Unix Command-Line Workspace Setup for Data Science Researchers

Ronald 'Ryy' G. Thomas

2026-05-17

One set of configuration files, two Macs and a Linux workstation: a dotfiles repository turns months of accumulated environment knowledge into a two-command bootstrap.

Introduction

Consider the situation that arises, perhaps more often than one might expect, when a new researcher joins a data science and biostatistics laboratory with three machines: a MacBook Pro for primary campus work, a Linux Mint workstation at home for long-running analyses, and a MacBook Air for travel and conferences. On the first day all three are unconfigured. By the third week, the campus MacBook has acquired a custom `.zshrc`, the Linux machine has accumulated PATH entries and aliases added in the moment and never documented, and the travel Air still presents the default macOS prompt. Six months later, reproducing a particular analysis command on the Linux machine requires either an unusually reliable memory or something closer to archaeology.

The Unix command-line workspace does not maintain itself. Left to accumulate without deliberate management, three machines will develop three divergent configurations, and the researcher who built them will be the only person with any clear picture of the differences. We have found that the solution is not, in practice, a matter of imposing more discipline on oneself; it is rather a question of establishing the right architecture from the start. That architecture is a version-controlled dotfiles repository, kept in git, that treats each machine as a reproducible deployment of a single authoritative source.

We shall describe that architecture here in three layers: the terminal emulator, the shell, and the dotfiles repository that ties both together. The presentation is addressed primarily to a researcher arriving with fresh hardware and no existing configuration, though the same procedure applies to anyone who has accumulated an ad-hoc setup and wishes to consolidate it. The three deliverables we shall present (an `install.sh` installer, a `Makefile`, and a `.gitignore` template) are intended as generic starting points; they merit adaptation to the specific tools and conventions of one's own laboratory, not copying verbatim.

Motivations

We have several reasons for approaching the problem this way. To begin with, three laptops managed by memory alone will diverge within weeks. The researcher who assembled them is, practically speaking, the only person who can reason about the differences, and that person is frequently unavailable at the moment the second machine is needed.

Beyond simple divergence, reproducing a broken analysis environment on a second machine is among the harder classes of reproducibility failure one encounters in practice. The dependencies are invisible, the state is implicit, and the only documentation is typically the original researcher's recollection. A new-machine bootstrap that demands several hours of manual configuration is, furthermore, a recurring tax on every hardware refresh, contract renewal, and collaborative handoff in a laboratory of any size.

We should also note that configuration files accumulated over years of daily use contain hard-won knowledge: project navigation shortcuts, editor settings, safety aliases, job-scheduling patterns. Unfortunately, this knowledge has no other repository. Losing a machine without version control means losing that knowledge entirely, and the cumulative cost of such losses is not trivial.

Finally, the Unix tools that make a data science workflow genuinely efficient, including fuzzy file finding, consistent vi-mode across all command-line clients, per-project directory jumping, and syntax-highlighted diffs, require deliberate configuration. The default settings are functional; the configured environment is qualitatively different in ways that compound over a working day.

Objectives

We have four concrete goals. First, we will configure a modern terminal emulator and shell on the primary machine, with settings documented and placed under version control from the first session. Second, we will create a `~/dotfiles` git repository outside any cloud-synchronisation path, containing shell configuration, editor settings, git configuration, system scripts, and a sensitive-files inventory. Third, we will write and test an `install.sh` with `--dry-run` support that bootstraps a fresh machine to the same state as the primary in under thirty minutes. Fourth, and perhaps most

importantly, we will verify the bootstrap on the second machine before the third is needed, so the repository becomes a tested deployment artefact rather than a wishlist.

What is a Unix Workspace?

A Unix command-line workspace is, at bottom, a layered system. Each layer carries a distinct responsibility, configures a distinct set of files, and tends to fail in a distinct way when misconfigured. We have found that understanding the layers is more useful than memorising any particular setting, because the layers generalise across machines, operating-system versions, and tool upgrades.

Layer	Responsibility	Configured via
Terminal emulator	Window, fonts, GPU rendering, input	`kitty.conf` or `ghostty/config`
Shell	Commands, history, aliases, scripting	`.zshrc`, `.zshenv`
Editor	Text editing, language servers	`.vimrc` or `nvim/init.lua`
Version control	Diff, history, multi-machine sync	`.gitconfig`, `.gitignore_global`
Line editor	Vi-mode in R, Python, database clients	`.inputrc`
Dotfiles repository	Deploys all of the above reproducibly	`install.sh`, `Makefile`

We may think of the dotfiles repository as the source code for the workspace itself. Just as application code can be cloned and built on a fresh server, a dotfiles repository can be cloned and installed on a fresh laptop, and the analogy holds in its details: the repository should live in git, be tested before deployment, and have a clear migration path for breaking changes. The second bootstrap, on a genuinely different machine, is the moment that surfaces assumptions invisible during the first.

Prerequisites

The primary campus machine and the travel laptop run macOS 13 (Ventura) or later on Apple Silicon. The home analysis machine runs Linux Mint 21 or later (Ubuntu 22.04 base). The shell configuration, dotfiles structure, and core installer logic are shared across all three; we will note macOS-specific and Linux-specific steps where they diverge.

