A Workflow Construct for the Modern Data Scientist

Ronald G. Thomas

2026-04-26

Figure 1: A closed silver laptop on a dark walnut desk, with a fanned stack of cream vellum sheets beside it, each sheet printed in fine sepia ink with a different schematic diagram and partially overlapping the next. A brass paperweight pins the topmost sheet; a fountain pen rests at the right. The composite of overlapping sheets makes more structure visible than any single sheet alone, evoking the post's thesis that a research workflow is a stack of thin, individually replaceable layers.

A workflow construct is a layered reference of the tools, configurations, and conventions that make day-to-day research work reproducible across machines and across years. Treating the construct as an explicit artifact, rather than as an accumulation of shell-history accidents, is what allows it to be reviewed, audited, and rebuilt.

Introduction

The handwritten outline reproduced in this post is a single notebook page titled 'Workflow Construct'. It summarizes a working data scientist's laptop in twelve rows: hardware, operating system,

file system, shell, editor, vim plugins, zsh functions, shell scripts, zsh aliases, applications, cloud, backup, and R packages. Each row names two to five concrete tools or conventions; together they describe a complete research compendium at the level of a single user's machine. The outline is dense, opinionated, and entirely undocumented, in the sense that reading the page does not by itself tell a newcomer how the rows fit together or why each entry was chosen.

The post has two purposes. The first is to document the existing construct as it stands, so that the reasoning behind each row is preserved alongside the row itself. The second is to propose a set of extensions appropriate to a 2026 polyglot data science practice, graded by their expected return on configuration effort. The extensions are not endorsements; each is presented with the failure mode it addresses, the cost of adopting it, and the conditions under which it is unnecessary.

The construct is presented as a reference rather than as a tutorial. Each row is a layer in a stack that other rows depend on, and the dependencies are noted where they are not obvious. Readers who want a complete tutorial for any single layer (zsh, Neovim, Docker, AWS) will find more thorough sources in the See Also section; the contribution here is the assembly, not the parts.

Motivations

The decision to write the construct down rather than to keep it in shell history was motivated by the following pain points, each of which is shared with most working data scientists:

- **Drift between machines.** A workflow that exists only as configuration files on a single laptop drifts whenever a new machine is provisioned. The new machine receives a partial transcription of the old one, missing precisely the conventions that the user has internalized and forgotten that they configured. Writing the construct down makes the drift visible.
- **Onboarding cost for collaborators.** A coauthor or student asked to reproduce an analysis on their own laptop arrives without the construct and produces results that fail in ways that are diagnostic only of the missing layer. A documented construct allows the collaborator to either match the layers that matter or to articulate the specific layer at which their setup differs.
- **Auditability of reproducible artifacts.** A reproducible research compendium (in the zzcollab sense) pins the project's R package versions, container image, and source code. It does not pin the user's editor, shell, or terminal multiplexer. Most of the time those are irrelevant. On the occasions they are not, for example an editor that strips trailing newlines or a shell that silently expands aliases inside scripts, the absence of a workflow-level record is what makes the failure hard to diagnose.
- **The construct itself as a research artifact.** Methods papers occasionally need to reference the computing environment used to produce a result. A frozen snapshot of the construct, dated and versioned alongside the analysis, is a more useful citation than a vague phrase like 'analyses were conducted in R on a workstation'.
- **Defense against tool churn.** The 2020s have seen rapid turnover in shell utilities, editor ecosystems, and cloud primitives. A construct expressed as a set of named layers rather than as a specific list of tools makes it easy to swap a layer without rewriting the entire stack. For example, `cd` can be replaced with `zoxide`, `tmux` with `zellij`, or AWS EC2 with Hetzner.

Objectives

The deliverables documented and proposed in this post are the following:

1. Reproduce the existing twelve-row construct with the rationale for each entry made explicit, using the handwritten outline as the source of truth.
2. Identify the dependencies between rows so that the construct can be rebuilt from scratch in a defensible order on a new machine.
3. Propose a set of layer-by-layer extensions appropriate to a 2026 polyglot data science practice, each graded as a strong recommendation, a conditional recommendation, or an optional addition.
4. Surface the limitations of a single-laptop construct (what it does and does not pin) and the failure modes that remain even after the construct is fully documented.

The post is not a step-by-step installation guide for any individual tool; the See Also section links to existing qblog setup posts that play that role for the most opinionated layers (zsh, Neovim, Quarto, AWS, dotfiles, zzcollab).

Figure 2: A weathered leather tool roll, unrolled on a wooden workbench, holding an organized row of distinct small hand tools each in its own stitched pocket.

What is the Workflow Construct?

A workflow construct is the set of choices that, taken together, define how a single user does research-grade computing on a single machine. The choices span a hardware layer, an operating system, a file-system layout, a shell, an editor, a set of language toolchains, a containerization layer, a cloud

or remote compute layer, and a backup strategy. Each layer is independent in principle: a user could swap macOS for Linux Mint without rewriting their R packages, or replace Vim with Neovim without changing their shell. In practice the layers interact (the editor plugins assume a shell, the shell scripts assume a file-system layout, the R packages assume a containerized dependency manifest), and the construct is the explicit record of which interactions have been resolved and which remain implicit.

A useful analogy is the operating-system kernel and its userland. The kernel is invisible to most users most of the time, but its presence is what makes any of the userland tools work. Likewise the workflow construct is invisible to a researcher who never touches a new machine, but its presence is what makes a new machine reach productivity in hours rather than weeks.

A concrete example: the construct documented below specifies that the user's projects live in `~/prj/`, that the directory is synced via Dropbox and committed to GitHub, that R packages are installed inside a Docker container with a pinned `renv.lock`, and that the container is rebuilt from a `Dockerfile` checked into the same repository as the analysis. With those four facts, a new analyst can reproduce the entire compute environment by cloning the repository, running `make r`, and starting work; without them, the same task requires a sequence of guesses about which package versions matched which paper.

