{print $1, $9}' access.log | head -5

# Step 3: filter to 500s only
awk '{print $1, $9}' access.log | awk '$2 == 500 {print $1}' | head -5

# Step 4: count and sort
awk '{print $1, $9}' access.log | awk '$2 == 500 {print $1}' | sort | uniq -
```

Each step builds on the last. If something looks wrong, you can remove the last command and inspect the intermediate output. This incremental approach is how experienced terminal users build even complex pipelines — not in one shot, but one verified step at a time.

Pipelines are not sequential — they're concurrent

An important and often misunderstood point: commands in a pipeline run simultaneously, not one after another. The moment `command1` starts writing output, `command2` starts reading it. This means a pipeline like:

```
cat huge-file.log | grep "ERROR" | wc -l
```

doesn't wait for `cat` to finish before starting `grep`. All three commands run at once, with data flowing between them. For large files, this is significantly faster than processing sequentially would be — and it means you can pipe the output of a long-running command into `less` and start reading before the command finishes.

Redirection Operators

Pipes connect commands to each other. Redirection operators connect commands to files.

Output redirection

```
command > file.txt          # redirect stdout to file (overwrite)
command >> file.txt         # redirect stdout to file (append)
command 2> errors.txt       # redirect stderr to file
command 2>> errors.txt      # append stderr to file
command > file.txt 2>&1     # redirect both stdout and stderr to file
command &> file.txt         # same, bash shorthand
command > /dev/null         # discard stdout
command 2> /dev/null        # discard stderr
command &> /dev/null        # discard all output
```

The `2>&1` syntax is worth understanding deeply. `2>` redirects file descriptor 2 (stderr). `&1` means "to wherever file descriptor 1 (stdout) is currently going." So `> file.txt 2>&1` means "redirect stdout to file.txt, then redirect stderr to wherever stdout is going" — which is also file.txt. Order matters: `2>&1 > file.txt` does something different (redirects stderr to the terminal, then stdout to the file).

Input redirection

```
command < file.txt          # read stdin from file
command < input.txt > output.txt    # read from file, write to file
```

Input redirection is less common than output redirection — most tools accept filenames directly. But it's useful with commands that only read from stdin:

```
# Send an email with body from a file
mail -s "Report" alice@example.com < report.txt

# Feed SQL from a file to a database client
psql mydb < schema.sql
```

Here documents and here strings

A here document (<<) lets you provide multi-line input to a command inline:

```
cat << EOF
Hello, world.
This is a multi-line
here document.
EOF

# With variable substitution
name="Alice"
cat << EOF
Dear $name,
Your report is ready.
EOF

# Without variable substitution (quoted delimiter)
cat << 'EOF'
This $variable will not be expanded.
EOF
```

Here documents are used constantly in shell scripts for generating configuration files, SQL queries, and API payloads:

```
# Generate a config file
cat > config.yaml << EOF
database:
  host: $DB_HOST
  port: $DB_PORT
  name: $DB_NAME
EOF

# Run a multi-line SQL query
psql mydb << EOF
BEGIN;
UPDATE users SET active = false WHERE last_login < NOW() - INTERVAL '90 days';
DELETE FROM sessions WHERE expires_at < NOW();
COMMIT;
EOF
```

A here string (`<<<`) is a single-line variant that feeds a string directly to a command's stdin:

```
wc -w <<< "count the words in this string"
base64 --decode <<< "SGVsbG8gV29ybGQ="
jq '.' <<< '{"name":"Alice","role":"admin"}'
```

`xargs`: Bridging the Gap

The pipe model works perfectly when the receiving command reads from stdin. But many commands don't read from stdin — they take arguments. `find`, `rm`, `cp`, `mv`, `grep` with a list of files — these commands expect their input as command-line arguments, not on stdin.

`xargs` bridges this gap. It reads items from stdin and passes them as arguments to a command:

```
# Delete all .log files found by find
find . -name "*.log" | xargs rm

# Search for a pattern in files found by rg
rg "TODO" -l | xargs grep -l "FIXME"

# Count lines in all TypeScript files
find . -name "*.ts" | xargs wc -l

# Run prettier on all changed files
git diff --name-only | xargs npx prettier --write
```

Handling filenames with spaces

The default `xargs` splits on any whitespace, which breaks filenames containing spaces. The `-0` flag (combined with `find`'s `-print0` or `rg`'s `--null`) uses null bytes as delimiters instead:

```
find . -name "*.ts" -print0 | xargs -0 wc -l
rg "TODO" -l --null | xargs -0 grep -l "FIXME"
```

This is the safe form to use in scripts — it handles all filenames correctly regardless of what characters they contain.

Controlling argument placement with `-I`

By default, `xargs` appends arguments at the end of the command. The `-I` flag lets you specify exactly where they go using a placeholder:

```
# The {} placeholder is replaced with each argument
find . -name "*.ts" | xargs -I {} cp {} {}.backup

# Open each found file in vim sequentially
rg "TODO" -l | xargs -I {} vim {}

# Create a directory structure
echo -e "src\ntests\ndocs" | xargs -I {} mkdir -p {}
```

Parallel execution with `-P`

`xargs` can run multiple instances of a command in parallel with `-P`:

```
# Process 4 files at once
find . -name "*.jpg" | xargs -P 4 -I {} convert {} -resize 800x600 {}.resized

# Run tests in parallel across multiple files
find tests/ -name "*.test.ts" | xargs -P 8 -I {} npx jest {}
```

`-P 4` runs up to 4 processes simultaneously. Combined with `-I {}`, each file gets its own process instance. For CPU-bound or IO-bound batch operations, this can dramatically reduce total processing time.

`-n`: controlling argument count

By default, `xargs` passes as many arguments as possible to each invocation. `-n` limits how many arguments each invocation receives:

```
echo "a b c d e f" | xargs -n 2 echo    # runs: echo a b, echo c d, echo e f
```

This is useful when a command has a maximum number of arguments it can handle, or when you want to process items one at a time:

```
cat urls.txt | xargs -n 1 curl -sO      # download each URL one at a time
```

Advanced Redirection Patterns

Redirecting to multiple destinations with `tee`

We covered `tee` in Chapter 4 in the context of file editing. Here's how it fits into the broader redirection picture.

`tee` reads stdin and writes it to both stdout *and* one or more files simultaneously — splitting the stream:

```
make build 2>&1 | tee build.log          # see output and save to file
npm test | tee test-results.txt | grep -E "PASS|FAIL"    # save full output,
```

`tee` with process substitution (covered next) can split a stream into multiple pipelines:

```
cat access.log | tee \
  >(grep "ERROR" | wc -l > error-count.txt) \
  >(grep "200" | wc -l > success-count.txt) \
  >(awk '{print $1}' | sort | uniq > unique-ips.txt) \
  > /dev/null
```

This reads the log file once and produces three output files simultaneously — error count, success count, and unique IPs. Without `tee` and process substitution, you'd need to read the file three times.

Process substitution <() and >()

Process substitution is one of the more powerful but less commonly known bash features. It allows a command's output or input to be treated as a file:

```
# <() - treat command output as a file
diff <(sort file1.txt) <(sort file2.txt)      # diff sorted versions of two files
diff <(ssh server1 cat /etc/hosts) <(ssh server2 cat /etc/hosts)  # diff remote files
comm <(sort list1.txt) <(sort list2.txt)      # find common and unique lines

# >() - treat a file argument as a command's stdin
tee >(gzip > output.gz) > /dev/null           # compress output on the fly
echo "data" | tee >(wc -w) >(wc -c) > /dev/null   # count words and chars simultaneously
```

Process substitution is useful whenever a command requires a filename argument but you want to provide processed output instead of an actual file — without creating a temporary file.

Named pipes (FIFOs)

For more complex inter-process communication, named pipes (FIFOs) let unrelated processes communicate through the filesystem:

```
# Create a named pipe
mkfifo /tmp/mypipe

# In terminal 1: write to the pipe
command1 > /tmp/mypipe

# In terminal 2: read from the pipe
command2 < /tmp/mypipe
```

```
# Clean up
rm /tmp/mypipe
```

Named pipes are less commonly needed than anonymous pipes, but they're the right tool when two processes can't easily be connected with `|` — for example, when they're running in different terminal sessions or started by different mechanisms.

Stderr and Error Handling in Pipelines

Stderr and pipelines interact in ways that catch developers by surprise.

Stderr bypasses pipes

By default, stderr is not passed through pipes — it goes directly to the terminal even when stdout is being piped:

```
# Only stdout goes to grep; errors appear on terminal regardless
find /etc -name "*.conf" 2>/dev/null | grep "ssh"

# Capture both stdout and stderr in the pipe
find /etc -name "*.conf" 2>&1 | grep "ssh"
```

Whether you want to include stderr in a pipeline depends on the situation. When processing command output, you usually want to suppress error messages with `2>/dev/null`. When debugging a failing command, you want to capture everything with `2>&1`.

Pipeline exit codes

By default, a pipeline's exit code is the exit code of the *last* command. This means a failing command in the middle of a pipeline can go unnoticed:

```
# This exits 0 even if rg fails (because wc -l succeeds)
rg "pattern" nonexistent-dir | wc -l
echo $?    # 0 - misleadingly indicates success
```

`set -o pipefail` (which we covered in Chapter 7 as part of `set -euo pipefail`) changes this — it makes a pipeline fail if *any* command in it fails:

```
set -o pipefail
rg "pattern" nonexistent-dir | wc -l
echo $?    # non-zero - correctly reflects the rg failure
```

Always use `set -euo pipefail` in scripts. The `pipefail` option in particular prevents a whole class of silent failures in multi-step pipelines.

Checking individual pipeline exit codes with `PIPESTATUS`

In bash, `PIPESTATUS` is an array containing the exit codes of each command in the most recent pipeline:

```
command1 | command2 | command3
echo "${PIPESTATUS[@]}"          # e.g., "0 1 0" - command2 failed
echo "${PIPESTATUS[0]}"          # exit code of command1
echo "${PIPESTATUS[1]}"          # exit code of command2
```

This is useful in scripts that need to handle failures in specific pipeline stages differently:

```
rg "ERROR" logs/ | sort | uniq -c | sort -rn > error-report.txt
rg_status=${PIPESTATUS[0]}

if [ $rg_status -eq 1 ]; then
  echo "No errors found - report is empty"
elif [ $rg_status -ne 0 ]; then
  echo "Search failed with code $rg_status" >&2
  exit 1
fi
```

Building One-Liners That Replace Scripts

The highest expression of the Unix pipeline model is the one-liner: a single command that performs a complete, meaningful task. One-liners aren't about showing off — they're about solving a problem at the speed of thought, without creating a file, opening an editor, or writing boilerplate.

Here are patterns that come up repeatedly in real development work, with the reasoning that connects each step.

Log analysis

```
# Top 10 most frequent errors in the last hour
grep "$(date '+%Y-%m-%d %H')" app.log \
  | grep "ERROR" \
  | awk '{$1=$2=$3=""; print $0}' \
  | sort | uniq -c | sort -rn \
  | head -10
```

Step by step: filter to the current hour -> filter to errors -> strip timestamps -> count occurrences -> sort by frequency -> show top 10.

```
# Response time percentiles from an access log
awk '{print $NF}' access.log \
  | sort -n \
  | awk 'BEGIN{c=0} {a[c++]=$1} END{
      print "p50:", a[int(c*0.5)],
      "p95:", a[int(c*0.95)],
      "p99:", a[int(c*0.99)]
    }'
```

Code analysis

```
# Find the most complex files (by line count) excluding generated code
find . -name "*.ts" \
  -not -path "*/node_modules/*" \
  -not -path "*/dist/*" \
  -not -name "*.d.ts" \
  | xargs wc -l \
  | sort -rn \
  | head -20
```

```
# Count TODOs per author using git blame
git ls-files "*.ts" \
  | xargs grep -l "TODO" \
  | xargs -I {} git blame --line-porcelain {} \
  | grep "^author " \
  | sort | uniq -c | sort -rn
```

```
# Find duplicate function names across the codebase
rg "^(export )?function \w+" --type ts -o \
  | sed 's/.*:export function //' \
```

```
  | sed 's/.*:function //' \
  | sort | uniq -d
```

Data processing

```
# Sum a column in a CSV, skipping the header
tail -n +2 sales.csv \
  | awk -F',' '{sum += $3} END {printf "Total: $%.2f\n", sum}'
```

```
# Find all unique dependency versions across a monorepo
find . -name "package.json" \
  -not -path "*/node_modules/*" \
  | xargs jq -r '.dependencies // {} | to_entries[] | "\(.key)@\(.value)"' \
  | sort | uniq -c | sort -rn \
  | awk '$1 > 1'    # show only packages with multiple versions
```

```
# Convert a .env file to JSON (for use with a config service)
cat .env \
  | grep -v '^#' \
  | grep '=' \
  | awk -F'=' '{print $1"="$2}' \
  | jq -Rn '[inputs | split("=") | {(.[0]): .[1:] | join("=")}] | add'
```

System investigation

```
# Find the largest directories in the current tree
du -sh */ 2>/dev/null \
  | sort -rh \
  | head -10
```

```
# Find files changed in the last 24 hours, sorted by modification time
find . -mtime -1 \
  -not -path "*/.git/*" \
  -not -path "*/node_modules/*" \
  | xargs ls -lt 2>/dev/null \
  | head -20
```

```
# Summarize git activity by author for the past month
git log --since="1 month ago" --format="%an" \
  | sort | uniq -c | sort -rn
```

Composing the Tools From This Book

The real value of understanding pipes and redirection is that it multiplies the value of everything else you've learned. Here's how the tools from previous chapters combine:

Chapter 1 (navigation) + Chapter 2 (search) + Chapter 9 (pipes)

```
# Find all TypeScript files modified this week, search them for a pattern,
# and open the results in less
find . -name "*.ts" -mtime -7 -not -path "*/node_modules/*" \
  | xargs rg "useEffect" -l \
  | xargs rg "useEffect" -n \
  | less
```

Chapter 3 (reading) + Chapter 4 (editing) + Chapter 9 (pipes)

```
# Read a log file, extract error messages, clean them up, and save a report
tail -n 10000 app.log \
  | grep "ERROR" \
  | sed 's/\[ERROR\] //' \
  | sed 's/[0-9]\{4\}-[0-9]\{2\}-[0-9]\{2\}T[0-9:\.Z]*//g' \
  | sort | uniq -c | sort -rn \
  | tee error-report.txt \
  | head -20
```

Chapter 5 (git) + Chapter 6 (APIs) + Chapter 9 (pipes)

```
# Get the last 5 commit hashes, fetch CI status for each from GitHub API,
# and display a summary
git log --oneline -5 --format="%H %s" | while read -r sha message; do
  status=$(curl -s \
    -H "Authorization: Bearer $GITHUB_TOKEN" \
    "https://api.github.com/repos/org/repo/commits/$sha/status" \
    | jq -r '.state')
  printf "%-10s %-8s %s\n" "${sha:0:7}" "$status" "$message"
done
```

Chapter 7 (automation) + Chapter 8 (process management) + Chapter 9 (pipes)

```
# Run tests in parallel, collect results, kill all if any fail
find tests/ -name "*.test.ts" -print0 \
  | xargs -0 -P 4 -I {} sh -c \
```