On the macOS machines, we begin by installing the minimum set of tools required before any configuration can proceed:

```
# Homebrew (if not yet installed)
/bin/bash -c \
  "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Core tools
brew install git gh pipx

# Generate an SSH key for GitHub (if none exists)
ssh-keygen -t ed25519 -C "your_email@university.edu"
# Then add the public key to your GitHub account settings
```

On Linux Mint, install the equivalents via apt:

```
sudo apt update && sudo apt install -y git pipx
# gh CLI: follow https://cli.github.com/manual/installation
# then: sudo apt install gh
```

On all three machines, confirm that GitHub access is working before continuing:

```
gh auth login     # follow the browser prompt
gh auth status    # should report 'Logged in to github.com'
```

In our experience, the initial setup on the primary machine requires roughly two to three hours, depending on familiarity with the tools; each subsequent machine should take no more than twenty to thirty minutes once the repository is in place.

Layer 1: Terminal Emulator

The terminal emulator occupies the outermost layer: it draws the window, renders text through the GPU, and forwards keystrokes to the shell. Two terminals are well suited to a data science workflow across macOS and Linux, and we shall consider both briefly before settling on one for the worked examples.

Kitty (`brew install --cask kitty` on macOS; `apt install kitty` or the official installer on Linux Mint) is GPU-accelerated with a built-in graphics protocol (`kitten icat`) that renders matplotlib and R plots inline without a separate display server. Its configuration is a single plain-text file at `~/.config/kitty/kitty.conf` and is identical across operating systems, which makes it the natural choice for a mixed macOS and Linux setup.

Ghostty is available for macOS (`brew install --cask ghostty`) and Linux. It offers sensible defaults and, on macOS, tighter native system integration. Its configuration lives at `~/.config/ghostty/config`, and it requires somewhat less initial configuration than Kitty to reach a usable state.

The choice between them is, in our view, a matter of genuine preference. Kitty is the stronger cross-platform choice here, because its configuration file is identical on the MacBook Pro, the Linux Mint workstation, and the MacBook Air. The worked examples below use Kitty for this reason and because its explicit configuration makes the choices visible.

Core Kitty Settings

```
# ~/.config/kitty/kitty.conf

# Fonts
font_family      JetBrainsMono Nerd Font Mono
font_size        14.0

# Cursor
cursor_shape           block
cursor_blink_interval  0

# Clipboard: selecting text copies immediately
copy_on_select clipboard

# Splits and tabs
map cmd+d   launch --location=vsplit --cwd=current
map cmd+t   new_tab
map cmd+w   close_window
map cmd+shift+] next_tab
map cmd+shift+[ previous_tab

# macOS integration
```

```
macos_option_as_alt                 yes
macos_quit_when_last_window_closed  yes
```

Install the Nerd Font variant required for prompt icons. On macOS:

```
brew install --cask font-jetbrains-mono-nerd-font
```

On Linux Mint, download JetBrainsMono Nerd Font from nerdfonts.com, extract the `.ttf` files to `~/.local/share/fonts/`, and run `fc-cache -fv` to register them.

Among the settings listed, `copy_on_select clipboard` has the largest day-to-day effect: selecting any text with the mouse copies it to the system clipboard without requiring a separate `Cmd+C` keystroke.

Layer 2: Shell Configuration

Environment Variables (.zshenv)

`.zshenv` is sourced by every shell instance, interactive or not, which makes it the right place for environment variables that every downstream process will need. We keep it minimal for exactly that reason: a variable that appears here should be one whose absence would break non-interactive shells or background jobs.

```
# ~/.zshenv
export DOTFILES="${DOTFILES:-$HOME/dotfiles}"
export PRJ_ROOT="${PRJ_ROOT:-$HOME/projects}"
export EDITOR="nvim"
export VISUAL="nvim"
```

`DOTFILES` and `PRJ_ROOT` are the two variables that all other configuration and project scripts should reference. Setting them here, with defaults, means a fresh machine that has not yet run `install.sh` will still receive sensible fallback values rather than errors.

Shell Configuration (.zshrc)

History

```
HISTFILE="$HOME/.zsh_history"
HISTSIZE=100000
SAVEHIST=100000
setopt SHARE_HISTORY HIST_IGNORE_DUPS
setopt INC_APPEND_HISTORY HIST_VERIFY
```

One hundred thousand lines is not, in our view, excessive for research work. Model training runs, pipeline executions, and database queries generate commands worth recalling weeks later. `SHARE_HISTORY` synchronises history across all open terminal sessions in real time; `INC_APPEND_HISTORY` writes each command immediately rather than at session end, so a crashed terminal loses nothing.

