Refactoring a Personal Toolbox: Scripts versus Shell Functions

Ronald ‘Ryy’ G. Thomas

2026-04-25

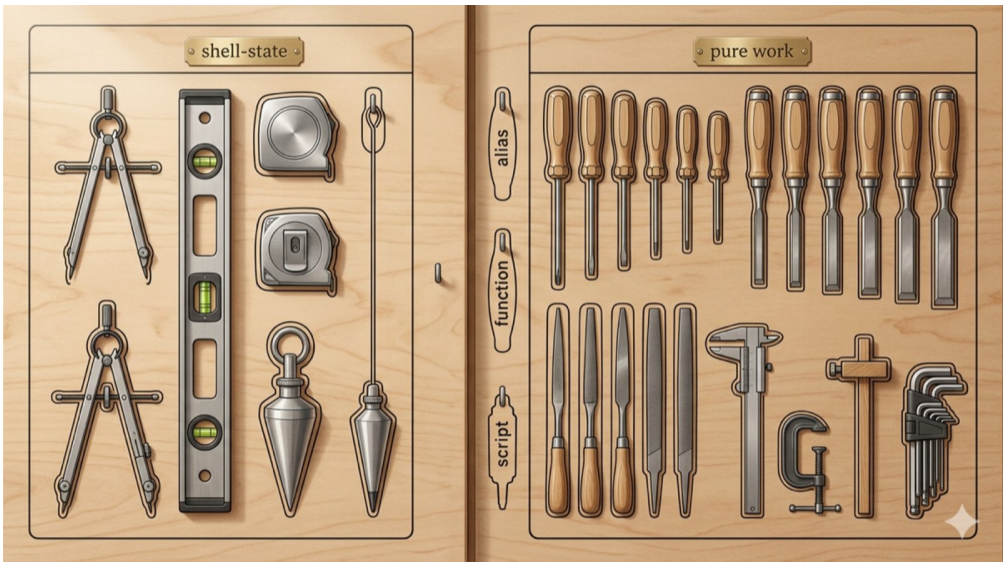

Figure 1: A workshop pegboard, organised so that each tool sits in its own labelled outline.

A personal toolbox earns its keep when each helper sits in the place that fits its job, with no doubt about which drawer to open.

1 Introduction

Most quantitative researchers maintain a personal toolbox of small shell helpers. It generally takes two forms: a `~/bin` directory populated with executable scripts, and a `.zshrc` (or `.bashrc`) file containing a layer of custom aliases and functions sourced into every interactive session. The contents accumulate gradually over years of incremental work, each entry typically introduced in response to a specific irritation: a one-liner written to bypass an awkward pipeline, a research-notes capture script drafted during a manuscript revision, a `git` workflow function codified after a co-author inadvertently committed a secret.

After several years of accretion, the toolbox remains functional but the boundary between 'script in `~/bin`' and 'function in `.zshrc`' has eroded. Some scripts in `~/bin` are aliases in disguise (a single `cd` followed by the launch of an editor). Some functions in `.zshrc` span hundreds of lines of program logic that have no need for the calling shell's state. The toolbox remains serviceable, but it has become difficult to reason about, difficult to audit for security, and noticeably slow to load on shell startup.

We document a principled refactor of such a toolbox. The worked example is drawn from a biostatistician's workflow (reproducible R analyses, Docker-based research compendia, frequent `git` commits, occasional HPC submission), but the underlying rule applies to any Unix-flavoured personal workspace.

A companion plan that operationalises the rule, with a concrete phase sequence and effort estimates, is referenced under See Also.

1.1 Motivations

- To establish a single rule that decides, for any helper, whether it belongs in `~/bin` or in shell config.
- To remove from `.zshrc` the logic that has no business mutating the current shell, lowering startup time and improving auditability.
- To bring versioned scripts under a linter (`shellcheck`) so that legacy quoting bugs surface before they bite.
- To eliminate microscripts that exist purely because the author did not realise the shell already has a primitive for them (the alias).
- To make the toolbox legible to future-self and to collaborators who inherit it during onboarding or Sabbatical handoff.

