Multi-Laptop macOS Bootstrap: Migrating Dotfiles to a Versioned Git Repository

Ronald 'Ryy' G. Thomas

2026-05-17

Two machines, one source of truth: a reproducible dotfiles repository makes adding a second workstation a two-command operation.

> **Note**
>
> **Multi-Machine Setup Series** (recommended reading order): Part 1: Migrating off Dropbox | Part 2: Multi-Laptop Bootstrap | Part 3: Security Audit

Introduction

Post 64, 'Migrating Off Dropbox: Beyond Dotfiles', covers migrating project content, shell history, and large binaries off cloud sync into a three-layer framework. This post picks up where that one ends: the dotfiles layer. After the Dropbox migration, every alias, every shell function, every

launchd scheduled job, and every editor setting lives in ~ as plain files, but with no `.git` folder, no version history, no diff capability, and no rollback mechanism.

The design for a better arrangement already existed. Post 24, 'Creating a GitHub Dotfiles Repository', laid out the correct architecture: a standalone `~/dotfiles` git repository outside any cloud-sync path, an `install.sh` that creates symlinks with timestamped backups, platform detection via `$OSTYPE`, a `Brewfile`, and an `.env.local` pattern for secrets. That post was written and published in February 2026 but never executed against the actual configuration.

This post documents the audit that identified three concrete blockers preventing multi-laptop deployment, and the migration procedure that resolves them. It extends post 24's design with five additions: launchd plist parameterisation, a pipx tool manifest, a vim plugin-manager strategy, an explicit sensitive-files inventory, and migration steps from the current half-installed state.

Motivations

- The `tn` incident (a `sed -i.bak` against a cloud-mounted path truncated a production script to zero bytes) demonstrated that cloud sync is not a substitute for version control; git is the correct synchroniser.
- Setting up a second MacBook without a deployment mechanism means manually recreating months of accumulated configuration from memory, which is both error-prone and time-consuming.
- Hardcoded absolute paths in scripts and launchd plists break silently on any machine where the directory layout differs even slightly from the primary machine.
- Sensitive files (`~/.aws`, `~/.ssh`, `~/.gnupg`) live adjacent to tracked dotfiles with no formal inventory and no explicit decision about what belongs in git versus what must remain local.
- Understanding the gap between a written architecture and an executed one is itself useful: post 24's design was sound but several category errors (vim plugin bundles treated as config, runtime caches mixed with user config) were only visible during an actual audit.

Objectives

1. Identify every file that must be tracked, excluded, or handled via a separate secrets mechanism, producing an explicit inventory.
2. Create `~/dotfiles` as a git repository outside any cloud-sync path, structured per post 24's layout, and push it to a private GitHub repository.
3. Write and test `install.sh` with `--dry-run` support, launchd `__USER__` substitution, and per-file symlinks for `~/bin`.
4. Verify that a clean bootstrap (on a fresh user account or VM) runs without hardcoded-path failures, undefined variable errors, or leaked secrets.

Figure 1: Ambiance image 1: a single open laptop on a wooden desk, terminal visible, soft natural light. Placeholder.

What is a Dotfiles Repository?

A dotfiles repository is a version-controlled directory that stores shell configuration files, editor settings, and system scripts in a form that can be deployed to any new machine with a single command. The analogy to application source code is direct: 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.

The concrete benefit is reproducibility. Without a dotfiles repository, two machines diverge the moment one is customised. With one, the command `git clone git@github.com:rgt47/dotfiles ~/dotfiles && cd ~/dotfiles && ./install.sh` brings a new laptop to the same state as the primary machine, minus secrets that must be installed by a separate mechanism.

Prerequisites

This post assumes:

- **Operating system:** macOS 13+ (Ventura or later); the architecture extends to Ubuntu 22.04+ with minor adjustments
- **Hardware:** Apple Silicon or Intel Mac (the scripts are architecture-agnostic)

- **Already installed:** git, Homebrew, zsh 5.9+, pipx (for Python CLI tools)
- **Background knowledge:** comfort editing dotfiles, writing bash scripts, and running shell commands; familiarity with post 24's design
- **GitHub account:** with SSH access configured (the repository must be private; never use a public repository for dotfiles)
- **Time required:** 3-4 hours for the initial migration; 30 minutes to bootstrap subsequent laptops

For those starting from scratch rather than migrating, post 24's step-by-step walkthrough is the right entry point. This post assumes post 64's Dropbox migration has been completed and that shell configuration files now live in ~ as plain, unversioned dotfiles.

