



zc-workstation

2026-10-16

Table of Contents

Start here	1
What zc-workstation is	1
Getting started	2
Writing a config	5
A new Mac in one command	7
A Linux machine	9
A Windows machine	11
Use it	14
Providers	14
Commands	21
Dashboard	28
Menu bar app and ZC Care	30
Agents and MCP	34
The toolbox	37
Overview	37
Terminal and shell	40
Command line	44
Git and GitHub	53
Runtimes and dotfiles	55
Containers and data	59
Desktop	62
Editors and AI	65
Keeping the Mac well	67
The project	69
Adding a provider	69
Threat model	70
Roadmap	72

Start here

What zc-workstation is

One YAML file says what a workstation needs: the packages, the dotfiles, the language runtimes. `zc-workstation` reads what this machine has, prints the difference, and only if you ask, closes it. Every apply is checked by planning again.

```
$ zc-workstation plan
CREATE brew      cask:ghostty      install with brew
CREATE brew      mas:uBlock Origin Lite install with mas
UPDATE dotfiles  .gitconfig          update; changed outside chezmoi, apply overwrites it
CREATE runtimes  java@temurin-25    install with mise
```

What it manages

PROVIDER	READS	CHANGES WITH
packages	A Brewfile, an apt or dnf package list, or a winget import file	brew, mas, apt-get, dnf, winget
dotfiles	A chezmoi source directory	chezmoi apply, one path at a time
tools	A mise config file	mise install
desktop	A wallpaper and a screen saver page	osascript, WebViewScreenSaver
settings	macOS preferences	defaults
services	Homebrew services and launchd agents	brew services, launchctl
localdir	Files written inline in the config	Direct writes inside one directory

It does not reimplement these tools. chezmoi already handles templates, secrets and per-machine differences, and mise already resolves versions. `zc-workstation` adds one plan across all of them, a dashboard, and the same rules for every surface.

One config, three systems

A target can name the systems it is for. The same file then describes the Mac, the Linux laptop and the Windows desktop:

```
targets:
- {name: brew,   provider: packages, os: [darwin], file: packages/Brewfile}
- {name: apt,    provider: packages, os: [linux],  file: packages/apt.txt}
- {name: winget, provider: packages, os: [windows], file: packages/winget.json}
```

```
- {name: dotfiles, provider: dotfiles, source: home}  
- {name: runtimes, provider: tools, file: mise.toml}
```

A target for another system is skipped, and shown as skipped.

The loop

1. **Plan** asks each provider what is missing and lists actions: CREATE, UPDATE, DELETE and MANUAL. Planning writes nothing.
2. **Apply** runs the actions in order, then plans again. A change that ran but did not take is reported as still outstanding, never as done.
3. **Saved plans** pin what was reviewed. `plan -out` writes the plan, and `apply FILE` refuses when the machine changed since.
4. **MANUAL** is what no tool should do for you, such as installing the package manager itself or trusting a mise config. The plan says what to run.

Five surfaces, one contract

The CLI is the contract. The [dashboard](#), the [menu bar app](#), the [MCP server](#), the agent skills and scripts all go through the same plan and apply, with the same refusals.

On a Mac, the menu bar app also has **ZC Care**, a window that cleans junk and what AI tools leave behind, uninstalls apps with their leftovers and finds what fills the disk.

A new Mac

Keep the config in its own repo, and a new Mac needs one command. See [A new Mac in one command](#).

Getting started

Install the tools it drives

`zc-workstation` runs your package manager, chezmoi and mise. Install the ones your config uses. A missing one shows up in the plan as a **MANUAL** step, so you can also start with `plan` and follow it.

macOS

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"  
$ brew install chezmoi mise mas
```

Ubuntu and Debian

```
$ sudo apt-get install -y curl git
$ sh -c "$(curl -fsLS get.chezmoi.io)" -- -b ~/.local/bin
$ curl https://mise.run | sh
```

Fedora

```
$ sudo dnf install -y chezmoi git
$ curl https://mise.run | sh
```

Windows

```
PS> winget install --id twpayne.chezmoi --exact
PS> winget install --id jdx.mise --exact
```

Install zc-workstation

The Homebrew cask, Linux packages and Windows zips come with each release. Before the first one, install with Go, or run `make install` in a clone.

Homebrew

```
$ brew tap zoolcoder/zc-workstation https://github.com/zoolcoder/zc-workstation
$ brew install --cask zc-workstation
```

Linux packages

Each release has `.deb`, `.rpm` and `.apk` packages.

```
$ sudo apt-get install ./zc-workstation*_linux_amd64.deb
```

Windows

Download the `windows_amd64` or `windows_arm64` zip from the releases page and put `zc-workstation.exe` on your PATH.

Go

```
$ go install github.com/zoolcoder/zc-workstation/cmd/zc-workstation@latest
```

Needs Go 1.26 or later.

Write a config

Start from `zc-workstation.example.yaml`. It declares a target for each system and points at the sample manifests in `example/`. Point it at your own dotfiles repository instead. [Writing a config](#) lists every key.

```
$ mkdir -p ~/.config/zc-workstation
$ cp zc-workstation.example.yaml ~/.config/zc-workstation/zc-workstation.yaml
$ zc-workstation validate
/Users/me/.config/zc-workstation/zc-workstation.yaml: 6 target(s), valid
```

Plan, then apply

```
$ zc-workstation plan -out plan.json
CREATE brew      brew:chezmoi  install with brew
CREATE dotfiles  .gitconfig   add
MANUAL runtimes  trust        mise does not trust .../mise.toml; review it, then run mise trust
.../mise.toml

1 of these cannot be automated and are yours to do.
$ zc-workstation apply plan.json
done CREATE brew brew:chezmoi
done CREATE dotfiles .gitconfig
applied and verified
```

Installs run attached to your terminal, so `sudo`, a license prompt or the App Store sign-in can ask you directly.

Check the machine

```
$ zc-workstation doctor
ok  binary      /opt/homebrew/bin/zc-workstation
ok  system      macbook darwin/arm64
ok  config      /Users/me/.config/zc-workstation/zc-workstation.yaml, 6 target(s)
ok  program brew /opt/homebrew/bin/brew
ok  program chezmoi /opt/homebrew/bin/chezmoi
ok  program mise /opt/homebrew/bin/mise
warn target brew 78 of 80 packages installed (brew)
ok  target apt   not for darwin, only linux
ok  target dotfiles 2 of 2 dotfiles in place
ok  target runtimes 7 of 7 tools installed
```

Then open the [dashboard](#) with `zc-workstation ui`.

Writing a config

The config is found in this order: `-config`, `$ZC_WORKSTATION_CONFIG`, `./zc-workstation.yaml`, then `~/.config/zc-workstation/zc-workstation.yaml`. `-config` reads standard input.

Keep the config in your dotfiles repository next to the manifests it names. Relative paths are relative to the config file, and `~` is your home directory, so the same file works on every machine.

```
version: 1
targets:
  - name: brew
    provider: packages
    os: [darwin]
    file: packages/Brewfile

  - name: dotfiles
    provider: dotfiles
    source: home
    mode: symlink

  - name: runtimes
    provider: tools
    file: mise.toml
```

Every target

KEY	MEANING
version	Always 1. A future format bumps it.
name	Lowercase kebab case, unique. <code>-target</code> selects by it.
provider	packages, dotfiles, tools, desktop, settings, services, keyboard, agents or localdir.
os	The systems the target is for: darwin, linux, windows. Empty means every system.

Per provider

PROVIDER	KEY	MEANING
packages	file	Required. The manifest: a Brewfile, a package list, or a winget import file.
	manager	brew, apt, dnf or winget. Empty picks the system's: brew on macOS, winget on Windows, apt on Linux unless only dnf is installed.

PROVIDER	KEY	MEANING
	upgrade	Also plan an UPDATE for each declared package with a newer version. Off by default.
dotfiles	source	Required. The chezmoi source directory.
	destination	Where the files land. Empty means your home directory.
	mode	file copies (the default); symlink links back into the source.
tools	file	Required. The mise config file whose tools to install.
	upgrade	Also plan an UPDATE for each runtime with a newer version that its request allows. Off by default.
desktop	wallpaper	The picture to show on every desktop.
	screensaver	The web page for WebViewScreenSaver to show. Set this, wallpaper, accent, or any of them. The target needs os: [darwin].
	accent	The macOS accent colour: blue, purple, pink, red, orange, yellow, green or graphite.
settings	defaults	Required on macOS. A macOS preferences domain, such as com.apple.dock or NSGlobalDomain, to its keys and values: a bool, an integer, a number or a string. A string starting with ~/ is a path in your home directory. The target needs os: [darwin].
	gsettings	Required on Linux. A GNOME schema, such as org.gnome.desktop.interface, to its keys and values: a bool, a number, a string or a list of strings. The target needs os: [linux].
	registry	Required on Windows. A registry key under HKCU, such as 'HKCU\Software\Microsoft\Windows\CurrentVersion\Themes\Personalize', to its values: a bool or an integer, stored as a DWORD, or a string. The target needs os: [windows].
services	services	Homebrew services to keep running. Set this, agents, or both. The target needs os: [darwin] or os: [linux].

PROVIDER	KEY	MEANING
	agents	launchd agent plists to keep loaded, the label being the file name without <code>.plist</code> ; on Linux, systemd user <code>.service</code> and <code>.timer</code> files to keep enabled and active; on Windows, Task Scheduler <code>.xml</code> files to keep registered and enabled. The target needs one <code>os</code> .
keyboard	file	Required. The keyd config to install as <code>/etc/keyd/default.conf</code> , such as <code>assets/keyd/default.conf</code> . The target needs <code>os</code> : <code>[linux]</code> .
localdir	path	Required. The directory to keep.
	files	File name to content. Names must stay inside path.
	prune	Also delete files in path the config does not list.

[Providers](#) describes what each one reads and runs.

Mistakes are errors

An unknown key is an error, not a warning: a typo should not silently mean the default. A key a provider ignores, such as `mode` on a `packages` target, is an error too. `validate` reports every problem at once:

```
$ zc-workstation validate
zc-workstation.yaml: target brew: manager "pacman" is not one of brew, apt, dnf, winget
target dotfiles: source is required for provider dotfiles
```

A new Mac in one command

zc-workstation is the tool; your machine's setup lives in a second repo, a *config repo*, which you can keep private. A new Mac then needs one command.

The config repo

A config repo holds a `zc-workstation.yaml` and the files its targets point at. Paths in the config are relative to it, so the repo works wherever it is cloned. A layout that works well:

```
zc-workstation.yaml  targets: packages, home, theme, runtimes, look
Brewfile             every app and command-line tool
mise.toml            runtimes
home/                chezmoi source for ~, with run_onchange_ scripts
apps/, scripts/      data and scripts the run scripts use
```

```

version: 1
targets:
- name: packages
  provider: packages
  os: [darwin]
  file: Brewfile
- name: home
  provider: dotfiles
  source: home
- name: theme
  provider: dotfiles
  source: ../zc-workstation/assets/theme
- name: runtimes
  provider: tools
  file: mise.toml
- name: look
  provider: desktop
  os: [darwin]
  wallpaper: ../zc-workstation/assets/wallpaper/zc-workstation-laptop-dark.png
  screensaver: ../zc-workstation/assets/screensaver/index.html
  accent: blue

```

The packages come first, so the targets after them find chezmoi and mise.

Setup steps that are not files

Some setup is a command, not a file: loading a launch agent, writing macOS defaults, creating a signing key. Put each one in a chezmoi [run script](#) in `home/`, named `run_onchange_after_NN-name.sh.tpl`. chezmoi runs it after the files are written, and again only when its content changes. To rerun it when a data file changes, put the file's hash in a comment:

```

#!/usr/bin/env bash
# macos.sh {{ include "../scripts/macos.sh" | sha256sum }}
"{{ .chezmoi.sourceDir | dir }}/scripts/macos.sh"

```

`{{ .chezmoi.sourceDir | dir }}` is the config repo itself. The plan shows each pending script as an UPDATE with chezmoi's R status. Scripts run attached to your terminal, so they can ask a question, such as the git identity.

The runtimes config

mise reads its global config from `~/.config/mise/config.toml`. Link it to the repo's `mise.toml` with a chezmoi symlink template, `home/dot_config/mise/symlink_config.toml.tpl`, holding `{{ .chezmoi.sourceDir | dir }}/mise.toml`. Then a tool you add with `mise use -g` lands in the repo.

bootstrap.sh

On a new Mac, in Terminal.app:

```
$ curl -fsSL https://raw.githubusercontent.com/ZoolCoder/zc-workstation/main/bootstrap.sh | bash -s -- OWNER/CONFIG-REPO
```

It:

1. installs Homebrew, which also installs the Xcode command line tools
2. installs git, gh, go, chezmoi and mise
3. logs you in to GitHub, so git can clone a private config repo
4. clones zc-workstation and the config repo into `~/Developer/Tools`, keeping either if it is already there
5. builds zc-workstation with `make install` into `~/local/bin`
6. trusts the config's `mise.toml`
7. shows the plan, applies it, and applies again if anything is left, such as runtimes planned before mise arrived

It is safe to run again. `ZC_TOOL_FROM` and `ZC_CONFIG_FROM` clone from somewhere else, such as a local copy in a VM test.

Keeping Macs in step

Change the config repo, `zc-workstation plan`, `zc-workstation apply`, commit and push. On another Mac, pull and apply. A shell function that pulls both repos, rebuilds zc-workstation when it changed, and applies keeps this to one word.

A Linux machine

zc-workstation runs the same way on Linux as on macOS: one config, a plan, an apply. The packages come from apt or dnf, background jobs are systemd user units, and GNOME settings are `gsettings` keys. CI applies a config in fresh Ubuntu 24.04 and Fedora 42 containers before each release, then checks that nothing is left and `doctor` passes.

Before the first plan

Install what zc-workstation drives. On a fresh install with no package lists yet, apply runs `sudo apt-get update` before the first apt install.

Ubuntu and Debian

```
$ sudo apt-get update
```

```
$ sudo apt-get install -y curl git
$ sh -c "$(curl -fsLS get.chezmoi.io)" -- -b ~/.local/bin
$ curl https://mise.run | sh
```

Fedora

```
$ sudo dnf install -y chezmoi git
$ curl https://mise.run | sh
```

Install zc-workstation from the release's `.deb` or `.rpm`, or with `go install`, as [Getting started](#) shows.

A config for Linux

Give each Linux-only target `os: [linux]`, so the same config also drives your Mac. Targets without `os`, such as dotfiles and runtimes, run everywhere.

```
version: 1
targets:
- name: apt
  provider: packages
  os: [linux]
  manager: apt          # or dnf on Fedora
  file: apt.txt
- name: home
  provider: dotfiles
  source: home
- name: runtimes
  provider: tools
  file: mise.toml
- name: jobs
  provider: services
  os: [linux]
  agents: [~/.config/systemd/user/backup.timer]
- name: gnome
  provider: settings
  os: [linux]
  gsettings:
    org.gnome.desktop.interface:
      color-scheme: prefer-dark
- name: keys
  provider: keyboard
  os: [linux]
  file: keyd.conf
```

`apt.txt` and `dnf.txt` hold one package per line, as the manager names them. `zc-workstation capture -target apt` writes the file from what the machine has.

- **Packages** install with `sudo apt-get install -y` or `sudo dnf install -y`. They run attached to your terminal, so `sudo` asks for your password.

- **Services** enables each unit with `systemctl --user enable --now` after a `daemon-reload`. Write the unit files with the `dotfiles` target, which comes first.
- **Settings** writes each key with `gsettings set`. GNOME applies it at once. A light theme is `color-scheme: prefer-light` with `gtk-theme: Adwaita`.
- **Keyboard** installs a keyd config with `sudo` and enables `keyd`. Start from `assets/keyd/default.conf`: Caps Lock is Escape on tap, Hyper on hold.

[Providers](#) has every key for each provider.

Plan, apply, check

```
$ zc-workstation plan -out plan.json
$ zc-workstation apply plan.json
$ zc-workstation doctor
```

`doctor` lists each program a target needs, so a missing `dnf` on an Ubuntu machine shows as a failing check: keep one packages target per manager, or give each its own config.

Not on Linux yet

The desktop provider (wallpaper, screen saver, accent colour) and the defaults keys of settings are macOS only. ZCBar and ZC Care are macOS apps. The [roadmap](#) tracks what comes next.

A Windows machine

On Windows, `zc-workstation` installs packages with `winget`, applies dotfiles with `chezmoi`, installs runtimes with `mise`, keeps registry settings and registers scheduled tasks.

Before the first plan

`winget` ships with Windows 11 and current Windows 10. Install `chezmoi` and `mise` with it, then download the `windows_amd64` or `windows_arm64` zip from the releases page and put `zc-workstation.exe` on your `PATH`.