1.2 Objectives

1. State a guiding principle that resolves the function-versus-script question in every encountered case.
2. Provide a categorisation matrix so that an existing toolbox can be triaged in a single sitting.
3. Sequence the refactor into seven small phases, each independently shippable and reversible.
4. Document the gotchas that only surface when scripts are extracted out of a long-lived shell environment.

Errors and better approaches are welcome; see the Feedback section at the end.

Figure 2: A second monitor displaying terminal panes during an active refactor session.

2 Background: function or script?

The function-versus-script rule developed here rests on a small set of recurring terms (process, state, namespace, scope, testing, reproducibility). Each is defined briefly below before the substantive discussion begins, so that the later prose can be read against a fixed vocabulary.

2.1 Terminology

Process. An operating-system execution unit with its own memory, environment, file descriptors, and a unique process identifier (PID). Every running program is a process. A shell, an editor, a script, and a database server are each a separate process; the operating system isolates them so that one cannot directly read or write another's memory. Communication between processes is restricted to explicit channels: arguments and environment passed at launch, file descriptors, signals, and exit codes.

State. The information a process holds in memory at a given moment. For a shell process, the state includes the current working directory, the values of environment and local variables, the defined aliases and functions, the shell options, the history list, the directory stack, and the job table. State is what distinguishes one interactive shell session from another, and it is what makes the function-versus-script question consequential at all: a function can read and write the calling shell's state, whereas a script cannot.

Namespace. The set of names (variables, functions, aliases) visible to a particular execution context. The interactive shell maintains one namespace; a child process launched from it receives a copy of the exported portion and nothing else. Two processes can hold variables of the same name without collision because each has its own namespace. The R analogy below uses the term in the same sense: an R session has a namespace into which `library()` calls bind names.

Scope. The region of code in which a given name is defined and accessible. Shell functions and scripts have distinct scopes by construction: a variable set inside a script is visible only within that script's process unless explicitly exported to children, while a variable set inside an interactive function (without `local`) is visible to the entire shell session. Scope is the language-level property that follows from process boundaries and namespace rules.

Testing. Exercising code in a controlled environment with known inputs and asserting that the observed outputs match expectations. In shell work, testing typically means invoking a script with prepared arguments, capturing its `stdout`, `stderr`, and exit code, and comparing them against reference values; static analysers such as `shellcheck` address a complementary class of issues without execution. A helper that cannot be invoked with a controlled input set, in isolation from a live interactive session, cannot be tested in this sense.

Reproducibility. The property that re-running the same code on the same inputs in the same environment yields the same outputs. Reproducibility requires both that the code be captured (version control, an executable file at a stable location) and that the execution environment be characterised (declared dependencies, pinned versions, an explicit shebang). A helper defined inline in a personal startup file fails the first requirement; a script in `~/bin` under `git` satisfies it.

2.2 How the shell executes code

To classify a helper sensibly, it helps to begin with how the shell actually executes code.

When a user opens a terminal, the shell launches as a long-lived process. It reads a startup file (`.zshrc`, `.bashrc`, or similar), constructs its environment (exported variables, defined aliases and functions, the current working directory, the history file, the directory stack, key bindings, completion definitions), and then waits for input at a prompt. Every command the user enters is interpreted in the context of that single process and the in-memory state it carries.

Within this model, a helper can run in one of two fundamentally different ways: inside the same process as the interactive shell, or inside a new process spawned from it. The distinction is mechanical and unambiguous, and it determines what the helper is permitted to change.

