BUILD THE ULTIMATE

AI / SOFTWARE-DEVELOPMENT WORKSTATION

with Omarchy

STEVINATOR HANDBOOK

Build the Ultimate AI/Software-Development Workstation with Omarchy

Volume II — Mastery & Maintenance
Modules 13 through 24

Edition 1 · 2026
Typeset with HTML/CSS · Rendered with WeasyPrint
Less spectacle. More working software.

Publication Note

This handbook documents a real, working Omarchy AI/software-development workstation. Commands, paths, and configuration examples are preserved exactly as verified against the current Omarchy release described in each module.

Omarchy evolves quickly. Where a module notes a version dependency, verify current behavior on your own installation before treating any instruction as permanent.

Guiding principle: less spectacle, more working software.

Table of Contents

13 Themes, Fonts, Backgrounds, and Visual Design on Omarchy

14 Plugins, Hooks, and Automation on Omarchy

15 Audio, Bluetooth, Network, and Displays on Omarchy

16 System Toggles and Runtime Controls on Omarchy

17 Defaults and XDG Integration on Omarchy

18 Updates and Maintenance on Omarchy

19 Restart, Repair, and Recovery on Omarchy

20 Boot, Limine, UKIs, and Secure Boot on Omarchy

21 Security and Hardening on Omarchy

22 Reproducible Workstation Setup on Omarchy

23 Troubleshooting Omarchy Like a Developer

24 Understand the Full Stack You Built

MODULE 13

13

Themes, Fonts, Backgrounds, and Visual Design on Omarchy

You now know how to customize how Omarchy behaves.

This module is about customizing how Omarchy looks.

About This Module

You now know how to customize how Omarchy behaves.

This module is about customizing how Omarchy looks.

Current Omarchy 4 / Quattro has a coordinated theming system that can style far more than just the wallpaper.

A theme can affect:

- the Omarchy Shell;
- Hyprland;
- terminal applications;
- Neovim;
- btop;
- Chromium;
- supported editors;
- notifications;
- menus;
- lock screen;
- background choices;
- icon presentation;
- unlock visuals.

This is one of Omarchy's strongest qualities.

Instead of theming each application independently, Omarchy provides a central visual layer.

The goal of this module is not to spend hours chasing perfect RGB values.

The goal is:

> ***Build a consistent, readable, professional development environment without breaking the system or creating theme-maintenance debt.***

What You Will Learn

By the end of Module 13, you will understand:

- how Omarchy themes work at a high level;
- how to switch themes;

- how to switch backgrounds;
- how fonts are integrated across the workstation;
- why Nerd Fonts matter;
- how to install additional supported fonts;
- how to choose a coding font;
- how text size and monitor scaling differ;
- where user backgrounds live;
- where user themes live;
- where Omarchy's built-in themes live;
- how `colors.toml` drives theme generation;
- how custom themes work;
- how Omarchy themes unsupported applications with templates;
- how current Omarchy treats third-party theme code safely;
- how to create a coherent visual baseline for development.

13.1 — Omarchy Themes Are System Themes

A normal desktop theme may only change:

```
window colors
+
wallpaper
```

Current Omarchy themes are broader.

The current official manual states that a theme can style:

- desktop;
- terminal;
- Neovim;
- btop;
- Chromium;
- Omarchy top bar;
- Omarchy menu;
- notifications;
- OSD;
- lock screen.

This means theme changes are intended to feel system-wide.

13.2 — The Current Theme Selector

Current official shortcut:

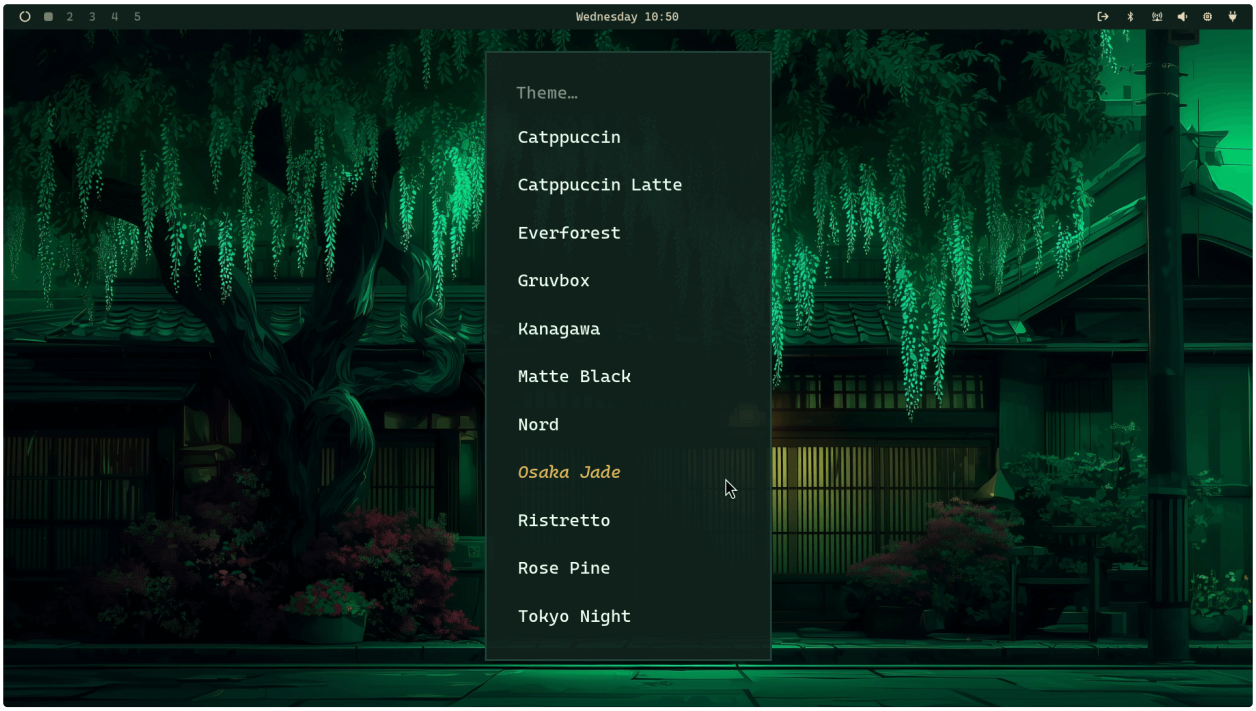
```
Super + Ctrl + Shift + Space
```

This opens the theme picker.

You can also use:

```
Super + Space
→ Style
→ Theme
```

The current official Omarchy manual describes a visual, filterable theme selector.



Screenshot 13.2-A
Omarchy theme selector open, listing themes with the active one highlighted

13.3 — Current Built-In Themes

The exact theme list can evolve.