Project Navigation

```
cdpath=($PRJ_ROOT $PRJ_ROOT/.. $HOME)
setopt auto_cd
```

With `cdpath` set, typing `cd myproject` from anywhere causes Zsh to search `$PRJ_ROOT/myproject` automatically. Combined with `auto_cd`, which permits typing a directory name alone without the `cd` prefix, navigating between analysis projects becomes a single word rather than a full path. The `$PRJ_ROOT` indirection is deliberate: changing the project tree location requires editing one line in `.zshenv`, not dozens of scripts.

Git Branch in the Prompt

```
autoload -Uz vcs_info
precmd() { vcs_info }
zstyle ':vcs_info:git:*' formats '%b '

PROMPT='%F{cyan}%m%f %F{green}%*%f '
PROMPT+='%F{yellow}${${PWD:A}/$HOME/~}%f '
PROMPT+='%F{red}${vcs_info_msg_0_}%f$ '
```

A prompt that shows the current git branch prevents the most common commit error we encounter in research projects: committing to the wrong branch when switching between experiments. The `vcs_info` hook is lightweight and adds no extra git processes to the shell startup path.

Modern CLI Replacements

Several tools have largely superseded the default Unix utilities for interactive use. On macOS, we install them in one step:

```
brew install eza bat fd ripgrep zoxide fzf git-delta lazygit \
             shellcheck
```

On Linux Mint, several are available via `apt` (`ripgrep`, `fd-find`, `bat`, `fzf`, `shellcheck`); `eza`, `zoxide`, `git-delta`, and `lazygit` are best installed from their GitHub releases or via `cargo install`. Wire each into Zsh with a conditional guard:

```
# eza: ls with git awareness and tree mode
command -v eza > /dev/null && {
  alias ls='eza --group-directories-first'
  alias ll='eza -l --git --group-directories-first'
  alias la='eza -la --git --group-directories-first'
  alias lt='eza --tree --level=2 --git-ignore'
}

# bat: syntax-highlighted cat; paging off for pipeline safety
command -v bat > /dev/null && \
  alias cat='bat --paging=never --style=plain'

# delta: side-by-side pager for git diff
command -v delta > /dev/null && \
  export GIT_PAGER='delta'

# zoxide: frequency-weighted directory jumping
command -v zoxide > /dev/null && \
  eval "$(zoxide init zsh --cmd j)"

# fzf: fuzzy search, with ripgrep as file source
command -v rg > /dev/null && {
  export FZF_DEFAULT_COMMAND='rg --files --hidden'
  export FZF_DEFAULT_OPTS='-m --height 50% --border --reverse'
}
```

The conditional guard (`command -v tool > /dev/null &&`) is the key portability decision. The same `.zshrc` can be deployed to a fresh machine before any tools are installed, skipping each alias gracefully until the corresponding package is present. Without this guard, a missing tool produces an error on every shell startup, which is the first indication that an assumption has not been met.

Fuzzy File Finders

Three short functions cover the most common interactive search patterns we encounter in a research compendium. Note that `open` in the `pp()` function is macOS-specific; on Linux Mint substitute `xdg-open` in its place:

```
# find any file, cd to its directory
ff() {
  local file
  file=$(rg --files "${1:-.}" 2>/dev/null \
         | fzf --select-1 --exit-0)
  [[ -n "$file" ]] && cd "$(dirname "$file")"
}

# find an R/Rmd/qmd file, open in editor
```

```
rr() {
  local f
  f=$(rg --files 2>/dev/null | rg '\.(R|Rmd|qmd)$' | fzf)
  [[ -n "$f" ]] && $EDITOR "$f"
}

# find a PDF, open it
pp() {
  local f
  f=$(rg --files 2>/dev/null | rg '\.pdf$' | fzf)
  [[ -n "$f" ]] && open "$f"
}
```

Readline Configuration (.inputrc)

Many command-line tools rely on the GNU Readline library for line editing: R, Python's REPL, `psql`, and most database clients. Without a `.inputrc`, each of these uses Emacs-mode keybindings while Zsh uses vi-mode, requiring the researcher to switch mental models with every tool change. This is, in our experience, sufficiently disorienting to warrant the single file that resolves it:

```
# ~/.inputrc
set editing-mode vi
set show-mode-in-prompt on
set vi-ins-mode-string \1\e[6 q\2
set vi-cmd-mode-string \1\e[2 q\2
set completion-ignore-case on
set show-all-if-ambiguous on
set colored-stats on
"\e[A": history-search-backward
"\e[B": history-search-forward
```

The cursor shape changes between insert mode (thin bar) and command mode (block), providing a persistent visual indicator that removes any ambiguity about the current editing state.

Git Configuration (.gitconfig)

```
[init]
    defaultBranch = main

[core]
    editor = nvim
    excludesfile = ~/.gitignore_global
    pager = delta
```

```
[pull]
    rebase = false

[alias]
    st  = status --short
    lg  = log --oneline --graph --decorate -20
    co  = checkout
    br  = branch

[diff]
    colorMoved = default

[merge]
    conflictstyle = diff3

[credential "https://github.com"]
    helper =
    helper = !/opt/homebrew/bin/gh auth git-credential
```

The `gh auth git-credential` helper delegates all GitHub authentication to the GitHub CLI. After `gh auth login` has run once on a given machine, no personal access tokens or SSH key passphrases are required for subsequent git operations on that machine. On Linux Mint, the `gh` binary lives at `/usr/bin/gh` rather than `/opt/homebrew/bin/gh`; replace the helper path in `.gitconfig` accordingly, or use `$(which gh)` to make it portable across both operating systems.