```
    'timeout 60 npx jest {} --json 2>/dev/null || echo "FAILED: {}"' \
  | tee test-output.log \
  | grep "FAILED" \
  | head -5
```

When Not to Use a Pipeline

Pipelines are powerful, but they're not always the right tool. Knowing when *not* to use them is as important as knowing how.

When you need error handling at each step. Pipelines don't make it easy to handle errors from intermediate commands differently. If you need to know whether step 3 of 7 failed and respond specifically to that failure, a script with explicit error checks is clearer and more maintainable than a pipeline with `PIPESTATUS` gymnastics.

When the logic becomes unreadable. A seven-command pipeline where each command has multiple flags and options is hard to debug, hard to modify, and hard for colleagues to understand. The threshold is fuzzy, but when you find yourself spending more time deciphering a pipeline than it would take to write it as a proper script, write the script.

When you need to process the same data multiple times. Pipelines are single-pass. If you need to make three different analyses of the same large dataset, running three separate pipelines means reading the data three times. In that case, save the intermediate result to a file and process it multiple times — or use `tee` with process substitution to split the stream.

When you need structured data across steps. Pipelines pass text. If your intermediate data has complex structure that's awkward to represent as text, passing it through a pipeline means serializing and deserializing at each step. At some point, a proper script or program that keeps data in memory as structured objects is the better choice.

The Unix pipeline model is one of the best tools ever designed for text processing and command composition. It's not a replacement for all other forms of programming — it's a complement to them.

Chapter Summary

The pipe and redirection system is the connective tissue of the Unix toolkit. Individual commands are useful. Commands composed into pipelines are transformative. The tools you've learned throughout this book — `rg`, `jq`, `sed`, `awk`, `curl`, `git`, `find`, `xargs` — reach their full potential only when combined.

The key habits to build:

- Build pipelines incrementally — verify each step before adding the next
- Always use `set -o pipefail` in scripts — silent pipeline failures cause some of the hardest-to-debug problems
- Use `2>/dev/null` to suppress noise when you only care about successful output; use `2>&1` when you need to capture everything
- Reach for `xargs -0` with `find -print0` or `rg --null` — null-delimited is always the safe choice for filenames
- Use `tee` when you need both to see output and save it — don't choose between them
- Know when to stop: a pipeline that requires five minutes of reading to understand should probably be a script

Exercises

1. Build a pipeline incrementally: start with `ps aux`, then add a filter to show only your own processes, then extract just the PID and command

name columns, then sort by memory usage. Verify the output at each step before adding the next command.

2. Use `tee` and process substitution to read a log file once and simultaneously produce three files: one containing only ERROR lines, one containing only WARN lines, and one containing a count of each log level.

3. Use `xargs -P` to process a batch of files in parallel. Try converting a directory of text files to uppercase using `tr` — run it with `-P 1`, `-P 4`, and `-P 8` and compare the time with the `time` command.

4. Write a pipeline that uses `git log`, `awk`, `sort`, and `uniq` to produce a leaderboard of commit counts by author for the current repository.

5. Find a multi-step task you currently do manually or with a script. Rewrite it as a pipeline. Then consider: is the pipeline clearer than the script? Faster? Harder to maintain? When would you choose each form?

Quick Reference

Pipes and redirection

Operator | What it does |

||| | `cmd1 \| cmd2` | Pipe stdout of cmd1 to stdin of cmd2 | | `cmd > file` | Redirect stdout to file (overwrite) | | `cmd >> file` | Redirect stdout to file (append) | | `cmd 2> file` | Redirect stderr to file | | `cmd > file 2>&1` | Redirect stdout and stderr to file | | `cmd &> file` | Same, bash shorthand | | `cmd > /dev/null` | Discard stdout | | `cmd 2> /dev/null` | Discard stderr | | `cmd < file` | Read stdin from file | | `cmd << EOF` | Here document (multi-line stdin) | | `cmd <<< "string"` | Here string (single-line stdin) |

Composition tools

Command | What it does |

||| | `cmd \| tee file` | Write to file and pass through to stdout | | `cmd \| tee >(cmd2)` | Split stream to another pipeline | | `diff <(cmd1) <(cmd2)` | Diff output of two commands | | `cmd1 \| xargs cmd2` | Pass stdin items as arguments | | `find . -print0 \| xargs -0 cmd` | Null-safe argument passing | | `xargs -I {} cmd {}` | Control argument placement | | `xargs -P 4 cmd` | Run 4 parallel instances | | `xargs -n 1 cmd` | One argument per invocation |

Error handling

Pattern | What it does |

||| | `set -o pipefail` | Fail pipeline if any command fails | | `echo "${PIPESTATUS[@]}"` | Exit codes of last pipeline's commands | | `cmd 2>&1 \| grep "ERROR"` | Include stderr in pipe | | `cmd 2>/dev/null \| next` | Suppress errors, pipe only stdout |

Working on Remote Machines

Most developers work locally most of the time. But at some point, every developer ends up on a remote machine — SSH'd into a production server to investigate an incident, working on a cloud-based development environment, running jobs on a remote build server, deploying to a staging environment, or debugging something that only reproduces in a specific infrastructure configuration.

The terminal is the native interface for remote machines. There is no GUI, no IDE plugin that fully abstracts the distance, no substitute for knowing how to work effectively over SSH. And yet most developers treat SSH as a single command — `ssh user@host` — and discover everything else only when they urgently need it and don't have time to learn it properly.

This chapter covers SSH from the ground up: configuration that makes connecting fast and painless, transferring files reliably, tunneling ports for local access to remote services, multiplexing connections for speed, and using `tmux` over SSH so that your work survives disconnections. By the end of it, working on a remote machine will feel almost as fluid as working locally.

SSH Basics and Key Authentication

SSH (Secure Shell) is the protocol for securely connecting to remote machines. The basic command:

```
ssh username@hostname
ssh username@192.168.1.100
ssh username@server.example.com
```

Key-based authentication

Password authentication works but is slow, inconvenient, and less secure than key-based authentication. Key-based authentication uses a cryptographic key pair — a private key that stays on your machine and a public key that goes on the remote server.

Generate a key pair if you don't have one:

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

`ed25519` is the modern, recommended key type — faster and more secure than the older `RSA`. The `-C` flag adds a comment to help identify the key. Accept the default location (`~/.ssh/id_ed25519`) and set a passphrase when prompted — the passphrase encrypts your private key so it's useless if stolen.

Copy your public key to a remote server:

```
ssh-copy-id username@hostname
```

This appends your public key to `~/.ssh/authorized_keys` on the remote server. After this, you can log in without a password.

If `ssh-copy-id` isn't available:

```
cat ~/.ssh/id_ed25519.pub | ssh username@hostname "mkdir -p ~/.ssh && cat >>
```

The ssh-agent

Typing your key passphrase every time you SSH somewhere defeats the convenience of key authentication. `ssh-agent` holds your decrypted private key in memory for the duration of a session:

```
eval "$(ssh-agent -s)"           # start the agent
ssh-add ~/.ssh/id_ed25519        # add your key (prompts for passphrase once)
ssh-add -l                       # list loaded keys
```

On macOS, the system Keychain integrates with `ssh-agent` so your passphrase is remembered across reboots. Add to `~/.ssh/config`:

```
Host *
  AddKeysToAgent yes
  UseKeychain yes
```

The SSH Config File

Typing `ssh -i ~/.ssh/id_ed25519 -p 2222 -l alice server.example.com` every time you connect to a server is tedious and error-prone. The SSH config file — `~/.ssh/config` — lets you define aliases and settings for every host you connect to regularly.

Basic host configuration

```
# ~/.ssh/config

Host myserver
  HostName server.example.com
  User alice
```

```
  Port 2222
  IdentityFile ~/.ssh/id_ed25519

Host staging
  HostName staging.example.com
  User deploy
  IdentityFile ~/.ssh/deploy_key

Host prod
  HostName production.example.com
  User deploy
  IdentityFile ~/.ssh/deploy_key
  ForwardAgent yes
```

With this config:

```
ssh myserver                    # connects as alice to server.example.com or
ssh staging                     # connects to staging with deploy key
ssh prod                        # connects to production
```

The alias becomes the argument to every SSH-related command — `ssh`, `scp`, `rsync`, `git` over SSH — which means you type `rsync -av files/ myserver:~/files/` instead of `rsync -av -e "ssh -i ~/.ssh/id_ed25519 -p 2222" files/ alice@server.example.com:~/files/`.

Wildcard patterns

```
# Settings that apply to all hosts
Host *
  ServerAliveInterval 60
  ServerAliveCountMax 3
  AddKeysToAgent yes
```

```
# Settings for all servers in a domain
Host *.example.com
  User alice
  IdentityFile ~/.ssh/id_ed25519

# Jump through a bastion for internal servers
Host internal-*
  ProxyJump bastion.example.com
  User alice
```

`ServerAliveInterval` and `ServerAliveCountMax` are worth setting globally — they send keepalive packets to prevent idle SSH connections from being dropped by firewalls and NAT devices, which is the most common cause of "why did my SSH session disconnect?"

Important config options

```
# Reuse existing connections (covered in detail in section 11.4)
ControlMaster auto
ControlPath ~/.ssh/control/%r@%h:%p
ControlPersist 10m

# Forward your local ssh-agent to the remote server
ForwardAgent yes

# Compress data (useful on slow connections)
Compression yes

# Connect through a jump host (bastion)
ProxyJump bastion.example.com

# Use a specific local port for a remote service
```

```
LocalForward 5432 localhost:5432
```

`ForwardAgent yes` allows the remote server to use your local SSH keys — useful when you need to pull from a private git repository on a remote server without copying your private key there. Use it only with servers you trust.

Transferring Files

`scp`: simple file copying

`scp` (secure copy) copies files between local and remote systems over SSH:

```
# Local to remote
scp file.txt alice@server.example.com:~/
scp file.txt myserver:~/documents/

# Remote to local
scp myserver:~/logs/app.log ./
scp alice@server.example.com:~/data/export.csv ./exports/

# Directory (recursive)
scp -r src/ myserver:~/project/src/

# With SSH config alias
scp -r dist/ staging:~/app/dist/
```

`scp` is simple and always available. For anything beyond copying a single file, `rsync` is the better choice.

`rsync`: the right tool for file transfer

`rsync` is faster and more capable than `scp` for most file transfer tasks. It transfers only the differences between source and destination, handles interruptions gracefully, and has a rich set of options for controlling exactly what gets transferred.

```
# Basic sync: local directory to remote
rsync -av src/ myserver:~/project/src/

# The flags you'll almost always want
rsync -avz --progress src/ myserver:~/project/src/
```

Breaking down the flags: - `-a` — archive mode: preserves permissions, timestamps, symlinks, and recurses into directories - `-v` — verbose: shows files being transferred - `-z` — compress data during transfer (useful on slow connections) - `--progress` — show transfer progress for each file

The trailing slash matters

`rsync`'s behavior with trailing slashes is a source of endless confusion. It's worth understanding precisely:

```
rsync -av src/ myserver:~/dest/     # copy CONTENTS of src into dest
rsync -av src  myserver:~/dest/     # copy src DIRECTORY into dest (creates dest/src/)
```

With a trailing slash on the source, `rsync` copies the *contents* of the directory. Without it, `rsync` copies the directory itself. When in doubt, use a trailing slash on the source.

Useful `rsync` options

```
# Dry run: show what would be transferred without doing it
rsync -avz --dry-run src/ myserver:~/project/

# Delete files on destination that no longer exist on source
rsync -avz --delete src/ myserver:~/project/

# Exclude files and directories
rsync -avz --exclude='node_modules/' --exclude='*.log' src/ myserver:~/proje

# Use a specific SSH key or port
rsync -avz -e "ssh -i ~/.ssh/deploy_key -p 2222" dist/ deploy@server:~/app/

# Limit bandwidth (useful to avoid saturating a connection)
rsync -avz --bwlimit=1000 large-files/ myserver:~/storage/    # 1000 KB/s li