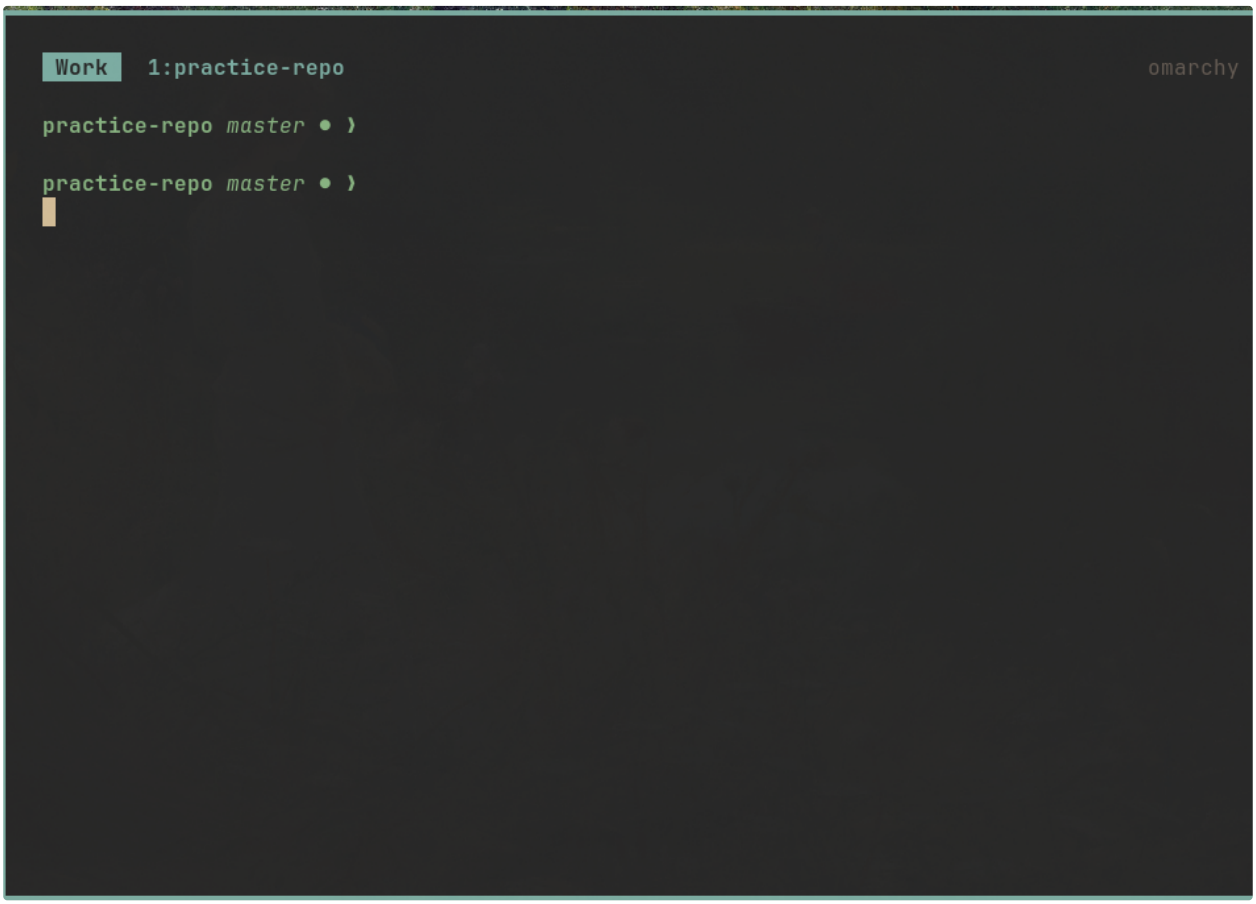
Current Omarchy documentation includes themes such as:

```
Tokyo Night
Catppuccin
Lumon
Ethereal
Everforest
Gruvbox
Miasma
Hackerman
Osaka Jade
Kanagawa
Nord
Matte Black
Vantablack
Ristretto
Retro 82
Flexoki Light
Rose Pine
Catppuccin Latte
White
```

and newer releases may add more.

Do not memorize the static list.

Inspect the current system instead.



Screenshot 13.3-A
a contrasting dark theme (e.g. Gruvbox)

13.4 — Discover Themes Through the CLI

Current official Omarchy CLI documentation includes:

```
omarchy theme list
```

to list available themes.

And:

```
omarchy theme set <name>
```

to apply a theme.

Example:

```
omarchy theme list
```

Then:

```
omarchy theme set catppuccin
```

Use the exact installed theme name returned by your system.

Screenshot 13.4-A
`omarchy theme list`

13.5 — Our Stevinator Baseline

The workstation used while developing this course currently uses:

```
Catppuccin
```

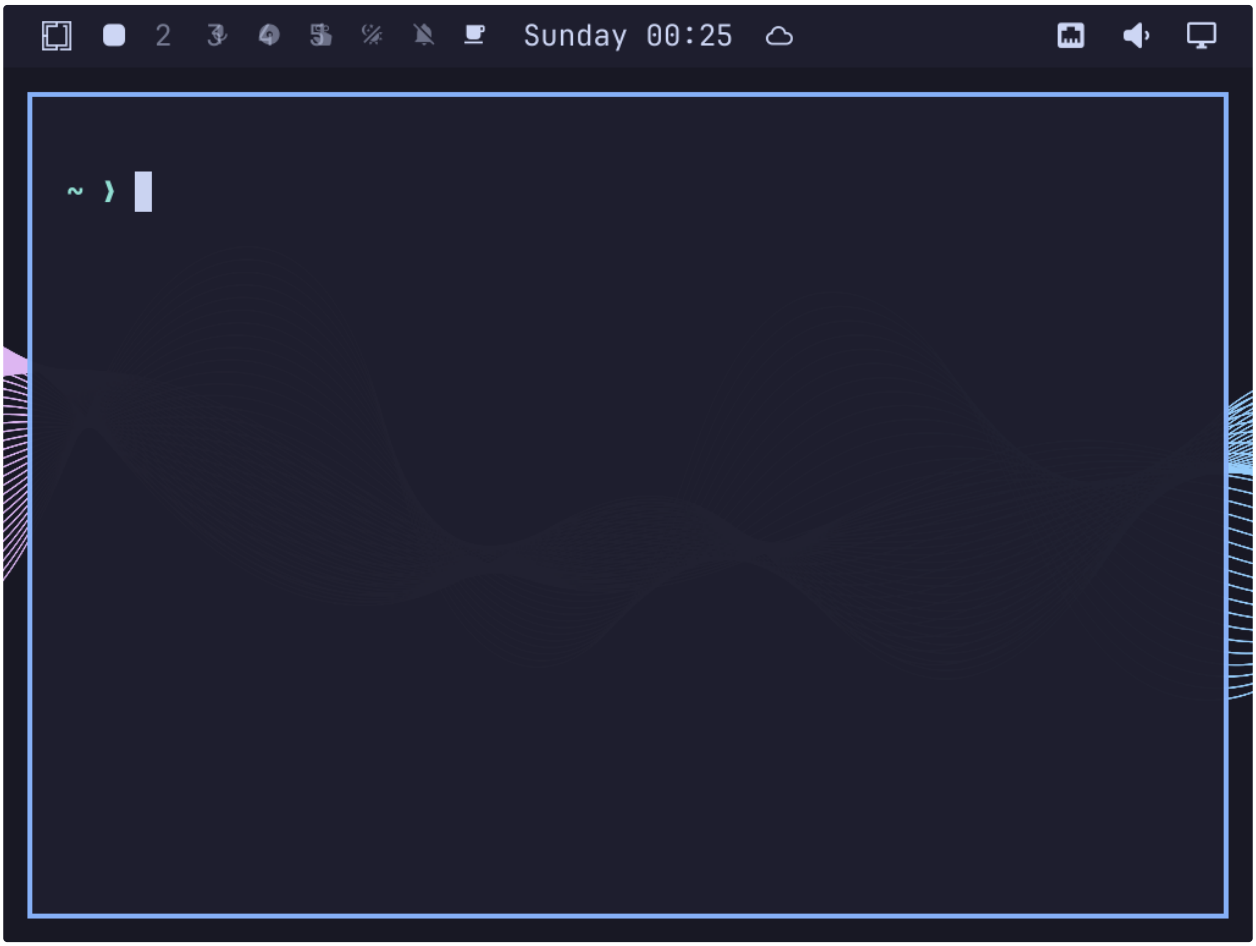
as its theme.