Theoretical Framing

The informal definition above is sufficient for daily use, but the construct is best understood as a special case of several recognized software-engineering frameworks. Naming the generalizations clarifies which design decisions are construct-specific and which are inherited from the underlying patterns, and identifies the rigorous formal counterparts to which the present hand-rolled instantiation could be migrated.

The construct generalizes along five axes.

Reference Architecture (layered / n-tier)

A reference architecture is a documented blueprint that prescribes the layers of a system and the permitted couplings between them, without prescribing the implementing tools. This is the sense used by Bass, Clements, and Kazman in *Software Architecture in Practice* (4th edition, 2021), and it is codified in TOGAF and ISO/IEC/IEEE 42010. The classical layered architecture pattern (Buschmann, Meunier, Rohnert, Sommerlad, and Stal, *Pattern-Oriented Software Architecture* vol. 1, 1996) is the variant in which each layer depends only on the layer immediately below it. The Workflow Construct is a personal-scope reference architecture in this sense: the technical section below is the blueprint, and each row names a layer whose tools can be substituted without disturbing the rows above.

Infrastructure as Code, applied at the workstation level

Infrastructure as Code, as articulated by Morris in *Infrastructure as Code* (2nd edition, 2020), treats the configuration of computing infrastructure as a versioned, declarative artifact that can be

applied repeatedly to produce a reproducible state. The construct's dotfiles repository, `Dockerfile`, and `renv.lock` together form a hand-rolled IaC instance scoped to a single workstation. The emerging term **Development Environment as Code** (DEaC) names this pattern specifically for developer workstations rather than for production infrastructure. The most rigorous formal counterpart is **Nix Home Manager / NixOS**, which replaces the construct's `install.sh` with a verified-reproducible build closure. The trade-off is that Nix is closer to all-or-nothing adoption, whereas the present construct admits incremental migration.

Reproducible Research Compendium (project tier)

Gentleman and Temple Lang (2007, *Journal of Computational and Graphical Statistics*) introduced the term 'research compendium' for a dynamically-generated document bundled with its data, code, and dependency manifest. Marwick, Boettiger, and Mullen ('Packaging Data Analytical Work Reproducibly Using R (and Friends)', *The American Statistician*, 2018) operationalized the pattern as an R package with `data/`, `R/`, and `vignettes/` directories plus a Dockerfile. The zzcollab framework implements this at the project level. The Workflow Construct sits one tier above the compendium: the compendium pins what an analysis depends on, the construct pins what the analyst's environment depends on, and together they form a two-tier reproducibility stack.

Internal Developer Platform and Golden Path (team tier)

The Platform Engineering discipline (Spotify's Backstage and Golden Path artifacts from 2020 onward; the CNCF Platform Working Group white paper from 2022 onward) defines an **Internal Developer Platform** as an opinionated, supported, paved-road set of conventions followed by default. The construct, scoped to a team rather than a single user, is a personal IDP. A team-scope construct with a published `install.sh` and shared scripts is structurally indistinguishable from a small-organization Golden Path; the distinction is one of governance, not of architecture.

Dev Container specification and Software Bill of Materials

Two narrower industry standards formalize specific subsets of the construct. The `devcontainer.json` specification (containers.dev) defines a schema-validated, portable record of the editor, runtime, and toolchain layers, suitable for use across IDEs, GitHub Codespaces, and CI runners. The Software Bill of Materials formats SPDX (ISO/IEC 5962:2021) and CycloneDX (Ecma TC54) formalize the version-matrix row of the construct's Reproducibility section as a machine-readable inventory of component versions and provenance.

Summary positioning

Taken together, the Workflow Construct is most precisely described as a layered Reference Architecture for a single-user research workstation, instantiated as ad hoc Infrastructure as Code, with the project tier following the Reproducible Research Compendium pattern. Adopting one of the formal counterparts (Nix Home Manager for the IaC layer, a `devcontainer.json` for the editor and toolchain layers, an SPDX SBOM for the version matrix) is a refinement, not a replacement; the layered architecture itself is unchanged. The technical realization follows in the next section.

Prerequisites

The construct documented below assumes the following:

- **Operating system literacy.** Reading and editing dotfiles (`.zshrc`, `.vimrc`, `.gitconfig`) without a graphical interface.
- **Version-control fluency.** Familiarity with `git` at the level of `clone`, `commit`, `push`, `branch`, `merge`, and `rebase`. Conventional Commits and signed commits are useful but not required.
- **Shell scripting comfort.** Reading and writing short bash or zsh scripts, with an understanding of environment variables, exit codes, and standard streams.
- **Containerization awareness.** Recognition that Docker images are layered, that `renv.lock` plus a `Dockerfile` define a reproducible R environment, and that a container is run, not installed.
- **Time investment.** Approximately one to two days to reconstruct the existing construct on a fresh laptop, plus perhaps a further half-day to layer in the recommended extensions discussed below. Most of the time is package installation rather than configuration.

The construct as documented is opinionated about Vim or Neovim as the editor, zsh as the shell, and Docker as the container runtime. Readers committed to alternative editors (VS Code, RStudio exclusively, Emacs) can still extract value from the structure, but several of the rows will need substitution rather than adoption.

The Layered Architecture

This section is the technical realization of the layered Reference Architecture introduced in the Theoretical Framing above. The thirteen rows of the construct correspond to thirteen layers in the architecture; each row names a single tool or convention that occupies the layer, with the understanding that the tool is replaceable while the layer is not. The table reproduces the handwritten outline as a typed reference, with two columns added: a brief rationale for each layer's tool choice, and the set of related qblog posts that document each layer in detail. Lines have been wrapped at 78 characters for portability.