Layer 3: The Dotfiles Repository

With the configuration files defined, we turn to the question of putting them under version control so that they can be deployed reproducibly to all three machines.

Repository Structure

```
~/dotfiles/
├── README.md
├── install.sh          # deployment script
├── Makefile            # convenience targets
├── .gitignore          # blocks sensitive files
├── shell/
│   ├── zshrc
│   ├── zshenv
│   └── inputrc
├── git/
│   ├── gitconfig
│   └── gitignore_global
├── editors/
```

```
│   └── vimrc
├── config/
│   ├── kitty/
│   │   └── kitty.conf
│   └── ghostty/         # alternative terminal
├── bin/
│   └── <custom scripts>
├── launchd/             # macOS scheduled jobs
│   └── *.plist
├── packages/
│   ├── Brewfile
│   └── pipx-tools.txt
└── secrets/
    └── README.md        # inventory only; no secrets in git
```

We store files without leading dots (`zshrc`, not `.zshrc`). The installation script adds the dot prefix when creating symlinks in `$HOME`, making the mapping explicit and the repository contents visible in standard directory listings without requiring `ls -a`.

Write .gitignore First

The `.gitignore` must be in place before the first `git add`. One commit with a credential in history requires a full `git filter-repo` rewrite and rotation of every exposed secret. Unfortunately, this is not a hypothetical concern; it is a predictable outcome of skipping this step. The template is at `analysis/configs/gitignore`. Key patterns include:

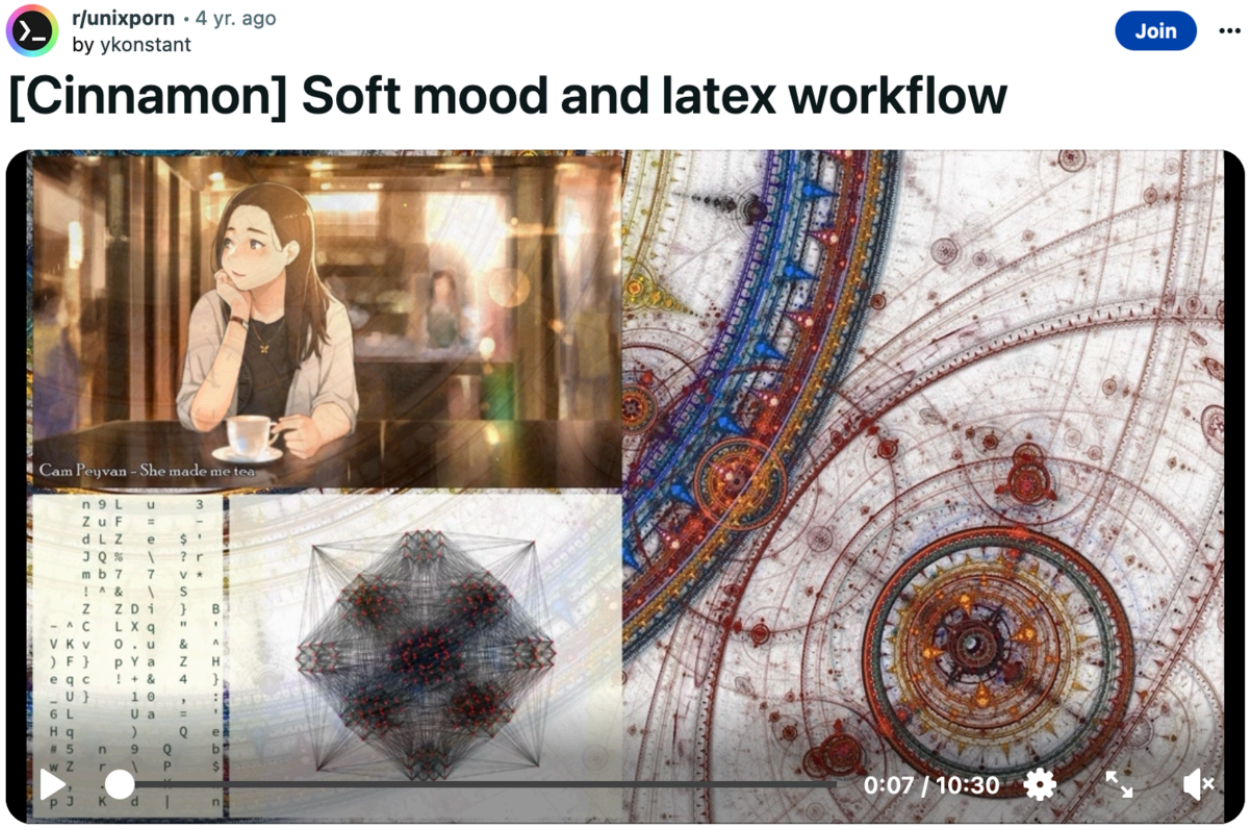
```
# Sensitive: never commit
secrets/*
!secrets/README.md
**/.env
**/.env.local
**/credentials*
**/*.pem
**/*.key
**/id_rsa*
**/id_ed25519*

# Per-machine state
*.backup.*
.DS_Store
.zsh_history
.zcompdump*

# Editor plugin directories (regenerated at install time)
config/vim/bundle/
config/vim/plugged/
```

```
config/nvim/plugged/
config/nvim/coc/
```