That is a preference.

It is not a course requirement.

Choose a theme that gives you:

- good contrast;
- readable code;
- comfortable long-session use;
- consistent UI.



Screenshot 13.5-A
Catppuccin theme active across desktop and top bar

13.6 — Theme Selection Is More Than Appearance

The wrong visual setup can create real development friction.

Examples:

```
low contrast
→ harder code scanning

tiny text
→ eye strain

high-saturation UI everywhere
→ distraction

poor foreground/background pairing
→ terminal readability problems
```

A developer workstation should optimize for:

```
readability
+
consistency
+
low friction
```

before novelty.

13.7 — Dark vs Light Themes

There is no universal winner.

Dark themes may be comfortable:

- in low-light environments;
- for long terminal sessions;
- on some OLED displays.

Light themes may be better:

- in bright rooms;
- for daytime work;
- for some visual accessibility needs.

Current Omarchy custom-theme support explicitly supports a light-mode declaration in theme configuration.

Choose based on your environment, not internet fashion.

Screenshot 13.7-A
a light theme active (e.g. Flexoki Light)

13.8 — Change the Background

Current official shortcut:

```
Super + Ctrl + Space
```

opens the background selector for the active theme.

You can also use:

```
Super + Space
→ Style
→ Background
```

Themes can ship with multiple matching backgrounds.

13.9 — Themes and Backgrounds Are Separate

This distinction matters.

```
Theme
→ colors and coordinated application/system styling

Background
→ wallpaper image
```

You can keep the same theme and choose a different background.

Example:

```
Catppuccin
+
Waves background
```

The course workstation currently uses exactly that kind of separation.

13.10 — Personal Background Location

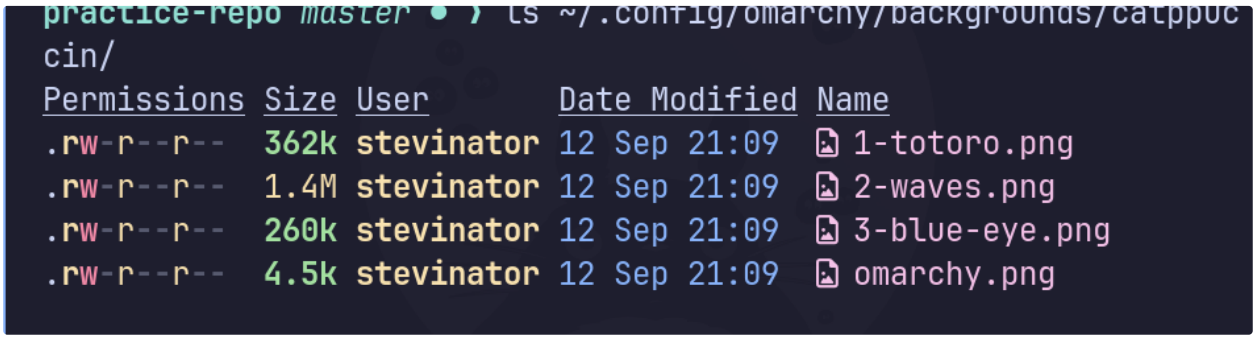
Current official Omarchy documentation states that extra backgrounds for a theme live under:

```
~/.config/omarchy/backgrounds/<theme-name>
```

Example:

```
~/.config/omarchy/backgrounds/nord
```

Any compatible image placed there can become part of the background choices for that theme.



Screenshot 13.10-A
custom background folder with images copied in

13.11 — Add a Personal Background Through the Menu

Current official workflow:

```
Super + Space
→ Install
→ Style
→ Background
```

This opens the relevant theme background location.

Then you can copy your image into that folder.

Afterward:

```
Super + Ctrl + Space
```

should include it in the background selector.

13.12 — Use Reasonable Background Images

A development wallpaper should not make the desktop harder to use.

Good:

```
moderate contrast
clean composition
not overly bright
```

Potentially distracting:

```
extreme detail
high-frequency patterns
very bright center behind windows
```

The wallpaper is visible less often in a tiling workflow, but it still affects the overall visual experience.