Layer	Choice	Rationale	Related posts
Hardware	MacBook + ThinkPad	Two machines covering macOS and Linux Mint surface area; ThinkPads have first-class Linux driver support.	03

Layer	Choice	Rationale	Related posts
Operating system	macOS, Linux Mint	macOS for daily driver and graphics-heavy tasks; Mint for a familiar Cinnamon desktop on Linux without the configuration cost of a tiling distribution.	03
File system	`~/prj` synced via Dropbox; tracked in Git	Single canonical project root; Dropbox handles continuous sync, Git handles intentional snapshots.	20, 21, 24
Shell	zsh	Modern parameter expansion, completion system, plugin ecosystem; default on macOS since Catalina.	01
Editor	Vim / Neovim	Modal editing, cross-machine parity, no graphical dependency, exact same configuration on the laptop and in an SSH session.	26, 30, 37
Vim plugins	`zzvim-r`, `ultisnips`, `vimtex`	R session integration, snippet expansion, LaTeX compilation; the minimum set for an R + LaTeX workflow inside Vim.	26, 30, 41
zsh functions	`ff` (fuzzy file finder)	Lightweight `fzf`-driven file selector; lighter than a plugin for a one-purpose helper.	01
Shell scripts	`zzcollab`, `tn`, `zzgit`, `lssince`	Project scaffolding, tmux session helpers, interactive git wizard with secret scanning, recently-modified file listing.	05, 49

Layer	Choice	Rationale	Related posts
zsh aliases	`ss`, `sk`, `d`, `j`, `zo`, `zp`, …	Two-letter shortcuts for high-frequency operations: open the most recent PDF of the current directory in Skim (`ss`), open Skim (`sk`), list the directory stack (`d`), jump to a frequent directory via autojump (`j`), and zzcollab project navigation to `docs/` (`zo`) and `analysis/report/` (`zp`).	01, 24
Applications	GitHub CLI (`gh`), R, RStudio, Docker	The minimum graphical or service-style tooling that does not fit inside a plugin or alias.	01, 14, 28
Cloud	AWS EC2 + dotfiles repository	EC2 for on-demand compute and Shiny hosting; a public dotfiles repository to reproduce the construct on any new machine.	22, 23, 24
Backup	GitHub, external SSD	Two-tier backup: GitHub for source-of-truth, SSD for binary artifacts and intermediate data that should not be committed.	20
R packages	`zzlongplot`, `zztab2fig`, `zztable1`, `zzpower`, `zzfishon`	Author-maintained packages for longitudinal plots, table-to-figure conversions, Table 1 generation, power calculations, and fishery-domain helpers.	14, 18, 19

The dependency order, when reconstructing the construct on a fresh machine, is roughly: operating

system, file system, shell, applications, editor, version control, cloud, then language toolchains and packages last. The order is not strict; it minimizes the number of times a step has to be repeated because a prerequisite was missing.

Figure 3: An architect's cross-section model of a building, built from stacked, laser-cut plywood layers, each layer labeled with a small tag on its edge, sitting on a drafting table beside a mechanical pencil.

Daily Workflow

Once the construct is in place, day-to-day operation reduces to a small set of repeatable patterns. The patterns matter more than the specific tools; a reader who substitutes Helix for Neovim or Hetzner for AWS can preserve the patterns unchanged.

Pattern	Command (representative)	Frequency
Start a new project	`mkdir ~/prj/NN-name && cd $_ && zzc analysis`	weekly
Resume an existing project	`j name` (autojump), then `tn` to attach a tmux session	daily
Edit a file	`vim path/to/file.R` (or `nvim`)	continuous
Run R inside the project container	`make r`	daily
Stage, scan, commit, push	`zzgit`	several times daily

Pattern	Command (representative)	Frequency
Sync dotfiles to a new machine	`git clone https://github.com/mygit/dotfiles ~/dotfiles && cd ~/dotfiles && ./install.sh`	per machine
Provision a cloud instance	`aws_create_instance.sh -p projname`	per project
Recently-modified files in a directory	`lssince 24h`	weekly

The pattern that pays the most compounded interest is the project root convention (`~/prj/NN-name/`), because it is the substrate on which all other automation can assume a path.

Recommendations for the Modern Data Scientist

The construct as documented is sufficient for a primarily R-centric biostatistics practice on macOS and Linux. The following extensions extend it for a 2026 polyglot practice without displacing any existing layer. Each is graded as **strong** (likely to repay the configuration cost within weeks), **conditional** (worth adopting only if the row's failure mode is already biting), or **optional** (interesting but not load bearing).

Modern command-line replacements (strong)

A small set of Rust-implemented utilities replace the historical Unix tools with substantially better defaults and materially faster operation on large directories. Each is a drop-in replacement for an existing alias and does not break anything if removed.

Replaces	Tool	Reason
`grep`	`ripgrep` (`rg`)	recursive by default, respects `.gitignore`, faster on large trees
`find`	`fd`	shorter syntax, parallel, respects `.gitignore`
`cat` (display)	`bat`	syntax highlighting, line numbers, page integration
`ls`	`eza`	git-aware, tree mode, more readable defaults
`cd`	`zoxide`	learns frequently-visited directories, fuzzy matching
`cd` selector	`fzf`	interactive selection across history, files, processes
`git diff` (display)	`delta`	side-by-side, syntax-highlighted, hunk-anchored

Replaces	Tool	Reason
`git` (porcelain)	`lazygit`	terminal UI for review-heavy workflows

The `j` shortcut in the construct is provided by **autojump**. Adopting `zoxide` is a substitution at the same layer, not an addition: the autojump source line is replaced with `eval "$(zoxide init zsh --cmd j)"`, preserving the `j` muscle memory while migrating the frecency database to zoxide's Rust implementation. Both tools rank directories by recent and frequent access; zoxide's incremental advantages are matching speed on large `cdpath` sets and a smaller dependency footprint (no Python, no separate `autojump.sh` sourcing).

The full installation, configuration, and migration walkthrough for these eight tools (including the autojump-to-zoxide database seeding) is documented in Modern CLI Replacements for the Shell Layer.