Initialise and Push

```
cd ~/dotfiles
git init
git add .
git status       # review; confirm no sensitive files present
git commit -m 'initial dotfiles import'
gh repo create yourhandle/dotfiles --private --source=. --push
```

The repository should be private. Even a `.gitconfig` that contains no credentials reveals preferred tooling, organisational paths, and workflow patterns that are, in our view, the researcher's own rather than public property.



install.sh

The installer creates symlinks from `~/dotfiles` into `$HOME`, handles launchd plist parameterisation on macOS, and installs packages via Homebrew and pipx where those tools are present. The full source is at `analysis/configs/install.sh`. On Linux Mint, where Homebrew is not standard, the

Homebrew section is skipped silently by the conditional guard (`command -v brew > /dev/null`); package installation there falls to `apt` and is handled separately. Several other design decisions in the script merit brief comment.

The `--dry-run` flag causes every action to be printed without execution. We recommend always running the dry-run on a new machine before the real install; it surfaces every hardcoded assumption that does not hold on the new hardware, and the cost of reviewing the output is small relative to the cost of diagnosing a failed deploy.

Before creating a symlink, existing files are moved to `filename.backup.TIMESTAMP`. This makes the installer idempotent: it can be re-run on an already-deployed machine without destroying local changes accumulated since the last install.

For `~/bin`, the installer creates one symlink per script (`~/bin/script -> ~/dotfiles/bin/script`) rather than a top-level directory link (`~/bin -> ~/dotfiles/bin`). A top-level symlink appears to work on the machine where it was created but dangles silently on any machine where the absolute path differs.

On macOS, `launchd` does not expand ~ in plist files. The installer substitutes the `__USER__` placeholder with the value of `$USER` when writing plist files to `~/Library/LaunchAgents/`, keeping the source templates machine-agnostic.

```
./install.sh --dry-run   # review all planned actions
./install.sh             # execute
```

Makefile

The Makefile provides a stable, memorable interface for common operations. The full source is at `analysis/configs/Makefile`.

Target	Action
`make install`	create symlinks; install Homebrew and pipx packages
`make update`	`git pull` then re-run `install.sh`
`make lint`	`shellcheck` on `bin/`; `zsh -n` on shell files
`make test`	dry-run + lint
`make help`	list all targets

Sensitive Files Inventory

Before the first `git add`, we recommend auditing `$HOME` for sensitive material:

```
~/.aws/           AWS credentials
~/.ssh/           SSH private keys
~/.gnupg/         GPG private keys
~/.docker/        registry authentication tokens
~/.npmrc          npm registry token
~/.config/rclone/ OAuth tokens for cloud storage
```

```
~/.env            project secrets sourced into shell
~/.password-store pass encrypted credential store
```

None of these items belong in git. Add each to `.gitignore` before the first commit. Create a `secrets/README.md` listing each item and its restore mechanism, whether that is 1Password, pass, or manual transfer via an encrypted drive. On a new machine, secrets are restored manually after `install.sh` runs; the installer prints a reminder checklist as its final output.

Bootstrapping Three Machines

With the dotfiles repository on GitHub, bootstrapping the second and third machines is reduced to two commands:

```
# On the new machine, after installing Homebrew and git:
git clone git@github.com:yourhandle/dotfiles ~/dotfiles
cd ~/dotfiles && ./install.sh
```

Sensitive files are then restored manually from secure storage; the installer prints a checklist derived from `secrets/README.md` as its final step.

We suggest the following sequence. First, on the MacBook Pro (Machine 1), complete the full setup described in this post, commit, and push to GitHub. Second, on the Linux Mint workstation (Machine 2), clone the repository, run `install.sh --dry-run`, inspect the output carefully, and only then execute the real install. Fix any failures before proceeding; the Linux machine is where cross-platform assumptions surface, and failures here almost always reflect something hardcoded for macOS that was invisible on Machine 1. Third, on the MacBook Air (Machine 3), clone and run. Because it shares the same operating system as Machine 1, it should be silent if Machine 2 succeeded.

The second bootstrap is, in our view, the most informative step in the entire process, and the fact that it runs on a different operating system makes it more informative still. It is the moment that converts the repository from a local configuration into a tested, cross-platform deployment artefact.