13.13 — Fonts

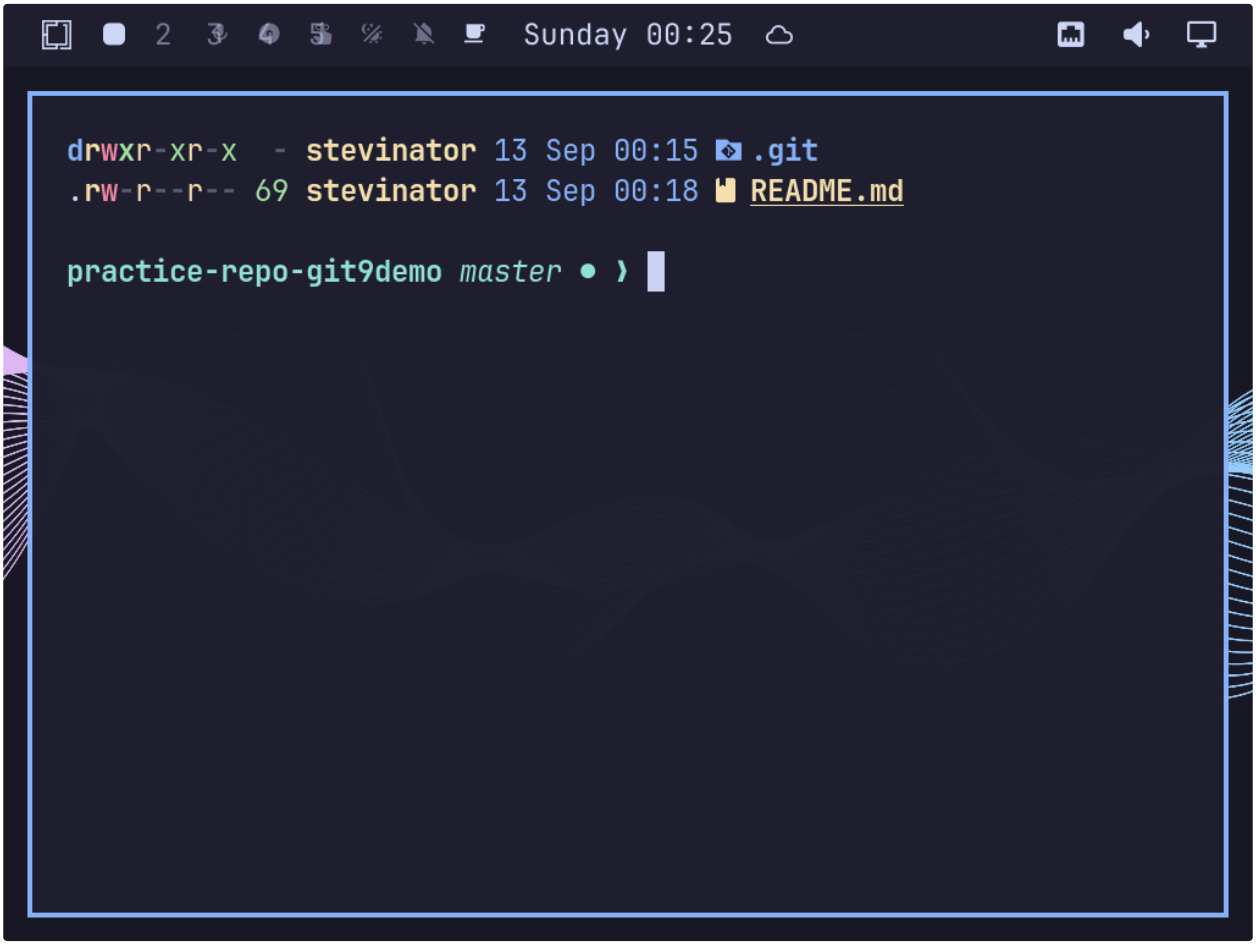
Current Omarchy official documentation states that the default font is:

```
JetBrainsMono Nerd Font
```

It is used as both:

- terminal font;
- system monospace font.

This makes it a strong default for developers.



Screenshot 13.13-A
JetBrainsMono Nerd Font rendering icon glyphs

13.14 — Why Nerd Fonts Matter

Nerd Fonts add extra icon glyphs commonly used by developer tools.

Examples include symbols used in:

- terminal prompts;
- status bars;
- file trees;
- Git indicators;
- Tmux;
- Neovim plugins.

Without a compatible Nerd Font, you may see:

```
□
?
missing glyph boxes
```

instead of icons.

That is why Omarchy installs Nerd Font variants of supported coding fonts.

13.15 — Change the Font

Current official workflow:

```
Super + Space
→ Style
→ Font
```

This changes the active monospace font across the relevant Omarchy-integrated surfaces.

Current documentation says this affects areas such as:

- terminal;
- bar;
- applications requesting the system monospace font.

13.16 — Discover Fonts Through the CLI

Current official Omarchy CLI documentation includes:

```
omarchy font list
```

to list available fonts.

Use:

```
omarchy font list
```

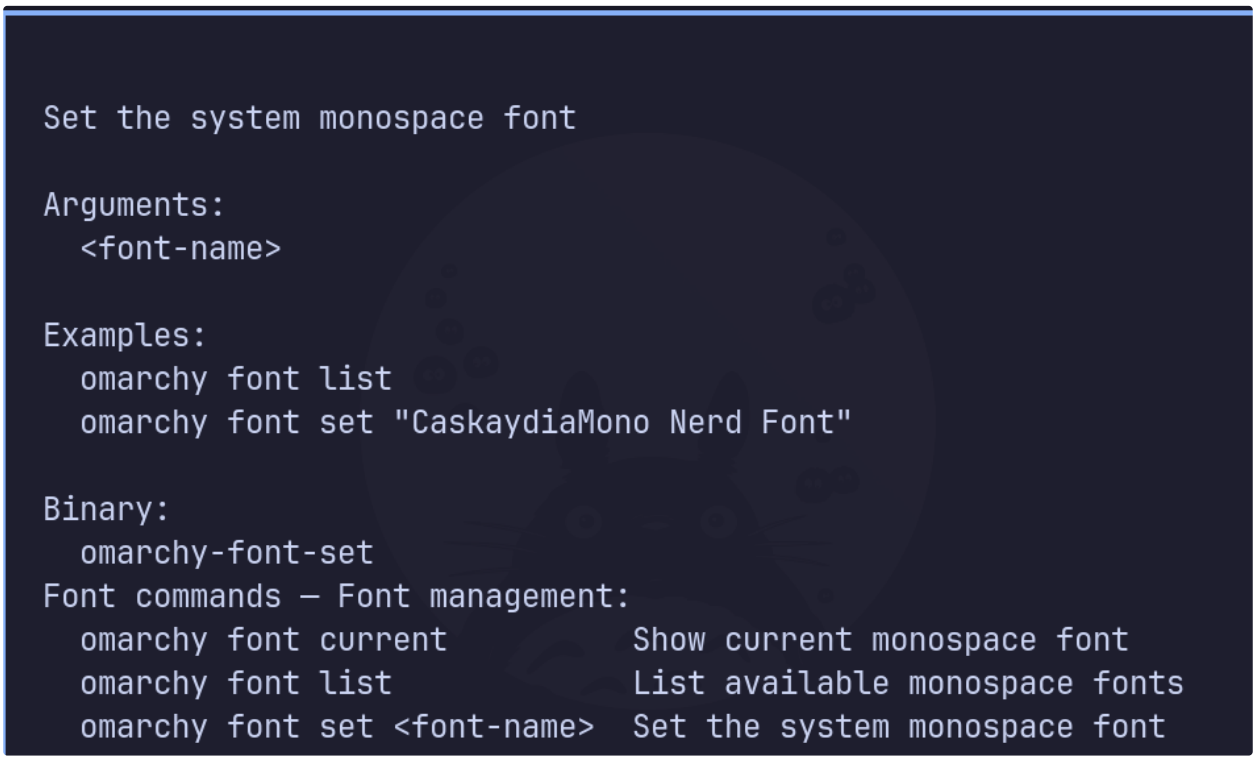
instead of assuming a static installed list.