# Resume an interrupted transfer
rsync -avz --partial --progress large-file.tar.gz myserver:~/
```

The `--dry-run` flag is worth using before any `rsync` command that includes `--delete` — it shows you exactly what will be changed without making any changes.

Syncing from remote to local

`rsync` works equally well in both directions:

```
# Pull logs from remote server
rsync -avz myserver:~/logs/ ./remote-logs/
```

```
# Back up a remote directory locally
rsync -avz --delete myserver:~/project/ ./backups/project/
```

SSH Multiplexing: Eliminating Connection Overhead

Every time you run an SSH command — `ssh`, `scp`, `rsync`, `git push` over SSH — it establishes a new TCP connection, performs the cryptographic handshake, and authenticates. On a fast local network this takes a fraction of a second. On a remote server with higher latency, it can take 1–3 seconds. Multiply that by dozens of operations and it adds up.

SSH multiplexing reuses an existing connection for subsequent SSH operations to the same host, reducing the overhead to near zero.

Enabling multiplexing

Add to `~/.ssh/config`:

```
Host *
  ControlMaster auto
  ControlPath ~/.ssh/control/%r@%h:%p
  ControlPersist 10m
```

Then create the control directory:

```
mkdir -p ~/.ssh/control
chmod 700 ~/.ssh/control
```

That's it. The next time you SSH to a host, a master connection is established. For the next 10 minutes (`ControlPersist 10m`), every subsequent SSH command to the same host reuses that connection instantly — no handshake, no authentication delay.

Practical impact

With multiplexing enabled:

```
ssh myserver "ls -la"              # first connection: ~1-2 seconds
ssh myserver "cat logs/app.log"    # reuses connection: ~50ms
scp myserver:~/data.csv ./         # reuses connection: near-instant
rsync -avz src/ myserver:~/src/    # reuses connection: near-instant
```

For workflows that involve many sequential SSH operations — deployments, log inspection, configuration changes — multiplexing is one of the highest-value configuration changes you can make.

Managing multiplexed connections

```
# Check if a master connection exists
ssh -O check myserver

# Stop the master connection
ssh -O stop myserver

# Exit all multiplexed connections
ssh -O exit myserver
```

Port Forwarding and Tunneling

SSH tunneling allows you to securely access services on a remote machine (or network) as if they were running locally. This is one of SSH's most powerful and underused features.

Local port forwarding

Local forwarding makes a port on the remote machine available as a local port:

```
ssh -L 5432:localhost:5432 myserver
```

This forwards local port 5432 to port 5432 on `myserver`. While the tunnel is open, connecting to `localhost:5432` connects you to the PostgreSQL database running on `myserver`. No need to expose the database port to the internet.

Common uses:

```
# Access a remote database locally
ssh -L 5432:localhost:5432 myserver
psql -h localhost -U alice mydb

# Access a remote web service
ssh -L 8080:localhost:3000 myserver
# Now visit http://localhost:8080 in your browser

# Access an internal service through a bastion host
ssh -L 8443:internal-service.private:443 bastion.example.com
# Connects to internal-service.private:443 through the bastion

# Run in background (-N: don't execute a command, -f: go to background)
ssh -fNL 5432:localhost:5432 myserver
```

The `-N` flag tells SSH not to execute a remote command — useful when all you want is the tunnel. The `-f` flag sends the process to the background, freeing your terminal.

In the SSH config file

For tunnels you use regularly, define them in `~/.ssh/config` so they activate automatically:

```
Host myserver-with-db
  HostName server.example.com
  User alice
  LocalForward 5432 localhost:5432
  LocalForward 6379 localhost:6379     # Redis too
```

```
ssh myserver-with-db    # connects AND opens both tunnels automatically
```

Remote port forwarding

Remote forwarding is the reverse — it makes a local port available on the remote machine:

```
ssh -R 8080:localhost:3000 myserver
```

This makes port 3000 on your local machine available as port 8080 on `myserver`. Anyone who connects to `myserver:8080` is forwarded to your local development server. Useful for sharing a local development server with a colleague or testing webhooks against a locally running service.

Dynamic port forwarding (SOCKS proxy)

Dynamic forwarding creates a SOCKS proxy that routes all traffic through the remote machine:

```
ssh -D 1080 myserver
```

Configure your browser or system to use `localhost:1080` as a SOCKS5 proxy, and all traffic routes through `myserver`. Useful for accessing geographically restricted services or browsing securely on untrusted networks.

`tmux` Over SSH: Surviving Disconnections

One of the most painful experiences in remote work is having an SSH connection drop mid-operation — a long-running database migration, a deployment, a compilation. If the process was running in the foreground of your SSH session, it's now dead. The connection drop sent a SIGHUP to the shell, which killed everything running in it.

`tmux` solves this completely. When you run processes inside a `tmux` session on the remote server, they continue running even when your SSH connection drops. Reconnecting and reattaching to the session brings you back to exactly where you were.

The essential remote workflow

```
# Connect to the server
ssh myserver

# Start a new named tmux session
tmux new -s deploy

# Run your long-running operation
npm run db:migrate
```

```
# If your connection drops, reconnect:
ssh myserver
tmux attach -t deploy            # everything is still there

# List running sessions
tmux ls

# Detach intentionally (leave session running)
# Ctrl+A d  (or Ctrl+B d with default prefix)
```

This workflow — SSH, start or attach to a `tmux` session, do your work, detach — should become automatic for any operation that might take more than a minute or might be interrupted.

Running persistent background jobs

For jobs that need to run unattended on a remote server:

```
ssh myserver

# Create a session for the job
tmux new -s batch-job

# Start the job
python scripts/process_data.py --input data/ --output results/

# Detach (job keeps running)
# Ctrl+A d

# Disconnect from SSH
exit
```

```
# Check on it later
ssh myserver
tmux attach -t batch-job

# Or just check if it's still running without attaching
tmux ls
ssh myserver "tmux ls"
```

`nohup` as a lighter alternative

For simpler cases where you just need a command to survive a disconnection without the full `tmux` workflow:

```
nohup python scripts/long_job.py > output.log 2>&1 &
disown $!
```

`nohup` ignores the SIGHUP signal, `&` backgrounds the process, `disown` removes it from the shell's job table so it won't be affected by the shell exiting, and `> output.log 2>&1` captures all output. This is a lighter-weight approach than `tmux` for fire-and-forget jobs.

Working Efficiently on Remote Machines

Copying your configuration

The first thing many developers do on a new remote machine is feel the friction of missing aliases, their preferred `vim` configuration, their `.bashrc` functions. A few patterns for managing this:

Minimal `.bashrc` snippet — keep a gist or a file in your dotfiles repository with the aliases and functions you can't live without, and paste it into `~/.bashrc` on new servers:

```
# Minimal remote .bashrc
alias ll='ls -lah'
alias gs='git status -s'
alias ..='cd ..'

# Append to remote .bashrc from local file
ssh myserver "cat >> ~/.bashrc" < ~/.config/remote-bashrc.sh
```

rsync your dotfiles — for machines you'll use regularly, sync your configuration:

```
rsync -avz ~/.vimrc ~/.bashrc ~/.tmux.conf myserver:~/
```

Dotfiles repository — the more complete solution is a dotfiles repository with a setup script that can be run on any new machine:

```
ssh myserver "git clone https://github.com/yourhandle/dotfiles && cd dotfiles
```

Running commands without a full session

For quick operations, you don't need an interactive shell — pass the command directly:

```
# Run a single command
ssh myserver "df -h"
ssh myserver "tail -n 50 logs/app.log"
ssh myserver "systemctl status nginx"

# Run a pipeline
ssh myserver "cat logs/app.log | grep ERROR | wc -l"
```

```
# Run a local script on the remote server
ssh myserver bash < scripts/check-health.sh

# Run a command on multiple servers
for server in web1 web2 web3; do
  echo "=== $server ==="
  ssh $server "uptime"
done
```

The last pattern — looping over servers and running the same command on each — is how you do basic multi-server operations without a dedicated orchestration tool.

Executing local scripts remotely

Sometimes you want to run a local script on a remote server without copying the script first:

```
# Pipe a local script to bash on the remote server
ssh myserver bash < scripts/deploy.sh

# With environment variables
ssh myserver "VERSION=$VERSION bash" < scripts/deploy.sh

# With arguments (heredoc approach)
ssh myserver << 'EOF'
cd /var/www/app
git pull origin main
npm install --production
pm2 restart app
EOF
```

The heredoc approach (`<< 'EOF'`) is particularly clean for multi-step remote operations — it reads naturally and keeps the remote commands visually grouped.

Inspecting and Debugging Remote Systems

Once connected, all the tools from previous chapters work exactly as they do locally. But a few tools and patterns are especially relevant on remote servers.

Disk space

Disk space issues are among the most common production problems. Check it first:

```
df -h                              # disk space for all filesystems
df -h /                            # disk space for the root filesystem
du -sh /var/log/*                  # size of each item in /var/log
du -sh /* 2>/dev/null | sort -rh | head -10    # largest top-level directorie
```

Memory

```
free -h                            # memory usage (Linux)
vm_stat                            # memory usage (macOS)
cat /proc/meminfo | head -20       # detailed memory info (Linux)
```

System load and uptime

```
uptime                              # load averages for 1, 5, 15 minutes
top                                 # live process and load view
w                                   # who is logged in and what they're doing
last | head -20                     # recent login history
```

Logs

On Linux systems using systemd:

```
journalctl -u nginx                 # logs for a specific service
journalctl -u nginx -f              # follow live log
journalctl -u nginx --since "1 hour ago"
journalctl -p err                   # only error-level entries
journalctl --disk-usage             # how much space logs are using
```

Traditional log files:

```
tail -f /var/log/nginx/access.log
tail -f /var/log/nginx/error.log
tail -f /var/log/syslog | grep -i "error\|warn"
```

Network

```
ss -tlnp                            # listening TCP ports and their processes
ss -tlnp | grep :80                 # what's on port 80
netstat -tlnp                       # older alternative to ss
curl -s http://localhost/health     # test a local service
curl -v https://example.com         # verbose HTTP request for debugging
```

`ss` is the modern replacement for `netstat` on Linux. `ss -tlnp` shows all TCP listening ports (`-t`), their addresses (`-l`), numeric ports (`-n`), and the process using each one (`-p`).

A Practical Remote Workflow

Here's how a complete remote debugging session might look, using the tools from this chapter and earlier ones:

```
# Connect (with multiplexing, subsequent commands are instant)
ssh myserver

# Start or attach to a tmux session so work survives disconnection
tmux new -s debug 2>/dev/null || tmux attach -t debug

# Orient yourself
uptime                                    # how long has it been running, wha
df -h /                                   # is the disk full?
free -h                                   # is memory under pressure?

# Find the problem
journalctl -u myapp -f &                  # tail logs in background
ps aux --sort=-%cpu | head -10            # what's using the most CPU?
ss -tlnp | grep myapp                     # is the app listening on the right

# Investigate a specific issue (using tools from Chapter 2)
rg "ERROR" /var/log/myapp/ --since 1h     # recent errors
tail -n 1000 /var/log/myapp/app.log | \
  grep "ERROR" | \
  awk '{print $NF}' | \
  sort | uniq -c | sort -rn | head -10   # most common error messages
```

```
# Make a change (using tools from Chapter 4)
sudo sed -i 's/workers=2/workers=4/' /etc/myapp/config.yaml

# Restart the service
sudo systemctl restart myapp
sleep 5
sudo systemctl status myapp              # did it come up?

# Watch the logs to confirm the fix
journalctl -u myapp -f

# Detach from tmux (session keeps running)
# Ctrl+A d

# Pull any relevant logs back to local machine for deeper analysis
exit
rsync -avz myserver:/var/log/myapp/ ./remote-logs/
```

Chapter Summary

SSH is the terminal's gateway to remote machines, and the difference between a developer who knows only `ssh user@host` and one who has internalized SSH config, multiplexing, tunneling, and `tmux` is measurable in hours saved per week. Production incidents are shorter. Deployments are smoother. The friction of working remotely drops to near zero.

The key habits to build:

- Set up key-based authentication for every server you access regularly — never type passwords over SSH
- Build out your `~/.ssh/config` file — aliases and per-host settings pay for the setup time immediately

- Enable SSH multiplexing globally — it costs nothing and makes repeated SSH operations dramatically faster
- Always use `tmux` for operations on remote servers that might take more than a minute or might be interrupted
- Use `rsync` instead of `scp` for anything more than a single small file
- Use local port forwarding to access remote databases and services locally rather than exposing them to the internet
- Keep a minimal dotfiles snippet you can quickly apply to new remote machines

Exercises

1. Generate an ed25519 SSH key pair if you don't have one. Copy it to a remote server using `ssh-copy-id`. Verify that you can log in without a password.

2. Add at least three hosts to your `~/.ssh/config` with meaningful aliases, the correct user and identity file, and `ServerAliveInterval 60`. Verify that you can connect to each using just the alias.

3. Enable SSH multiplexing in your `~/.ssh/config`. Connect to a server, then in a second terminal run `ssh -O check <hostname>` to verify the master connection exists. Run five quick commands over SSH and compare the response time to connections without multiplexing.

4. Use local port forwarding to access a remote database (PostgreSQL, MySQL, Redis, or any other) from your local machine. Connect to it using a local client and verify the connection works.

5. Start a long-running command inside a `tmux` session on a remote server. Detach from the session, close your SSH connection, reconnect, and reattach to verify the command is still running.

6. Write a shell script that uses `rsync` to back up a remote directory to a local path. Include a `--dry-run` mode triggered by a flag, use `--delete` to

mirror the remote, and exclude common noise directories like `node_modules` and `.git`.

Quick Reference

SSH basics

Command | What it does |

||| | `ssh-keygen -t ed25519` | Generate an ed25519 key pair | | `ssh-copy-id user@host` | Copy public key to remote server | | `ssh-add ~/.ssh/id_ed25519` | Add key to ssh-agent | | `ssh myserver` | Connect using ~/.ssh/config alias | | `ssh myserver "command"` | Run a single command remotely | | `ssh myserver bash < script.sh` | Run local script remotely |

File transfer

Command | What it does |

||| | `scp file.txt myserver:~/` | Copy file to remote home directory | | `scp myserver:~/file.txt ./` | Copy file from remote to local | | `rsync -avz src/ myserver:~/dest/` | Sync directory to remote | | `rsync -avz --dry-run src/ myserver:~/dest/` | Preview what would sync | | `rsync -avz --delete src/ myserver:~/dest/` | Mirror (delete remote extras) | | `rsync -avz myserver:~/logs/ ./logs/` | Pull remote directory locally |

Port forwarding

Command | What it does |

||| | `ssh -L 5432:localhost:5432 myserver` | Forward local 5432 to remote 5432 | | `ssh -fNL 5432:localhost:5432 myserver` | Same, backgrounded | | `ssh -R 8080:localhost:3000 myserver` | Expose local 3000 as remote 8080 | | `ssh -D 1080 myserver` | SOCKS proxy through remote |

Multiplexing

Command | What it does |

||| | `ssh -O check myserver` | Check if master connection exists | | `ssh -O stop myserver` | Stop master connection |

Remote diagnostics

Command | What it does |

||| | `df -h` | Disk space usage | | `free -h` | Memory usage | | `uptime` | Load averages | | `ss -tlnp` | Listening ports and processes | | `journalctl -u service -f` | Follow service logs | | `w` | Who is logged in |

Conclusion — Building CLI Fluency Over Time

You've covered a lot of ground. Navigation and search. File reading and editing. Version control, data processing, automation, process management, pipes and redirection, terminal configuration. Dozens of tools, hundreds of commands, thousands of combinations.

If you've been reading linearly, you might feel a mixture of excitement and mild overwhelm — there's a lot here, and no one absorbs all of it at once. If you've been dipping in and out of chapters as specific needs arose, you've probably already put some of it to use. Either way, the question at this point isn't "did I learn everything in this book?" The question is: what do you do next?

This final chapter is about that question. How to build on what you've learned. How CLI fluency actually develops over time. What to do when you get stuck. Where to go deeper. And why the investment you've started making in these skills will pay compounding returns for the rest of your career.

How Fluency Actually Develops

Learning terminal tools is not like learning an API or a framework. With a framework, you study the documentation, understand the abstractions, build a project, and after a few weeks you're productive. The knowledge

is largely declarative — you either know how `useEffect` works or you don't.