2.3 Functions: code in the current process

A shell function is a named block of code, defined as `name() { ... }` (or `function name { ... }`), that executes in the *current* shell process. Because it runs there, it has direct access to that process's state. It can:

- Change the working directory via `cd`, with the change persisting for subsequent commands at the same prompt.
- `export`, set, or unset environment variables that future commands in the session will inherit.
- Define or redefine aliases, options (`setopt`), key bindings, and completion functions.
- Manipulate the directory stack (`pushd`, `popd`), the command history, and the job table.

Functions are most commonly defined in a startup file so that they are available in every interactive session. The cost is borne by every shell invocation: each function is parsed in full before the prompt appears, regardless of whether it is ever called.

2.4 Scripts: code in a child process

A shell script is a file with execute permission that the shell runs by spawning a *child process*. The mechanism is the standard Unix `fork` and `execve` pair: the parent shell calls `fork`, which produces a near-identical copy of the shell process (the child), and the child then calls `execve` to replace its own program image with the interpreter named in the script's shebang line (`#!/usr/bin/env bash`, `#!/usr/bin/env zsh`, and so on). The parent typically waits for the child to finish; the child runs the script and eventually exits.

2.4.1 The parent-child relationship

From the operating system's perspective the parent and child are two distinct processes. Each has its own process identifier (PID), its own address space (a private region of memory neither process can read from the other), its own working directory, its own copy of the environment, and its own file descriptor table. At the moment of `fork`, the child receives a snapshot of the parent's exported environment, the parent's open file descriptors (`stdin`, `stdout`, `stderr`), the command-line arguments passed at invocation, and a copy of the parent's working directory. That snapshot is

the entirety of what crosses the boundary. After the fork, the two processes are independent: state changes the child makes are invisible to the parent, and state changes the parent makes are invisible to the child. When the child exits, the parent collects three pieces of information and nothing else: the bytes the child wrote to `stdout`, the bytes the child wrote to `stderr`, and the integer exit code returned by the child's final instruction.

2.4.2 What this means for `cd` (and everything like it)

The practical consequence is that a script cannot mutate the state of the shell that invoked it. The point is precise enough to deserve a careful statement: a script *can* call `cd`, and the call succeeds. The child process moves to the new directory and any subsequent commands in the script run from there. What the script cannot do is `cd` the *parent shell*, because the parent and child each hold their own working directory, and the child's was a copy from the start. A common source of early confusion is the user who places `cd /some/project` inside a script, executes it, and is puzzled to find the prompt unchanged. The script ran, the child process did `cd`, the child exited, and the parent shell, never having shared a working directory with the child, remained where it was.

The same argument applies to environment variables, aliases, shell options, the directory stack, and every other element of shell state: a script can change them inside its own process, but the changes are discarded the moment that process terminates. This is the same process isolation that allows a crashing program to leave the rest of the system intact.

2.4.3 The one explicit exception: `source`

The shell built-in `source` (equivalently, `.`) does *not* spawn a child process. It reads the named file and executes its commands directly in the *current* shell. A `cd` inside a sourced file therefore does change the calling shell's working directory, and an `export` inside a sourced file does set a variable in the calling shell's namespace. Sourcing is the mechanism by which `.zshrc` contributes its definitions to the interactive session in the first place, and it is the standard workaround when a helper genuinely needs to leave the user in a new directory but is too long to live as a function.

The cost is the loss of isolation: a sourced file shares the calling shell's state and can read, modify, or clobber any name defined there. An error in a sourced file (an unset variable under `set -u`, a stray `exit`) terminates the interactive session rather than a child process. For these reasons, sourcing is reserved for files whose authors explicitly intend to operate on the parent's state; ordinary helpers should be invoked, not sourced.

2.5 Why the distinction matters

These two execution models are not interchangeable. Some helpers genuinely require the function form because their entire purpose is to mutate the parent shell's state: a wrapper that prepares a project directory and leaves the user inside it, a helper that loads a credential into the current session via `export`, a function that toggles a shell option for the remainder of the session. Other helpers fit equally well in either form, but the choice has measurable consequences for testability, auditability, startup performance, security review, and the reader's mental model of where logic lives.