Screenshot 13.16-A
`omarchy font list`

Check:

```
omarchy font --help
```

for the current font command surface on your installed version.



Screenshot 13.16-B
`omarchy font --help` command surface

13.17 — Install Additional Fonts

Current official Omarchy documentation lists one-click Nerd Font options through:

```
Super + Space
→ Install
→ Style
→ Font
```

Current documented options include:

- Cascadia Mono;
- Meslo LG Mono;
- Fira Code;
- Victor Code;
- Bitstream Vera Mono;
- Iosevka.

The exact available list may grow.

13.18 — Our Stevinator Font Baseline

The workstation used for this course currently uses:

```
JetBrainsMono Nerd Font
```

This is also the current Omarchy default.

We are keeping it because it is:

- readable;
- common;
- well-supported;
- developer-focused;
- compatible with Nerd Font glyphs.

No change is required simply because customization is possible.

13.19 — How to Choose a Coding Font

Evaluate:

```
0 vs O
1 vs l vs I
braces
parentheses
punctuation
small-size readability
bold readability
terminal glyph support
```

A coding font should reduce ambiguity.

Test with real code, not only a font preview page.

13.20 — Ligatures

Some coding fonts support ligatures.

For example:

```
!=
=>
===
```

may render as more visually connected glyphs.

Some developers love this.

Others prefer literal characters.

Ligatures are aesthetic, not necessary.

Choose what helps you read code.

13.21 — Font Size vs Monitor Scaling

These are different concepts.

Font/Text Size

Changes:

```
how large text appears
```

Monitor Scale

Changes:

```
how large the entire UI appears
```

Do not solve every text-size problem by changing monitor scale.

13.22 — Omarchy Text Size

Current Omarchy versions expose text-size control under:

```
omarchy display text size
```

Check current syntax:

```
omarchy display text size --help
```

or:

```
omarchy display --help
```

This is useful when:

```
display resolution is correct
UI scale is correct
but text is too small or large
```

13.23 — When to Change Scale Instead

Change monitor scale when the whole UI is physically too small or large because of display density.

Example:

```
13-inch 4K laptop
```

may need scaling.

A:

```
32-inch 1440p monitor
```

often works naturally at scale 1.

Use Module 11's monitor configuration principles.

13.24 — Avoid Solving the Same Problem in Three Places

A messy visual setup might use:

```
Hyprland scale
+
app-specific zoom
+
terminal font size
+
browser zoom
+
text size override
```

all to solve the same issue.

Prefer the lowest sensible layer.

Conceptually:

```
whole UI too small
→ monitor scale

text broadly too small
→ text/font setting

one app only
→ app-specific setting
```

13.25 — Theme Files

Current built-in Omarchy themes live under:

```
/usr/share/omarchy/themes
```

Your own themes belong under:

```
~/.config/omarchy/themes
```

This follows the same system-wide rule from earlier modules:

```
/usr/share/omarchy
→ Omarchy-owned

~/.config
→ user-owned
```

13.26 — Inspect the Built-In Themes

Run:

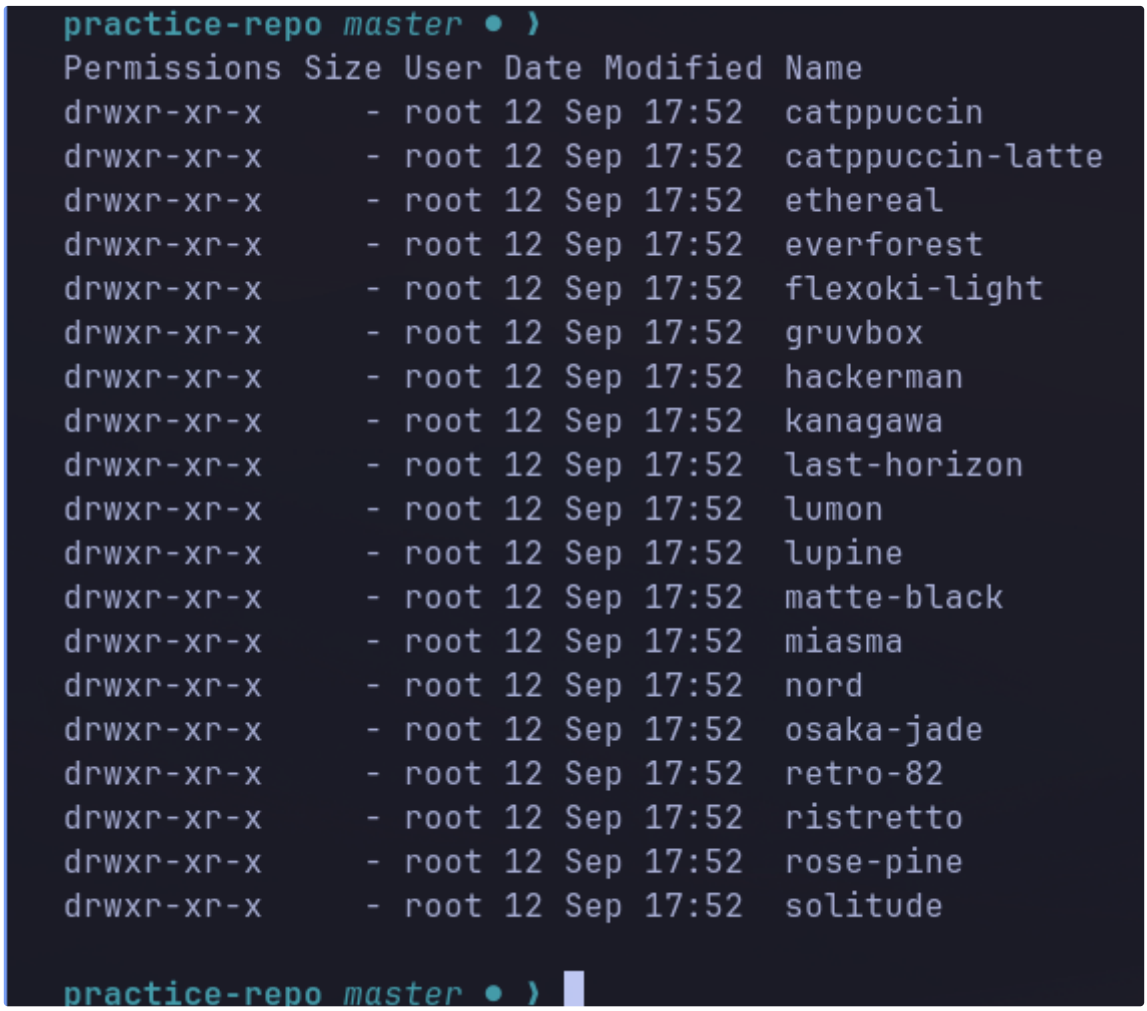
```
ls /usr/share/omarchy/themes
```

Then inspect one:

```
ls /usr/share/omarchy/themes/catppuccin
```

Do not edit the built-in theme.