```
PS> winget install --id twpayne.chezmoi --exact
PS> winget install --id jdx.mise --exact
```

A config for Windows

The packages target reads a `winget export` file. Give it os: `[windows]`, so the same config also drives your Mac and Linux machines.

```
version: 1
targets:
  - name: winget
    provider: packages
    os: [windows]
    file: winget.json
  - name: home
    provider: dotfiles
    source: home
  - name: runtimes
    provider: tools
    file: mise.toml
```

Write `winget.json` from what the machine has with `winget export -o winget.json`, or start from `example/winget.json`: Git, chezmoi, mise, WezTerm, PowerShell 7, starship, atuin and the JetBrainsMono Nerd Font. chezmoi names files the same way on every system; use its templates for what differs, such as `{{ if eq .chezmoi.os "windows" }}`.

Settings and scheduled tasks

A settings target keeps registry values under HKCU, and a services target keeps Task Scheduler tasks registered. Each needs `os: [windows]`.

```
- name: windows
  provider: settings
  os: [windows]
  registry:
    'HKCU\Software\Microsoft\Windows\CurrentVersion\Themes\Personalize':
      AppsUseLightTheme: false
      SystemUsesLightTheme: false
- name: tasks
  provider: services
  os: [windows]
  agents: [tasks/weekly-update.xml]
```

Write a task file with Task Scheduler's **Export**, or with `Export-ScheduledTask -TaskName weekly-update > tasks/weekly-update.xml`. [settings](#) and [services](#) have the details.

PowerShell and WezTerm

The ZoolCoder theme, a dotfiles target with `source: ../zc-workstation/assets/theme`, writes the same look as on the Mac:

- `~/.config/wezterm/wezterm.lua`: the colours, font and glass, with PowerShell 7 as the shell.
- `~/.config/starship.toml`, the prompt, which starship reads on Windows too.
- `~/.config/zoolcoder/theme.ps1`, the colours of PowerShell's line editor.

example/home has a PowerShell 7 profile, Documents/PowerShell/Microsoft.PowerShell_profile.ps1, that starts mise, starship and atuin and loads the colours. chezmoi writes it only on Windows. If OneDrive moves your Documents folder, \$PROFILE points into OneDrive: move the file in the chezmoi source to OneDrive/Documents/PowerShell.

Plan, apply, check

```
PS> zc-workstation plan -out plan.json
PS> zc-workstation apply plan.json
PS> zc-workstation doctor
```

Not on Windows yet

The desktop and keyboard providers are macOS and Linux only. ZCBar and ZC Care are macOS apps.

Use it

Providers

A provider reads the machine and returns actions. It never writes during a plan. Each one also counts what is in place, for `status`, `doctor` and the dashboard.

packages

Installs what a manifest declares. **It never uninstalls:** a package you drop from the manifest stays installed, because removing software is your call.

MANAGER	MANIFEST	READS INSTALLED	INSTALLS WITH
brew	A Brewfile: <code>tap</code> , <code>brew</code> , <code>cask</code> and <code>mas</code> entries	<code>brew list --formula</code> , <code>brew list --cask</code> , <code>brew tap</code> , <code>mas list</code>	<code>brew tap</code> , <code>brew install</code> , <code>brew install --cask</code> <code>--adopt</code> , <code>sudo mas install ID</code>
apt	One package per line, <code>#</code> comments	<code>dpkg-query</code> , packages in state <code>ii</code>	<code>sudo apt-get install -y</code>
dnf	One package per line, <code>#</code> comments	<code>rpm -qa</code>	<code>sudo dnf install -y</code>
winget	The file <code>winget export</code> writes	<code>winget list --id ID</code> <code>--exact</code> , per package	<code>winget install --id ID</code> <code>--exact --silent</code>

On a machine with no apt package lists yet, such as a fresh install or a container, apply runs `sudo apt-get update` before the first install.

For `list` and the Inventory page, versions and descriptions come from `brew info --json=v2 --installed` and `mas list` on macOS, `dpkg-query` on apt systems, and `rpm -qa` on dnf systems. `winget` items list without them. If the details cannot be read, items still list, without versions.

Newer versions and undeclared packages come from:

MANAGER	NEWER VERSIONS	INSTALLED ON PURPOSE	UPGRADES WITH
brew	<code>brew outdated --json=v2</code> , skipping pinned formulae; <code>mas outdated</code>	<code>brew leaves --installed --on-request</code> , <code>brew list --cask</code> , <code>brew tap</code> , <code>mas list</code>	<code>brew upgrade</code> , <code>brew upgrade --cask</code> , <code>mas upgrade ID</code>
apt	<code>apt list --upgradable</code>	<code>apt-mark showmanual</code>	<code>sudo apt-get install --only-upgrade -y</code>
dnf	<code>dnf list --upgrades</code>	<code>dnf repoquery --userinstalled</code>	<code>sudo dnf upgrade -y</code>

winget has neither yet. These reads are best effort: if one fails, items still list without it.

- With `upgrade: true`, the plan has one `UPDATE` per outdated declared package, such as `upgrade with brew, 3.7c to 3.8`. Without it, nothing is upgraded. A manager that cannot report newer versions gets one `MANUAL` step instead.
- `capture` writes a Brewfile (`tap`, `brew`, `cask` and `mas` lines) or a package list from the "installed on purpose" reads. Dependencies are left out, since the manager pulls them in again.
- Taps are installed before the formulae that need them.
- Other Brewfile entries, such as `vscode` or `go`, are `MANUAL`: the plan names them, and you install them yourself.
- `mas` entries are `MANUAL` until `mas` is installed.
- If the manager itself is missing, the plan is one `MANUAL` step to install it.
- An app installed outside the manager is not seen. Its cask is installed with `--adopt`, so Homebrew takes over the app already there; if the app differs from the cask, Homebrew refuses, and you delete the app or remove the entry.

dotfiles

Applies a `chezmoi` source directory. Use chezmoi's naming (`dot_zshrc`, `private_`, `executable_`, `.tmpl`) and its templates for per-machine differences. Its secrets support keeps passwords out of the repository.

The plan is `chezmoi status`:

CHEZMOI	PLAN	MEANING
A	CREATE	The file is not there yet.
M	UPDATE	The file differs from the source.
D	DELETE	The source removes it, with chezmoi's <code>remove_</code> prefix.
R	UPDATE	A <code>run_</code> script will run.

In the inventory each file names the app it configures: the folder under `.config`, `.local/share`, `Library/Application Support` or `AppData`, else the file's own name (`.zshrc` is `zsh`, `.gitconfig` is `git`). A login item under `Library/LaunchAgents` is `launchd`.

A file edited since chezmoi last wrote it says so in the plan: `changed outside chezmoi`, `apply` overwrites it. Copy your edit into the source first if you want to keep it.

Apply runs `chezmoi apply --force` for exactly the planned paths and their parent directories, nothing else. `--force` stands in for chezmoi's own overwrite prompt, because you already reviewed the plan.

zc-workstation ships one source of its own, `assets/theme`: the ZoolCoder colours for Ghostty, tmux,

starship, SketchyBar, JankyBorders, bat, btop, eza, yazi, lazygit and fastfetch, each with a dark and a light variant. Its shell settings, such as `BAT_THEME_DARK` and `FZF_DEFAULT_OPTS`, sit in `~/.config/zoolcoder/theme.zsh`. The source creates `~/.zshrc.local` to load it, once: an existing `~/.zshrc.local` is left alone and needs the line itself, and an existing `.zshrc` needs one line, `source ~/.zshrc.local`. After apply, reload what is running: `sketchybar --reload`, `~/.config/borders/bordersrc`, and `tmux source-file ~/.config/tmux/tmux.conf`.

With `mode: symlink`, each file is a link into the source directory, so an edit in place lands in the repository. A file `chezmoi` must render, such as a template or a private file, is still copied.

tools

Installs the runtimes a `mise` config pins. Only the tools declared in that file count. Tools from your global `mise` config or a parent directory's config are ignored.

- The plan lists one `CREATE` per `tool@version` that is not installed, such as `java@temurin-25`.
- Apply runs `mise install` in the config's directory.
- In the inventory, each runtime shows the version `mise` resolved next to the one requested, such as `24.21.0` for `lts`, and the newer version `mise outdated` reports. Versions installed but asked for by no config in this target, such as a global tool or an old node, show as `not declared`.
- With `upgrade: true`, the plan has one `UPDATE` per outdated runtime. Apply runs `mise upgrade` for that `tool@request` in the config's directory, so a pin such as `3.13` moves only within `3.13`.
- `capture` writes a `[tools]` table of every installed version, using the request when `mise` knows it.
- `mise` refuses to read a config it does not trust. The plan then shows a `MANUAL` step: read the file, then run `mise trust`.
- `doctor` checks that the shell runs each command from `mise`, not from another copy earlier on `PATH`, such as Homebrew's `node`. The fix is to activate `mise` last in the shell profile.

desktop

Sets how a Mac looks: the desktop picture, the page the `WebViewScreenSaver` screen saver shows, and the accent colour. macOS only, so the target needs `os: [darwin]`.

- **Wallpaper.** macOS keeps one picture for every Space on every display, in its wallpaper store, `~/Library/Application Support/com.apple.wallpaper/Store/Index.plist`. System Events sees only the Spaces in view, so the plan reads the store itself, with a small JavaScript for Automation script run by `osascript`. If any Space shows another picture, the plan has one `UPDATE`. Apply sets the picture through System Events, rewrites every Space in the store, and restarts `WallpaperAgent` so the desktops redraw. The first run asks you to let your terminal control System Events. Another tool that also sets the wallpaper, such as a login agent, will undo this; turn it off.
- **Accent.** `accent` is one of blue, purple, pink, red, orange, yellow, green or graphite. The plan reads `AppleAccentColor` with defaults; apply writes it and the matching `AppleHighlightColor`. Apps

show the new colour once they restart; log out to see it everywhere.

- **Screen saver.** The page, with the fonts and images its CSS loads from its folder inlined, becomes one `data: URL`. The plan compares it with the saver's default page, `WVSSDefaultAddressURL` in the saver's `Info.plist`, read with `plutil`. If they differ, apply writes it there with `plutil` and signs the saver again, ad hoc, with `codesign`. The saver shows its default page while its own list of pages, set in its Options, is empty.
- If `WebViewScreenSaver` is not installed, in `~/Library/Screen Savers` or `/Library/Screen Savers`, the plan is a **MANUAL** step: run `assets/screensaver/build-saver.sh`. It builds the saver from source with Xcode and installs it for you. The Homebrew cask is disabled, because the release build fails the Gatekeeper check; a local build is not quarantined.
- Choosing `WebViewScreenSaver` as the screen saver is yours to do, once, in System Settings, Screen Saver. macOS keeps that choice where no tool should write it.
- A file that does not exist, such as a wallpaper not rendered yet, is a **MANUAL** step.

The saver runs sandboxed: it cannot read files outside its bundle, and macOS keeps its preferences in a container no terminal may open. That is why the page travels inside the saver itself. Query options such as `?light` do not reach a `data: URL`, so the saver shows the page's default, dark look.

settings

Keeps preferences at the values the config declares: on macOS the keys `defaults` reads and writes, on GNOME the keys of `gsettings`, and on Windows the user's registry values. A target drives one system: `os: [darwin]` with `defaults`, `os: [linux]` with `gsettings` or `os: [windows]` with `registry`.

```
- name: macos
  provider: settings
  os: [darwin]
  defaults:
    NSGlobalDomain:
      KeyRepeat: 2
      NSWindowResizeTime: 0.001
    com.apple.dock:
      autohide: true
    com.apple.screencapture:
      location: ~/Screenshots
```

- The plan reads each key with `defaults read`. A key that is not set is a **CREATE**; one with another value is an **UPDATE**. Apply writes it with the type of the YAML value: `-bool`, `-int`, `-float` or `-string`.
- The Dock, Finder and the screenshot settings are read only when their process starts, so a change to `com.apple.dock`, `com.apple.finder` or `com.apple.screencapture` adds an action that restarts it.
- A key the config stops declaring keeps its value. Some keys need a logout to take effect.

On GNOME, `gsettings` maps a schema to its keys. A dark desktop with the dark Adwaita theme, a

faster key repeat and the dock's apps:

```
- name: gnome
  provider: settings
  os: [linux]
  gsettings:
    org.gnome.desktop.interface:
      color-scheme: prefer-dark # prefer-light or default for a light one
      gtk-theme: Adwaita-dark # Adwaita for a light one
    org.gnome.desktop.peripherals.keyboard:
      delay: 250
      repeat-interval: 20
    org.gnome.shell:
      favorite-apps: [org.gnome.Nautilus.desktop, com.mitchellh.ghostty.desktop]
```

- The plan reads each key with `gsettings get` and compares values whatever the type `gsettings` prints, so `uint32 250` matches `250`.
- Apply runs `gsettings set` with the value as `GVariant` text: a string is quoted, a list becomes an array of strings. GNOME applies a change at once; nothing restarts.
- A schema or key GNOME does not know is a `CREATE` whose apply fails with `gsettings'` message.

On Windows, `registry` maps a key under `HKCU` to its values. Dark mode, file extensions shown and the taskbar on the left:

```
- name: windows
  provider: settings
  os: [windows]
  registry:
    'HKCU\Software\Microsoft\Windows\CurrentVersion\Themes\Personalize':
      AppsUseLightTheme: false
      SystemUsesLightTheme: false
    'HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced':
      HideFileExt: 0
      TaskbarAl: 0
```

- Only keys under `HKCU` (or `HKEY_CURRENT_USER`): the user's own settings, which need no administrator. Quote a key in single quotes, so YAML keeps its backslashes.
- The plan reads each value with `reg query`. A bool or an integer is a `REG_DWORD`, so `false` matches `0x0`; a string is a `REG_SZ`. A string starting with `~/` is a path in your home folder.
- Apply runs `reg add` with the type. Windows applies most values at once; Explorer reads some only when it starts, so sign out to see them.

services

Keeps background services running: Homebrew services, launchd agents on macOS, systemd user units and timers on Linux, and Task Scheduler tasks on Windows. A target drives one system, `os`:

[darwin], os: [linux] or os: [windows]. It starts what is stopped and never stops anything.