Verification

After each bootstrap, we verify the key invariants:

```
# 1. Symlinks resolve correctly
ls -la ~/.zshrc ~/.gitconfig ~/.config/kitty

# 2. Environment variables are set
echo "$DOTFILES"
echo "$PRJ_ROOT"
```

```
# 3. CLI tools are available
for t in eza bat fd rg fzf zoxide delta lazygit; do
  command -v "$t" > /dev/null \
    && echo "$t: ok" || echo "$t: missing"
done

# 4. Git prompt shows branches in a repo
cd /path/to/any/git/repo
# Branch name should appear in prompt in red

# 5. Readline vi-mode works in R
R --quiet
# Press Escape; cursor should change to a block
# Press k to move up in history
# Press i to return to insert mode

# 6. Dry-run and lint pass cleanly
cd ~/dotfiles && make test
```

A clean `make test` with no warnings confirms that the repository is in a deployable state and can be trusted for subsequent machines.

Daily Workflow

After setup, the dotfiles repository is maintained as a standard git project.

Command	Action
`cd ~/dotfiles && git diff`	Review uncommitted config changes
`make test`	Lint and dry-run before committing
`make install`	Re-link after adding a new config file
`make update`	Pull from GitHub and re-run install
`brew bundle dump --force`	Refresh Brewfile after new installs
`j <partial>`	Jump to a frequently visited directory
`ll` / `lt`	eza long listing / tree view
`rr`	Fuzzy-find and open an R/Rmd/qmd file
`pp`	Fuzzy-find and open a PDF
`ff`	Fuzzy-find any file, cd to its directory

`brew bundle dump --force` applies only to the macOS machines; on the Linux Mint workstation, package state is managed through `apt` and there is no direct equivalent for snapshotting installed packages into the dotfiles repository. The remaining commands in the table are cross-platform.

The single most useful habit we have found is running `make update` on each machine at the start of every work session. The operation takes roughly ten seconds and ensures that a setting committed on the primary machine the previous day is available everywhere by the time the next session begins.

Things to Watch Out For

1. **Write `.gitignore` before `git init`, not after.** If any sensitive file is committed even once, removing it requires a full `git filter-repo` rewrite and rotation of every exposed credential. Unfortunately this is not a recoverable situation in any practical sense; the secret must be treated as compromised regardless of how briefly the commit was visible. The template in `analysis/configs/gitignore` covers the standard sensitive categories and merits careful review before running `git add .`.
2. **Do not place the dotfiles repository inside Dropbox, iCloud, or Google Drive.** Git writes to `.git/index`, `.git/HEAD`, and ref files on every operation. A cloud provider racing to synchronise those writes can corrupt the index or truncate files to zero bytes. We have observed this failure mode. The repository must be local to each machine; git with a GitHub remote is the synchronisation mechanism, not the cloud provider.
3. **Run `install.sh --dry-run` on the second machine before the real run.** The first dry-run on a fresh environment surfaces every hardcoded assumption that was invisible on the primary machine. Look especially for paths that reference a directory that does not exist on the new hardware; these tend to produce silent failures rather than useful errors.
4. **Do not track editor plugin directories.** The directories where plugin managers deposit downloaded code, `~/.config/vim/bundle/` or `~/.config/nvim/plugged/` for instance, are large, platform-specific, and regenerated automatically on first launch. Track only the plugin manifest (the `vimrc Plug` lines or `nvim/lua/plugins.lua`) and add the download directories to `.gitignore` before the first commit. A reliable heuristic: if a directory is regenerated by running a tool, it is not user configuration.
5. **Per-file symlinks in `~/bin`, not a top-level directory link.** A `~/bin -> ~/dotfiles/bin` symlink appears to work on the machine where it was created but dangles silently on any machine where the absolute path differs. The installer creates one symlink per script; this is more verbose but correct across all machines.
6. **The launchd `__USER__` substitution is macOS-only; it produces no output on Linux Mint.** On the MacBook Pro and MacBook Air, the installer writes substituted plists to `~/Library/LaunchAgents/`. To modify a plist, edit the source template in `~/dotfiles/launchd/` and re-run the installer. Editing the substituted copy in `~/Library/LaunchAgents/` directly is problematic: it is overwritten on the next `install.sh` run and the change is lost. On the Linux Mint workstation, the equivalent mechanism is systemd user units; see Opportunities for the extension path.
7. **`SHARE_HISTORY` propagates across all terminal sessions in real time.** A command typed in one tab is immediately available in all others, which is useful but produces interleaved history when several sessions are active concurrently. If this behaviour proves disorienting, replacing `SHARE_HISTORY` with `INC_APPEND_HISTORY` alone restricts sharing to session boundaries rather than individual commands.

Uninstall / Rollback

To remove the dotfiles installation and restore the previous state:

```
# 1. Reverse symlinks (restore .backup.TIMESTAMP files)
for f in ~/.zshrc ~/.zshenv ~/.gitconfig \
         ~/.gitignore_global ~/.inputrc; do
  backup=$(ls "${f}.backup."* 2>/dev/null | tail -1)
  if [ -L "$f" ] && [ -n "$backup" ]; then
    rm "$f" && mv "$backup" "$f"
  fi
done

# 2. Remove per-file ~/bin symlinks
find ~/bin -maxdepth 1 -type l -delete

# 3. Unload and remove launchd agents (macOS only)
launchctl bootout gui/$UID \
  ~/Library/LaunchAgents/local.*.plist 2>/dev/null
rm -f ~/Library/LaunchAgents/local.*.plist

# 4. Remove ~/dotfiles (optional)
rm -rf ~/dotfiles
```

If no `.backup.TIMESTAMP` files exist, as is the case on a first-run against a machine with no prior dotfiles, step 1 will leave empty broken symlinks that must be removed manually. The `--dry-run` flag shows exactly which files would be backed up before the real install, which is the most reliable way to confirm the rollback path before committing to a production machine.