Terminal fluency is more like learning a language. The vocabulary is the individual commands. The grammar is the pipeline model. The idioms are the patterns — `sort | uniq -c | sort -rn`, `xargs -0`, `2>&1`, `tee >(...)` — that experienced users reach for automatically. And like a language, fluency comes not from studying but from *use*. From finding yourself in situations where you need to express something and figuring out how to express it. From making mistakes and correcting them. From seeing how other people solve problems and incorporating their approaches into your own.

This means the path forward from this book is not to read more books. It's to use what you've learned, deliberately and consistently, until it becomes automatic.

The three stages of CLI fluency

Most developers pass through three recognizable stages on the way to terminal fluency:

Stage 1: Conscious lookup. You know tools exist but have to look up the syntax every time. You remember that `find` can search by modification time but not the exact flag. You know `jq` can filter arrays but have to check the docs for `select`. This stage feels slow and effortful, but it's not a failure — it's the normal starting point.

Stage 2: Conscious recall. Commands come to you without looking them up, but you still have to think about them. You compose pipelines deliberately, pausing to consider each step. You remember most flags but occasionally check `man` for the obscure ones. This stage is where most developers who use the terminal regularly spend most of their time.

Stage 3: Unconscious fluency. Commands and pipelines flow without deliberate thought. When you see a problem, a solution assembles itself — not because you're clever, but because you've seen similar problems enough times that the pattern recognition is automatic. This is the stage where the terminal starts to feel like a superpower rather than a tool.

The distance between stages is measured in *repetitions*, not in time. A developer who uses the terminal for an hour a day will develop fluency faster than one who uses it for an hour a week, regardless of how long either has been "using the terminal." Deliberate practice accelerates the process — actively trying new commands, building pipelines from scratch rather than copying them, choosing the terminal over the GUI even when the GUI is faster in the short term.

One New Tool or Trick Per Week

The most practical advice for continuing to develop after finishing this book is also the simplest: learn one new tool, flag, or technique per week, and use it until it sticks.

One per week sounds modest. Over a year it's fifty new capabilities added to your toolkit. Over five years it's two hundred and fifty. The compounding effect is real — each new tool interacts with everything you already know, creating new combinations you couldn't have built before.

The key is specificity. "Get better at the terminal" is not a useful goal. "Learn `git bisect` this week and use it to find a bug in a real project" is. Some concrete starting points, roughly ordered from foundational to advanced:

Week 1–4: The foundations - Master `fzf` — install it, learn `Ctrl+R`, `Ctrl+T`, and `Alt+C`, use nothing else for history and file search for a month - Learn `git log` thoroughly — `--oneline --graph --all`, `-S`, `-p --follow`, `--since`, `--author` - Get comfortable with `awk` field extraction

and basic filtering — these come up constantly - Build your first real `Makefile` for a project you work on

Week 5–8: Intermediate composition - Learn interactive rebase — use it to clean up commits before every pull request for a month - Master `jq` transformations — reshaping, filtering, aggregating, building JSON from shell variables - Write your first multi-function shell script with proper error handling and `trap` - Set up `direnv` and move all your project environment variables into `.envrc` files

Week 9–12: Advanced tools - Learn `tmux` — set up a persistent dev session and use it for two weeks straight - Master `xargs` — `-0`, `-I {}`, `-P` for parallelism - Learn `git bisect` and use it on a real problem - Build a complete automation script: something that fetches data, transforms it, and does something useful with the result

Beyond: specialization - If you work with infrastructure: `ssh` multiplexing, `rsync`, `ansible` basics, `kubectl` workflows - If you work with data: `csvkit`, `sqlite3` pipelines, `awk` aggregations - If you work on performance: `strace`/`dtrace`, `perf`, flame graphs from the terminal - If you work on security: `openssl` command-line usage, certificate inspection, network debugging

Building Instincts, Not Just Knowledge

There's a difference between knowing that `rg "pattern" -l | xargs sed -i '' 's/old/new/g'` is how you rename something across a codebase, and *reaching for it automatically* when you need to rename something across a codebase. The first is knowledge. The second is instinct. This book can give you the first. Only practice can give you the second.

The fastest way to build instincts is to impose constraints on yourself — to deliberately choose the terminal even when another option is available and easier.

Choose the terminal over the GUI for one week. Instead of using your IDE's search, use `rg`. Instead of clicking through git history in a GUI client, use `git log`. Instead of opening a CSV in Excel to inspect it, use `csvstat` and `head`. You'll be slower at first. That's the point — the friction forces you to practice, and practice builds the muscle memory that eliminates friction.

When you solve a problem with multiple commands, turn it into a function or alias. The act of generalizing a solution cements the understanding. If you spent ten minutes figuring out how to find the ten most-changed files in a git repository, the next ten minutes should be spent turning that into a `git-hotspots` alias so you never have to figure it out again.

When you see someone else solve a problem at the terminal, ask how. Experienced terminal users are usually happy to share their approach, and you'll often learn an idiom or a flag you'd never have found on your own. Reading other people's dotfiles on GitHub is a similarly efficient way to discover approaches you wouldn't have invented yourself.

Explain what you know to someone else. Teaching is one of the most effective ways to deepen understanding. If you can explain why `2>&1` works the way it does, or what the difference between `kill -15` and `kill -9` actually is, you understand it. If you can't explain it, you've identified a gap worth filling.

When You Get Stuck

Every terminal user gets stuck. Commands that should work don't. Pipelines that look right produce wrong output. Error messages that seem to have nothing to do with what you did. Getting unstuck quickly is a skill in itself.

The diagnostic sequence

When a command doesn't work as expected, run through this sequence:

1. Read the error message carefully. This sounds obvious, but many developers scan error messages for keywords rather than reading them. "No such file or directory" and "Permission denied" and "command not found" each mean something specific. Read the whole message.

2. Check the man page. `man <command>` contains the authoritative documentation for every Unix tool. It's dense and not always beginner-friendly, but it's complete. If you're not sure what a flag does or what the exact syntax is, the man page will tell you.

3. Try `--help`. Many modern tools (especially those covered in this book — `rg`, `fd`, `bat`, `jq`) have excellent `--help` output that's more readable than the man page and covers the most common use cases.

4. Simplify. If a pipeline of six commands isn't working, remove commands from the end until you find where the problem starts. If a `sed` expression isn't matching, test the pattern in isolation with a simple input. Divide and conquer.

5. Check your assumptions. Does the file exist? Is the variable set? Is the command the one you think it is? (`type command`, `which command`). Is the data in the format you expect? (`head file`, `cat file | jq '.'`). The most common debugging insight is "the input wasn't what I assumed."

6. Search for the specific error. A search for the exact error message — especially for common tools like `git`, `bash`, `sed`, and `awk` — will usually find a Stack Overflow answer or documentation that addresses exactly your situation.

Resources for going deeper

`man` pages. Every tool covered in this book has a man page. `man grep`, `man find`, `man bash`, `man jq`. They're not always easy reading, but they're authoritative and always available — even on a remote server with no internet connection.

`tldr` (`brew install tldr`). Community-maintained simplified man pages with practical examples. Where `man` tells you everything, `tldr` tells you the things most people actually use. It's the right first stop when you know what a tool does but can't remember the syntax:

```
tldr find
tldr xargs
tldr jq
```

explainshell.com. Paste any shell command and it will annotate each part with an explanation drawn from the man pages. Invaluable for understanding commands you've found online but don't fully understand.

shellcheck.net (or `brew install shellcheck`). A static analysis tool for shell scripts that catches common mistakes — unquoted variables, incorrect comparison operators, portability issues. Run it on any script you write before considering it done:

```
shellcheck scripts/deploy.sh
```

The Bash Manual. The GNU Bash reference manual is exhaustive and freely available online. For anything beyond basic scripting — arrays, string manipulation, process substitution, arithmetic — it's the definitive source.

The Compounding Value of CLI Fluency

It's worth being explicit about why all of this matters — not just in the abstract "become a better developer" sense, but concretely.

Speed at the right moments

Most of what developers do day-to-day doesn't benefit much from terminal fluency. Writing code, reviewing pull requests, attending meetings, thinking about architecture — the terminal isn't relevant to any of these. But there are moments — debugging a production incident, investigating an unfamiliar codebase, processing a data export, automating a deployment — where terminal fluency is the difference between resolving something in five minutes and spending an hour on it. These moments are high-stakes, often time-pressured, and disproportionately visible to the people around you.

Portability across environments

A developer who relies primarily on GUI tools is dependent on having those GUI tools available. A developer who is fluent at the terminal can be productive on any Unix system — a new laptop, a remote server, a Docker container, a CI environment — with nothing but a shell. This portability has concrete value every time you need to debug something in production, set up a new machine, or work in an environment where your usual tools aren't available.

Understanding what your tools actually do

Fluency at the terminal comes with a deeper understanding of how software and operating systems work. You understand processes, file descriptors, environment variables, signals, standard streams. You understand why "it

works on my machine" happens and how to diagnose it. You understand what your build tools, deployment scripts, and CI pipelines are actually doing — not just whether they succeed or fail. This understanding makes you better at debugging, better at designing systems, and better at communicating with the people who work on infrastructure and operations.

The confidence to automate

Perhaps the most important effect of CLI fluency is a change in how you think about repetitive tasks. Developers who aren't comfortable at the terminal tend to tolerate repetition — doing the same five-step process manually because automating it feels like a bigger project than it's worth. Developers who are fluent at the terminal automate reflexively, because they know it takes ten minutes and will save ten hours. That difference in attitude toward automation compounds over a career into a massive difference in both productivity and the quality of the systems they build.

A Note on Tools Changing

Everything in this book is accurate as of writing, but tools change. Some of the "modern alternatives" covered here — `rg`, `fd`, `bat`, `eza` — will likely be superseded by something better in the years ahead. New tools will emerge that solve problems the current generation doesn't handle well.

The core principles won't change. Stdin, stdout, stderr, pipes, redirection — these are the foundation of Unix and will remain so for the foreseeable future. The tools built on that foundation will evolve, but the mental model transfers. If you understand why `rg` is better than `grep` for development use, you'll understand why whatever replaces `rg` is better than `rg`. If you understand how `xargs` bridges the gap between pipes and argument-based commands, you'll understand any tool that solves the same problem differently.

Invest in understanding the principles. The specific tools are learnable in an afternoon once the principles are solid.

What a Fluent Terminal User Looks Like

It can be useful to have a concrete picture of what you're working toward — not as a benchmark to measure yourself against anxiously, but as a description of habits and instincts that develop naturally with practice.

A fluent terminal user doesn't spend time remembering syntax. Not because they've memorized everything, but because they've internalized the patterns well enough that the right command assembles itself when they need it, and they know immediately where to look when it doesn't.

They build pipelines the way a writer constructs sentences — not by carefully planning each word before starting, but by beginning with a clear intention and refining as they go. They run a command, see the output, add the next step, see the output, continue.

They automate without fanfare. When they find themselves doing something for the second time, they pause for ten minutes and write a function or a script. Not because they're disciplined about it, but because it feels faster and more natural than doing the same thing manually again.

They are comfortable in unfamiliar environments. On a new machine, in a Docker container, SSH'd into a production server — they orient themselves quickly because the tools are the same everywhere and the mental model travels with them.

They treat their shell configuration as a living document — adding aliases and functions when they find a pattern worth capturing, occasionally pruning things that no longer serve them, occasionally discovering that a tool they configured years ago is something they use every day without thinking about it.

And they remain curious. The terminal is deep. Even developers who have used it for twenty years occasionally discover a flag, a tool, or a technique they didn't know existed. That discovery is one of the pleasures of the environment — the sense that there is always something new to find, and that finding it will make the work a little faster, a little cleaner, or a little more satisfying.

Final Words

The terminal is not a relic. It is not a hazing ritual that experienced developers impose on newcomers to prove a point. It is not something you need to master before you're allowed to call yourself a real developer.

It is a tool — the most composable, portable, and expressive tool available to a software developer — and like any tool, its value is proportional to how well you use it.

You've made a serious start. You understand the philosophy. You know the essential commands. You can navigate, search, read, edit, version-control, process data, manage processes, compose pipelines, and automate workflows entirely from the command line. That's not a small thing.

What happens next depends on what you do with it. The developers who get the most from the terminal are not the ones who read the most about it. They're the ones who use it — consistently, deliberately, and with a willingness to slow down occasionally in order to learn something that will make them permanently faster.

Open a terminal. Find a problem. Solve it with the tools you have. When those tools aren't quite enough, find the one you're missing. Add it to your toolkit. Repeat.

The shell is patient. It will be there every time you come back to it, exactly as you left it, ready to do whatever you ask.

Resources

Essential references

`man <command>` — The authoritative documentation for every tool on your system. Always available, always accurate, always worth checking before giving up.

`tldr <command>` (`brew install tldr`) — Practical, example-focused summaries of common commands. The right first stop for "what are the most useful things I can do with this tool?"

explainshell.com — Paste any shell command to get an annotated breakdown of every component. Invaluable for understanding commands found online.

shellcheck.net / `brew install shellcheck` — Static analysis for shell scripts. Catches the mistakes that are easy to make and hard to debug.

Going deeper

"The Unix Programming Environment" by Kernighan and Pike — Written in 1984, still one of the best books ever written about the Unix philosophy and how to think at the command line. The examples are dated; the thinking is not.

"Classic Shell Scripting" by Robbins and Beebe — The most thorough treatment of portable shell scripting available. For anyone who writes scripts regularly and wants to understand the deeper mechanics.

"Data Science at the Command Line" by Jeroen Janssens — Applies Unix tools to data science workflows. Excellent for developers who work with data and want to push the terminal toolkit further than this book goes.

"The Linux Command Line" by William Shotts — A comprehensive introduction that goes deeper into Linux-specific tools and the internals of the shell. Freely available online at linuxcommand.org.

Tools to explore next

Having completed this book, these are the tools most worth exploring next, roughly in order of broad applicability:

`parallel` (GNU Parallel) — A more powerful alternative to `xargs -P` for parallelizing shell commands. Handles progress reporting, error collection, and complex job scheduling that `xargs` can't.

`pv` (pipe viewer) — Adds a progress bar to any pipeline. When processing large files, `cat large-file.log | pv | grep "ERROR" | wc -l` shows throughput and estimated completion time.

`watch` — Runs a command repeatedly and displays the output, refreshing every N seconds. Useful for monitoring live system state without writing a polling loop.

`nc` (netcat) — A Swiss Army knife for network connections. Testing whether a port is open, sending data over TCP, simple client-server communication — all from the terminal.

`ssh` config and multiplexing — If you SSH into remote servers regularly, a well-configured `~/.ssh/config` with connection multiplexing dramatically reduces latency and eliminates repeated authentication.

`rsync` — The right tool for syncing files between local and remote systems. More efficient and reliable than `scp` for anything beyond a single small file.

`awk` programs — This book covered `awk` for common tasks. `awk` is actually a complete programming language, and for developers who work heavily

with structured text data, investing in learning it properly pays significant dividends.

`vim` or `neovim` — A terminal-based editor that, once learned, keeps your hands on the keyboard and integrates naturally with all the pipeline tools in this book. The learning curve is steep; the payoff for developers who commit to it is substantial.

Communities and ongoing learning

commandlinefu.com — A community-curated collection of useful shell one-liners. Browsing it occasionally is one of the best ways to discover idioms and techniques you wouldn't have found otherwise.

GitHub dotfiles — Search GitHub for "dotfiles" to find other developers' shell configurations, aliases, and functions. Reading a well-maintained dotfiles repository is like getting a tour of someone else's terminal workflow.

r/commandline and **r/bash** — Active communities for questions, discoveries, and discussion about terminal tools and shell scripting.

Your own `TIL` (Today I Learned) file — Many developers keep a simple text file or note where they record terminal tricks and commands they discover. A year of consistent TIL entries is a surprisingly rich personal reference.

About the Author

About the Author

Vijay Mathew is a software architect and hands-on engineer with two decades of experience building programming languages, distributed services, developer platforms, search systems, and embedded software. He shapes systems at the intersection of languages, tooling, and operations - balancing design rigor with the craft of shipping reliable code.

Currently a founding engineer and software architect at Fractl, Vijay is leading the design of a declarative language and runtime for domain modeling, defining core abstractions around execution, contracts, and developer ergonomics. Across previous roles he has built high-throughput identity and authorization services, search and delivery platforms, embedded systems, and analytics tooling, learning to ship large-scale software with clarity.