```
- name: services
  provider: services
  os: [darwin]
  services: [sketchybar, borders, colima]
  agents: [~/Library/LaunchAgents/local.weekly-update.plist]
```

- A service is in place when `brew services list` shows it started, or `scheduled` for one that runs on a timer. Otherwise apply runs `brew services restart`, which starts it at every login and recovers one that failed.
- An agent is in place when `launchctl print gui/<uid>/<label>` finds it; the label is the file name without `.plist`. Otherwise apply loads it with `launchctl bootstrap`. The plist can come from a `dotfiles` target listed earlier in the config, which writes it first.
- A changed plist is not reloaded. Reload it from a `chezmoi run_onchange_` script that hashes the plist.

On Linux, agents names `.service` and `.timer` files:

```
- name: services-linux
  provider: services
  os: [linux]
  agents: [~/config/systemd/user/backup.timer, ~/units/syncthing.service]
```

- A unit is in place when `systemctl --user show` reports it enabled and active; a timer is active while it waits for its next run.
- Otherwise apply runs `systemctl --user daemon-reload`, then `systemctl --user enable --now` with the file's path, which links in a unit that lives outside `systemd`'s folders.
- `services` works on Linux too: Homebrew on Linux runs `brew services` on `systemd`.

On Windows, agents names Task Scheduler task files, the `.xml` that Task Scheduler's **Export** or **Export-ScheduledTask** writes:

```
- name: tasks
  provider: services
  os: [windows]
  agents: [tasks/weekly-update.xml]
```

- The task's name is the file name without `.xml`. It is in place when `Get-ScheduledTask` finds it and it is not disabled; PowerShell reports the state in English whatever the system language.
- A missing task is registered with `Register-ScheduledTask` from the file, in whatever encoding it has. A disabled one is enabled.
- A changed file is not registered again. Re-register it by hand, or from a `chezmoi run_onchange_` script that hashes the file.

keyboard

Remaps keys on Linux with `keyd`, which works under X11, Wayland and the console alike. The target installs a keyd config as `/etc/keyd/default.conf` and keeps the keyd service enabled and running. Linux only, so the target needs `os: [linux]`.

```
- name: keys
  provider: keyboard
  os: [linux]
  file: assets/keyd/default.conf
```

`assets/keyd/default.conf` makes Caps Lock Escape when tapped and Hyper (Control, Alt, Shift and Super together) when held, as Karabiner does on the Mac. Point `file` at your own config for other remaps.

- `keyd` itself comes from your packages: `keyd` on Debian 13 and Ubuntu 25.04 or later, which name the program `keyd.rvaiya`, or the `keyd` COPR on Fedora. Until it is installed the plan shows a `MANUAL` step.
- The plan compares the file byte for byte. Apply copies it with `sudo install -m 644`, then runs `keyd reload` when `keyd` is running.
- `keyd` is in place when `systemctl show keyd` reports it enabled and active; otherwise apply runs `sudo systemctl enable --now keyd`.

agents

Declares, once, what every AI coding agent gets: MCP servers, a rules file, skills, and the models Ollama serves. It writes them into Claude Code, Codex, Cursor and Gemini CLI. It adds and updates, and never removes a server, skill or model the file does not name.

```
- name: ai
  provider: agents
  file: agents.yaml
```

`agents.yaml`:

```
clients: [claude, codex, cursor, gemini] # leave out for all four
servers:
  context7:
    command: npx
    args: [-y, "@upstash/context7-mcp@2.1.0"] # pin a version; doctor warns otherwise
  github:
    url: https://api.githubcopilot.com/mcp/
    headers: {Authorization: "Bearer ${GITHUB_TOKEN}"}
rules: AGENTS.md # relative to agents.yaml
skills: [skills/review] # folders holding a SKILL.md
models: [qwen3:8b, nomic-embed-text]
```

CLIENT	MCP SERVERS	RULES	SKILLS
claude	<code>~/.claude.json</code>	<code>~/.claude/CLAUDE.md</code>	<code>~/.claude/skills</code>
codex	<code>~/.codex/config.toml</code>	<code>~/.codex/AGENTS.md</code>	<code>~/.codex/skills</code>
cursor	<code>~/.cursor/mcp.json</code>	kept in Cursor's settings	none
gemini	<code>~/.gemini/settings.json</code>	<code>~/.gemini/GEMINI.md</code>	none

- A server is set by name in `mcpServers`; every other key in the file stays as it was. Each file is replaced in one rename, read again just before the write, so an agent writing it in between keeps its change.
- Codex reads TOML. Its servers live in a marked block of `config.toml`. A server Codex already declares outside the block is a `MANUAL` step, since a table declared twice breaks the file. An `env` value written `${NAME}` becomes an `env_vars` entry, which Codex passes through from your shell.
- The rules file lands in a marked block, so what you wrote around it stays.
- A skill is in place when every file in its folder matches.
- A model is in place when `ollama list` shows it; otherwise apply runs `ollama pull`. `ollama serve` must be running. `list` shows each model.
- Keep keys out of the file: write `${NAME}` and keep the value in the keychain or a password manager. `doctor` warns about plain-text keys.

localdir

Keeps a directory holding the files written inline in the config. Writes go through `os.Root`, so a symlink or `..` cannot lead a write outside the directory. With `prune`, files the config does not list are deleted, after you confirm.

It is useful for a handful of small files, and it is the worked example for [writing your own provider](#).

Commands

COMMAND	WHAT IT DOES
<code>plan</code>	Read live state and list what would change. The default.
<code>apply [FILE]</code>	Make the changes, then plan again to verify. With a saved plan, exactly those.
<code>status</code>	What is in place per target, without diffing. Targets for another system show as skipped.

COMMAND	WHAT IT DOES
list	Every package, app, dotfile and runtime the config declares, one line each: its state, installed version, the app it configures, and why it is there. Then what is installed but declared nowhere.
add ITEM	Declare a package or runtime in one target's manifest, with why it is there. Installs nothing; plan and apply do.
capture	Print the manifest one target would declare to match this machine. Needs -target.
validate	Check the config offline.
doctor	Machine health: binary, system, config, the programs targets need, and each target. It also warns about weak security settings, keys without a passphrase, unsigned commits, secrets in the shell history and risky AI agent configs, each with its fix. Every check .
upgrade	Save what is outdated to the state folder, then apply -upgrade. The review skill reads the snapshot to say what moved from what to what.
history	What was applied, newest first: when, which target, from the terminal, the dashboard or an agent, and whether it worked. Drift changes show here too.
drift	Plan, and save how many changes are outstanding to ~/.local/state/zc-workstation/drift.json, which ZCBar and SketchyBar show. -last prints the saved one; -plist prints a launchd agent that runs it every morning.
ui	The dashboard .
mcp	The MCP server on standard input and output.
version	The version.

Flags

-config FILE	Config file, or - for standard input.
-target NAME	Limit to this target; repeat for several.
-out FILE	plan: save the plan for apply FILE. capture: write the manifest to FILE instead of standard output.
-force	capture -out: overwrite FILE if it exists.
-state LIST	list: only these states, comma-separated, such as outdated, not declared.

<code>-kind LIST</code>	<code>list</code> : only these kinds, comma-separated, such as <code>cask,mas</code> .
<code>-adoc</code>	<code>list</code> : the declared packages and runtimes as an AsciiDoc page, one table per section.
<code>-why TEXT</code>	<code>add</code> : one line on why the item is there.
<code>-section NAME</code>	<code>add</code> : the manifest section to add it under; a new name starts a new section.
<code>-json</code>	<code>plan, status, list</code> and <code>doctor</code> : machine-readable.
<code>-exit-code</code>	<code>plan</code> : exit 2 for changes, 3 for manual-only.
<code>-upgrade</code>	<code>plan, apply</code> : also upgrade outdated packages and runtimes, as if every <code>packages</code> and <code>tools</code> target said <code>upgrade: true</code> .
<code>-only RESOURCE</code>	<code>plan, apply</code> : only the change to this resource, such as <code>brew:tmux</code> ; repeat for several. A saved plan remembers <code>-upgrade</code> and <code>-only</code> .
<code>-yes</code>	<code>apply</code> : skip the confirmation prompt for deletions.

Inventory

```
$ zc-workstation list
```

TARGET	KIND	NAME	STATE	VERSION	APP	ABOUT
brew	brew	starship	in place	1.26.0	-	Cross-shell prompt for astronauts
brew	brew	tmux	outdated	3.7c -> 3.8	-	Terminal multiplexer
brew	cask	ghostty	in place	1.3.1	-	Ghostty Terminal emulator that uses platform UI and GPU
brew	cask	google-chrome	missing	-	-	-
brew	brew	jq	not declared	1.8.1	-	Lightweight and flexible command-line JSON processor
dotfiles	dotfile	.config/ghostty/config	in place	-	-	ghostty
runtimes	runtime	node	in place	24.21.0	-	requested lts

States:

- `in place`: installed, and nothing newer is known.
- `outdated`: installed, and the manager has a newer version. A pinned formula never shows as outdated.
- `missing`: declared, not installed.
- `differs`: a dotfile that does not match its source, or a script still to run.
- `manual`: an entry `zc-workstation` does not install.
- `not declared`: installed on purpose, but no target declares it. For brew these are formulae installed on request, casks, taps and App Store apps; a dependency never shows.

ABOUT is the item's why, from its manifest line, else the package manager's description; JSON has both, as `why` and `desc`, with the `section`. Versions and descriptions come from the package manager; see [Providers](#). A target for another system is listed as skipped, one that cannot be read with its error. The reads run side by side, so a full list takes about two seconds on a Mac with a few hundred packages.

```
$ zc-workstation list -state outdated
$ zc-workstation list -state "not declared" -kind cask,mass
```

Why each item is there

A manifest says what is installed; its comments say why. A comment at the end of an entry's line is that entry's why. A comment line that opens a block, after a blank line or at the top of the file, is the heading of the section below it.

```
# Git
brew "git-delta" # Side-by-side, syntax-highlighted git diffs
brew "lazygit" # Git in a terminal UI, opened from tmux with prefix g
```

The same works in apt and dnf package lists and in a mise config's `[tools]` table: `go = "1.26" # Builds zc-workstation.`

`list`, the dashboard and the MCP `list` tool show the why. `list -adoc` turns it into a page that names every tool, links its homepage, and says why you have it; keep that page in your config repository and generate it again after a change:

```
$ zc-workstation list -adoc > docs/software.adoc
```

Add

`add` writes one entry, with its why, into a target's manifest. It installs nothing: `plan` and `apply` do, so you review the change like any other edit.

```
$ zc-workstation add -target packages -section Git -why "Fixup commits that find their target" brew:git-absorb
added brew:git-absorb to /Users/me/config/Brewfile; review and install it with: zc-workstation plan -target packages
$ zc-workstation add -target runtimes -why "Builds zc-workstation" runtime:go@1.26
```

- The item is `kind:name`: `brew`, `cask` or `tap` for a Brewfile, the manager's name for apt and dnf, `runtime` for mise. A bare name is a formula, a package or a runtime.
- A formula or cask from a third-party tap, such as `brew:ciroslab/cli/tart`, brings its tap line too.

- A runtime is `name@version`; without a version it asks for `latest`.
- In a manifest with sections, `-section` is required, and the error lists them. An item the manifest already declares is refused.
- App Store apps need their id: write the `mas` line yourself. `winget` files are not edited.

Remove

`remove` uninstalls one item now and drops its line from the manifest, so the next plan does not bring it back. `Apply` only ever adds; `remove` is how something leaves.

```
$ zc-workstation remove -target packages brew:htop
removed brew:htop and dropped it from /Users/me/config/Brewfile
$ zc-workstation remove -target packages mas:409201541
```

- The item is `kind:name` as `list` shows it. An item the manifest does not declare is still uninstalled, so leftovers go through the tool too.
- An App Store app that is not declared is named by its id.

Capture

`capture` writes the manifest that would declare what the machine has: a Brewfile for brew, one package per line for apt and dnf, and a `[tools]` table for mise. Use it to start a config from a machine you already set up, or to see what a manifest misses.

```
$ zc-workstation capture -target brew > Brewfile.new
$ zc-workstation capture -target brew -out Brewfile.new
captured brew to Brewfile.new
```

It writes standard output unless you pass `-out`, and it refuses to replace a file without `-force`. Diff the result against your manifest and copy what you want; a capture lists everything installed on request, including what you only tried once. `winget` has its own: `winget export -o FILE`.

Installs ask in the terminal

`Apply` runs installers attached to your terminal, so `sudo`, a license prompt or an App Store sign-in can ask you. Their output goes to standard error, so `-json` output stays parseable.

With no terminal, as under an AI agent or `launchd`, `sudo` asks for your password in a macOS dialog instead, through `SUDO_ASKPASS`; Touch ID still comes first when `sudo` is set up for it. A `SUDO_ASKPASS` you set yourself is left alone.

Deletions ask first

A plan with a DELETE stops and asks you to type yes. Without a terminal, and without `-yes`, apply is cancelled. Agents never get `-yes`.

Drift watch

`drift -plist` prints a launchd agent that runs `drift` at 9:00 (`-hour` moves it). Keep it in your dotfiles and loaded by a `services` target, like any other agent:

```
$ zc-workstation drift -plist > Library/LaunchAgents/dev.zoolcoder.zc-workstation.drift.plist
```

The agent carries the PATH of the shell that printed it, since launchd starts agents with almost none. It only plans; nothing is applied. A change in the count is written to `history`, and SketchyBar's `drift` item shows the count while it is above zero.

Exit codes

0	Nothing to do, or the apply converged and was verified.
1	Something failed.
2	With <code>-exit-code</code> : changes are outstanding.
3	With <code>-exit-code</code> : nothing left but <code>MANUAL</code> items.

JSON

```
$ zc-workstation plan -json
{
  "changes": 1,
  "manual": 1,
  "actions": [
    {
      "op": "CREATE",
      "target": "brew",
      "resource": "cask:ghostty",
      "detail": "install with brew",
      "manual": false
    },
    {
      "op": "MANUAL",
      "target": "runtimes",
      "resource": "trust",
      "detail": "mise does not trust /Users/me/dotfiles/mise.toml; review it, then run mise trust /Users/me/dotfiles/mise.toml",
      "manual": true
    }
  ]
}
```

```
]
}
```

Reference

The usage text, as `zc-workstation help` prints it:

zc-workstation sets up this machine from a YAML file: packages, dotfiles (chezmoi) and runtimes (mise), on macOS, Linux and Windows.

Usage:

```
zc-workstation plan      [flags]      show what would change (default)
zc-workstation apply    [flags] [file]  make the changes; with a saved plan, exactly those
zc-workstation status   [flags]      what is in place right now, per target
zc-workstation list     [flags]      every package, dotfile and runtime, with versions
zc-workstation capture  -target t  write t's manifest from what this machine has
zc-workstation validate [flags]      check the config offline
zc-workstation doctor   [-json]      health: system, config, programs, targets
zc-workstation upgrade  [flags]      save what is outdated, then upgrade it
zc-workstation history  [-n 20] [-json] what was applied, newest first
zc-workstation drift    [-last] [-json] count what differs from the config, and save it;
                                -plist prints a launchd agent that runs it daily
zc-workstation add      -target t -why w [-section s] kind:name
                                declare an item in t's manifest; plan and apply install it
zc-workstation mcp      stdio MCP server for agents
zc-workstation remove   -target t kind:name
                                uninstall an item and drop it from t's manifest; mas:ID for an undeclared
app
zc-workstation ui       [-addr a]      the admin page, on loopback only
zc-workstation version
```

Exit codes:

```
0 nothing to do, or the apply converged and was verified
1 something failed
2 with -exit-code: changes are outstanding
3 with -exit-code: nothing left but MANUAL items
```

Flags:

```
-config string  config file, or "-" for standard input; default: $ZC_WORKSTATION_CONFIG,
                then ./zc-workstation.yaml, then ~/.config/zc-workstation/zc-workstation.yaml
-target value   limit to this target; repeat for several
-yes           skip the confirmation prompt for deletions
-exit-code     plan: exit 2 for changes, 3 for manual-only
-upgrade       plan and apply: also upgrade outdated packages and runtimes
-only value    plan and apply: only the change to this resource, such as
                brew:tmux; repeat for several
-json          plan, status and list: machine-readable
-out string    plan: save the plan, to apply exactly what you reviewed;
                capture: write the manifest to this file, not stdout
-force        capture: overwrite the -out file
-state list    list: only these states, comma-separated (missing,outdated,...)
-kind list     list: only these kinds, comma-separated (brew,cask,dotfile,...)
```

`-adoc`

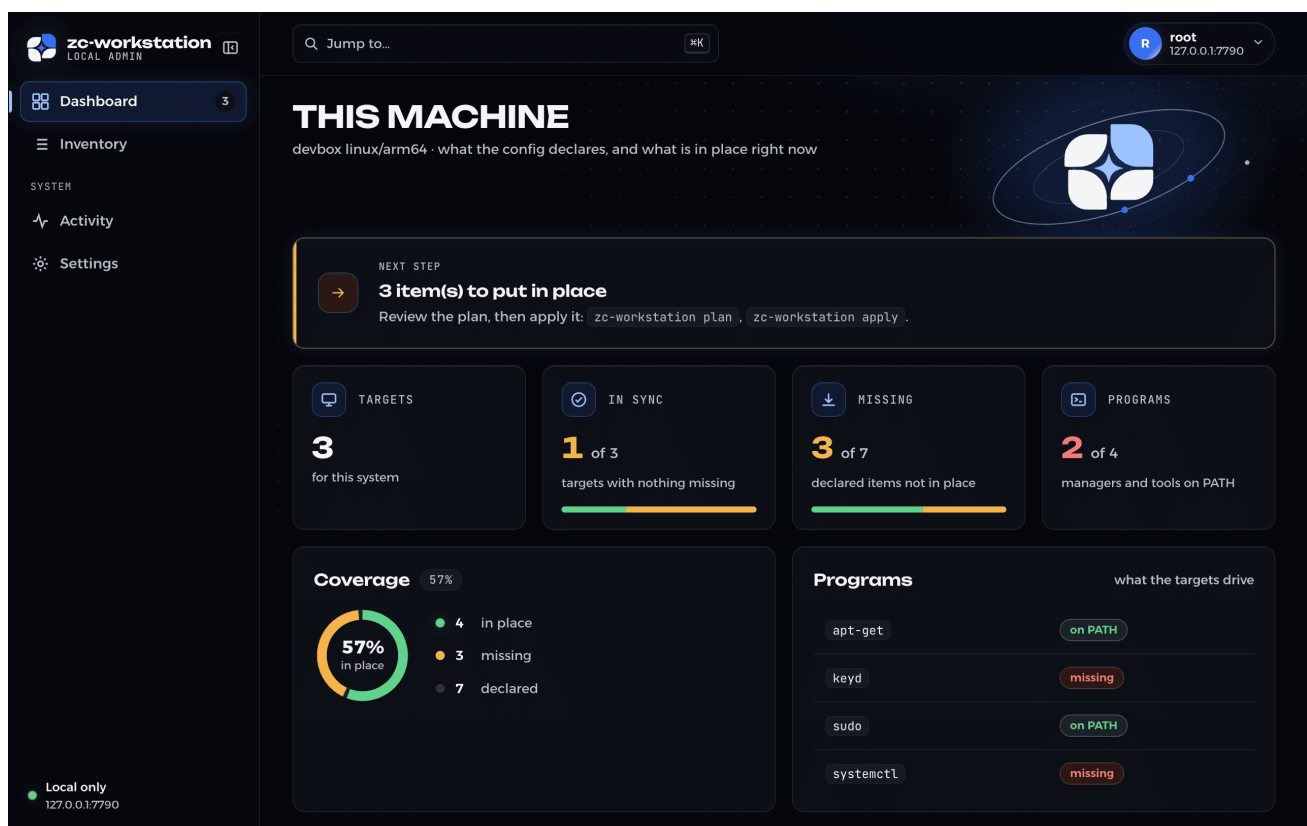
list: the declared packages and runtimes, with why each is there, as an AsciiDoc page

Dashboard

```
$ zc-workstation ui
zc-workstation ui on http://127.0.0.1:7780/ (this machine only; Ctrl-C to stop)
```

The config is read again on every request, so an edit shows on reload.

This machine



The front page reads every target and shows:

- **Next step:** targets that cannot be read, else the number of items still to put in place, else all clear.
- **Targets:** how many are for this system, and how many are skipped.
- **In sync:** targets with nothing missing.
- **Missing:** declared items not yet in place, across all targets.
- **Programs:** the managers and tools the targets drive, such as `brew`, `chezmoi` and `mise`, and whether each is on `PATH`.
- **Coverage:** the share of declared items in place.

- **Targets table:** each target with its provider, its systems, a bar of what is in place, the live summary, and the result of the last plan made on the page.

Inventory

The **Inventory** page lists every package, app, dotfile and runtime the config declares, read from this machine:

- **Tools, Apps, Configured** and **Runtimes:** how many of each are in place. Apps are casks, App Store apps and winget packages.
- **Configured apps:** the programs the dotfiles configure, such as `ghostty` or `zsh`, with their file count. One with a file not in place shows amber.
- **Everything:** one row per declared item with its state, version, app and description, then the items installed but declared nowhere. A newer version shows next to the installed one. The chips above the table count what is not in place, outdated and not declared; each one filters the table. Search with `/`, sort by column, and export CSV. The name links to the project's homepage when the manager knows it.

Undeclared and outdated items do not count against the bars: a bar measures what the config declares.

`zc-workstation list` prints the same table.

One target

A target's page shows its live state, the same item table for that target, and plans on demand. The plan lists every action, and the apply panel takes over from there.

Apply is deliberate

- Apply needs a plan made on the page first.
- You type the target's name to confirm.
- The page plans again before applying, and refuses if the result differs from what you saw.
- A plan that deletes is not applied from the page. The page shows the terminal command instead, where the deletion prompt runs.
- Installs that need `sudo` or another prompt ask in the terminal that runs `zc-workstation ui`. For apt and dnf targets, applying from a terminal is simpler.

Every plan and apply is recorded on the **Activity** page, in a JSON lines file under the user data directory (`-data` changes it).

Look

The page wears the ZoolCoder identity: paper and ink in light, deep space and electric blue in dark, with Unbounded for titles, JetBrains Mono for labels and Alexandria for text. Pick light, dark or the system's choice from the account menu. The fonts are served by the dashboard itself, under the SIL Open Font License, so the page loads nothing from the internet.

The same orbit art dresses the Mac itself, and the [desktop provider](#) sets it:

- `assets/wallpaper` holds the wallpaper, dark and light, at 3024×1964 for a 14-inch MacBook Pro. The title sits top left, clear of the notch and the Dock. `assets/wallpaper/render.sh` draws it again, plus 5120×3200 for 5K displays.
- `assets/screensaver/index.html` is the screen saver: the orbits turn, the stars twinkle, and a clock shows the time and date. The scene drifts slowly so nothing stays in one place. In a browser, add `?light` for paper, or `?still` for no motion; the system's reduced motion setting also stops it. `assets/screensaver/build-saver.sh` builds and installs the saver that shows it.

Local only

The page has no login. It listens on loopback only, answers 403 to a foreign Host header and to a cross-site POST, and caps request bodies.

Menu bar app and ZC Care

ZCBar is a macOS menu bar app over the CLI. It shows what `zc-workstation` would change, the Mac's vitals and updates in a panel, and opens **ZC Care**, a window that cleans, protects and tunes the Mac. Each change ZCBar makes is one you approve first.

It lives in `apps/menubar`, a Swift package for macOS 14 and later.

Install

From the Homebrew tap, with the CLI it needs:

```
$ brew tap zoolcoder/zc-workstation https://github.com/zoolcoder/zc-workstation
$ brew install --cask zcbar
```

Each release builds ZCBar for Apple silicon and Intel and signs it ad hoc, not yet with a Developer ID, so macOS blocks the first start: open it once from the Finder with Control-click, **Open**. `gh attestation verify ZCBar_<version>.zip --repo zoolcoder/zc-workstation` checks the zip was built by the release workflow.

From a clone of the repository:

```
$ make app-install
```

installed ~/Applications/ZCBar.app; open it once, **then** tick Open at login

make app builds bin/ZCBar.app, signed ad hoc. make app-install copies it to ~/Applications. An app opened from the Finder gets no shell environment, so if ZC_WORKSTATION_CONFIG is set, app-install hands that path to the app once. **Choose Config...** in the panel changes it later.

The panel

Click the menu bar mark, or press `⌘Z`, to open the panel. Its tabs:

TAB	SHOWS
Mac	Disk, memory, CPU, battery, network and sensors, the largest junk folders to free up, and advice. Each tile opens a detail.
Overview	Each target with what is in place and missing, Apply per target, and a link into the dashboard .
Updates	Outdated packages and runtimes, with Update per item and Update All...
Health	doctor findings with Fix , Copy Fix and Ignore .
Services	Homebrew services and launchd agents, with Start , Run Now and their logs.

Its settings turn on notifications, the `⌘Z` hotkey, the weekly Smart Care scan and opening at login. If SketchyBar is installed, ZCBar hands it the count of open items for its `zcbar` item.

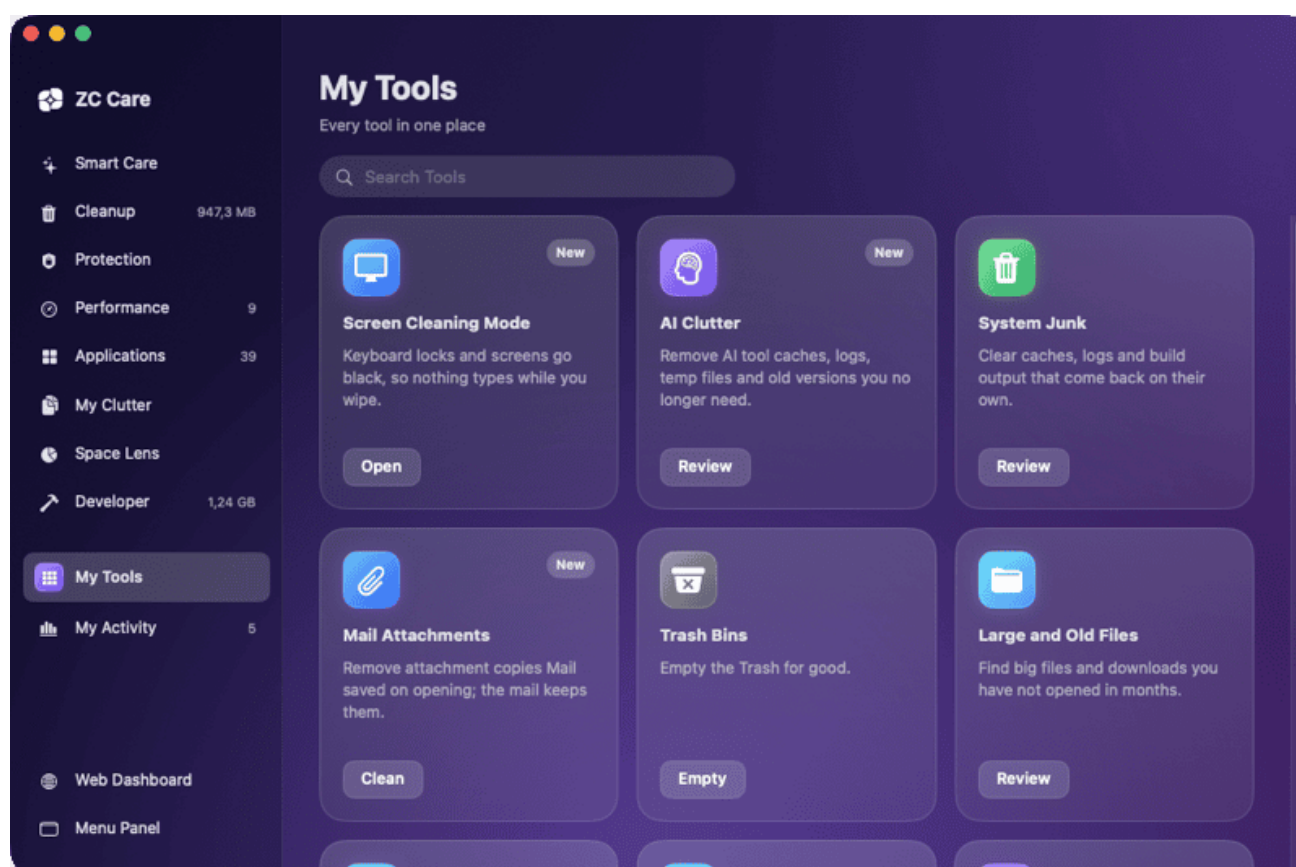
ZC Care

Open ZC Care in the panel opens a window with one module per job. A module opens on an intro that says what it does. Its **Scan** reads the Mac; nothing changes until you press a module's clean or remove button and confirm.



MODULE	DOES
Smart Care	Scans everything at once and sums it up in cards: junk, clutter, updates and health. Run cleans up, then updates.
Cleanup	Clears caches, logs, build output, npx packages and Mail downloads. Clears AI clutter : Claude Code temp files and old versions, and the caches and logs of Claude, Cursor, ChatGPT, Windsurf, Codex, Gemini CLI and Chrome DevTools MCP. Empties the Trash.
Protection	Gatekeeper's verdict on each app, the apps that run in the background, the privacy permissions apps hold, and each browser's history, cookies and cache, cleared only while that browser is closed.
Performance	Login items, background apps and maintenance tasks: free RAM, flush DNS, thin Time Machine snapshots, reindex Spotlight, and clean Homebrew, Docker and simulators.
Applications	Every app with its size and last use. Uninstalls an app with what it keeps in the Library. Adopt replaces an app installed by hand with its Homebrew cask and declares it. Finds what removed apps left behind.
My Clutter	Large files, duplicates, similar images, old downloads and installers. Similar images are photos Vision finds alike; the one with the most pixels stays.

MODULE	DOES
Space Lens	A map of what fills your home folder.
Developer	Dependencies of projects untouched for 90 days, such as <code>node_modules</code> and <code>virtualenvs</code> , and local AI models from Ollama, LM Studio and Hugging Face.
My Tools	Every tool as a searchable card, including Screen Cleaning Mode , which blacks out the screens and locks the keyboard until you click Done .
My Activity	Mac health, disk use, the space ZC Care freed by kind, recommendations from the last scan, and recent activity.



`zcbars://care` opens the window, and `zcbars://care/<module>` opens a module by the start of its name, such as `zcbars://care/cleanup`. `zcbars://panel/<tab>` opens a panel tab.

What it never touches

- Chats, settings and sessions of AI tools: `~/.claude/projects`, `~/.codex/sessions` and the Claude Code version `~/.local/bin/claude` runs.
- A Claude Code temp folder used in the last day: a running session owns it.
- Caches of apps that are open.

Where it goes

Junk, caches and AI clutter are deleted: each tool makes them again. Files, apps, leftovers, dependencies and models go to the Trash, where you can put them back. A Homebrew app is uninstalled through `zc-workstation remove`, so it also leaves the Brewfile. Maintenance tasks that need an administrator ask for your password in a macOS dialog.

ZC Care keeps its last scan in `~/Library/Caches/dev.zoolcoder.zcbar` and the activity log in `~/Library/Application Support/dev.zoolcoder.zcbar`, outside the caches it clears.

Permissions

Reading the privacy permissions needs Full Disk Access for ZCBar. An ad hoc signed build is a new app to macOS, so each rebuild asks again. Emptying the Trash asks once to let ZCBar control the Finder.

Develop

```
$ make app-test      # swift test, the ZCBarCore tests
$ make app           # bin/ZCBar.app
$ make app-icon      # redraw the icons from apps/menubar/icon/mark.svg
```

ZCBarCore holds the logic, with no UI, and is what the tests cover. ZCBar is the SwiftUI app over it. CI builds and tests both on a macOS runner.

Agents and MCP

MCP server

`zc-workstation mcp` serves these tools over standard input and output:

<code>status</code>	Live state per target. Read-only.
<code>list</code>	Every declared item with its state, version, newer version and app, then what is installed but not declared. Takes <code>state</code> and <code>kind</code> filters. Read-only.
<code>add</code>	Declare a package or runtime in one target's manifest, with its why and section. Edits the manifest only.
<code>remove</code>	Uninstall a package and drop it from the manifest. Destructive: only once the user okayed it.
<code>capture</code>	The manifest one target would declare to match this machine. Writes nothing. Read-only.

validate	Check the config offline. Read-only.
plan	Plan and save it. Returns a <code>plan_id</code> . Changes nothing.
apply	Apply a saved plan by <code>plan_id</code> . Refuses when live state moved.
doctor	Install health, security settings and AI agent configs. Read-only.
history	What was applied, newest first, and from where. Read-only.
drift	What the last daily drift check found. Read-only.

`apply` takes only the id of a plan the server saved, never a plan body. A plan that deletes is cancelled and the reply names the terminal command for the user to run.

- **Plans expire.** A saved plan older than an hour is refused; the agent plans again and shows the user the new one.
- **Every call is logged.** Each tool call, its arguments and whether it worked go to `~/.local/state/zc-workstation/mcp-audit.jsonl`. An `apply` also lands in `zc-workstation history` as applied from an agent.
- **Tools say what they do.** Each tool carries MCP annotations: read-only tools say so, `add` edits a file, `remove` is destructive, and `apply` changes the machine. A client can ask before the ones that write.
- **Resources.** The server offers the config (`zc-workstation://config`), the last 50 applies (`zc-workstation://history`) and each target's manifest (`zc-workstation://manifest/<target>`), so an agent reads them without a tool call.
- **Prompts.** Each skill is also an MCP prompt, for clients without skills.

Claude Code plugin

The repository is a plugin: `skills/`, `commands/`, `hooks/` and `.mcp.json`.