A toolbox that has drifted out of alignment shows two symmetric inversions: scripts that should have been aliases (because the 'script' is a single command line that does not even merit its own file), and functions that should have been scripts (because the 'function' performs no operation that requires access to the parent shell's state). The next section defines the rule that resolves both.

3 Guiding principle

The rule is short:

> A helper should be a shell function only when it must mutate the calling shell. Otherwise it should be a versioned, executable script in `~/bin`.

The remainder of this section unpacks the rule in terms readily familiar to an audience trained in scope, reproducibility, and the boundaries between processes.

3.1 A familiar analogy: R functions and Rscript files

R users have lived with the same boundary for years without naming it explicitly. A pure helper that takes inputs, returns a value, and produces no side effects is written as an R function, often grouped with related functions in an R package. A one-shot batch job that loads a dataset, fits a model, writes an `.rds` file, and emits a PDF report is written as a script and executed with `Rscript` or rendered through Quarto. No experienced R user would propose to embed an entire batch pipeline in `.Rprofile`, even though `.Rprofile` is technically capable of running it. The pipeline does not require access to the interactive session's namespace, and locating it there would slow every R session, obscure it from version control review, and prevent any tooling from analysing it as a self-contained unit.

The shell case is the same boundary at a different scale. A shell function corresponds to a function loaded into the user's current R session: it can read and modify that session's globals, and it lives or dies with the session. A shell script corresponds to `Rscript` invoked from the command line: it receives a fresh interpreter, communicates back through `stdout` and an exit code, and exits cleanly. The process boundary that makes `Rscript` reliable for batch work is the same boundary that makes a shell script reliable for non-interactive use. Conflating the two in shell-land carries the same costs as it would in R-land; the shell community has historically been less attentive to the conflation.

3.2 Scope, in four lines

Concretely, a helper *must* be a function when it does any of the following in a way that should outlast the helper's run:

- Change the working directory.
- Export, set, or unset shell variables.
- Modify aliases, options, key bindings, completion definitions.
- Manipulate the directory stack, history, or job table.

Each item on this list shares a single property: the change must persist in the calling shell's process, which is precisely what a child process is structurally incapable of accomplishing. Anything outside the list (running a program, reading or writing files, formatting output, fetching from a URL, calling `git`, calling `make`, calling a database, parsing JSON) communicates with the surrounding world through file descriptors and exit codes alone, which is exactly what the script form is designed for. The rule scales cleanly: a function that *both* runs a long pipeline *and* needs to `cd` at the end is two helpers, one calling the other.

3.3 Why the inversion is expensive

A long shell function in `.zshrc` carries five practical costs that map directly onto practices already familiar from `renv`, package versioning, and unit testing.

Startup latency. A function in `.zshrc` is parsed every time the shell starts. A script is parsed only when invoked. A two-hundred- line zsh function that the user calls a few times a day is re-parsed dozens of times for each one execution. The latency cost is the same shape as `library()` calls in `.Rprofile`: convenient on the surface, expensive in aggregate.

Lint and static analysis. `shellcheck` reads scripts; it does not introspect functions buried in dotfiles. Quoting bugs, unset variables, and incorrect test predicates that would be caught immediately in a script silently survive in a `.zshrc` function. The R parallel is `lintr` against package code versus `lintr` against ad-hoc chunks in `.Rprofile`.

Version control granularity. A script is a file. Its diffs are local, its history is local, blame is meaningful, and its commits do not interact with unrelated config changes. A function in a shared `.zshrc` competes with `PATH` exports, plugin lines, keybindings, and PROMPT changes for git history attention. Any non-trivial helper deserves its own git history, the same way a non-trivial R helper deserves its own `R/foo.R`.