Audit Findings: Three Blockers

Before writing any code, an audit of `~/bin`, `~`, `~/.config`, and `~/.local` identified three issues that must be resolved before dotfiles can safely be exported to a second machine.

Blocker 1: No Version Control

The shell configuration living in ~ has no `.git` directory. There is no history, no diff capability, and no rollback. As the `tn` incident showed (covered in post 64), filesystem-level recovery is not guaranteed for in-place edits; git is the only reliable mechanism.

The repository must live at `~/dotfiles`, outside any cloud-sync path. Git writes constantly to `.git/index`, `.git/HEAD`, and `.git/refs/...`; a cloud provider syncing those writes simultaneously causes race conditions that corrupt the index and can truncate files to zero bytes. The `~/dotfiles` directory is local to each machine; git itself (with GitHub as the remote) is the synchroniser.

Blocker 2: Hardcoded Absolute Paths

Every script in `~/bin/launchd/`, the launchd plists, the `cdpath` in `.zshrc`, and the `vz` alias hardcode machine-specific absolute paths. On a second laptop with even a slightly different directory layout, every such reference fails silently.

The fix is a single environment variable sourced from `~/.zshenv`:

```
# ~/.zshenv
export DOTFILES="${DOTFILES:-$HOME/dotfiles}"
export PRJ_ROOT="${PRJ_ROOT:-$HOME/prj}"
```

Every script and config then references `$DOTFILES` and `$PRJ_ROOT` rather than literal paths.

Blocker 3: Sensitive Files Without an Inventory

Confirmed sensitive items in ~:

```
~/.aws      AWS credentials (plaintext)
~/.docker   registry tokens
~/.env      project secrets
~/.gnupg    private keys
~/.npmrc    npm registry token
~/.password-store  pass database (encrypted)
~/.ssh      SSH private keys
~/.vpn      VPN configuration
```

These must never be committed to git, even in a private repository, because they appear in every cloned working tree and persist in history forever. A `.gitignore` written before the first `git add` is the only reliable control; after the first commit, sensitive data requires a full `git filter-repo` rewrite and credential rotation.

Installation: Creating ~/dotfiles

The migration proceeds in three phases: copy, restructure, and commit.

Phase 1: Copy Existing Content

```
# Create the new home outside cloud sync
mkdir -p ~/dotfiles
cd ~/dotfiles
mkdir -p shell git editors system config bin launchd packages secrets

# Shell configuration: drop leading dots
cp ~/.zshrc            shell/zshrc
cp ~/.zshenv           shell/zshenv
cp ~/.gitignore_global git/gitignore_global
cp ~/.lintr            lintr
cp ~/.mbsyncrc         mbsyncrc

# bin scripts: copy real files (not symlinks)
cp -RL ~/bin/* bin/
rm -f bin/*.bak.*

# XDG configs (selective; leave runtime state behind)
for d in ghostty kitty karabiner gh rclone mutt nvim; do
    [ -d ~/.config/$d ] && cp -R ~/.config/$d config/$d
done
```

Do not copy `~/.config/vim/` (22M of downloaded plugin bundles), `~/.config/coc/` (1.6M of language-server cache), or any directory under `~/.config/` that contains runtime state rather than user configuration. These will be regenerated on each laptop by the plugin manager.

Phase 2: Write .gitignore Before git init

This step is not optional. Write the `.gitignore` file before the first `git add`. One commit with a credential in history requires a full rewrite.

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

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

# Plugin manager output (regenerated at install time)
config/vim/bundle/
config/vim/plugged/
config/nvim/plugged/
config/nvim/coc/
```

Phase 3: Commit and Push

```
cd ~/dotfiles
git init
git add .
git status   # review carefully before committing
git commit -m 'initial dotfiles import'
gh repo create rgt47/dotfiles --private --source=. --push
```

Review `git status` output before committing. If any file in the listing looks sensitive, add it to `.gitignore` and re-check before proceeding.

Configuration

install.sh

The installer creates symlinks from `~/dotfiles` into `$HOME`, handles launchd plist parameterisation, and installs packages via Homebrew and pipx. A `--dry-run` flag prints every action without executing it.