```
$ claude plugin marketplace add zoolcoder/zc-workstation
$ claude plugin install zc-workstation
```

- `setup` installs the binary, writes a config and runs `doctor`.
- `plan-apply` plans, explains, asks for a yes, and applies that plan by id.
- `add-software` turns "install X" or "I need a tool for Y" into a manifest entry: it picks the package, writes it with `add` under a section with a why, then plans and applies after a yes. Nothing is installed by hand, so the next machine gets it too.
- `review` reads `doctor` and `list`, then reports what is new in the tools you rely on, what is broken or drifted, and up to three ideas. It changes nothing until you pick.
- Slash commands: `/zc-workstation:plan`, `/zc-workstation:apply`, `/zc-workstation:doctor`, `/zc-`

```
workstation:add <tool or need> and /zc-workstation:review.
```

- The `PreToolUse` hook asks before any `apply` with `-yes` or without a saved plan file.

What is new after an update

`review` reports features worth trying in the versions you moved to. It reads the release notes between the old and the new version, from GitHub releases when the homepage is on GitHub, and keeps only what touches each item's why.

To know what an update moved, upgrade with `zc-workstation upgrade`. It saves the outdated list to `~/.local/state/zc-workstation/upgrades/`, then runs `apply -upgrade`; `review` reads the newest file there.

```
$ zc-workstation upgrade           # everything outdated
$ zc-workstation upgrade -only brew:tmux
```

One setup for every agent

The `agents provider` writes the same MCP servers, rules and skills into Claude Code, Codex, Cursor and Gemini CLI, and pulls the Ollama models you name. A new machine gets them with the rest of `apply`.

Other agents

`adapters/` holds the same rules for Codex, Cursor, Gemini CLI, Copilot and cloud agents.

The toolbox

Overview

zc-workstation does not choose your tools; your config does. This section describes the setup it was built for: a macOS developer workstation for Go, Java, Vue and Flutter work with Claude Code at its centre. Each page says what a tool is for, how the workstation sets it up, and the keys and commands worth knowing. Take what you like into your own Brewfile.

Every tool here is declared in a Brewfile or `mise.toml` with a comment saying why it is there, so `zc-workstation list` shows the same reasons:

```
$ zc-workstation list -kind cask,brew
```

The colours, terminal, prompt and bar themes live in this repository under `assets/theme`. The shell, git, app settings and keyboard config live in a private config repository, applied by the `dotfiles` provider. See [A new Mac in one command](#) for that layout.

How the pieces fit

```
Karabiner (Hyper+T) -> Ghostty -> tmux (dev) -> Claude Code | lazygit | zsh
                                                                starship · atuin · fzf · zoxide
SketchyBar <- macmon, ZCBar JankyBorders around the focused window
Raycast -> repos, branches, zcbar:// URLs
```

A typical start: Hyper+D opens Ghostty running `dev`, you pick a repository, and tmux opens Claude Code, lazygit and a shell in it. ZCBar in the menu bar says when packages are outdated or something drifted, and ZC Care keeps the disk clean. See [Menu bar app and ZC Care](#).

Terminal and shell

TOOL	WHAT IT IS FOR
Ghostty	Ghostty, the terminal: GPU-rendered, native on macOS, with a drop-down quick terminal.
tmux	tmux, the session manager: panes, windows, popups and the dev layout.
zsh, antidote and plugins	zsh with antidote: the shell, its plugins, aliases and helper functions.
Starship	Starship, the prompt: folder, git state, pull request and versions at a glance.

TOOL	WHAT IT IS FOR
Atuin	Atuin, shell history: searchable, per folder, and free of secrets.

Command line

TOOL	WHAT IT IS FOR
fzf	fzf, the fuzzy finder behind completion, the session picker and dev.
zoxide	zoxide, a smarter cd that learns the folders you use.
eza	eza, ls with icons, git status and trees.
bat	bat, cat with syntax highlighting and git changes; also the man page pager.
fd and ripgrep	fd and ripgrep, fast find and grep that respect .gitignore.
jq and yq	jq and yq, query and edit JSON and YAML from the command line.
Yazi	Yazi, a terminal file manager with previews.
ffmpeg, sox, jpegoptim, pngquant and poppler	Convert and shrink video, audio, images and PDFs.
just, xh, hyperfine and other helpers	Project recipes, HTTP calls, benchmarks, command examples, Python scripts and shell linting.
GnuPG, oathtool and mailctl	Signatures, one-time codes and mail accounts kept in a file.

Git and GitHub

TOOL	WHAT IT IS FOR
git, delta, difftastic and git-absorb	git and its helpers: delta diffs, structural diffs, automatic fixups and secret scanning.
lazygit	lazygit, git in a terminal UI.
gitleaks	gitleaks, the secret scanner that runs before every commit.
GitHub CLI	gh, GitHub from the terminal: pull requests, issues, Actions and auth.

Runtimes and dotfiles

TOOL	WHAT IT IS FOR
mise	mise , the runtime manager: Java, Node, Python and Gradle versions.
chezmoi	chezmoi , the dotfiles engine behind the dotfiles provider.
direnv	direnv , per-folder environment variables.
Go, gopls, delve and golangci-lint	The Go toolchain, its language server, debugger and linter.
jdtls, typescript-language-server and CocoaPods	The Java and TypeScript language servers, and CocoaPods for Flutter on iOS.

Containers and data

TOOL	WHAT IT IS FOR
Colima, Docker and friends	Colima , Docker , lazydocker , dive and the Kubernetes tools.
PostgreSQL, pgcli and TablePlus	A local PostgreSQL 17 server, pgcli and TablePlus .
Cloud CLIs	AWS , Google Cloud , Hetzner , Cloudflare Tunnel , Fly.io , Neon , Supabase and the other deploy tools.

Desktop

TOOL	WHAT IT IS FOR
SketchyBar and JankyBorders	SketchyBar , the menu bar replacement, and JankyBorders around the focused window.
Karabiner-Elements and the Hyper key	Karabiner-Elements : Caps Lock as Escape and Hyper, and Hyper shortcuts for apps.
Raycast	Raycast , the launcher, with clipboard history and script commands.
Google Chrome	Chrome , the default browser, with a small set of extensions.
Everyday apps	Shottr , Handy , Bruno , Obsidian , Proxyman , the meeting apps and the fonts.

Editors

TOOL	WHAT IT IS FOR
Visual Studio Code	VS Code, the editor, with settings that follow the system theme.
IntelliJ IDEA and Neovim	IntelliJ IDEA through JetBrains Toolbox, and Neovim in the terminal.

AI

TOOL	WHAT IT IS FOR
Claude Code and AI tools	Claude Code, Codex, the AI desktop apps, CodexBar, Ollama, and how the workstation wires them in.

Keeping the Mac well

TOOL	WHAT IT IS FOR
btop, macmon and fastfetch	btop, macmon and fastfetch: what the Mac is doing.
Mac helpers	asimov, pam-reattach, mkcert and the other helpers that fix small Mac annoyances.

The menu bar app

ZCBar and ZC Care are part of zc-workstation itself, not a separate install. The panel shows the Mac's vitals, the plan, outdated packages, `doctor` findings and services. ZC Care cleans junk and AI clutter, uninstalls apps with their leftovers and finds idle dependencies. See [Menu bar app and ZC Care](#).

Terminal and shell

Ghostty

Ghostty is the terminal every shell, tmux session and Claude Code session runs in. It is fast, it draws with the GPU, and it behaves like a Mac app: native tabs, splits, notifications and the menu bar. It is the first thing you open on the workstation.

On the workstation

- Installed by the `ghostty` cask; the config is `assets/theme/dot_config/ghostty/config`, applied by the `theme` target.
- Always dark, with the ZoolCoder colour theme and JetBrains Mono Nerd Font at 14 pt.

- A glass look: 92% opacity with a blur, so the wallpaper shows faintly.
- A notification when a command that ran over a minute finishes while you look elsewhere.
- A bell frames its split and plays a short sound, so you see which Claude Code session is waiting.
- 50 MB of scrollback, copy on select, and the Option key as Alt.

Keys and commands

KEY OR COMMAND	DOES
Cmd+`	Drop-down quick terminal from any app
Cmd+Shift+C	Claude Code in a split on the right, continuing this folder's last conversation
Cmd+Shift+O	Glass on or off, for screen sharing
Cmd+1 to Cmd+8	Jump to a tab; new tabs open at the end so the numbers stay put

See also

- [tmux](#)
- [zsh, antidote and plugins](#)
- [Claude Code and AI tools](#)
- [Ghostty website](#)

tmux

tmux keeps work sessions alive and splits one terminal into panes. On the workstation it is how a repository becomes a workspace: Claude Code, lazygit and a shell side by side, one session per repo or branch.

On the workstation

- Installed by the `tmux` formula; the config and its theme are in `assets/theme/dot_config/tmux`.
- The prefix is Ctrl+Space, easier to reach than Ctrl+B.
- New panes and windows open in the current folder.
- The `dev` shell function builds the standard layout (see below).

Keys and commands

KEY OR COMMAND	DOES
Ctrl+Space or -	Split side by side or one above the other
Ctrl+Space h j k l	Move between panes; H J K L resize
Alt+h j k l	Move between panes without the prefix
Alt+1 to Alt+4	Jump to a window
Ctrl+Space g	lazygit in a popup
Ctrl+Space a / A	Claude Code in a new window / in a popup
Ctrl+Space s	Pick a session with fzf, with a live preview
Ctrl+Space r	Reload the config
dev [repo] [branch]	Pick a repo under ~/Developer and open it with Claude Code, lazygit and a shell. With a branch, it opens a git worktree for it.
ta tn tls tk	Attach, new, list, kill sessions

See also

- [Ghostty](#)
- [lazygit](#)
- [Claude Code and AI tools](#)
- [tmux website](#)

zsh and plugins

zsh is the login shell. antidote loads its plugins from a plain list and compiles them into one file, so the shell starts fast. The plugins add the things you feel on every keystroke: suggestions, highlighting and fuzzy completion.

On the workstation

- antidote, starship, fzf and the plugins come from the Brewfile; `.zshrc` and `~/.config/zsh/*.zsh` come from the config repo's chezmoi source.
- Plugins, in load order: Oh My Zsh's git, docker-compose, kubectl, gradle, sudo, extract, colored-man-pages, cypath and copybuffer; fzf-tab; zsh-you-should-use; zsh-autopair; zsh-autosuggestions; fast-syntax-highlighting.
- The ZoolCoder theme colours fzf, eza, bat and the highlighting, and follows light and dark mode.

Keys and commands

KEY OR COMMAND	DOES
Tab	Completion as a fuzzy menu with previews (fzf-tab)
Esc Esc	Put <code>sudo</code> in front of the last command
Ctrl+O	Copy the typed command line
<code>update</code>	Pull zc-workstation and the config, rebuild, apply, upgrade packages and runtimes, then run the doctor
<code>wtf</code>	Claude explains why the last command failed and puts a fixed command on the prompt; <code>wtf ci</code> explains the branch's last failed Actions run
<code>aic</code>	Claude writes a conventional commit message for the staged changes; you edit it before it commits
<code>aipr</code>	Claude writes the pull request title and body from the branch; you edit it, then it pushes and opens the PR
<code>gt [query]</code>	Pick one Gradle test class with fzf and run only it
<code>mkcd</code> , <code>killport</code> , <code>serve</code> , <code>ports</code>	Make and enter a folder, free a port, serve this folder over HTTP, list listening ports

See also

- [Starship](#)
- [fzf](#)
- [Atuin](#)
- [zsh, antidote and plugins website](#)

Starship

Starship draws the prompt. It shows where you are and what state the repository is in, so you rarely need `git status` to know whether you are mid-rebase or ahead of the remote.

On the workstation

- Installed by the starship formula; `assets/theme/dot_config/starship.toml` holds the layout and colours.
- Shows the system, the host over SSH, the folder, the git branch, a rebase or merge in progress, changed and ahead/behind counts, lines added and removed, the open pull request, the age of the last commit, and the Node, Java and package versions in use.

- A red mark after a command that failed, and the time a slow one took.

See also

- [zsh, antidote and plugins](#)
- [GitHub CLI](#)
- [Starship website](#)

Atuin

Atuin replaces the shell's history file with a database you search on Ctrl+R. It remembers the folder, the exit code and how long each command took, so finding "that docker command I ran in this repo last week" takes a second.

On the workstation

- Installed by the `atuin` formula; its config comes from the config repo and its theme from `assets/theme/dot_config/atuin`.
- Commands that look like they hold a secret are never saved.
- With `atuin login`, history syncs between your Macs, end-to-end encrypted; without it, it stays on this machine.

Keys and commands

KEY OR COMMAND	DOES
Ctrl+R	Search all history; Ctrl+R again switches to this host, this session or this folder

See also

- [zsh, antidote and plugins](#)
- [fzf](#)
- [Atuin website](#)

Command line

fzf

fzf turns any list into a type-to-filter picker. The workstation uses it everywhere a list would otherwise be typed by hand: tab completion, tmux sessions, repositories, test classes.

On the workstation

- Installed by the `fzf` formula; coloured by the ZoolCoder theme.
- Used by `fzf-tab`, `dev`, `cpj`, `gt` and the `tmux` session picker.

Keys and commands

KEY OR COMMAND	DOES
Ctrl+T	Insert a file path
Alt+C	cd into a folder
** then Tab	Fuzzy completion of a path or a process

See also

- [zsh, antidote and plugins](#)
- [zoxide](#)
- [tmux](#)
- [fzf website](#)

zoxide

`zoxide` remembers the folders you visit and jumps to the best match for a few letters. `z zc-w` lands in `zc-workstation` from anywhere.

On the workstation

- Installed by the `zoxide` formula and hooked into `zsh`.

Keys and commands

KEY OR COMMAND	DOES
<code>z part</code>	Jump to the most used folder matching <code>part</code>
<code>zi</code>	Pick among the matches with <code>fzf</code>

See also

- [fzf](#)
- [zsh, antidote and plugins](#)
- [zoxide website](#)

eza

eza is a modern `ls`. It shows icons, file types in colour, and each file's git status, and it can print a tree.

On the workstation

- Installed by the `eza` formula; `assets/theme/dot_config/eza` holds the light and dark themes.

Keys and commands

KEY OR COMMAND	DOES
<code>ls, ll, la</code>	List, long list, list hidden files
<code>lt3</code>	A tree three levels deep, folders first

See also

- [bat](#)
- [Yazi](#)
- [eza website](#)

bat

bat prints files with syntax highlighting, line numbers and git change marks. It is also the pager for man pages, so they are coloured and searchable.

On the workstation

- Installed by the `bat` formula; its themes are in `assets/theme/dot_config/bat`.

Keys and commands

KEY OR COMMAND	DOES
<code>bat file</code>	Show a file with highlighting
<code>catp</code>	The same, by its alias

See also

- [eza](#)
- [fd and ripgrep](#)

- [bat website](#)

fd and ripgrep

fd finds files by name and ripgrep finds text inside them. Both skip what `.gitignore` ignores and are many times faster than `find` and `grep`, which matters in large repositories.

On the workstation

- Installed by the `fd` and `ripgrep` formulae.
- `rg` is aliased to `rg --smart-case`: lowercase searches ignore case, mixed case does not.

Keys and commands

KEY OR COMMAND	DOES
<code>fd name</code>	Find files whose name matches
<code>rg text</code>	Search the repository for text

See also

- [fzf](#)
- [bat](#)
- [fd and ripgrep website](#)

jq and yq

jq filters JSON: pick fields, select items, reshape the output. yq does the same for YAML with the same expression language, and converts between YAML and JSON. Together they read every `-json` output of `zc-workstation` and its YAML config.

On the workstation

- Installed by the `jq` and `yq` formulae. `yq` is Mike Farah's Go version.
- `gh` takes the same expressions with `--jq`, so a GitHub query needs no pipe.

Keys and commands

KEY OR COMMAND	DOES
<code>jq . file.json</code>	Pretty-print JSON
<code>jq -r '[][.name']</code>	Print a field of every item, without quotes

KEY OR COMMAND	DOES
<code>jq '.[] select(.x > 1)'</code>	Keep the items that match
<code>jq -s '.[0] * .[1]' a.json b.json</code>	Merge two JSON files, the second winning
<code>yq '.targets[].name' file.yaml</code>	Query YAML
<code>yq -i '.key = "value"' file.yaml</code>	Edit YAML in place
<code>yq -o json file.yaml</code>	Convert YAML to JSON

With zc-workstation

The targets that are not fully in place:

```
$ zc-workstation status -json | jq -r '.[ ] | select(.in_place < .declared) | .summary'
41 of 43 dotfiles in place
```

Each planned change on one line:

```
$ zc-workstation plan -json | jq -r '.actions[] | "\(.op) \(.target) \(.resource)'"
UPDATE macos com.apple.finder FXPreferredViewStyle
```

The targets that run on Linux:

```
$ yq '.targets[] | select(.os[0] == "linux") | .name' zc-workstation.yaml
apt
```

See also

- [GitHub CLI](#)
- [Commands](#)
- [jq website](#)
- [yq website](#)

Yazi

Yazi browses folders in the terminal with previews of text, images and archives. Quitting it leaves the shell in the folder you ended up in.

On the workstation

- Installed by the `yazi` formula; `assets/theme/dot_config/yazi` holds its config and the ZoolCoder flavour.

Keys and commands

KEY OR COMMAND	DOES
<code>y</code>	Open Yazi; quit with <code>q</code> to <code>cd</code> where you were

See also

- [eza](#)
- [Yazi website](#)

Media tools

Five command-line tools for the media that ends up in a repository or a message: video and audio with `ffmpeg` and `sox`, images with `jpegoptim` and `pngquant`, and PDFs with `poppler`.

On the workstation

- Installed by the `ffmpeg`, `sox`, `jpegoptim`, `pngquant` and `poppler` formulae.
- `poppler` brings `pdftotext`, `pdftoppm` and `pdftinfo`.

Keys and commands

KEY OR COMMAND	DOES
<code>ffmpeg -i in.mov -c:v libx264 -crf 23 out.mp4</code>	Convert a screen recording to a small MP4
<code>ffmpeg -i in.mp4 -vf "fps=12,scale=800:-1" out.gif</code>	Make a GIF for a README or a pull request
<code>ffmpeg -i in.mp4 -vn -c:a copy out.m4a</code>	Take the audio out of a video
<code>sox in.wav -r 16000 -c 1 out.wav</code>	Resample audio to 16 kHz mono, as speech models expect
<code>jpegoptim --max=85 --strip -all *.jpg</code>	Shrink JPEGs and drop their metadata
<code>pngquant --quality=65-80 --strip --ext .png --force *.png</code>	Shrink PNGs in place and drop their metadata

KEY OR COMMAND	DOES
<code>pdftotext -layout in.pdf -</code>	Print a PDF's text, keeping its columns
<code>pdftoppm -png -r 150 in.pdf page</code>	Render each page to a PNG
<code>pdfinfo in.pdf</code>	Show a PDF's pages, size and producer

With zc-workstation

The screenshots in these docs are PNGs. Before committing one, shrink it:

```
$ pngquant --quality=65-80 --strip --ext .png --force docs/modules/ROOT/images/*.png
```

`--strip` drops metadata, such as which machine and app made the image.

See also

- [Everyday apps](#), for Shottr
- [ffmpeg documentation](#)
- [pngquant](#)
- [poppler](#)

just, xh and helpers

Small tools that each replace a longer command or a lookup: run a project's recipes, call an HTTP API, time two commands against each other, see how a command is used, run a Python script with its packages, download a file, and check a shell script.

On the workstation

- Installed by the `just`, `xh`, `hyperfine`, `tlrc`, `uv`, `wget`, `bash` and `shellcheck` formulae.
- `tlrc` installs the command `tlldr`.
- `bash` is Bash 5. macOS ships Bash 3.2, so a script that needs associative arrays or `mapfile` starts with `#!/usr/bin/env bash` and finds Homebrew's.
- `uv` runs Python scripts and tools without a virtualenv made by hand; `mise` keeps the Python versions.

Keys and commands

KEY OR COMMAND	DOES
<code>just / just -l</code>	Run the default recipe of the <code>justfile</code> / list the recipes
<code>just --choose</code>	Pick a recipe with <code>fzf</code>
<code>xh :8080/healthz</code>	GET from localhost, with the response coloured
<code>xh POST :8080/items name=demo count:=3</code>	POST JSON: <code>=</code> for a string, <code>:=</code> for a raw value
<code>xh -b example.com/api</code>	Print only the response body
<code>hyperfine --warmup 3 'cmd a' 'cmd b'</code>	Time two commands and say which is faster
<code>hyperfine --export-markdown bench.md 'cmd'</code>	Save the result as a Markdown table
<code>tldr tar / tldr -u</code>	Short examples for a command / update the pages
<code>uv run --with requests script.py</code>	Run a script with a package, no virtualenv to set up
<code>uvx ruff check .</code>	Run a Python tool once without installing it
<code>uv tool install ruff</code>	Install a Python tool on PATH
<code>wget -c URL</code>	Resume a download that stopped
<code>shellcheck -x script.sh</code>	Lint a script, following the files it sources

With zc-workstation

Before committing a change to a shell script, such as `bootstrap.sh` or the files in `scripts/`, lint it:

```
$ shellcheck -x bootstrap.sh scripts/*.sh
```

To check that a change made `plan` faster, time both builds:

```
$ hyperfine --warmup 2 'old/zc-workstation plan' 'bin/zc-workstation plan'
```

See also

- [fzf](#), which `just --choose` uses
- [mise](#), for Python versions
- [just manual](#)
- [xh](#)

- [hyperfine](#)
- [uv](#)
- [ShellCheck](#)

GnuPG, oathtool, mailctl

Three tools for keys and accounts: GnuPG for signatures and encrypted files, oathtool for the six-digit codes of two-factor login, and mailctl for mailboxes, aliases and mail DNS kept in a YAML file.

On the workstation

- Installed by the `gnupg` and `oath-toolkit` formulae, and `mailctl` from the `zoolcoder/tap` tap.
- Sign commits with an SSH key rather than GPG: `doctor` checks for it and prints the commands. GnuPG is for the projects and downloads that still use it.
- `oath-toolkit` installs the command `oathtool`.

Keys and commands

KEY OR COMMAND	DOES
<code>gpg --verify file.sig file</code>	Check a download against its signature
<code>gpg --list-secret-keys</code>	List your own keys
<code>gpg -d file.gpg</code>	Decrypt a file
<code>oathtool --totp -b -</code>	Print the current code for a base32 secret read from standard input
<code>mailctl plan</code>	Show what would change in the mail accounts
<code>mailctl apply</code>	Make those changes
<code>mailctl audit</code>	Check MX, SPF, DKIM and DMARC records and deliverability

Give `oathtool` its secret on standard input, from a password manager or a file only you can read. A secret typed as an argument lands in the shell history.

`mailctl` works like `zc-workstation`: `plan` first, then `apply`. It deletes nothing at the provider unless you pass `-prune`, and it deletes mailboxes only with `-prune-mailboxes` as well.

See also

- [git tools](#), for commit signing
- [Security](#), for the `doctor` checks on keys

- [GnuPG documentation](#)
- [OATH Toolkit](#)

Git and GitHub

git and diff tools

The workstation installs a current git and a few helpers that make it easier to read and to keep clean.

On the workstation

- git from Homebrew, newer than the one Xcode ships.
- delta shows diffs side by side with syntax highlighting, in git and lazygit; `assets/theme/dot_config/git/zoolcoder-delta.gitconfig` holds its colours.
- difftastic compares by syntax, not by lines (`git dft`).
- git-absorb turns staged changes into fixup commits for the commits they belong to.
- Global git hooks run a gitleaks scan before every commit, then the repository's own hook.
- git maintenance runs in the background on the repositories you use.

Keys and commands

KEY OR COMMAND	DOES
<code>gs gd gds glog</code>	status, diff, staged diff, the last 20 commits as a graph
<code>git dft</code>	A structural diff
<code>git absorb</code>	Create fixup commits for what is staged

See also

- [lazygit](#)
- [GitHub CLI](#)
- [gitleaks](#)
- [git, delta, difftastic and git-absorb website](#)

lazygit

lazygit makes the git operations that are fiddly on the command line quick: staging single hunks, interactive rebase, fixups, cherry-picks and resolving conflicts. It sits in the dev layout next to

Claude Code, so you review what an agent changed before you commit it.

On the workstation

- Installed by the `lazygit` formula; diffs go through delta.

Keys and commands

KEY OR COMMAND	DOES
<code>lazygit</code>	Open it in this repository
<code>Ctrl+Space g</code>	Open it in a tmux popup

See also

- [git, delta, difftastic and git-absorb](#)
- [tmux](#)
- [lazygit website](#)

gitleaks

gitleaks looks for keys, tokens and passwords in what you are about to commit. On the workstation it runs as a global pre-commit hook in every repository, so a secret is stopped before it reaches history, which is far cheaper than rotating it later.

On the workstation

- Installed by the `gitleaks` formula.
- A global hook chain runs it on staged changes, then hands over to the repository's own hook, so pre-commit and git-lfs keep working.

Keys and commands

KEY OR COMMAND	DOES
<code>gitleaks git</code>	Scan a repository's whole history

See also

- [git, delta, difftastic and git-absorb](#)
- [gitleaks website](#)

GitHub CLI

gh handles the GitHub side of the work without a browser: opening and checking pull requests, watching CI and reading logs. It is also how git authenticates to GitHub.

On the workstation

- Installed by the `gh` formula; `bootstrap.sh` logs you in on a new Mac.

Keys and commands

KEY OR COMMAND	DOES
<code>prc prv prl</code>	Create a pull request, open it in the browser, list them
<code>gh run watch</code>	Follow a CI run
<code>aipr</code>	Let Claude write the pull request, then open it

See also

- [git, delta, difftastic and git-absorb](#)
- [Claude Code and AI tools](#)
- [GitHub CLI website](#)

Runtimes and dotfiles

mise

mise installs and switches language runtimes per project. Enter a folder with a `.nvmrc`, `.sdkmanrc` or `.python-version` and the right version is on PATH; no SDKMAN, nvm or pyenv needed.

On the workstation

- Installed by the `mise` formula; the `runtimes` target applies `mise.toml` through the `tools` provider.
- Pins Temurin 25, 21 and 17, GraalVM Community 25, Gradle, Node LTS and Python 3.13.
- Reads the version files projects already have.
- A config mise does not trust yet is a `MANUAL` step, never trusted for you.

Keys and commands

KEY OR COMMAND	DOES
<code>mise use java@21</code>	Pin a version in this folder
<code>mise ls</code>	What is installed and active
<code>zc-workstation list -kind runtime</code>	Runtimes with their state and why

See also

- [chezmoi](#)
- [direnv](#)
- [mise website](#)

chezmoi

chezmoi applies a source folder of dotfiles to your home folder, with templates for what differs per machine and scripts for setup steps a file cannot express. zc-workstation drives it path by path, so a plan shows each file it would write.

On the workstation

- Installed by the `chezmoi` formula.
- The `home` target applies the config repo's `home/` in symlink mode: editing `~/.zshrc` edits the repo.
- The `theme` target applies `assets/theme` from this repository: colours for the terminal, the bar and the command-line tools.
- `run_onchange_` scripts set up git, macOS, the screen saver, scheduled jobs, app settings and Claude Code.

Keys and commands

KEY OR COMMAND	DOES
<code>zc-workstation plan -target home</code>	What would change in your dotfiles
<code>chezmoi diff</code>	The raw diff, if you want it

See also

- [mise](#)
- [zsh, antidote and plugins](#)
- [chezmoi website](#)

direnv

direnv loads a project's `.envrc` when you enter its folder and unloads it when you leave, so each project gets its own variables without polluting the shell.

On the workstation

- Installed by the `direnv` formula and hooked into `zsh`.

Keys and commands

KEY OR COMMAND	DOES
<code>direnv allow</code>	Trust this folder's <code>.envrc</code> after reading it

See also

- [mise](#)
- [direnv website](#)

Go tools

Go builds `zc-workstation` itself. Homebrew installs the toolchain and three tools that editors and AI agents lean on: `gopls`, the language server; `delve`, the debugger; and `golangci-lint`, the linter that also enforces which package may import which.

On the workstation

- Installed by the `go`, `gopls`, `delve` and `golangci-lint` formulae.
- Code generators a project needs, such as `sqlc` or `oapi-codegen`, are pinned with a `tool` directive in that project's `go.mod` and run through `go tool`, never installed globally.
- A project drives everything through `make`: `make lint`, `make test`, `make build`. The targets wire up the flags that raw commands miss.
- VS Code and Claude Code find `gopls` on `PATH`, so both get the same definitions, references and diagnostics.

Keys and commands

KEY OR COMMAND	DOES
<code>go test ./internal/plan -run TestName</code>	Run one test
<code>go test -race ./...</code>	Run every test with the race detector

KEY OR COMMAND	DOES
<code>go tool <name></code>	Run a tool pinned in <code>go.mod</code>
<code>go get -tool example.com/cmd@v1.2.3</code>	Pin a tool in <code>go.mod</code>
<code>golangci-lint run</code>	Lint with the project's <code>.golangci.yml</code>
<code>dlv test ./internal/plan</code>	Debug the tests of one package
<code>dlv debug ./cmd/app --plan</code>	Debug a program with its arguments
<code>gopls check file.go</code>	Print a file's diagnostics, as an editor sees them

With zc-workstation

The repository's `.golangci.yml` turns on depguard with rules such as `plan-performs-no-io`: the `plan` package may not import `net/http` or run a command. `make lint` fails when one of these import boundaries is crossed, before review sees it.