Use it as a reference.



Screenshot 13.26-A
`/usr/share/omarchy/themes` directory listing

13.27 — `colors.toml`

Current official Omarchy theme documentation identifies:

```
colors.toml
```

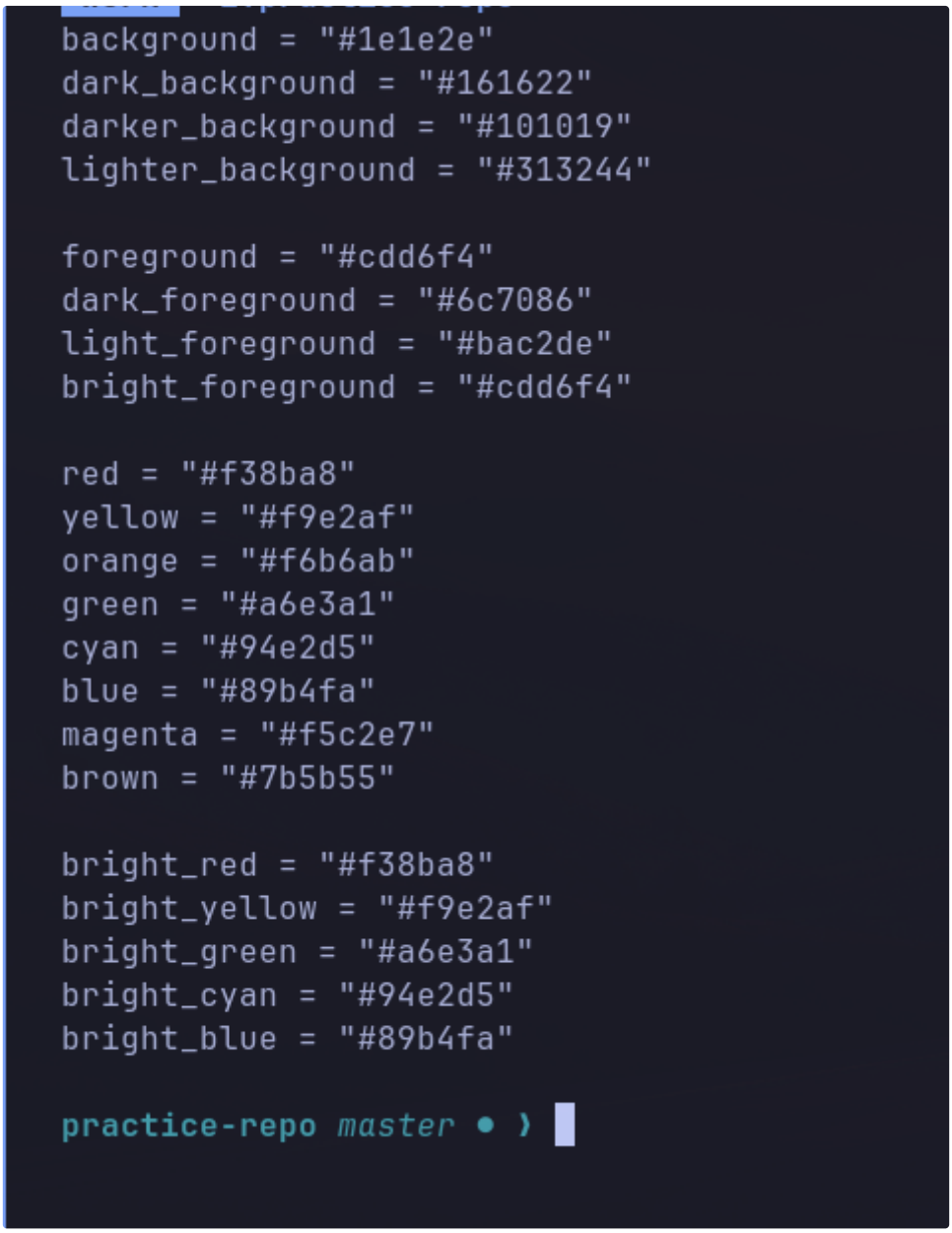
as the main custom-theme file.

It defines the color set Omarchy uses to generate configurations for multiple applications.

Conceptually:

```
colors.toml
        ↓
Omarchy theme generation
        ↓
terminal
Hyprland
Neovim
btop
Chromium
shell
supported editors
```

This is the core of Omarchy's coordinated theming model.



Screenshot 13.27-A
`colors.toml` content

13.28 — Semantic Theme Colors

Current Omarchy 4 has moved toward a semantic color system.

A theme defines foundational values such as:

```
background
foreground
accent
urgent
```

plus standard palette colors.

The Omarchy Shell can then map those into semantic UI roles.

This is much more maintainable than hard-coding unrelated colors in every component.

13.29 — Create a Custom Theme

Current official workflow:

```
1. copy an existing theme
2. place it under ~/.config/omarchy/themes
3. edit colors.toml
4. select the theme
```

Example:

```
cp -r /usr/share/omarchy/themes/catppuccin \
  ~/.config/omarchy/themes/stevinator
```

Then:

```
nvim ~/.config/omarchy/themes/stevinator/colors.toml
```

Do not do this unless you actually want to build a custom theme.

For most users, built-in themes are enough.