```
#!/bin/bash
# install.sh: deploy dotfiles into $HOME via symlinks.
# Idempotent: existing files are backed up with a timestamp
# suffix before being replaced.
set -euo pipefail

DRY_RUN=0
[[ "${1:-}" == "--dry-run" ]] && DRY_RUN=1

DOTFILES="$(cd "$(dirname "$0")" && pwd)"

log()      { echo "[INFO]  $*"; }
warn()     { echo "[WARN]  $*" >&2; }
do_or_say() {
  [[ $DRY_RUN -eq 1 ]] && echo "[DRY] $*" || "$@"
}

link_file() {
  local src="$1" dest="$2"
  if [ -e "$dest" ] && [ ! -L "$dest" ]; then
    warn "$dest exists, creating backup"
    do_or_say mv "$dest" \
      "${dest}.backup.$(date +%Y%m%d_%H%M%S)"
  fi
  do_or_say mkdir -p "$(dirname "$dest")"
  do_or_say ln -sfn "$src" "$dest"
  log "Linked $src -> $dest"
}

# Shell, git, editors ------------------------------------------------
link_file "$DOTFILES/shell/zshrc"  "$HOME/.zshrc"
link_file "$DOTFILES/shell/zshenv" "$HOME/.zshenv"
link_file "$DOTFILES/git/gitconfig"         "$HOME/.gitconfig"
link_file "$DOTFILES/git/gitignore_global"  "$HOME/.gitignore_global"
link_file "$DOTFILES/editors/vimrc" \
  "$HOME/.config/vim/vimrc"

# XDG configs --------------------------------------------------------
for d in "$DOTFILES"/config/*/; do
```

```
  name="$(basename "$d")"
  link_file "$d" "$HOME/.config/$name"
done

# ~/bin scripts: per-file symlinks (not a top-level link) -----------
do_or_say mkdir -p "$HOME/bin"
for f in "$DOTFILES"/bin/*; do
  link_file "$f" "$HOME/bin/$(basename "$f")"
done

# launchd plists: substitute __USER__ then install ------------------
do_or_say mkdir -p "$HOME/Library/LaunchAgents"
for plist in "$DOTFILES"/launchd/*.plist; do
  name="$(basename "$plist")"
  target="$HOME/Library/LaunchAgents/$name"
  do_or_say sed "s|__USER__|$USER|g" "$plist" > "$target"
  log "Installed $target (with __USER__ -> $USER)"
done

# Package managers --------------------------------------------------
if command -v brew >/dev/null 2>&1; then
  do_or_say brew bundle \
    --file="$DOTFILES/packages/Brewfile"
fi
if command -v pipx >/dev/null 2>&1 && \
   [ -f "$DOTFILES/packages/pipx-tools.txt" ]; then
  while read -r tool; do
    [[ -z "$tool" || "$tool" =~ ^# ]] && continue
    do_or_say pipx install --force "$tool"
  done < "$DOTFILES/packages/pipx-tools.txt"
fi

log "Install complete. Manual follow-ups:"
log "  - Copy shell/env.local.example to ~/.env.local"
log "  - Restore ~/.aws, ~/.ssh, ~/.gnupg from secure storage"
log "  - Reload launchd: launchctl bootstrap gui/\$UID \
~/Library/LaunchAgents/*.plist"
log "  - Restart shell: source ~/.zshrc"
```

The full source is at `analysis/configs/install.sh`.

Three deviations from post 24's original `install.sh` deserve explanation:

1. **`--dry-run` flag:** prints every operation without executing, allowing review before the first real run on a new machine.
2. **XDG-config and `~/bin` loops:** a new helper script or config directory is picked up automatically without editing `install.sh`.

3. **launchd `__USER__` substitution:** macOS `launchd` does not expand ~ in plist files, so absolute paths are required; the installer writes the substituted plist to `~/Library/LaunchAgents/` rather than tracking the substituted form in git.

Makefile