What Did We Learn?

Lessons Learnt

Conceptual Understanding:

We began with a model in which the workspace is a single undifferentiated collection of files. We leave with a layered model: the terminal emulator, shell, editor, line editor, and dotfiles repository are independent layers, each with independent configuration files and independent failure modes. Treating them as a coherent system from the start produces a qualitatively different working environment from one assembled ad hoc.

We also found, perhaps more clearly than expected, that cloud sync and git are not substitutes. Cloud sync provides availability across machines; git provides history, diff, rollback, and the ability to deploy a specific state to a specific machine. The dotfiles repository needs git.

On the question of the three-machine requirement, we shall emphasise it once more: the second bootstrap is more informative than the first. Setting up the primary machine is configuration; setting up the second machine is testing. Assumptions invisible on the primary surface immediately on the second, and this is the mechanism by which the requirement earns its cost.

A sensitive-files inventory written before the first commit is a one-time investment that prevents a class of irreversible errors. Credential rotation after an accidental git commit is, at a minimum,

a half-day operation, and it must be assumed thorough regardless of how briefly the commit was visible.

Technical Skills:

The `link_file` pattern, which backs up existing files with a timestamp before creating a symlink, makes `install.sh` idempotent: it can be re-run on an already-deployed machine without destroying local changes. The `--dry-run` flag is inexpensive to implement and essential for validating any installer before committing to a production machine. Ten seconds of review prevents an afternoon of recovery.

Conditional tool guards (`command -v tool > /dev/null &&`) in `.zshrc` allow the same configuration file to deploy before tools are installed and to a fully-configured machine without producing errors. The `$OSTYPE` detection pattern (`case "$OSTYPE" in darwin*) ... linux*) ... esac`) allows a single `.zshrc` to configure different `PATH` and alias settings for macOS and Linux without forking the repository into two maintenance paths.

Gotchas and Pitfalls:

The order of operations is not negotiable: `.gitignore` before `git init`, `git status` review before `git add`, dry-run before the real run. Each shortcut creates a harder problem to undo, and the hardest of them, a committed credential, cannot be undone at all.

Editor plugin directories in `~/.config/vim/` or `~/.config/nvim/` are large, platform-specific, and silently break cross-machine deploys if tracked in git. The reliable rule is: if a directory is regenerated by running a tool, it is not user configuration.

We should note one dependency that is not obvious: `delta` as a git pager requires both `export GIT_PAGER='delta'` in `.zshrc` and `[core] pager = delta` in `.gitconfig`. Either setting alone is insufficient.

Limitations

Several limitations of the current approach merit explicit statement.

The configuration has been tested on macOS 15.4 (Sequoia) with Apple Silicon and on Linux Mint 21.3 (Ubuntu 22.04 base). Other distributions, NixOS, and enterprise-managed workstations may require adjustments to `PATH`, plugin paths, and package names that we have not yet worked out.

The `install.sh` deploys to `$HOME` only; system-level configuration, including `/etc/` files and system-wide launchd daemons, is out of scope. The launchd plist handling is macOS-only; on Linux Mint the equivalent mechanism is systemd user units, which require a separate code path in the installer (noted under Opportunities).

The secrets workflow is manual throughout. The installer prints a reminder but provides no automation for restoring `~/.ssh`, `~/.aws`, or `~/.gnupg` on a new machine. This is by design, in that automating credential restoration raises its own security concerns, but it does mean that the bootstrap is not fully hands-off.

Finally, no automated CI is applied to the repository itself. A GitHub Actions workflow that runs `install.sh --dry-run` on macOS and Ubuntu runners after every push would catch cross-platform breakage before it propagates to a live machine bootstrap.

Opportunities for Improvement

Several extensions would meaningfully improve the setup described here.

First, a `bootstrap.sh` that installs Homebrew and git before cloning would reduce the first command on a fresh machine to a single `curl | bash` invocation. We note that reviewing the script content before running it is essential in this pattern, but the convenience is real.

Second, a GitHub Actions workflow that runs `install.sh --dry-run` and `make lint` on macOS and Ubuntu runners after every push would catch cross-platform breakage before it reaches a live machine. This is perhaps the highest-value addition for anyone maintaining the repository across multiple operating systems.

Third, writing a systemd user-unit equivalent of the launchd plist section, conditioned on `linux*` in `$OSTYPE`, would allow the repository to serve Linux development servers without a separate maintenance path.