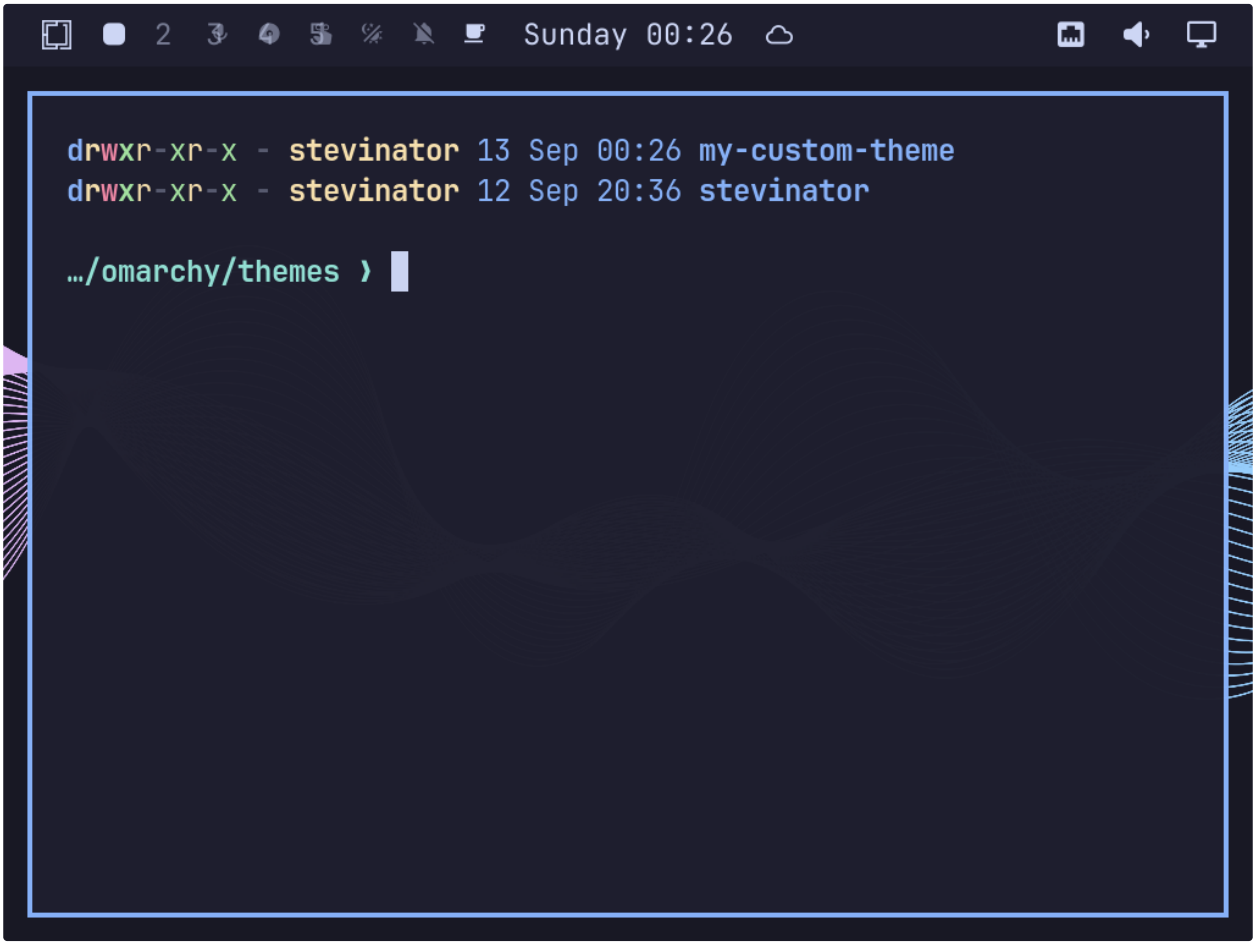
13.30 — User Themes Automatically Appear in the Theme Menu

Current official Omarchy documentation states that themes placed in:

```
~/.config/omarchy/themes
```

are included in the theme selection menu.

This gives you a clean user-owned customization layer.



Screenshot 13.30-A
`~/.config/omarchy/themes` with a user theme

13.31 — Aether

Current Omarchy includes an application called:

```
Aether
```

for creating themes visually.

The current official custom-theme documentation says it can be launched from the apps menu:

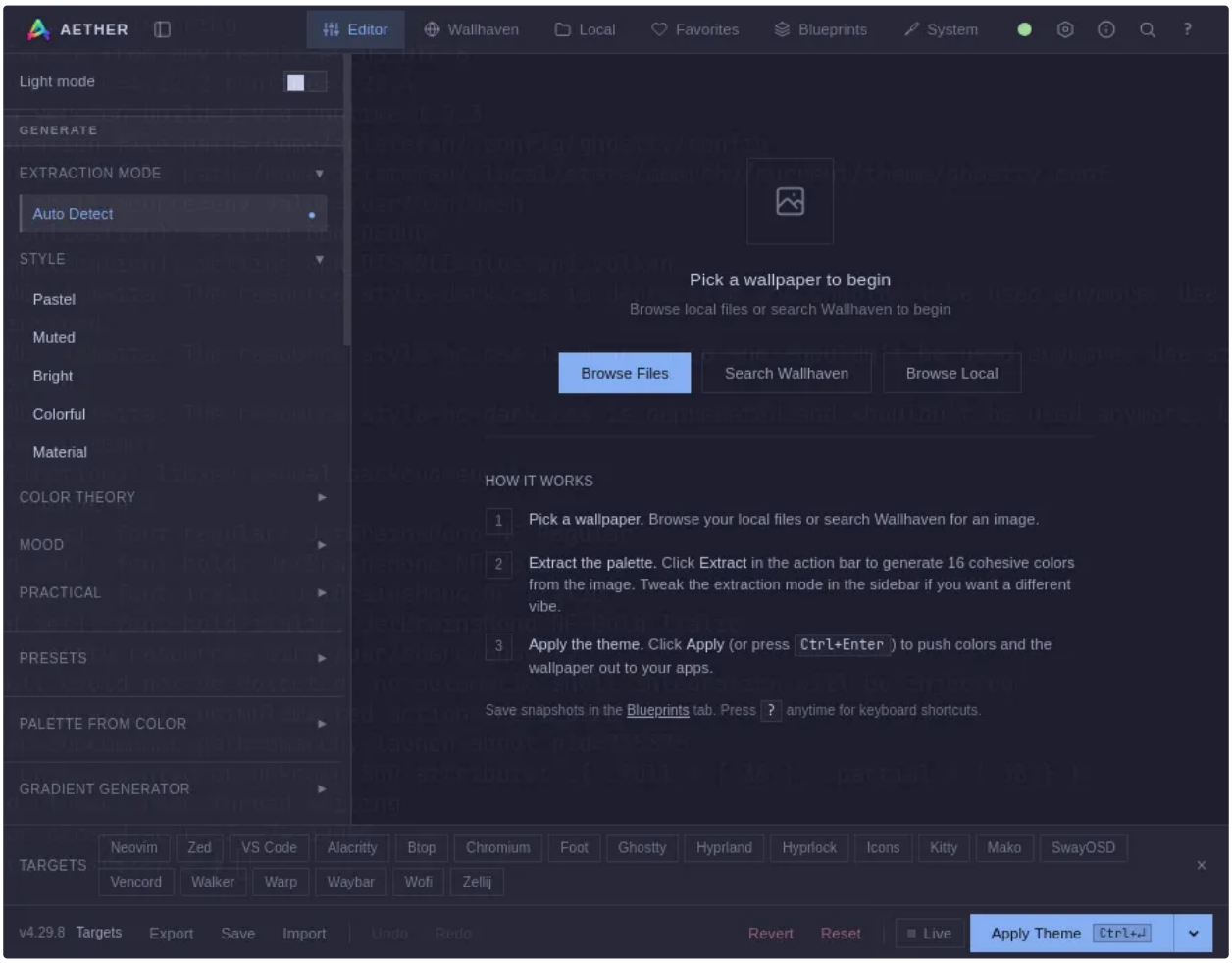
```
Super + Alt + Space
```

Aether allows you to:

- experiment with theme colors;
- search for backgrounds;

- create a custom theme through a GUI workflow.