```
.PHONY: install update lint backup test help

help:
    @echo "Targets:"
    @echo "  install   create symlinks; install brew/pipx packages"
    @echo "  update    git pull, then re-run install.sh"
    @echo "  lint      shellcheck bash; zsh -n zsh scripts"
    @echo "  backup    snapshot current dotfiles"
    @echo "  test      install.sh --dry-run plus lint"

install:
    ./install.sh

update:
    git pull origin main
    ./install.sh

lint:
    find bin -type f -name '*.sh' -exec shellcheck {} \;
    find shell -type f -exec zsh -n {} \;

backup:
    @d=backups/$$(date +%Y%m%d-%H%M%S) && mkdir -p "$$d" && \
    for f in .zshrc .zshenv .gitconfig .vimrc; do \
      [ -e "$$HOME/$$f" ] && cp "$$HOME/$$f" "$$d/"; \
    done

test:
    ./install.sh --dry-run
    $(MAKE) lint
```

The full source is at `analysis/configs/Makefile`.

Capturing Package Manifests

Before the first push, capture the current toolchain so that a new laptop gets the same packages:

```
brew bundle dump \
  --file=~/dotfiles/packages/Brewfile --force

# pipx tools
pipx list --short \
  | awk '{print $1}' > ~/dotfiles/packages/pipx-tools.txt
```

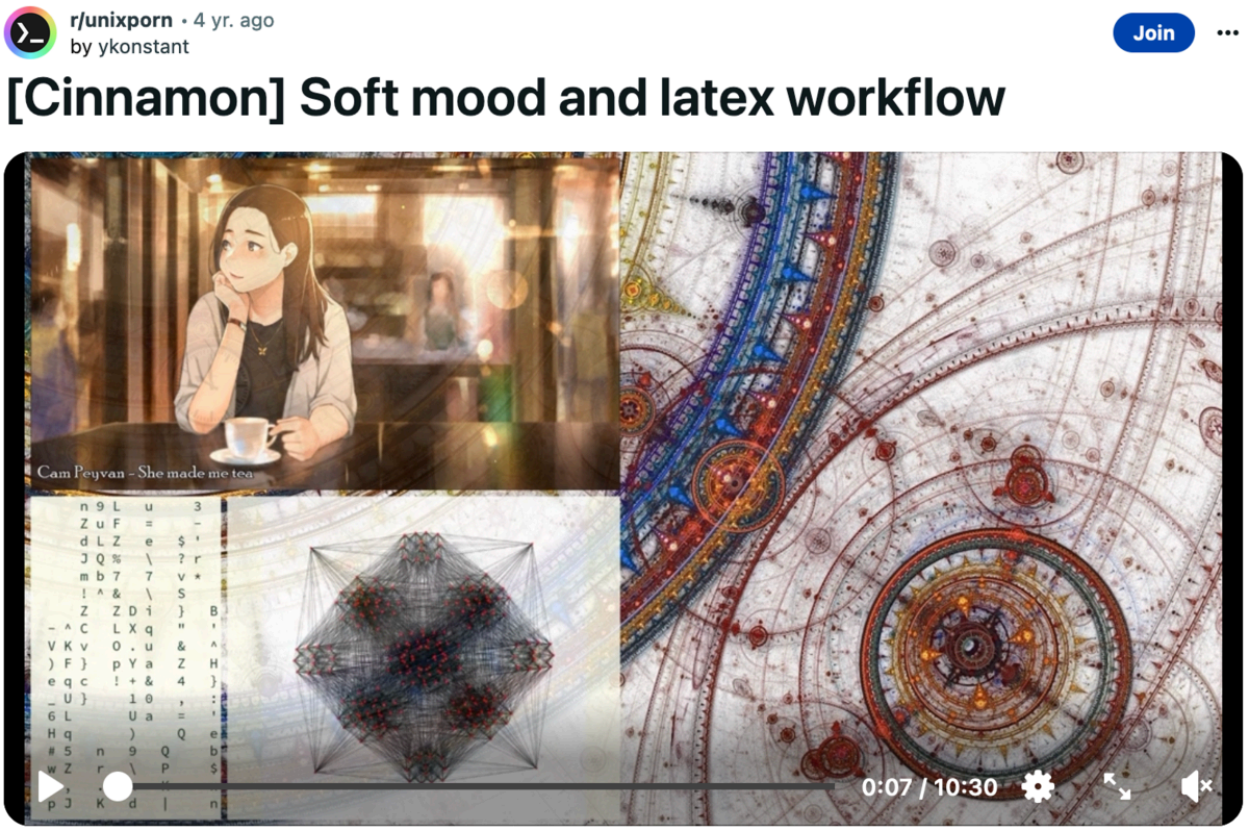

Figure 2: Ambiance image 2: a terminal showing git log output and file structure, warm overhead lighting, shallow depth of field. Placeholder.

The Migration Procedure

With `~/dotfiles` populated and committed, deploy to `$HOME`:

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

After the installer runs:

1. Reload launchd agents:

```
launchctl bootstrap gui/$UID \
  ~/Library/LaunchAgents/*.plist
```

2. Restart the shell: `source ~/.zshrc`
3. Verify a sample alias or function resolves correctly.

To bootstrap a second laptop:

```
git clone git@github.com:rgt47/dotfiles ~/dotfiles
cd ~/dotfiles && ./install.sh
```

Then manually restore `~/.aws`, `~/.ssh`, and `~/.gnupg` from secure storage (1Password, `pass`, or hand-transfer). These are never in git; the installer prints a reminder.

Verification

```
# 1. Confirm symlinks are in place
ls -la ~/.zshrc ~/.gitconfig ~/.config/ghostty

# 2. Check that DOTFILES and PRJ_ROOT resolve correctly
echo "$DOTFILES"
echo "$PRJ_ROOT"

# 3. Verify launchd jobs loaded without error
launchctl list | grep local.

# 4. Confirm pipx tools are installed
pipx list --short

# 5. Dry-run passes cleanly on the current machine
cd ~/dotfiles && make test
```

Step 5 (`make test`) runs `install.sh --dry-run` and `make lint`, which exercises `shellcheck` against every script in `bin/`. A clean run with no warnings confirms the repository is in a deployable state.

Daily Workflow

After setup, the dotfiles repository becomes a standard git project.

Command	Action
`cd ~/dotfiles && git diff`	Review uncommitted config changes

Command	Action
`make test`	Lint + dry-run before committing
`make install`	Re-link after adding a new config file
`make update`	Pull from GitHub and re-run install
`make backup`	Snapshot current ~ dotfiles
`brew bundle dump --force`	Refresh Brewfile after new installs
`pipx list --short > pipx-tools.txt`	Update pipx manifest

The repository should be managed like any other codebase: commit small changes, test before pushing, and keep commit messages descriptive enough to understand what changed and why.

Things to Watch Out For

1. **Write `.gitignore` before `git init`, not after.** If any sensitive file is committed even once, it must be removed with `git filter-repo` and every credential it contained must be rotated. There is no lighter-weight remediation.
2. **Do not place the dotfiles repo inside Dropbox, iCloud, or Google Drive.** Git writes to `.git/index`, `.git/HEAD`, and ref files continuously; a cloud provider racing to sync those writes can corrupt the index or truncate files to zero bytes. This is the same failure mode as the `tn` incident (2026-05-03).
3. **The launchd `__USER__` substitution writes to `~/Library/LaunchAgents/`, not back to the repo.** If you edit a plist in `~/dotfiles/launchd/` and re-run `install.sh`, the substituted file is overwritten in place. Never `git add` the file from `~/Library/LaunchAgents/`; always edit the template in `~/dotfiles/launchd/`.
4. **vim plugin bundle directories must not be tracked.** The 22M `~/.config/vim/` directory is almost entirely downloaded plugin code managed by vim-plug or Vundle. Track only the `vimrc` and the plugin manifest; the installer should run the plugin manager on first boot. The same applies to `~/.config/coc/` (1.6M of language-server cache). Add both to `.gitignore` before the first commit.
5. **`~/bin` must be a real directory populated by per-file symlinks, not a symlink to any directory outside `$HOME`.** A top-level symlink to an absolute path dangles silently on any machine with a different directory layout. The installer creates individual symlinks from `~/dotfiles/bin/<script>` to `~/bin/<script>`.
6. **Test `install.sh --dry-run` on a non-primary user account before running it on a second laptop.** The first dry-run on a fresh environment will surface hardcoded paths and assumptions invisible on the primary machine. Identify and fix them before the real bootstrap.

Uninstall / Rollback

To remove the dotfiles installation and restore the previous state:

```
# 1. Reverse the symlinks (restores any .backup.TIMESTAMP files)
for f in ~/.zshrc ~/.zshenv ~/.gitconfig ~/.gitignore_global; 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 ~/bin per-file symlinks (leaves non-symlinked files)
find ~/bin -maxdepth 1 -type l -delete