Secret scanning at the staging step (strong)

The single most preventable category of credential incident is a secret committed to a public repository. The `zzgit` script listed in the construct already wraps `gitleaks` for this purpose; the recommendation is to ensure that `gitleaks` is actually installed on every machine that participates in the construct, and to add a `pre-commit` hook for the residual case in which a commit is made through an IDE or a script that bypasses `zzgit`. See post 49 for the design rationale and post 51 for the relationship between standalone scripts and shell functions.

LLM-augmented editing (conditional)

Two integration points are worth distinguishing. The first is an in-editor assistant (Cursor, Zed, Neovim with `codecompanion.nvim` or `avante.nvim`, Claude Code or `aider` invoked at the shell) that proposes diffs in response to a prompt about the current buffer. The second is an out-of-editor research assistant (Claude, ChatGPT, or a local model via Ollama) that answers questions about a paper, a function reference, or a methodology without producing a diff. The first integration is the one that is conditional: it amplifies code-velocity for users who are already shell- and editor-fluent and is dilutive for users who use it as a substitute for learning the underlying tools. The second is unambiguously useful and should be adopted when the construct is otherwise stable.

A specific configuration appropriate to the existing construct is Claude Code at the shell (`claude` invoked from inside `~/prj` with the project's `CLAUDE.md` defining conventions) plus a Neovim plugin for in-buffer prompts on the rare occasion that the diff is small enough to be reviewed inline. The pattern reverses the modal of a graphical AI editor: the assistant lives at the shell where the user already does their version-control work, and the editor remains a deterministic text manipulation tool. The full installation, configuration, and `CLAUDE.md` authoring conventions for this integration are documented in LLM-Augmented Editing for the Workflow Construct, which also catalogues the in-buffer-plugin, graphical-editor, and browser-based alternatives and the trade-offs between them.

Polyglot language management (conditional)

R is the construct's primary language; Python is its frequent auxiliary. A version manager that handles both (and Node, Ruby, and Go for ecosystem dependencies) eliminates the class of incidents in which a project requires R 4.4 but the system provides R 4.5 silently.

Two competitive choices exist: `mise` (formerly `rtx`, Rust-based, fast) and `asdf` (older, plugin-based, shell-portable). Either works; the recommendation is to pick one and document the choice in the construct, because mixing both produces hard-to-diagnose path-precedence issues.

For Python specifically, `uv` (also Rust-based) replaces `pip`, `virtualenv`, `pyenv`, `pip-tools`, and `poetry` with a single tool that resolves and installs in a fraction of the time the stack of older tools requires. Adoption is a strong recommendation if Python appears in the construct at all.

Quarto as the document layer (strong)

Quarto subsumes the role previously divided between R Markdown, Jupyter, and standalone LaTeX, and produces both HTML and PDF output from a single source. The construct's qblog repository is already a Quarto site; the recommendation is to extend the convention to research papers (`paper.qmd` rendered to PDF via the `quarto pdf` engine) and to internal reports, retiring standalone `.Rmd` and `.tex` files where the cross-format benefit is real. Existing posts 17 and 29 cover the migration path.

Containerization beyond Docker Desktop (conditional)

Docker remains the construct's container runtime. Two extensions are worth knowing about:

- **`devcontainer.json`** (the VS Code / GitHub Codespaces format) lets the same container definition serve a local Docker invocation, a Codespace, and a CI runner. The cost is a single JSON file in the repository root; the benefit is that the construct's container layer becomes portable across graphical editors that the user does not personally use.
- **Apptainer (formerly Singularity)** is the right runtime for HPC clusters that disallow Docker for security reasons. It reads OCI images so a Dockerfile composes; the substitution is surgical (the Makefile's `make r` target invokes `apptainer exec` instead of `docker run`).

Both are conditional on the user actually crossing into a multi-environment setting; for a single-laptop practice neither is necessary.

Secrets management (strong if cloud is used)

The construct's cloud row depends on AWS credentials, which by default live in `~/.aws/credentials` as a plain-text file. The broader concern, which the cloud-credentials problem specializes, is the inventory of secrets the laptop accumulates over time (API keys, tokens, SSH keys), most of which end up exported in `.zshrc` or pasted into `.env` files. Three recommendations together address the full inventory:

- **AWS IAM Identity Center (formerly SSO)** with `aws sso login` produces short-lived credentials that expire automatically. This is the recommended primary path for AWS credentials specifically; long-lived `~/.aws/credentials` access keys should be rotated and removed once SSO is in place.
- **`pass`** (the Unix password store) is the recommended general-purpose secret store for non-AWS API keys (Anthropic, OpenAI, Zotero, GitHub tokens). It is open-source, GPG-keyed, git-synchronisable, and small enough (around 700 lines of bash) to audit. Secrets are injected at process-start time via a `with-secret` shell helper rather than exported into the environment, so they are visible only to the wrapped command.
- **`gitleaks` at staging time**, via the `zzgit` secret scanner, catches the residual cases in which a secret reaches the working tree despite the upstream prevention. Tier 1 and Tier 2 reduce the volume of cases that reach the safety net; Tier 3 catches the remainder.

Open-source alternatives to `pass` include **`gopass`** (a multi-store-aware Go reimplementation, useful for users managing multiple stores) and **`bitwarden-cli`** against a self-hosted **`vaultwarden`** server (a 1Password-like UX without the proprietary dependency). The proprietary **`1Password CLI`** (`op`) is structurally similar to the `pass`-plus-`with-secret` pattern and may be appropriate for users already paying for 1Password; the construct's default recommendation is open-source given the trust requirement on the secrets layer (a single subverted code path could compromise every system the operator authenticates to).

The full installation, configuration, and migration walkthrough for `pass`, AWS IAM Identity Center, and the `with-secret` injection pattern is documented in AWS and pass: A Three-Tier Secrets Scheme.