Fourth, `git-crypt` merits evaluation for a small set of near-sensitive configuration files, rclone config and mbsyncrc for instance, that are not credentials but are not entirely public either. The alternative, manual scrubbing before each commit, is problematic at scale.

Fifth, completing the editor plugin manifest and adding a post-install hook that runs `:PlugInstall` in headless vim would make the editor ready on a new machine without manual intervention.

Wrapping Up

The three-laptop problem is a microcosm of the reproducibility problem that motivates most of the Unix best-practice literature. A workflow that exists only in a researcher's memory cannot be audited, handed off, or recovered from hardware failure. A workflow that exists in a version-controlled repository can be all three.

The investment described in this post is roughly a few hours on day one. It pays every time a machine is refreshed, every time a new collaborator joins the laboratory, and every time the primary laptop requires repair. In conclusion, four points are worth emphasising.

First, a dotfiles repository is, in the fullest sense, the source code for the workspace. It should live in git, be tested before deployment, and have a clear migration path for breaking changes. Second, cloud sync is not version control. A dotfiles repository inside Dropbox or iCloud has no history, no diff, no rollback, and is subject to sync-race corruption that git does not protect against. The repository must be local to each machine, with GitHub as the remote. Third, write `.gitignore` before `git init`. This is a prerequisite, not a best practice; one sensitive commit requires a full rewrite and credential rotation. Fourth, verify on the second machine before the third is needed. The second bootstrap surfaces every assumption invisible on the primary machine, and the time to find those assumptions is before the third machine is needed, not during it.

See Also

Related posts on this site:

- Migrating Off Dropbox: Beyond Dotfiles (post 64): the broader context, covering project content, append-only history files, and the three-layer framework for workflow independence from cloud sync.
- Multi-Laptop macOS Bootstrap (post 65): a deeper audit of three specific blockers encountered when executing the dotfiles migration on a real workflow.
- Setting Up pass, the Unix Password Manager (post 66): GPG key generation, pass initialisation, and migration of credentials out of cloud sync.
- Multi-Laptop Security: Hardening the Bootstrap (post 67): a security audit of the infrastructure built in this post, covering FileVault, GPG subkey architecture, and per-machine SSH keys.
- Modern CLI Replacements (post 53): the full reference for `eza`, `bat`, `fd`, `zoxide`, `delta`, and `lazygit`.
- Secrets Management for Data Scientists (post 55): the three-tier credentials scheme that extends the sensitive-files inventory introduced here.

Key resources:

- Dotfiles community guide: curated showcase of community dotfiles repositories
- Atlassian dotfiles guide: alternative bare-repository approach without `install.sh`
- shellcheck: static analysis for bash scripts; used by `make lint`
- Zsh documentation: official manual for options, completion, and scripting
- launchd.info: comprehensive reference for macOS launchd plist format
- git-filter-repo: the correct tool for removing sensitive data from git history

Reproducibility

Tested configuration:

Component	Version
Operating system	macOS 15.4 / Linux Mint 21.3
zsh	5.9
git	2.45.x
Homebrew	4.3.x
Kitty	0.36.x
eza	0.18.x
bat	0.24.x
ripgrep	14.x
fzf	0.54.x
zoxide	0.9.x
delta	0.17.x
shellcheck	0.10.x
Last verified	2026-05-31

Configuration deliverables:

- `analysis/configs/install.sh` (full installer source)
- `analysis/configs/Makefile` (build targets)
- `analysis/configs/gitignore` (`.gitignore` template)

To reproduce on a new machine:

```
git clone git@github.com:yourhandle/dotfiles ~/dotfiles
cd ~/dotfiles
make test       # dry-run + lint
make install    # deploy symlinks and packages
```

Rendered on 2026-05-17 at 16:18 PDT. Source: ~/prj/qblog/posts/68-unix-workspace-setup/unix-workspace-setup/analysis/report/index.qmd

---

Let's Connect

I would enjoy hearing from readers who are setting up a research computing environment and have questions about adapting this configuration to a different toolchain, who use a different secrets mechanism or synchronisation strategy and wish to compare notes, who have extended the launchd section to work with systemd on Linux and are willing to share the approach, or who have spotted an error or a better approach to any of the code presented here. - You just want to say hello and connect.

- **GitHub:** rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgthomas47@gmail.com

Related posts in this cluster

This post is part of the *Workflow Construct* series. Recommended reading order:

1. Post 15: A Workflow Construct for the Modern Data Scientist
2. **Post 16: Unix Command-Line Workspace Setup for Data Science** (this post)
3. Post 17: Multi-Laptop macOS Bootstrap
4. Post 18: Setting Up Git for Data Science Workflows
5. Post 19: Setting Up Neovim as a Data Science IDE
6. Post 20: Extending the R-Vim Workflow with LaTeX

7. Post 21: Modern CLI Replacements for the Shell Layer
8. Post 22: LLM-Augmented Editing for the Workflow Construct
9. Post 23: Configuring Yabai as a Tiling Window Manager
10. Post 24: A pocket terminal with ttyd and Tailscale
11. Post 25: Install Linux Mint on a MacBook Air