# 3. Unload launchd agents
launchctl bootout gui/$UID \
  ~/Library/LaunchAgents/local.*.plist

# 4. Remove the installed plists
rm ~/Library/LaunchAgents/local.*.plist

# 5. Remove the ~/dotfiles repo (optional)
rm -rf ~/dotfiles
```

If the `.backup.TIMESTAMP` files were not created (first-run on a machine with no prior dotfiles), step 1 produces empty symlinks that must be removed manually. The `--dry-run` flag on `install.sh` shows exactly which files would be backed up before the real run.

Figure 3: Ambiance image 3: two laptops open side by side on a shared desk, matching terminal sessions, muted earth tones. Placeholder.

What Did We Learn?

Lessons Learnt

Conceptual Understanding:

- A dotfiles repository is the source code for a development environment. Like application code, it should live in git, be tested before deployment, and have a migration path for breaking changes.
- Cloud-sync and git are fundamentally different synchronisation mechanisms with conflicting write patterns. Using cloud storage as a substitute for git version control is not a safe design; it provides neither the branching, diffing, nor rollback capabilities of git, while adding race-condition risk.
- The distance between a written architecture and an executed one accumulates debt. Post 24's design was correct, but the gap between the written plan and the actual filesystem meant three months of configuration changes had no version history.
- An explicit inventory of sensitive files, written before the first `git add`, is a one-time cost that prevents a very expensive recovery. Credential rotation after an accidental commit is not hypothetical; it has happened to experienced engineers.

Technical Skills:

- The `link_file` pattern (back up existing files with a timestamp, then create a symlink) makes `install.sh` idempotent and re-runnable without destroying local changes.
- The `__USER__` substitution pattern for launchd plists is a clean solution to the constraint that macOS `launchd` does not expand ~ in plist files.
- A `--dry-run` flag in shell scripts (print every action without executing) is low-cost to implement and essential for validating an installer on a production machine before committing.
- Separating package manifests (`Brewfile`, `pipx-tools.txt`, `nvim-plugins.lua`) from the install script means adding a new tool requires one line in a manifest file, not an edit to the installer logic.

Gotchas and Pitfalls:

- The order of operations matters: `.gitignore` before `git init`, `git status` review before `git add`, `--dry-run` before the real run. Any of these steps skipped creates a problem that is harder to fix than to prevent.
- Runtime caches and downloaded plugin code are frequently mixed with user configuration in `~/.config`. The rule is: if a directory would be regenerated by running a tool, it is not user configuration and must not be tracked in git.
- `~/bin` as a top-level symlink to an absolute path works on the primary machine but produces a dangling symlink on any other machine. The per-file symlink pattern is more verbose but correct across environments.
- The `rclone` configuration file (`~/.config/rclone/rclone.conf`) contains OAuth tokens and must be scrubbed of credentials before it is committed, even to a private repository.

Limitations

- **Single-user scope:** `install.sh` deploys to `$HOME` and makes no attempt to set up system-level configuration (`/etc/`, `/usr/local/`, or system-wide launchd daemons).
- **macOS-centric launchd:** the launchd plist handling is macOS only. A Linux equivalent using systemd user units would require a parallel code path that this version does not provide.
- **No conflict resolution:** if `install.sh` is run on a machine that already has dotfiles managed by a different tool (e.g., chezmoi or GNU Stow), the backup-and-replace strategy may interfere with that tool's state.
- **Secrets workflow is manual:** the installer prints a reminder to restore `~/.aws`, `~/.ssh`, and `~/.gnupg`, but provides no automation. On a new laptop, secret restoration remains a manual step that must be performed correctly.
- **Plugin manifest not yet complete:** the vim plugin manifest (`editors/vimrc` + a `packages/nvim-plugins.lua` manifest) is planned but not yet written. The 22M plugin bundle is excluded from git, which means the first `vim` launch on a new machine will require a manual `:PlugInstall`.

Opportunities for Improvement

1. Add a `bootstrap.sh` that installs Homebrew and git before cloning the dotfiles repo, making the first command on a truly fresh laptop a single `curl | bash` invocation (with appropriate scrutiny of what that script does).