Secure remote access (conditional)

`ssh` over the public internet is adequate for an EC2 instance with a static IP and a known SSH port. Two extensions address the case in which the construct spans multiple machines that should reach each other regardless of network topology:

- **Tailscale** (a WireGuard-based mesh VPN) gives every machine a stable hostname reachable from every other machine without port-forwarding. The construct's MacBook, ThinkPad, and EC2 instance become reachable as `mac`, `thinkpad`, and `ec2-prod` respectively, regardless of which network any of them is on.
- **mosh** replaces `ssh` over high-latency or intermittent links, surviving roaming and brief disconnections that would drop a `ssh` session.

Both are conditional on the user actually working remotely or on multiple machines. For a single-laptop practice neither is required.

Knowledge management (optional)

A research workflow accumulates references, notes, and methodological observations that do not belong in a code repository. Three layers are worth naming:

- **Zotero** plus the **Zotero MCP server** for reference management with programmatic access from an LLM context.

- **Obsidian** or **Logseq** for plain-text notes with bidirectional links and a graph view, both of which sync over Dropbox or Git alongside the rest of the construct.
- **Pandoc** as the universal document converter, which makes the notes portable to LaTeX, Quarto, or HTML without re-authoring.

The optional grade reflects that knowledge management is personal: a researcher with an established system should not disrupt it for the sake of construct consistency. In the qblog curriculum (`WORKFLOW_REFACTOR_PLAN.md` at the qblog root), knowledge management is positioned as a separate cross-cutting tier alongside the workstation, project, and team-cloud tiers, paired with the LLM-augmented-editing recommendation above as the corresponding research-side counterpart.

Modern data tooling (conditional)

For data sets that exceed comfortable in-memory size in R or Python, three additions are worth considering:

- **DuckDB** as an embedded analytical database, queried from R via `duckdb` or from Python via `duckdb` directly. Replaces most ad-hoc uses of `data.table` or `polars` for query-shaped workloads.
- **Apache Arrow / Parquet** as the on-disk format for intermediate data, with `arrow` in R and `pyarrow` in Python exchanging columnar data without copy.
- **Polars** as the Rust-implemented DataFrame library in Python (and increasingly in R via `polars`), competitive with `data.table` for in-memory work.

The conditional grade reflects that for clinical-trial-scale data sets (hundreds of subjects, tens of variables) none of these are necessary; for cohort or registry data they begin to matter.

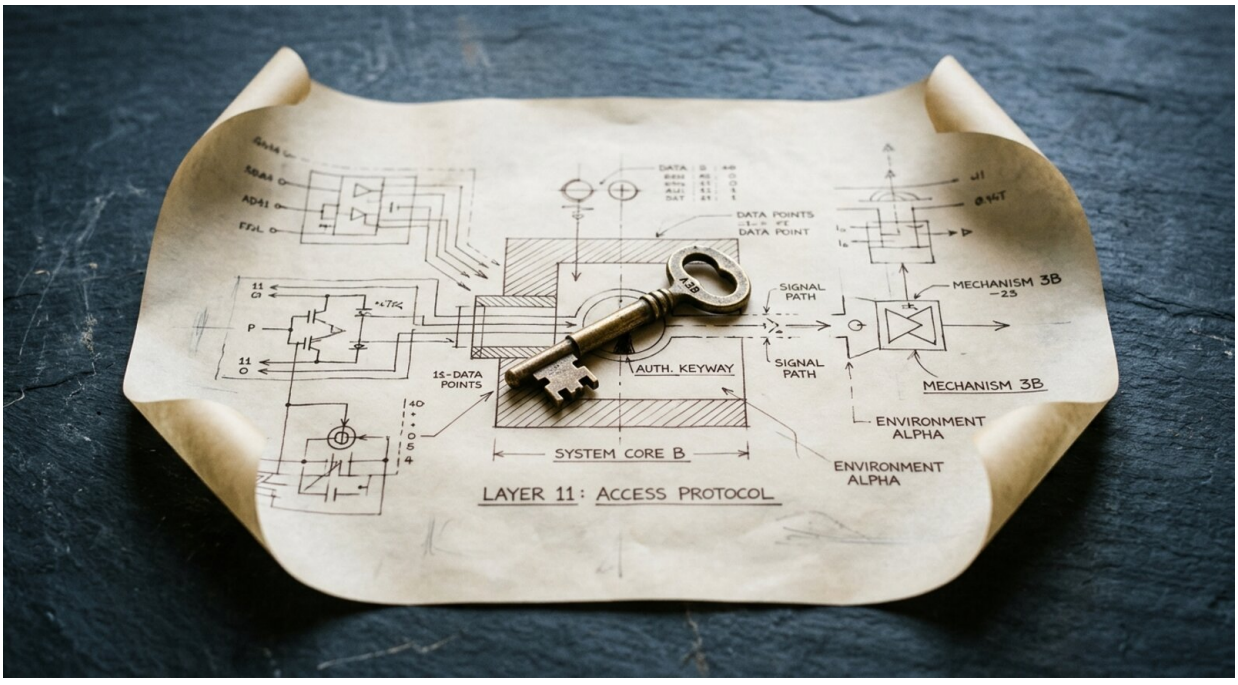

Figure 4: A single brass key resting on an open, hand-drawn technical diagram labeled with layer and access-protocol annotations, on a dark slate surface.

Things to Watch Out For

Five gotchas have surfaced in the maintenance of this construct over the past several years. Each is small in isolation; in aggregation they are the leading causes of construct drift.