Testability. A script can be invoked under a controlled environment with arguments, redirected I/O, and a captured exit code. The same logic embedded in a `.zshrc` function can only be tested by sourcing the dotfile and calling the function in a live shell, which is roughly as defensible as testing R code by copy-pasting it into the console.

Reuse across hosts and contexts. Scripts in `~/bin` are picked up by anything that respects `PATH`: cron, launchd, Make rules, Docker containers that mount the home directory, HPC submission scripts, scheduled GitHub Actions runners that source the toolbox. Functions are visible only to interactive zsh sessions. A research-backup helper hidden in `.zshrc` cannot be wired to a launchd job; the same logic in `~/bin` can be scheduled in seconds. This boundary maps almost exactly to the line between 'helpers exported by a package' and 'helpers defined inline in a notebook' in R work.

3.4 Why the inversion happens

A second pattern to notice: the inversions are not random. Helpers that grow up as one-liners (`cd somewhere && open something`) are written as scripts because the author did not yet know about aliases. Helpers that grow up gradually (a git workflow that started as a snippet, then accreted secret-scanning, then commit- message templating, then a confirmation prompt) end up as functions because they were edited each time a new shell was open and `.zshrc` was already in

the editor. The forces driving each inversion are completely different, but the symptom (a toolbox where category does not predict location) is identical.

Recognising both forces matters because the corrective for each is different. The microscript that should have been an alias becomes an alias and disappears. The accumulated function that should have been a script becomes a script and gets the full hardening treatment: shebang, `set -euo pipefail`, quoted variables, a `-h`/`--help` flag, a `shellcheck` pass, and a place in version history.

3.5 When the rule cuts the other way

For completeness, the rule does sometimes route a current script *into* the function form. A helper that the author scripted but that genuinely needs to leave the user in a different directory is mis-categorised as a script. The give-away symptom: 'why does my shell not stay in the project directory after I run this?'. That helper's logic should move to a function, or, more cleanly, the script should be retained for non-interactive callers (cron, make) and a thin shell function should call it and apply the final `cd`.

3.6 What the rule is not

The rule is not 'all logic must leave `.zshrc`'. The dirstack helpers, the `ff` fzf-then-cd shortcut, the navigation jumps to project subdirectories: all are legitimately functions, because they exist precisely to mutate the calling shell. The point is to keep the function tier narrow and obviously appropriate, so that when a future reader sees a function in `.zshrc`, the existence of that function is itself evidence that it had to be one.

4 Symptoms of an unaligned toolbox

Three signs reliably indicate that a toolbox needs a refactor along these lines.

The shell startup file has crossed a thousand lines. Most of the bulk is unlikely to be configuration. It will be logic that should have been extracted long ago.

Microscripts in `~/bin` outnumber non-trivial scripts. A directory with thirty entries, half of which are under five lines, is mostly aliases waiting to be promoted.

At least one shell function imports a non-trivial external program. A `.zshrc` function that calls `gitleaks`, `aws`, `pandoc`, or `jq` is almost certainly mis-located. External programs imply a real workflow, and real workflows belong in versioned files.

5 A categorisation matrix

The matrix below sorts every helper a typical toolbox will contain into one of five fates. Triaging an existing toolbox is a single-sitting exercise: read each helper, ask the four questions in the principle, and place it.

Fate	Trigger
Keep as function	Helper must `cd`, `export`, modify aliases or history
Move from function to script	Function does no shell-state work; it just runs a program
Convert script to alias	Script is a single command line with no real argument parsing
Keep as script (harden)	Script is correctly placed but lacks shebang, quoting, lint
Decommission	Helper duplicates another, is referenced nowhere, or is dead

The first row is the smallest in any toolbox the author has inspected, typically four to twenty helpers. The remaining rows absorb everything else.

Figure 3: Editor showing a shell function being lifted into its own file under ‘~/bin’.

6 A seven-phase refactor

Each phase below is independently shippable and reversible. The ordering is by leverage: the early phases produce most of the visible improvement.