```
$ make lint
go vet ./...
0 issues.
```

See also

- [VS Code](#)
- [mise](#)
- [Go documentation](#)
- [golangci-lint](#)
- [delve](#)

Language servers

Language servers give an editor or an AI agent what an IDE knows: where a symbol is defined, who calls it, and what does not compile. The workstation installs them once, from Homebrew, so every tool uses the same copy. CocoaPods fetches the iOS libraries a Flutter app depends on.

On the workstation

- Installed by the `jdtls`, `typescript-language-server` and `cocoapods` formulae.
- Claude Code uses them through the `jdtls-lsp` and `typescript-lsp` plugins from `claude-plugins-official`; `gopls-lsp` uses `gopls` from the [Go tools](#).

- jdtls runs on the Java that mise makes active.
- Flutter runs `pod install` itself when it builds for iOS.

Keys and commands

KEY OR COMMAND	DOES
<code>/plugin install jdtls-lsp@claude-plugins-official</code>	Give Claude Code the Java language server
<code>/plugin install typescript-lsp@claude-plugins-official</code>	Give Claude Code the TypeScript language server
<code>cd ios && pod install --repo-update</code>	Fetch a Flutter app's iOS pods with fresh specs
<code>pod outdated</code>	List pods with newer versions

When a Flutter iOS build fails over pods after a Flutter upgrade, remove `ios/Pods` and `ios/Podfile.lock`, then build again.

See also

- [Claude Code and AI tools](#)
- [IntelliJ IDEA and Neovim](#)
- [Eclipse JDT Language Server](#)
- [typescript-language-server](#)
- [CocoaPods guides](#)

Containers and data

Colima and Docker

Colima runs the Docker engine in a small Linux VM, with no Docker Desktop and no licence to think about. The Docker CLI, Compose and Buildx talk to it as usual. The terminal UIs make containers and clusters easy to read.

On the workstation

- `colima`, `docker`, `docker-compose`, `docker-buildx` from the Brewfile; the `services` target keeps colima running.
- `lazydocker` shows containers, logs and images; `dive` looks inside image layers.
- `k9s` (with the ZoolCoder skin), `kubectx` and `stern` for Kubernetes.

Keys and commands

KEY OR COMMAND	DOES
dps	Containers with name, status and ports
dcu dcd dcl	Compose up, down, follow logs
lazydocker	Containers in a terminal UI
k9s	A cluster in a terminal UI
kubectx kubens	Switch context and namespace

See also

- [PostgreSQL, pgcli and TablePlus](#)
- [Colima, Docker and friends website](#)

PostgreSQL

A local database server for development, a friendly command-line client and a GUI. The server runs as a Homebrew service only when you start it.

On the workstation

- postgresql@17, pgcli, pgbadger and the tableplus cask from the Brewfile.

Keys and commands

KEY OR COMMAND	DOES
pgstart pgstop pgstatus	Start, stop, check the server
pg	pgcli, with completion and highlighting
pglog pgreport	Follow the log, or turn it into a pgbadger report

See also

- [Colima, Docker and friends](#)
- [PostgreSQL, pgcli and TablePlus website](#)

Cloud CLIs

One command-line tool per cloud or hosting service the workstation deploys to. Each logs in once through the browser and keeps its token under the home folder.

On the workstation

- Installed by the `awscli`, `oci-cli`, `hcloud`, `cloudflared`, `flyctl`, `railway`, `render`, `koyeb`, `neonctl` and `supabase` formulae and the `gcloud-cli` cask.
- Tokens stay out of the dotfiles: `chezmoi` manages none of the folders where these tools keep their logins, such as `~/.aws` and `~/.config/gcloud`.
- `supabase start` runs its local stack in Docker, so Colima must be running.

Keys and commands

KEY OR COMMAND	DOES
<code>aws sso login --profile work</code>	Log in to AWS through single sign-on
<code>aws sts get-caller-identity</code>	Show which AWS account and role are active
<code>gcloud auth login</code>	Log in to Google Cloud
<code>gcloud config configurations list</code>	List the Google Cloud accounts and projects set up
<code>hcloud context create <name></code>	Add a Hetzner project with its API token
<code>hcloud server list</code>	List the Hetzner servers
<code>cloudflared tunnel --url http://localhost:8080</code>	Share a local service on a temporary public URL
<code>fly deploy</code>	Build and deploy the app in this folder to Fly.io
<code>fly logs</code>	Follow a Fly.io app's logs
<code>railway up</code>	Deploy this folder to Railway
<code>neonctl branches create --name dev</code>	Branch a Neon database, data included
<code>neonctl connection-string dev</code>	Print the connection string of a branch
<code>supabase start</code>	Run Supabase locally
<code>supabase migration new <name></code>	Start a new SQL migration
<code>supabase db push</code>	Apply the migrations to the linked project

With zc-workstation

The cloud CLIs sit under a `# Cloud` section of the Brewfile. A new one goes in the same section with its reason, and one you stop using leaves with its package:

```
$ zc-workstation add -target packages -section Cloud -why "DigitalOcean from the command line" brew:doctl  
$ zc-workstation remove -target packages brew:railway
```

See also

- [Colima and Docker](#)
- [PostgreSQL](#)
- [GitHub CLI](#)

Desktop

SketchyBar

SketchyBar replaces the macOS menu bar with one you script. It shows the focused app, CPU, memory, power and temperatures, battery, the clock, and how many things ZCBar says need you. JankyBorders draws a coloured border around the focused window, so you always know where keys go.

On the workstation

- Installed from the FelixKratz tap; both run as services from the `services` target.
- Config and plugins in `assets/theme/dot_config/sketchybar`; JankyBorders in `assets/theme/dot_config/borders`.
- The `macos` target hides the macOS menu bar so the two do not overlap.
- Temperatures and power come from `macmon`, without `sudo`.
- The `zcbar` item shows ZCBar's count of open items; clicking it opens the panel.

See also

- [ZCBar and ZC Care](#)
- [Karabiner-Elements and the Hyper key](#)
- [SketchyBar and JankyBorders website](#)

Karabiner and Hyper

Karabiner remaps the keyboard. Caps Lock, the most wasted key, becomes Escape when tapped and Hyper (Cmd+Ctrl+Opt+Shift) when held. Hyper plus a letter then opens an app from anywhere, without clashing with any app's own shortcuts.

On the workstation

- Installed by the `karabiner-elements` cask; `karabiner.json` comes from the config repo.

Keys and commands

KEY OR COMMAND	DOES
Caps Lock (tap)	Escape
Hyper+T	Ghostty
Hyper+V	VS Code
Hyper+I	Intellij IDEA
Hyper+B	Chrome
Hyper+A	Bruno
Hyper+M	Teams
Hyper+O	Outlook
Hyper+S	A Shottr area screenshot
Hyper+D	A new Ghostty window running <code>dev</code>

See also

- [Raycast](#)
- [Ghostty](#)
- [Karabiner-Elements and the Hyper key website](#)

Raycast

Raycast replaces Spotlight as the launcher: apps, files, clipboard history, window layouts and small scripts, all from one shortcut.

On the workstation

- Installed by the `raycast` cask.
- Script commands from the config repo: **Open Repo**, **Copy Branch** and **Kube Context**.
- It can open ZCBar through `zcbars://` URLs.

Keys and commands

KEY OR COMMAND	DOES
Raycast hotkey	Open Raycast
Open Repo	Pick a repository and open it
Copy Branch	Copy the current git branch
Kube Context	Switch the Kubernetes context

See also

- [Karabiner-Elements and the Hyper key](#)
- [ZCBar and ZC Care](#)
- [Raycast website](#)

Google Chrome

Chrome is the default browser and the one DevTools work happens in.

On the workstation

- Installed by the `google-chrome` cask; `defaultbrowser` makes it the default from a script.
- Extensions: uBlock Origin Lite, Bitwarden, JSON Formatter, Vue.js devtools, Vimium and Dark Reader.

Keys and commands

KEY OR COMMAND	DOES
Hyper+B	Open Chrome

See also

- [Karabiner-Elements and the Hyper key](#)
- [Google Chrome website](#)

Everyday apps

Small apps that each do one job well.

On the workstation

- **Shottr**: screenshots to annotate, blur or read text from (Hyper+S).
- **Handy**: system-wide dictation with a local Whisper model; open source.
- **Bruno**: an API client whose collections are files in git.
- **Obsidian**: Markdown notes in a local folder.
- **Proxyman**: inspect and change HTTP and HTTPS traffic.
- **TablePlus**: a database GUI.
- **AnyDesk**: remote desktop, to help or be helped.
- **Microsoft Teams and Zoom**: company chat and video meetings.
- **JetBrains Mono Nerd Font**: the font of the terminal, editors and bar, with the icons that eza, starship and SketchyBar draw.
- **Cairo**: a font for Arabic text.

See also

- [Raycast](#)
- [Everyday apps website](#)

Editors and AI

VS Code

VS Code is the main editor, next to IntelliJ IDEA for Java and Neovim in the terminal. Its settings are managed, so a new Mac opens it exactly as the old one had it.

On the workstation

- Installed by the `visual-studio-code` cask; `settings.json` comes from the config repo.
- Catppuccin Latte in light mode and Mocha in dark mode, following the system.
- JetBrains Mono Nerd Font with ligatures, no minimap, coloured bracket pairs.
- Language servers for Go, Java and TypeScript come from the Brewfile, so editors and AI tools share them.

See also

- [IntelliJ IDEA and Neovim](#)
- [Visual Studio Code website](#)

IntelliJ and Neovim

IntelliJ IDEA is the editor for Java and Spring Boot; JetBrains Toolbox installs and updates it. Neovim covers quick edits in the terminal, and `vim` and `vi` open it.

On the workstation

- `jetbrains-toolbox` cask and `neovim` formula.
- IntelliJ's editor and terminal fonts are set to JetBrains Mono Nerd Font by a setup script.

Keys and commands

KEY OR COMMAND	DOES
Hyper+I	Open IntelliJ IDEA
<code>vim file</code>	Neovim

See also

- [Visual Studio Code](#)
- [IntelliJ IDEA and Neovim website](#)

Claude Code and AI

Claude Code is a first-class part of the workstation. It opens in a Ghostty split or a tmux pane in the folder you are in, its sessions show in the bar when they wait for you, and its limits sit in the menu bar. `zc-workstation` itself is a Claude Code plugin, so an agent can plan and apply changes to the machine, with your yes.

On the workstation

- Codex, Claude, ChatGPT and GitHub Copilot desktop apps come from the Brewfile; Ollama runs models locally.
- CodexBar shows Claude Code and Codex session and weekly limits with reset countdowns.
- `claude-squad` runs several Claude Code sessions side by side (`cs`).
- A setup script installs the `zc-workstation` plugin and Claude's notification hook.
- ZC Care's Cleanup clears what AI tools leave behind and never touches their chats or the version in use.

Keys and commands

KEY OR COMMAND	DOES
Cmd+Shift+C (Ghostty)	Claude Code in a split, continuing here
Ctrl+Space a / A (tmux)	Claude Code in a window / popup
cc cr	Continue the last session / pick one
cpj	Pick a repository and open Claude there in tmux
cu	Token and cost reports
wtf aic aipr	Explain a failure, write a commit message, write a pull request

See also

- [tmux](#)
- [Ghostty](#)
- [ZCBar and ZC Care](#)
- [Claude Code and AI tools website](#)

Keeping the Mac well

System monitors

btop shows processes, CPU, memory, disks and network in one screen. macmon reads Apple Silicon power, temperatures, GPU and Neural Engine use without sudo, and feeds the bar. fastfetch prints a summary of the system.

On the workstation

- btop, macmon and fastfetch formulae, each themed in assets/theme/dot_config.

Keys and commands

KEY OR COMMAND	DOES
btop	Everything the Mac is doing
macmon	Power and temperatures
dust	Which folders use the disk

See also

- [SketchyBar and JankyBorders](#)
- [ZCBar and ZC Care](#)
- [btop, macmon and fastfetch website](#)

Mac helpers

Command-line helpers that make the Mac behave better for development.

On the workstation

- **asimov**: keeps `node_modules`, Gradle `build` and other rebuildable folders out of Time Machine; runs once a day.
- **pam-reattach**: Touch ID for `sudo` also works inside `tmux`.
- **mkcert**: trusted HTTPS certificates for local development.
- **terminal-notifier**: notifications from scripts, such as the weekly update.
- **mas**: installs App Store apps from the Brewfile.
- **fileicon**: sets a custom icon on a folder or an app from the command line (`fileicon set ~/Developer icon.png`).
- **A weekly update** launch agent runs `update` and notifies you.

See also

- [btop, macmon and fastfetch](#)
- [ZCBar and ZC Care](#)
- [Mac helpers website](#)

The project

Adding a provider

A provider turns a target into actions. It is one interface:

```
type Provider interface {
    Plan(ctx context.Context, t config.Target) ([]plan.Action, error)
    Status(ctx context.Context, t config.Target) (string, error)
}
```

Five optional interfaces feed the dashboard, `doctor`, `list`, `capture` and `add`:

```
// Measure counts what is in place, for the progress bars.
type Measurer interface {
    Measure(ctx context.Context, t config.Target) (Live, error)
}

// Programs lists what the provider runs, for the Programs panel.
type Requirer interface {
    Programs(t config.Target) []string
}

// Items lists every declared item, for list and the Inventory page.
type Lister interface {
    Items(ctx context.Context, t config.Target) ([]Item, error)
}

// Capture writes the manifest that matches this machine, for capture.
type Capturer interface {
    Capture(ctx context.Context, t config.Target) ([]byte, error)
}

// Add declares one entry in the manifest, for add. It edits the file only.
type Adder interface {
    Add(t config.Target, e Entry) error
}
```

A line-based manifest gets sections and whys from `internal/manifest: manifest.Notes` reads them for `Items`, and `manifest.Insert` and `manifest.Line` write an entry for `Add`.

Steps

1. Add a package under `internal/provider/`. `tools` is the smallest that drives another program; `localdir` writes files itself.
2. Add its name to `config.Providers`, and its fields and checks to `Target.validateFields`.
3. Register it in `engine.With`.

4. Deny its import from `internal/ui` in `.golangci.yml`, so the page goes through the engine.
5. Document it in [Providers](#).

Run programs through `sh.Runner`

A provider that runs other programs takes an `sh.Runner`, never `os/exec` directly:

- `Output` reads, and its error carries the program's `stderr`.
- `Run` installs, attached to the command's streams. Under `mcp`, those are buffers, never the protocol stream.
- Has checks `PATH`.

Tests use `sh.Fake`. It answers commands from a table and records what `Run` was given, so a test never touches the machine:

```
r := &sh.Fake{
  Programs: []string{"mise"},
  Out:      map[string]string{"mise ls --json -C /cfg": `{...}`},
}
actions, err := tools.New(r).Plan(ctx, target)
_ = actions[0].Do(ctx)
// r.Ran[0] == "mise install -C /cfg java@temurin-25"
```

A test against the real program skips when the program is missing. The dotfiles tests run `chezmoi` in temporary directories this way.

Rules

1. **Plan writes nothing.** Read live state, compare, and return actions. Every write lives in an action's `Do`.
2. **Apply does what the plan showed.** Act on the planned item only. The dotfiles provider applies one path, not the whole source.
3. **Say what breaks.** `Detail` is what a person reads to decide. Never put a value that could be a secret in it.
4. **MANUAL has no `Do`.** Use it for what no tool should do, and say what a person must do instead.
5. **Writes are verified.** The engine plans again after apply. A provider whose write is accepted but not kept shows up as still outstanding; do not hide that by caching.

Threat model

What it protects

- **Nothing is applied that was not shown.** A saved plan refuses to apply when the machine changed since. The dashboard plans again before applying. The dotfiles provider applies only the planned paths.
- **Deletions need a person.** The CLI asks, the dashboard sends you to the terminal, and the MCP server cancels.
- **Nothing is uninstalled.** The packages provider only installs.
- **Nothing is upgraded unless a target asks.** `upgrade: true` is per target, and each upgrade is a planned `UPDATE` you review first.
- **Capture never replaces a file.** It writes standard output, or a new file; replacing one needs `-force`.
- **Untrusted mise configs stay untrusted.** Trusting one is a `MANUAL` step, never automatic.
- **Writes stay inside a target.** `localdir` writes through `os.Root`: a symlink or `..` cannot lead a write out of the directory.
- **The protocol stream stays clean.** Under `mcp`, installers get empty input and their output is captured, never written to the protocol stream.
- **The dashboard stays local.** Loopback only, `Host` checked against DNS rebinding, cross-site writes refused, strict CSP.
- **An agent's plan goes stale.** The MCP server refuses a saved plan older than an hour, and writes every tool call to `mcp-audit.jsonl` in its state folder. Every apply, from the terminal, the dashboard or an agent, lands in `zc-workstation history`.

What doctor checks

`zc-workstation doctor` reads, and changes nothing. A weak setting is a warning, not a failure: the choice is yours.

- **macOS:** FileVault, the firewall and stealth mode, a password after sleep, Gatekeeper, System Integrity Protection, automatic updates, File Sharing, Remote Login, Screen Sharing, automatic login, AirDrop open to everyone, Touch ID for `sudo`, and servers listening on every interface.
- **Every system:** SSH private keys without a passphrase, git commits left unsigned, and secrets in the shell history. `gitLeaks` scans `~/.zsh_history`, `~/.bash_history`, the fish history and Atuin's, with `--redact`: the report names the rule and the count, never the secret.
- **AI agents:** API keys written in plain text in the MCP configs of Claude Code, Claude Desktop, Cursor, Gemini, Windsurf and Codex; MCP servers that `npm`, `bunx` or `uvx` start without a version; Claude Code rules that allow any shell command, and `bypassPermissions`.

What it does not

- **A config or manifest you did not write.** A Brewfile can name any package, a `chezmoi` source

can run scripts, and a mise config can install any tool. Review them as you would a script.

- **The package sources.** Homebrew, apt, dnf, winget and mise verify what they download; `zc-workstation` adds nothing to that.
- **sudo.** Installs that run `sudo` ask for your password in the terminal, or, with no terminal, in a macOS dialog that only you can answer. The dialog is a script in the user cache folder; the password goes straight to `sudo` and is never stored or seen by an agent.
- **An attacker who already has your shell.**

Verifying a release

Each release is built by GitHub Actions from its tag. `checksums.txt` lists every archive and package, carries a keyless cosign signature, and each file has SLSA build provenance and an SPDX SBOM.

1. Check the signature on the checksums. It proves the release workflow of this repository signed them:

```
cosign verify-blob checksums.txt --bundle checksums.txt.sigstore.json \
  --certificate-identity-regexp '^https://github.com/ZoolCoder/zc-
workstation/.github/workflows/release.yml@refs/tags/v' \
  --certificate-oidc-issuer https://token.actions.githubusercontent.com
```

2. Check the file you downloaded against the checksums:

```
shasum -a 256 --ignore-missing -c checksums.txt
```

3. Check its provenance, which names the commit and workflow that built it:

```
gh attestation verify zc-workstation_darwin_arm64.tar.gz --repo zoolcoder/zc-workstation
```

The SBOM, `<archive>.sbom.json`, lists the Go modules built into each binary.

Report a problem through the repository's security policy, never a public issue.

Roadmap

`aurora-mac` sets up one kind of machine, a Mac, with about 850 lines of shell. Its successor, **zc-workstation**, sets up macOS, Linux and Windows from one repo. It is a new repository. `aurora-mac` stays as it is until `zc-workstation` reaches parity on macOS, and then it is archived.

Decisions

QUESTION	DECISION
Engine	chezmoi applies the dotfiles. A thin Go CLI does the glue that chezmoi does not do.
Platforms	Phased: macOS parity first, then Linux, then native Windows.
Repository	New repo github.com/ZoolCoder/zc-workstation , MIT. The CLI binary is <code>zc-workstation</code> , with a short alias <code>zcw</code> .
Scaffold	Generated from go-tool-template with <code>zoolfleet new</code> (class <code>oss</code> , license MIT), so it gets the plan/apply engine, <code>doctor</code> , MCP, skills, evals, Antora docs and release workflows that the other tools already share.

Found on GitHub

Searched first, so that the new code is only the glue nobody else provides.

PROJECT	LICENSE	VERDICT	WHY
twpayne/chezmoi (Go, ~22k stars)	MIT	Adopt	Dotfiles on macOS, Linux and Windows. Per-OS templates, <code>run_onchange_</code> scripts, secrets from password managers, and a symlink mode. It replaces <code>link()</code> in <code>install.sh</code> and every per-OS branch we would otherwise write. Used as an external binary, not as a library.
jdx/mise (~35k stars)	MIT	Keep	Already manages Java, Node and Python, and works on all three systems. Its <code>aqua</code> and <code>github</code> backends can also install CLI tools (<code>starship</code> , <code>lazygit</code> , <code>yazi</code> , <code>delta</code> , <code>atuin</code>) the same way everywhere, which shrinks the per-OS package lists.
go-tool-template (private) and toolshell	MIT	Adopt	Our own. Provider/plan/apply engine, <code>toolshell/doctor</code> checks, MCP server, admin UI, docs and release pipeline.
charmbracelet/huh	MIT	Adopt	The pick list for <code>catalog</code> . It replaces <code>zfz</code> , so the picker also works in a Windows console.
goreleaser	MIT	Adopt	Builds the binaries and publishes a Homebrew tap formula, a Scoop/winget manifest and Linux packages.

PROJECT	LICENSE	VERDICT	WHY
topgrade-rs/topgrade	GPL-3.0	Optional	Upgrades every package manager on a machine. <code>update</code> calls it only when it is installed, as a separate program. It is never vendored or linked, so its GPL does not reach this repo.
comtrya , rotz , dotter	MIT	Skip	Rust tools that overlap with chezmoi. They are smaller and younger, and comtrya says it is early in its development.
yadm , dotbot	GPL-3.0 / MIT	Skip	Symlinks plus shell commands, with weaker Windows support than chezmoi.
Package manager wrappers (pm, pkm, tux)	various	Skip	Shell, Unix only. The package provider is ours, about 200 lines of Go.

How it fits together

```

zc-workstation/
  cmd/zc-workstation/      CLI: init, plan, apply, update, doctor, catalog, mcp
  internal/provider/
    dotfiles/              runs chezmoi (source: home/)
    packages/              brew + Brewfile, apt, dnf, winget; one manifest per OS
    tools/                 mise install
    settings/              macOS defaults, GNOME gsettings, Windows registry
    services/              launchd, systemd --user timers, Task Scheduler
    keyboard/              keyd on Linux
  home/                    chezmoi source state; *.tmpl where the OS changes the file
  packages/
    Brewfile               macOS
    apt.txt, dnf.txt        Linux
    winget.json             Windows (winget import format)
  docs/                   Antora

```

- `plan` shows what each provider would create, update or delete, and changes nothing.
- `apply` runs a plan after a person says yes, from the CLI, the UI or an agent through MCP.
- `doctor` replaces `doctor.sh`, built on `toolshell/doctor`, and adds `--json` for agents and CI.
- `update` replaces the `update` alias: pull the repo and the overlay, `apply`, upgrade (topgrade when installed), then `doctor`.
- The overlay stays a separate private repo. It is a second chezmoi source with its own state file and its own package manifests, applied after the public one. The path moves to `~/.config/zc-workstation/overlay`.
- The scripts in `bin/` (`git-tidy`, `gh-summary`, `prompt-pr`, `ci-watch`, `catalog`) become subcommands

only where they must run on Windows. Unix-only helpers stay shell scripts under `home/`.

What carries over

PART	MACOS	LINUX	WINDOWS
zsh, antidote, starship, atuin	yes	yes	starship and atuin in PowerShell
tmux	yes	yes	inside WSL only
Terminal	Ghostty	Ghostty	WezTerm until Ghostty ships for Windows
git, delta, lazygit, yazi, mise	yes	yes	yes
VS Code, IntelliJ settings	yes	yes	yes
Claude Code config and hooks	yes	yes	yes, hooks rewritten where they call Unix tools
Theme (Catppuccin Latte/Mocha follows the system)	yes	GNOME first	yes
Caps Lock and Hyper key	Karabiner	keyd	PowerToys Keyboard Manager
Launcher	Raycast	GNOME default	PowerToys Run
Menu bar, window borders	SketchyBar, JankyBorders	none	none
Scheduled jobs	launchd	systemd --user	Task Scheduler
System settings	<code>macos.sh</code>	<code>gsettings</code>	registry
Clean-machine test	<code>scripts/vm-test.sh</code> (tart)	container in CI	Windows runner in CI

Where it stands

PIECE	STATE	NOTES
Engine, CLI, MCP server, dashboard	Done	From go-tool-template. The dashboard shows coverage, missing items and the programs each target needs.
os filter per target	Done	One config for every machine. Targets for another system are skipped.
packages: brew, mas, apt, dnf, winget	Done	Installs only, never uninstalls. Read-only status matches the aurora-mac Brewfile: 78 of 80 installed.
dotfiles through chezmoi	Done	Applies only the planned paths. <code>mode: symlink</code> is supported, which settles the copy-or-link risk below.
tools through mise	Done	Untrusted configs stay a MANUAL step.
settings and services providers	Done on macOS, Linux and Windows	macOS defaults, Homebrew services and launchd agents; GNOME gsettings and systemd user units and timers; the Windows registry under HKCU and Task Scheduler.
Security checks in doctor	Done on macOS	`doctor.sh's FileVault, firewall, screen lock, Gatekeeper, SIP, update, File Sharing, Touch ID and exposed-server checks, as warnings with their fix.
list, capture, upgrades	Done	list shows outdated and undeclared packages and runtimes in about two seconds. capture writes a Brewfile, package list or mise config from the machine. <code>upgrade: true</code> plans upgrades per target.
aurora-mac config moved to a private config repo	Done	A private zc-workstation-config repo holds the Brewfile, the mise config and a chezmoi home/source, so nothing personal enters this repo.
aurora-mac steps beyond links	Done	The git identity, hooks and signing key, app settings (VS Code, IntelliJ, Chrome, CodexBar), <code>gradle.properties</code> , git maintenance and Claude Notify are chezmoi <code>run_onchange_</code> scripts in the config repo; macOS defaults and background services are settings and services targets.

PIECE	STATE	NOTES
add, remove, -upgrade and -only	Done	add declares an item with why it is there. remove uninstalls and drops it from the manifest. sudo asks in a macOS dialog when there is no terminal.
bootstrap.sh, theme and desktop provider	Done on macOS	A new Mac in one command from the config repo. The ZoolCoder theme, wallpaper and screen saver.
ZCBar menu bar app and ZC Care	Done on macOS	The panel over the CLI, and a window that cleans junk and AI clutter, uninstalls apps, finds clutter and idle dependencies, and logs what it freed. Similar Images is still to come. See Menu bar app and ZC Care .
CI on macOS, Linux and Windows runners	Done	ci.yml tests the CLI on Linux, macOS and Windows runners, and builds and tests ZCBar on macOS.
The toolbox	Done	A manual page per tool in the reference setup: what it is for, how it is configured, its keys. See Overview .
Repository hardening	Ready to apply	Git flow with develop, rulesets for main, develop and v* tags, pinned actions, CodeQL and govulncheck. scripts/harden.sh applies the GitHub settings.
First release, v0.1.0	Next	Binaries, the Homebrew cask, Linux packages and Windows zips through goreleaser.

Next

The concrete tasks are in [TODO.md](#). In order:

1. **Release.** Apply the repository settings, cut v0.1.0, and run bootstrap.sh from the cask on a clean macOS VM.
2. **ZC Care.** Similar Images in My Clutter, and browser privacy data in Protection.
3. **ZCBar distribution.** A Developer ID signature and notarisation, so Full Disk Access survives an update, and a cask for the app.
4. **Linux and Windows.** Phases 2 and 3 below.

Phases

Each phase ends with a check that proves it, not a date.

PHASE	WORK	DONE WHEN
0. Scaffold	<code>zoolfleet new go-tool-template</code> as <code>zc-workstation</code> (oss, MIT). Push the empty repo, with CI green.	<code>make lint test build</code> passes on macOS, Linux and Windows runners.
1. macOS parity	Move the config folders into <code>home/</code> as <code>chezmoi</code> source state. Write the dotfiles, packages, tools, settings and services providers. Port <code>doctor.sh</code> checks and <code>catalog</code> . Port the overlay. Release through goreleaser and a Homebrew tap.	On a clean macOS VM, <code>zc-workstation apply</code> gives the same result as <code>install.sh</code> . <code>doctor</code> is clean on both of your Macs. <code>aurora-mac</code> is no longer used.
2. Linux	Templates for Linux paths, <code>apt.txt</code> and <code>dnf.txt</code> , systemd timers, gsettings theme, keyd. Ubuntu LTS and Fedora.	<code>apply</code> then <code>doctor</code> pass in Ubuntu and Fedora containers in CI.
3. Windows	<code>winget.json</code> , PowerShell profile, WezTerm, PowerToys, registry settings, Task Scheduler. WSL gets the Linux profile.	<code>apply</code> then <code>doctor</code> pass on a Windows CI runner.
4. Retire <code>aurora-mac</code>	Archive the repo on GitHub with a pointer to <code>zc-workstation</code> . Rename <code>aurora-mac-overlay</code> to <code>zc-workstation-overlay</code> . Update <code>CLAUDE.md</code> and memory paths.	Nothing on any machine still reads <code>~/Developer/Tools/aurora-mac</code> .

Docs

The docs move to `zc-workstation` with the code and change shape:

- **Per-OS getting started.** One page each for macOS, Linux and Windows, in place of today's macOS-only setup page.
- **Generated reference.** The command reference comes from the CLI's help, and the package list comes from the manifests, so neither can drift.
- **Split by purpose.** Tutorials (first setup), how-to (add a package, add a dotfile, write an overlay), reference (commands, manifests, providers) and explanation (why `chezmoi`, the threat model).
- **Workstation notes.** Claude Code, Git, Flutter, PostgreSQL, disk management and maintenance stay as notes modules, each marked with the systems it applies to.
- **No personal paths.** Before the repo goes public, the five pages that still name personal paths are made generic. The personal detail moves to the overlay.
- **Published.** The template's `docs.yml` workflow publishes the site to GitHub Pages.

Risks

- **chezmoi copies files by default.** Editing `~/.zshrc` directly no longer edits the repo. Either use `chezmoi edit` and `chezmoi re-add`, or set `mode = "symlink"` for the files that you edit often. Decide this in phase 1.
- **Windows has no tmux or zsh.** The Windows profile is PowerShell-first, with WSL for the Unix workflow. Parity there means the same tools and theme, not the same shell.
- **Two dotfile engines in phase 1.** Until parity, aurora-mac and zc-workstation must not both manage one machine. Test zc-workstation on a VM, then switch each Mac once.

Not planned

- Managing servers or fleets of machines. That is Ansible's job.
- Nix or home-manager. They do not cover native Windows.
- Applying a plan without a person's yes.