- **Dropbox is not a version-control system.** The file-system row pairs Dropbox sync with Git tracking precisely because Dropbox provides continuous sync but no notion of an intentional snapshot. Relying on Dropbox alone leaves the user with no rollback target between an accidental deletion and the most recent typing event.
- **`renv.lock` does not pin the system R version.** A project that restored cleanly on R 4.4 may fail to compile a package on R 4.6 because the package's source has incompatibilities with a newer R. The construct's container layer addresses this by pinning R itself in the Dockerfile; outside of Docker, the language manager (`mise` or `asdf`) is the equivalent control.
- **zsh aliases do not expand inside scripts by default.** A script that calls `ss` will fail with `command not found` even if `ss` is aliased in `.zshrc`. The fix is to use the underlying command in scripts and reserve aliases for interactive shells, or to invoke the script with `zsh -i` to load the interactive configuration (with the corresponding overhead).
- **GitHub Actions runs Linux by default; macOS and Windows runners are billed at higher rates.** A construct that develops on macOS and tests on Linux benefits from this alignment; one that depends on macOS-specific tooling (a Homebrew cask, a Quartz graphics device) needs an explicit `runs-on: macos-latest` and a budget for the additional minute-rate.
- **AWS EC2 instances accumulate Elastic IP charges when stopped.** A construct that leaves an EC2 instance in the stopped state to resume work the next morning continues to be billed for the IP, which is approximately the same as the instance's hourly rate when running. The teardown discipline (release the IP and stop the instance, or release the instance and the IP both) is part of the construct, not an afterthought.

Verification

The construct can be partially verified with the following set of one-line commands, run on a freshly provisioned machine after the dotfiles repository is cloned and the install script has run. Each command should succeed (return zero, non-empty output); failures indicate a layer that did not install cleanly.

```
# Shell layer
zsh --version
echo $SHELL                          # /bin/zsh or /usr/local/bin/zsh

# Editor layer
nvim --version | head -1
vim --version | head -1

# Version control layer
git --version
```

```
gh --version

# Container layer
docker --version
docker run --rm hello-world

# Language layer
R --version
Rscript -e 'sessionInfo()' | head -3

# Cloud layer
aws --version
aws sts get-caller-identity            # confirms credentials are loaded

# zzcollab + custom scripts
which zzcollab zzgit lssince tn

# R packages (inside the project container)
make r                                  # then in R: library(zzlongplot)
```

A green pass on all twelve commands indicates the construct is materialized. A failure on any single command localises the problem to the corresponding layer.

Uninstall / Rollback

The construct is additive: each layer is composed of files under `~/.config/`, `~/.zshrc`, or analogous locations, and can be removed without affecting the other layers. A clean rollback of the entire construct (for instance, when handing the laptop back at the end of a contract) follows this order:

```
# 1. Stop and remove all running containers and images
docker ps -aq | xargs -r docker rm -f
docker images -aq | xargs -r docker rmi -f
docker system prune -af --volumes

# 2. Remove project root (assuming nothing unsynced remains)
ls ~/prj                                # confirm contents first
rm -rf ~/prj

# 3. Remove dotfiles
rm -rf ~/dotfiles
rm -f ~/.zshrc ~/.vimrc ~/.gitconfig ~/.aws/credentials

# 4. Remove plugin manager state
rm -rf ~/.oh-my-zsh ~/.cache/zsh ~/.zinit
rm -rf ~/.local/share/nvim ~/.cache/nvim
```

```
# 5. Optionally uninstall the system-level applications
brew uninstall --cask docker rstudio
brew uninstall awscli gh git neovim ripgrep fd bat eza zoxide fzf
```

The order matters: containers and project data first (because they hold work), dotfiles next (because they hold configuration), plugin caches third (because they hold derived state), and system-level applications last. The reverse order would leave orphaned containers or dangling configuration files.

Lessons Learned

Maintaining the construct over several years has produced lessons grouped into three buckets: conceptual, technical, and gotcha-shaped.

Conceptual

- **The construct is a stack, not a list.** The temptation when documenting a setup is to flatten it into a list of tools. Treating it as a layered stack (hardware, OS, file system, shell, editor, language, container, cloud, backup) makes the dependencies explicit and the substitutions surgical. Swapping `tmux` for `zellij` does not require rewriting the editor configuration, because the substitution lives in a single layer.
- **Layers should be replaceable individually.** A construct in which the editor's plugins assume a specific shell, or in which the shell's aliases assume a specific terminal multiplexer, is a construct that resists incremental modernization. The discipline is to push such couplings into optional helper functions (`tn` works whether or not the user is in tmux) rather than into the layer itself.
- **Documentation lives next to the code, not in a wiki.** The outline reproduced here is precisely the kind of artifact that used to live on a corporate wiki and was lost when the wiki was decommissioned. Committing the construct to a public Git repository (the dotfiles repo, plus the qblog post) ensures that it is recoverable from any machine with internet access.
- **The construct is more durable than any individual tool.** Editors, shells, and cloud primitives turn over on a roughly five-year cycle; the convention `~/prj/NN-name/` has been stable for longer. The construct's stability comes from naming the convention rather than the tool.

Technical

- **Dotfiles repositories should be installable, not just readable.** A dotfiles repository that requires a manual symlink for every file is one that nobody (including its author) will fully install. An `install.sh` script that uses `stow`, `chezmoi`, or a hand-rolled symlink loop is the difference between an inert dotfiles repository and a reproducible construct. See post 24.

- **Shell scripts beat zsh functions for cross-shell portability.** A script written as `#!/usr/bin/env bash` works identically when invoked from zsh, bash, or `cron`. A zsh function works only inside an interactive zsh shell. The construct's shell-scripts row reflects this principle: tools intended to be invoked from `cron`, from CI, or from a collaborator's terminal are scripts. Tools intended to wrap the user's interactive flow (where zsh-only constructs are worth the conciseness) are functions.
- **Container images should be built from `Dockerfile`, not pulled from a personal registry.** A registry image is opaque; a Dockerfile is auditable. The construct's R package work rebuilds from `Dockerfile` plus `renv.lock` on every analyst's machine, which means a divergence between two analysts' results is debuggable rather than mysterious.
- **Backup is two-tier or it is not backup.** GitHub holds the source of truth and the version history. An external SSD holds the binary artifacts (PDF outputs, compiled containers, intermediate data) that should not be committed. A single-tier backup that puts everything in one place is a single point of failure for both privacy and recovery.