6.1 Phase 1: extract the largest function

In every toolbox the author has examined, a single function dominates by line count. Often it is a custom git commit workflow or a research-notes capture pipeline that grew over the years. That function alone typically contains 30 to 70 percent of the non-config bulk in `.zshrc`.

The extraction is mechanical:

- Create a new file in `~/bin` named after the helper. Mark it executable.
- Add a shebang that matches the syntax in use. Functions written with zsh-only constructs (`${(f)...}`, `${array:#}`, `setopt` semantics) require `#!/usr/bin/env zsh`; do not assume bash will parse them.
- Apply standard hardening: `set -u` always, `set -e` if the function does not deliberately tolerate failure, `set -o pipefail` if any command piping is involved.
- Decide on helpers. Sub-functions called only by the extracted helper can move with it (inline, or in a `lib/` directory the helper sources). External helpers used by other functions stay in shell config.
- Delete the original function from the shell startup file.
- Verify by exercising the helper end-to-end on a scratch workspace, including any failure paths.

This phase alone is usually responsible for a measurable shell startup speedup and for moving the largest single block of code under linting.

6.2 Phase 2: extract the small functions

After the large extraction, several smaller functions remain that also fail the principle: a Mathematica wrapper, a fuzzy file finder that calls vim, a make invocation that runs from any subdirectory. Each is a one-file extraction at most. Time per helper is small; the cumulative effect is roughly another fifty to eighty lines out of `.zshrc`.

6.3 Phase 3: convert microscripts to aliases

Inspect every script under fifteen lines in `~/bin`. Any whose body reduces to a single command pipeline (with no flag parsing, no branching, no output processing) is more honestly expressed as an alias. Examples that look exactly like aliases when written inline:

```
nohup /Applications/Ghostty.app/Contents/MacOS/ghostty \
  >/dev/null 2>&1 &
```

Promote each such script to an alias in shell config and delete the file. The benefit is twofold: the helper now appears in `alias` output (so it is discoverable by listing aliases), and the toolbox shrinks by one file per promotion.

6.4 Phase 4: standardise the rest

The remaining scripts in `~/bin` are correctly scripts. They typically need light hardening:

- Add a shebang, `#!/usr/bin/env bash` for bash-portable scripts.
- Add `set -euo pipefail` unless the script explicitly chooses a more permissive policy and documents why.
- Replace backticks with `$(...)`.
- Quote variable expansions: `"$1"`, `"$PWD"`, and so on.
- Resolve `shellcheck` warnings, or annotate with a `disable` comment that names a reason.
- Add a `-h`/`--help` flag for any script over thirty lines.
- Decide an extension policy and apply it consistently; standard Unix practice is to drop `.sh` from executables and reserve the extension for files that are explicitly sourced. Confirm that any scheduler entries (launchd, systemd, cron) referring to the script by name are updated together.

This is the long-tail phase. It can be done one helper at a time, amortised over normal work.

6.5 Phase 5 (optional): autoload remaining functions

For shell startup files that still feel heavy after phases 1 to 4, move the remaining functions out of the startup file into a function directory and `autoload` them:

```
fpath=(~/.zsh/functions $fpath)
autoload -Uz d ff za zw zy zf zt zs zp zr z0 zm ze zo zc zg
```

Each function lives in its own file under the function path. They load on first call, not on shell start. Skip this phase if the remaining functions are few or if having their bodies visible in the startup file is more valuable than the small startup cost.

6.6 Phase 6: install a guardrail

Add a `shellcheck` pre-commit hook scoped to `~/bin/*`, excluding binaries, R scripts, Python scripts, and any `archive/` directory. The point is not to catch a flood of issues at install time (phase 4 already handled those) but to prevent regressions as the toolbox continues to evolve. A single Make target, `make lint`, that runs `shellcheck` over every executable is sufficient.