His daily toolkit spans TypeScript, Clojure, Java, Scheme, C, C++, Python, and PostgreSQL, together with a strong grounding in distributed systems, performance-sensitive services, and developer tooling. Vijay pairs technical leadership, greenfield design, and remote collaboration with an intense focus on readable, testable, and maintainable code.

Open source contributions - Nex - the practical, expression-oriented programming language focused on readable control flow, contracts, and teachable engineering discipline. (https://github.com/vijaymathew/nex) - Slogan - a high-level dynamic language that compiles to efficient machine code with parallel execution. (http://schemer.in/slogan) - Reo - contributions to a programming language and environment designed for animating data analysis. (https://github.com/anvetsu/reo)

About the Author

Learn more at schemer.in and keep up with Vijay on GitHub at github.com/vijaymathew.

Bibliography

This bibliography lists all tools, projects, and written works referenced throughout the book, organized by category. Where a tool has a primary home page, repository, or documentation site, that is listed as the reference. All URLs were verified at time of writing.

Books

Kernighan, Brian W., and Rob Pike. *The Unix Programming Environment*. Prentice Hall, 1984. > The foundational text on Unix philosophy and command-line thinking. Chapter 3 in particular — on using the shell — remains one of the clearest articulations of the pipe-and-filter model ever written.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Prentice Hall, 1988. > Not a command-line book per se, but the cultural artifact closest to the original Unix design sensibility. Useful context for understanding why the tools were designed the way they were.

Robbins, Arnold, and Nelson H.F. Beebe. *Classic Shell Scripting*. O'Reilly Media, 2005. > The most thorough treatment of portable POSIX shell scripting available. Covers edge cases and portability concerns that most scripting guides skip entirely.

Robbins, Arnold. *sed & awk*, 2nd ed. O'Reilly Media, 1997. > The definitive reference for both tools. Goes significantly deeper than this book on `awk` as a programming language and `sed` as a full stream editor.

Shotts, William. *The Linux Command Line*, 2nd ed. No Starch Press, 2019. > A comprehensive introduction to Linux-specific tools and shell internals. Freely available online at linuxcommand.org.

Janssens, Jeroen. *Data Science at the Command Line*, 2nd ed. O'Reilly Media, 2021. > Applies the Unix toolkit to data science workflows — fetching, cleaning, exploring, and modeling data entirely from the terminal. Freely available online at datascienceatthecommandline.com.

Cooper, Mendel. *Advanced Bash-Scripting Guide*. The Linux Documentation Project, 2014. > An exhaustive reference for bash scripting. Freely available at tldp.org. Dense but comprehensive — the right place to look for anything not covered in the bash man page.

Raymond, Eric S. *The Art of Unix Programming*. Addison-Wesley, 2003. > A detailed exploration of Unix philosophy, design patterns, and culture. Freely available online at catb.org/esr/writings/taoup. Essential reading for understanding *why* the tools work the way they do, not just how.

Core Unix Tools

These are tools that ship with or are standard on virtually every Unix and Linux system.

bash — GNU Bourne Again Shell https://www.gnu.org/software/bash/ https://www.gnu.org/software/bash/manual/

zsh — Z Shell https://www.zsh.org/

awk — Pattern scanning and processing language. Covered in Chapter 4. POSIX specification: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html Gawk (GNU awk) manual: https://www.gnu.org/software/gawk/manual/

sed — Stream editor. Covered in Chapter 4. GNU sed manual: https://www.gnu.org/software/sed/manual/sed.html

grep — Pattern search. Covered in Chapter 2. GNU grep manual: https://www.gnu.org/software/grep/manual/

find — File search. Covered in Chapter 1. GNU findutils: https://www.gnu.org/software/findutils/

curl — Data transfer tool. Covered in Chapter 6. https://curl.se/ https://curl.se/docs/manpage.html

git — Distributed version control. Covered in Chapter 5. https://git-scm.com/ https://git-scm.com/doc

make — Build automation tool. Covered in Chapter 7. GNU make manual: https://www.gnu.org/software/make/manual/

less — Terminal pager. Covered in Chapter 3. http://www.greenwoodsoftware.com/less/

tmux — Terminal multiplexer. Covered in Chapter 8. https://github.com/tmux/tmux https://github.com/tmux/tmux/wiki

cron — Job scheduler. Covered in Chapter 7. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/cront

strace — System call tracer (Linux). Covered in Chapter 9 (Process Management). https://strace.io/

lsof — List open files. Covered in Chapter 9 (Process Management). https://github.com/lsof-org/lsof

xargs — Build and execute command lines from stdin. Covered in Chapter 9 (Pipes and Redirection). GNU findutils (includes xargs): https://www.gnu.org/software/findutils/

Modern CLI Tools

These are third-party tools that extend or replace classic Unix utilities, referenced throughout the book.

ripgrep (`rg`) — Fast recursive search. Covered in Chapter 2. Burns, Andrew. ripgrep. 2016–present. https://github.com/BurntSushi/ripgrep

fd — Fast and user-friendly alternative to `find`. Covered in Chapters 1 and 8. Peter, David. fd. 2017–present. https://github.com/sharkdp/fd

bat — Syntax-highlighted `cat` replacement. Covered in Chapters 3 and 8. Peter, David. bat. 2018–present. https://github.com/sharkdp/bat

eza — Modern replacement for `ls`. Covered in Chapters 1 and 8. https://github.com/eza-community/eza > Note: `eza` is the maintained fork of the original `exa` project by Benjamin Sago (https://github.com/ogham/exa), which is no longer actively developed.

fzf — General-purpose fuzzy finder. Covered in Chapter 8. Junegunn, Choi. fzf. 2013–present. https://github.com/junegunn/fzf

zoxide — Smarter `cd` command. Covered in Chapter 8. Bhatt, Ajeet. zoxide. 2020–present. https://github.com/ajeetdsouza/zoxide

jq — Command-line JSON processor. Covered in Chapters 2 and 6. Dolan, Stephen (original author). jq. 2012–present. Currently maintained by the jq community. https://jqlang.github.io/jq/ https://jqlang.github.io/jq/manual/

yq — YAML/JSON processor. Covered in Chapter 6. Humphrey, Mike. yq. 2017–present. https://github.com/mikefarah/yq https://mikefarah.gitbook.io/yq/

csvkit — Suite of utilities for working with CSV files. Covered in Chapter 6. Schafer, Christopher. csvkit. 2011–present. https://csvkit.readthedocs.io/

httpie — Modern HTTP client. Covered in Chapter 6. Reberski, Jakub, and contributors. HTTPie. 2012–present. https://httpie.io/ https://github.com/httpie/cli

starship — Cross-shell prompt. Covered in Chapter 8. The Starship Contributors. Starship. 2019–present. https://starship.rs/ https://github.com/starship/starship

direnv — Environment switcher for the shell. Covered in Chapter 7. Petazzoni, Jérôme (original author). direnv. 2012–present. https://direnv.net/ https://github.com/direnv/direnv

htop — Interactive process viewer. Covered in Chapter 9 (Process Management). Hisham, Muhammad (original author). htop. 2004–present. Currently maintained by the htop community. https://htop.dev/ https://github.com/htop-dev/htop

Oh My Zsh — Framework for managing zsh configuration. Covered in Chapter 8. Robrussell, Robby. Oh My Zsh. 2009–present. https://ohmyz.sh/ https://github.com/ohmyzsh/ohmyzsh

shellcheck — Static analysis tool for shell scripts. Referenced in Chapter 12. Kola, Vidar. ShellCheck. 2012–present. https://www.shellcheck.net/ https://github.com/koalaman/shellcheck

tldr — Simplified, community-driven man pages. Referenced in Chapter 12. The tldr-pages contributors. tldr-pages. 2014–present. https://tldr.sh/ https://github.com/tldr-pages/tldr

GNU Parallel — Shell tool for executing jobs in parallel. Referenced in Chapter 12. Tange, Ole. GNU Parallel. 2010–present. https://www.gnu.org/software/parallel/

pv — Monitor the progress of data through a pipe. Referenced in Chapter 12. Wood, Andrew. pv. 2002–present. http://www.ivarch.com/programs/pv.shtml

Terminal Emulators

iTerm2 — macOS terminal emulator. Covered in Chapter 8. https://iterm2.com/

Warp — Modern terminal emulator. Covered in Chapter 8. https://www.warp.dev/

Alacritty — GPU-accelerated terminal emulator. Covered in Chapter 8. https://alacritty.org/ https://github.com/alacritty/alacritty

Ghostty — Fast, native terminal emulator. Covered in Chapter 8. https://ghostty.org/

Online Resources

explainshell.com https://explainshell.com/ > Parses and annotates shell commands using man page content. Referenced in Chapter 12.

commandlinefu.com https://www.commandlinefu.com/ > Community-curated collection of shell one-liners and idioms. Referenced in Chapter 12.

linuxcommand.org https://linuxcommand.org/ > Home of William Shotts' *The Linux Command Line*, freely available in full. Referenced in Chapter 12.

datascienceatthecommandline.com https://datascienceatthecommandline.com/ > Home of Jeroen Janssens' *Data Science at the Command Line*, freely available in full. Referenced in Chapter 12.

The GNU Bash Reference Manual https://www.gnu.org/software/bash/manual/bash.htm > The authoritative reference for bash syntax, builtins, and behavior. Referenced throughout.

POSIX.1-2017 Shell & Utilities Specification https://pubs.opengroup.org/onlinepubs/9699919799/ > The formal specification for POSIX-compliant shell behavior and standard utilities. The authoritative source for portable shell scripting.

The Bash Hackers Wiki https://wiki.bash-hackers.org/ > Community documentation covering bash features and idioms in depth. Particularly good on parameter expansion, arrays, and process substitution.

Greg's Wiki (Wooledge Bash Guide) https://mywiki.wooledge.org/BashGuide > One of the most reliable community resources for bash scripting best practices. The associated BashFAQ (https://mywiki.wooledge.org/BashFAQ) is an excellent reference for common scripting questions.

Standards and Specifications

POSIX.1-2017 (IEEE Std 1003.1-2017) The Open Group. *The Open Group Base Specifications Issue 7, 2018 Edition*. 2018. https://pubs.opengroup.org/onlinepubs/9699919799/ > The formal standard that defines the behavior of shell utilities including `grep`, `find`, `awk`, `sed`, `xargs`, and the shell itself. Where tool behavior differs between macOS and Linux, POSIX is the arbiter of what is "correct."

RFC 7159 / RFC 8259 — JSON Bray, T. (ed.). *The JavaScript Object Notation (JSON) Data Interchange Format*. IETF, 2017. https://datatracker.ietf.org/doc/html/rfc8259 > The formal specification for JSON, the data format processed by `jq` throughout Chapter 6.

RFC 4180 — CSV Shafranovich, Y. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. IETF, 2005. https://datatracker.ietf.org/doc/html/rfc4180 > The closest thing to a formal CSV specification. Explains why naive field-splitting on commas fails for real-world CSV files.

Historical and Cultural References

McIlroy, M. D., E. N. Pinson, and B. A. Tague. "Unix Time-Sharing System: Foreword." *The Bell System Technical Journal* 57, no. 6 (1978): 1902–1903. > Contains McIlroy's famous statement of the Unix philosophy, including the principle of writing programs that do one thing well and work together — the philosophical foundation of Chapter 9 (Pipes and Redirection) and the book as a whole.

Ritchie, Dennis M., and Ken Thompson. "The UNIX Time-Sharing System." *Communications of the ACM* 17, no. 7 (1974): 365–375. > The original paper describing Unix. Freely available through the ACM digital library. Worth reading for the historical context of where these tools came from.

Salus, Peter H. *A Quarter Century of Unix*. Addison-Wesley, 1994. > A history of Unix from its origins at Bell Labs through the early 1990s. Provides context for why the tools are designed the way they are and how the culture of the command line developed.

Tool versions, repository locations, and maintainer information are subject to change. Readers encountering broken links or outdated information are encouraged to search for the current location of any tool using its name — active open source projects are generally easy to find.