Gotcha-shaped

- **macOS aliases for `find` and `sed` are BSD-flavoured and silently differ from GNU.** A script that uses `sed -i ''` on macOS will fail on Linux Mint with the same `sed -i` syntax, because GNU `sed` reads the empty string as a filename. The construct addresses this by aliasing GNU coreutils (`gsed`, `gfind`, `ggrep`) on macOS, or by writing scripts in a language-portable subset.
- **Dropbox file watchers conflict with `vim`'s default swap-file behavior on shared paths.** Symptom: `E325: ATTENTION` on every file open. Fix: configure `vim` to write swap files to `~/.vim/swap` outside the synced directory.
- **`docker` on Apple Silicon defaults to ARM-native images.** An R package that is unavailable as an ARM build will install from source and may fail in ways that are not reproduced on a collaborator's x86_64 Linux machine. The construct's Dockerfile pins `--platform=linux/amd64` to keep the build cross-machine consistent at the cost of a small runtime overhead under emulation.
- **`renv::restore()` in a container reads the lockfile and installs to the host's R library if the container's library path is not exported correctly.** The construct's Dockerfile sets `RENV_PATHS_CACHE=/home/analyst/.cache/R/renv` to confine the cache to the container; without this line the cache is shared with the host, which produces the failure mode in which the analysis succeeds locally but fails in CI even though the lockfile is identical.

Limitations

The construct as documented above has the following honest limitations:

- **It is opinionated about the editor.** A reader who uses VS Code, RStudio exclusively, or a graphical IDE will need to substitute several rows. The construct can be adapted to such users, but the effort is non-trivial.

- **It is single-user.** The construct describes one person's laptop. Team-scale conventions (shared package mirrors, group CI runners, shared secret stores) are out of scope; the zzcollab framework provides those at the project level.
- **The cloud layer is AWS-specific.** A construct on a university or hospital network may need to substitute the institution's HPC, a private OpenStack cloud, or a different commercial provider. The patterns transfer; the specific scripts in the construct's shell-scripts row do not.
- **Backup is local-plus-GitHub.** The construct does not prescribe an off-site cold backup (Glacier, Backblaze B2, encrypted external drive at a relative's house). For research-critical data the user's institution's backup policy is the authoritative layer; the construct supplements it rather than replacing it.
- **The R packages row is author-maintained.** A reader who does not author R packages in the `zz*` namespace will substitute different package choices in the same row. The pattern (a small set of personal packages tracked alongside the construct) generalizes; the names do not.
- **The construct does not address security hardening.** Full-disk encryption, firmware passwords, screen-lock policies, and secure-boot configuration are out of scope. The recommendation is to follow the operating system vendor's hardening guidance separately and to treat the construct as layered on top of a hardened machine.

Opportunities for Improvement

Several extensions are plausible and would be appropriate for subsequent revisions of the construct:

1. **A construct-as-code repository.** A single repository (e.g., `mygit/workflow-construct`) that contains the dotfiles, the install script, the layer-by-layer documentation, and a CI job that boots a fresh container, runs the install script, and exercises the verification commands. The CI job would prove that the construct is in fact reproducible rather than aspirationally so.
2. **A construct profile for collaborators.** A simplified subset of the construct (no zzcollab, no zz-prefix R packages, no personal aliases) that a coauthor can install on their machine to reproduce the project-specific layers without inheriting personal preferences.
3. **A `mise.toml` at the project root.** A polyglot version manifest that pins R, Python, and Node versions per project, replacing the current implicit reliance on the system-installed R. See the polyglot recommendation above.
4. **A construct snapshot in every paper's reproducibility appendix.** A frozen, dated copy of the construct (with versions for every layer) committed alongside the paper's analysis repository, so that the construct is recoverable from the paper's archive even if the dotfiles repository later moves or is deleted.
5. **An LLM-context file (`CLAUDE.md` or equivalent).** A human-and-machine-readable file that documents the construct's conventions for an in-shell coding assistant. The file already exists for the qblog repository; the recommendation is to template it for new projects so that the assistant has the construct in its context from the first invocation.

6. **A health-check script.** A `~/bin/construct-check` command that runs the verification block above on demand, reports a green or red status per layer, and exits non-zero if any layer is broken. Useful as a periodic check on long-lived machines.

Wrapping Up

A workflow construct is not the same as a list of tools. It is the explicit record of how a working data scientist's laptop is assembled, layer by layer, from hardware up to language packages. The handwritten outline reproduced and annotated above documents one such construct as it stood at the start of 2026, with twelve named rows and a concrete tool or convention in each. The annotations make explicit the reasoning that lived implicitly in the author's habits.

The recommendations section proposes a set of evidence-graded extensions that bring the construct forward into a 2026 polyglot practice without displacing any existing layer. The extensions are not prescriptions; each is presented with the specific failure mode it addresses, the conditions under which adoption is worthwhile, and the cost of adoption. The intent is that a reader can read the construct, identify the failure modes that match their own current pain, and adopt the corresponding extensions without inheriting the parts that do not fit their practice.

The decision to write the construct down was the highest-leverage single act in its maintenance. Subsequent decisions (replacing `grep` with `ripgrep`, adding `zzgit`, layering in Tailscale) were small in isolation; the documentation is what makes them discoverable, reviewable, and reversible.

In conclusion, four points merit emphasis. First, a workflow construct is layered (hardware, OS, file system, shell, editor, language, container, cloud, backup): each layer should be documented, its dependencies pinned, and it should be treated as individually replaceable. Second, documentation belongs next to the code; a construct preserved only in the author's habits is not a construct. Third, extensions should be graded by failure mode: a tool is warranted because of the failure mode it closes, not because it is new. Fourth, the convention outlives any individual tool: `~/prj/NN-name/` has outlasted three editors and two cloud providers, and naming the convention rather than the tool is what produces that durability.