6.7 Phase 7: decommission

Triage each remaining helper for redundancy. Common findings: two clipboard-to-notes helpers that do roughly the same thing under different names, a PDF viewer launcher that exists in two versions because the author once wanted to try a different viewer, an installer left behind from a tool that was removed years ago. Pick one of each, retire the rest. Git history preserves the removed code; the working tree should not.

Figure 4: A clean desk after the refactor; the same tools are present, but each sits where it belongs.

7 Things to watch out for

These pitfalls are easy to underestimate before starting and easy to recognise once seen.

1. **Zsh-only constructs.** Many shell functions accumulate zsh conveniences that bash will not parse: `${(f)var}` for newline splitting, `(s::)` for string-splitting flags, parameter modifiers like `${var:#}`. When extracting such a function to a script, the shebang must match the syntax. A bash shebang on a zsh function will fail in unhelpful ways.
2. **Schedulers refer to file names.** If `~/bin/foo.sh` is referenced by a launchd plist, a systemd unit, a cron entry, or a Makefile, renaming to `~/bin/foo` will silently break the scheduler. Search for every occurrence before renaming, and land both changes in the same commit.
3. **`PATH` ordering.** A script extracted from a function inherits `PATH` from the calling shell, which is usually fine. The exception is when the function previously ran inside a block that prepended a directory to `PATH`. The new script needs to do its own `PATH` adjustment or use absolute paths.
4. **Working directory at invocation.** A function inherits the caller's `$PWD`. A script does too, but if the script was previously a function that called itself recursively or that `cd`'d for its own purposes, that behaviour is now subprocess-local and may surprise the parent.
5. **History and dirstack contamination.** A function that previously ran `pushd`/`popd` to navigate around silently lost that capability when extracted to a script. If the helper genuinely needed the navigation, the helper was mis-classified and the rule routes it back to the function tier.
6. **`shellcheck` false positives.** External programs called with computed arguments (`"${args[@]}"`) sometimes trigger `SC2068`, `SC2086`, or `SC2154`. Disable per line with a comment that names the reason; do not blanket-disable the warning across the project.
7. **Secrets in extracted code.** A function previously sourced from `.zshrc` may have benefited from environment variables set elsewhere in the same file. The extracted script does not have that benefit. Either source the relevant env file explicitly inside the script, or read secrets via a tool such as `pass` or a credential helper.

8 Daily workflow after the refactor

The refactor pays back every day in three ways.

Discoverability. A new helper added to `~/bin` shows up under tab completion against `$PATH`. A new function added to the shell config shows up only in the current session and only after a reload.

Auditability. Every helper in `~/bin` is one `git log` away from full history; one `shellcheck` away from lint; one `cat` away from a complete reading. A helper buried in a thousand-line startup file requires the reader to first locate it.

Composability. Scripts in `~/bin` can be called from anything that respects `PATH`: Make rules, scheduled jobs, Docker containers mounted with the home directory, HPC submission scripts, peer collaborators who cloned the dotfiles repository. Functions can only be called from interactive shells

that have sourced them. The composability gain is the most important long-term return on the refactor.

9 What did we learn?

9.1 Lessons learnt

Conceptual:

- The function-versus-script question reduces to a single test: does the helper need to mutate the calling shell? Almost every other consideration follows from that one decision.
- The forces that produce inversions in each direction are different. Microscripts grow up because the author did not know about aliases; oversized functions grow up because the startup file is the file that happens to be open.
- A mature toolbox is mostly scripts, a small set of legitimate functions, and a layer of aliases. Any other distribution is evidence of accumulated misalignment.

Technical:

- A long zsh function may not be portable to bash, even when the prose-level intent looks generic. Confirm the shebang against the syntactic features actually in use.
- `shellcheck` is the highest-leverage tool to apply at the end of an extraction. It catches more than half of the latent quoting and unset-variable bugs in legacy scripts.
- Naming and extension conventions only need to be applied consistently within a toolbox. The choice of `.sh` versus no extension matters less than the absence of a third option appearing for no reason.