2. Write a systemd user-unit equivalent of the launchd plist section, conditioned on `[[ "$OSTYPE" == "linux-gnu"* ]]`, so the same dotfiles repository works on Ubuntu development VMs.
3. Add GitHub Actions CI that runs `install.sh --dry-run` and `make lint` on macOS and Ubuntu runners after every push, catching cross-platform breakage before a laptop bootstrap.
4. Complete the vim plugin manifest and add a post-install hook to `install.sh` that runs `:PlugInstall` in headless vim, making the editor ready without manual intervention.
5. Evaluate `git-crypt` for encrypting a small set of near-sensitive configs (rclone config, mbsync-crc) so they can live in the repository without requiring manual scrubbing of credentials.

Wrapping Up

The gap between a correct architecture and an executed one is easy to accumulate and expensive to close. Post 24 described the right design for a portable dotfiles repository in February 2026; this post documents the audit that revealed why it had not been executed, identifies the three blockers (no version control, hardcoded absolute paths, and sensitive files without an inventory), and provides the concrete scripts and migration procedure.

The highest-leverage change is also the simplest: `git init` in a directory outside cloud sync, with a `.gitignore` written before the first commit. Everything else, the `install.sh` parameterisation, the launchd `__USER__` substitution, the plugin-manifest strategy, builds on that foundation.

In conclusion, four points merit emphasis. First, cloud sync is not version control; a dotfiles repository outside any cloud-mounted path, with GitHub as the remote, is the correct design. Second, `.gitignore` must be written before `git init`; this is not a best practice but a prerequisite, since one sensitive commit requires a full history rewrite and credential rotation. Third, `install.sh`

`--dry-run` should precede every real install; the cost of reviewing the plan is ten seconds, and the cost of an unexpected overwrite can be an afternoon. Fourth, per-file symlinks in `~/bin` are more verbose than a top-level symlink but are the only pattern that works reliably across machines with different directory layouts.

See Also

Related posts on this site:

- Migrating Off Dropbox: Beyond Dotfiles (post 64): the preceding post in this series; covers migrating project content, shell history, and large binaries off cloud sync into a three-layer framework. This post implements the dotfiles layer.
- Creating a GitHub Dotfiles Repository (post 24, 2026-02-11): the architectural design this post implements.
- Setting Up pass, the Unix Password Manager (post 66): the next step after this post; covers GPG key generation, pass initialisation, and migrating credentials out of cloud sync.
- Multi-Laptop Security: Hardening the Bootstrap (post 67): a security audit of the infrastructure established in posts 64-66; covers FileVault, GPG subkey architecture, per-machine SSH keys, and quarterly review procedures.

Key resources:

- Atlassian Dotfiles Guide: alternative bare-repository approach without `install.sh`
- shellcheck: static analysis for bash scripts; used in `make lint`
- pipx documentation: managing Python CLI tools without virtualenv sprawl
- launchd.info: comprehensive reference for macOS launchd plist format and semantics
- git-filter-repo: the correct tool for removing sensitive data from git history (faster and safer than `git filter-branch`)

Reproducibility

Tested configuration:

Component	Version
Operating system	macOS 15.4
zsh	5.9
git	2.45.x
Homebrew	4.3.x
pipx	1.5.x
shellcheck	0.10.x
Last verified	2026-05-31

Configuration artifacts:

- `analysis/configs/install.sh`: full installer source
- `analysis/configs/Makefile`: build targets
- `analysis/configs/gitignore`: `.gitignore` template
- `analysis/configs/zshenv`: `$DOTFILES` / `$PRJ_ROOT` exports

To reproduce end-to-end:

```
git clone git@github.com:rgt47/dotfiles ~/dotfiles
cd ~/dotfiles
make test       # dry-run + lint
make install    # deploy symlinks and packages
```

Rendered on 2026-05-17 at 09:27 PDT. Source: ~/prj/qblog/posts/65-multi-laptop-bootstrap/multi-laptop-bootstrap/analysis/report/index.qmd

---

Let's Connect

Have questions, suggestions, or spot an error? Let me know.

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

I would enjoy hearing from you if:

- You spot a hardcoded assumption in `install.sh` that breaks on your setup.
- You have extended the launchd section to work with systemd on Linux.
- You use a different secrets mechanism (`pass`, `gopass`, or a hardware token) and want to compare notes.
- You just want to say hello and connect.

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
3. **Post 17: Multi-Laptop macOS Bootstrap** (this post)
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