See Also

- Setup posts in this repository that document individual layers in detail:
  - `01-configtermzsh`: zsh configuration and the alias conventions referenced above.
  - `03-installmintonmacbook`: Linux Mint on the ThinkPad row.
  - `20-researchbackupsystem` and `21-researchmanagement`: the file-system row and its backup discipline.
  - `22-serversetupawscli` and `23-serversetupawsconsole`: the AWS EC2 cloud row, both scripted and console-based.
  - `24-setupdotfilesongithub`: the dotfiles installer.
  - `25-setupgit`: the version-control layer.
  - `26-setupneovim`, `30-setupRvimtex`, `37-simplevimplugin`, `41-ultisnipspythonpost`: the editor and plugin rows.

  - `28-setupormodifytorrtoolsanalysisrepo`, `32-sharermdcodeviadocker`, `33-shareshinycodeviadocker`: the container layer.
  - `49-zzgit`: the interactive git wizard listed in the shell-scripts row.
  - `50-textbookdevelopment`, `51-scripts-vs-functions`: related discussions of layered authoring practice.
  - `53-modern-cli-replacements`: the worked walkthrough for the modern command-line replacements recommendation.
  - `55-secrets-management`: the worked walkthrough for the secrets-management recommendation, leading with `pass` as the open-source primary.
  - `60-llm-augmented-editing`: the worked walkthrough for the LLM-augmented-editing recommendation, leading with Claude Code at the shell and the per-project `CLAUDE.md` convention.

- External references for the recommended extensions:
  - Modern CLI replacements: `ripgrep` (https://github.com/BurntSushi/ripgrep), `fd` (https://github.com/sharkdp/fd), `bat` (https://github.com/sharkdp/bat), `eza` (https://github.com/eza-community/eza), `zoxide` (https://github.com/ajeetdsouza/zoxide), `delta` (https://github.com/dandavison/delta), `lazygit` (https://github.com/jesseduffield/lazygit).
  - Polyglot version managers: `mise` (https://mise.jdx.dev/), `uv` (https://docs.astral.sh/uv/).
  - Quarto: https://quarto.org/.
  - Containers: `devcontainer.json` spec (https://containers.dev/), Apptainer (https://apptainer.org/).
  - Secrets: AWS IAM Identity Center (https://docs.aws.amazon.com/singlesignon/), `1Password CLI` (https://developer.1password.com/docs/cli/).
  - Remote access: Tailscale (https://tailscale.com/), `mosh` (https://mosh.org/).
  - Data tooling: DuckDB (https://duckdb.org/), Apache Arrow (https://arrow.apache.org/), Polars (https://pola.rs/).

- Canonical references for the Theoretical Framing:
  - Bass, L., Clements, P., and Kazman, R. (2021). *Software Architecture in Practice*, 4th edition. Addison-Wesley. (Reference architecture; layered pattern.)
  - Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley. (The 'Layers' architectural pattern.)
  - ISO/IEC/IEEE 42010:2022, 'Software, systems and enterprise: Architecture description.'
  - Morris, K. (2020). *Infrastructure as Code: Dynamic Systems for the Cloud Age*, 2nd edition. O'Reilly. (Definitive IaC treatment; the workstation analogue is direct.)
  - Gentleman, R., and Temple Lang, D. (2007). 'Statistical Analyses and Reproducible Research.' *Journal of Computational and Graphical Statistics* 16(1): 1 to 23. (Original 'research compendium' formulation.)
  - Marwick, B., Boettiger, C., and Mullen, L. (2018). 'Packaging Data Analytical Work Reproducibly Using R (and Friends).' *The American Statistician* 72(1): 80 to 88. (Operational compendium pattern; basis for zzcollab.)
  - CNCF Platform Working Group white paper, 'Platforms' (https://tag-app-delivery.cncf.io/whitepapers/platforms/). (Internal Developer Platform definition.)

– Spotify Engineering, 'How We Use Backstage and the Golden Path to Pave the Road for Engineers' (https://engineering.atspotify.com/).
– SPDX (ISO/IEC 5962:2021): https://spdx.dev/. CycloneDX (Ecma TC54): https://cyclonedx.org/.
– Nix and Home Manager: https://nixos.org/; Home Manager (https://nix-community.github.io/home-manager/). (Verified-reproducible counterpart to ad hoc IaC.)

Reproducibility

The construct was developed and verified on the following software stack:

Component	Version	Notes
MacBook OS	macOS 15 (Sequoia)	primary daily driver
ThinkPad OS	Linux Mint 22 (Wilma)	Cinnamon edition
Shell	zsh 5.9	minimum 5.0
Editor	Neovim 0.11, Vim 9.1	both supported
Version control	git 2.43, gh 2.62	
Container runtime	Docker Desktop 4.36 / Docker Engine 27	
R	4.5.2	inside `rocker/tidyverse:4.5.2`
AWS CLI	2.18	with `aws sso` configured
Quarto	1.6.43	for the qblog site itself
zzcollab	1.0.x	profile: `ubuntu_x11_analysis`
Author R packages	`zzlongplot`, `zztab2fig`, `zztable1`, `zzpower`, `zzfishon`	tagged releases
Recommended extensions	`ripgrep` 14, `fd` 9, `bat` 0.24, `eza` 0.20, `zoxide` 0.9, `delta` 0.17, `lazygit` 0.44, `uv` 0.5, `mise` 2026.04, `tailscale` 1.78, `1password-cli` 2.30	optional layers

Date of last verification: 2026-04-26.

Feedback

Corrections, suggestions, and questions are welcome. Please open an issue or pull request on the GitHub repository or send an email to user@example.com. Substitutions for any single layer (for example, a Helix-based editor row, an Emacs-based editor row, or a non-AWS cloud row) are particularly welcome and will be incorporated into a subsequent revision of this post.

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** (this post)
2. Post 16: Unix Command-Line Workspace Setup for Data Science
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 21: Modern CLI Replacements for the Shell Layer
7. Post 25: Install Linux Mint on a MacBook Air