Gotchas:

- Renaming a script that a scheduler refers to silently breaks the schedule. Always grep for the file name across `launchd/`, `systemd/`, `Makefile`, and any related repositories before renaming.
- Functions that depend on shell-specific syntax cannot be blindly retitled with a bash shebang. Prefer the zsh shebang unless the syntax has been explicitly portabilised.

9.2 Limitations

The refactor described here addresses helpers in shell-land specifically. It does not:

- Address Python, R, or other-language helpers whose execution model differs from shell scripts.
- Replace a real configuration management system. Tools such as Ansible, Nix, or chezmoi solve a larger problem and may be appropriate when the personal toolbox grows past a few dozen files or needs to be deployed to multiple machines.
- Provide automated migration. Each helper still needs a human to read it, classify it, and verify the extraction.
- Compose with shell frameworks that load logic from many files (Oh My Zsh, prezto). Those frameworks have their own conventions; the rule still applies, but the implementation details change.

9.3 Opportunities for improvement

1. Encode the rule as a small linter that scans `.zshrc` for functions and flags any whose body never references shell state. This would surface candidates for extraction over time rather than at one big-bang refactor.
2. Add a CI workflow that runs `shellcheck` over `~/bin` on every push to the dotfiles repository.
3. Generate `~/bin/nav` (a navigation cheat sheet) from the `autoload`ed function metadata, so adding a new shortcut automatically updates the help output.
4. Build a thin packaging layer around shared helpers so they can be installed on collaborator machines without copying files by hand. A simple Make target plus a manifest is usually sufficient.
5. Track helper invocation frequency via the shell's `precmd` hook for one week per year, and use the result to prune helpers nobody actually calls.

10 Wrapping up

A personal toolbox is software, even when it is small, even when it is private, even when it grew organically. Treating it as software (with a rule for how it is structured, lint applied to the parts that should be linted, version control applied to the parts that change, and a maintenance pass when the bulk crosses some threshold) pays back every day for the next decade.

The rule that drives the refactor is short enough to remember: function only when shell state must change, script otherwise. The work to apply it is finite, ordered, and reversible. A single afternoon, divided across the seven phases above, is enough to move a toolbox from accumulated drift back into alignment.

In conclusion, four points merit emphasis. First, the rule reduces every helper-placement decision to a single question about shell state. Second, the largest single win is extracting the dominant function out of the shell startup file and into `~/bin`. Third, microscripts that read like one-liners belong as aliases, not as files. Fourth, a `shellcheck` guardrail prevents the same drift from re-accumulating.

11 See Also

Related posts:

- Setting up dotfiles on GitHub: versioning the shell startup file alongside the rest of the configuration.
- A research backup system: a worked example of a helper that belongs as a script invoked from a scheduler, not as a function in a shell.

Key resources:

- shellcheck: static analyser for shell scripts.
- zsh manual: parameter expansion: authoritative reference for zsh-only constructs encountered during extraction.
- Bash hackers wiki: portable shell: inventory of constructs that fail to port between shells.

- The twelve-factor app: the configuration discipline that the rule echoes for shell environments.

---

12 Reproducibility

Tested configuration:

Component	Version
Operating system	macOS 15.4
Shell	zsh 5.9
shellcheck	0.10.0
Last verified	2026-04-25

Configuration files:

- `analysis/configs/refactor_plan.md`: the seven-phase plan in the format used to drive the worked refactor.

---

13 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.

13.1 Related posts in this cluster

This post is part of the *Shell Scripting and Git Tooling* series. Recommended reading order:

1. **Post 41: Refactoring a Personal Toolbox: Scripts versus Shell Functions** (this post)
2. Post 43: A Mac Workflow for Tracking Daily Research Progress