This can be easier than editing `colors.toml` manually.



Screenshot 13.31-A
Aether theme editor with extraction controls and app targets

13.32 — Light Theme Declaration

Current official Omarchy custom-theme documentation supports:

```
mode = "light"
```

at the top of:

```
colors.toml
```

for light themes.

This lets Omarchy coordinate light-mode behavior across supported applications.

13.33 — Icon Themes

Current official custom-theme documentation says a theme can include:

```
icons.theme
```

to select a matching icon set.

Current built-in icon options are based around Yaru variants.

This allows file-manager icons to match the overall theme.

This is optional.

13.34 — Unlock Visuals

Current themes can include:

```
unlock.png
```

and:

```
preview-unlock.png
```

for boot-unlock/decryption visuals.

Current Omarchy exposes these under:

```
Style
→ Unlock
```

if the theme provides them.

This is an advanced cosmetic feature.

13.35 — Theme Safety

This is an important current Omarchy 4 behavior.

A local theme you created yourself can contain whatever configuration you deliberately put into it.

But when Omarchy installs a third-party theme repository, current official documentation says it strips files that could execute code or alter executable behavior.

Examples of files removed from third-party theme installs include:

- `.lua` files;
- terminal configs;
- `vscode.json`.

This is a security improvement.

13.36 — Why Theme Safety Matters

A theme downloaded from the internet should ideally change:

```
appearance
```

not:

```
what commands your terminal launches
```

Current Omarchy therefore distinguishes visual theme assets from files that could execute or trigger code.

This is a good security principle:

> ***Visual customization should not silently become code execution.***

13.37 — Third-Party Themes Still Require Judgment

Even with filtering, treat external themes as third-party content.

Good practice:

```
use trusted source
inspect repository
check recent maintenance
understand what is installed
```

Do not assume "theme" automatically means zero risk.

13.38 — Custom Theme Templates for Unsupported Apps

Current Omarchy allows you to theme applications not handled by default.

User templates live under:

```
~/.config/omarchy/themed/
```

A template filename ends with:

```
.tpl
```

Omarchy replaces theme placeholders when the active theme changes.

13.39 — Theme Template Variables

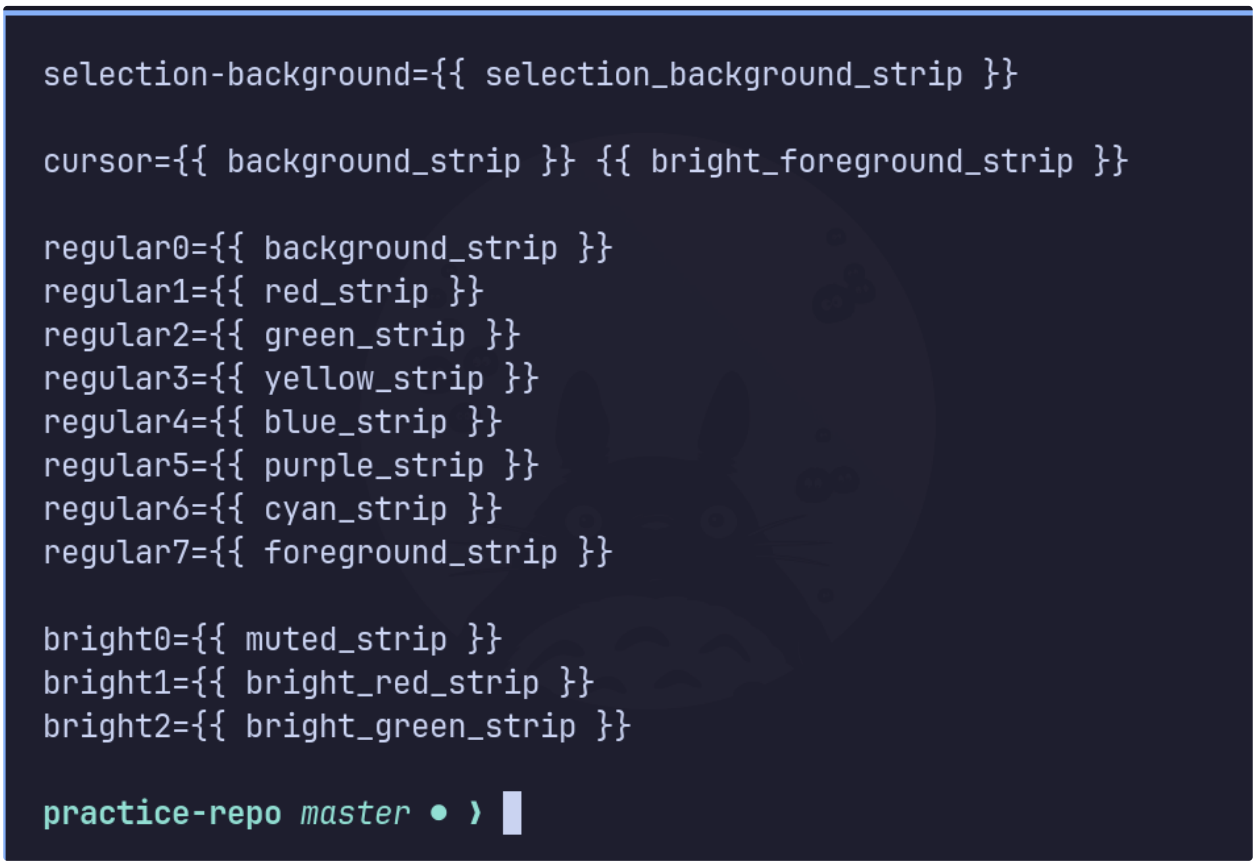
Current official documentation supports variables such as:

```
{{ background }}
{{ foreground }}
{{ accent }}
{{ red }}
{{ color0 }}
...
{{ color15 }}
```

The generated config then receives the active theme's colors.

Conceptually:

```
template
+
active theme colors
        ↓
generated app config
```



Screenshot 13.39-A
a real `.tpl` file with `{{ variable }}` placeholders

13.40 — Why Templates Are Powerful

Suppose an unsupported terminal utility has a config file:

```
~/.config/mytool/theme.toml
```

You could create an Omarchy template that regenerates it when the theme changes.

Now:

```
change Omarchy theme
↓
mytool changes with it
```

This keeps visual consistency beyond the built-in supported applications.

13.41 — Do Not Template Everything

Theme automation is useful when an app genuinely benefits from coordinated colors.

It is unnecessary for:

- apps where you already like the built-in theme;
- applications that manage themes internally;
- tools where theme generation would complicate maintenance.

Use this capability deliberately.

13.42 — Machine-Level Shell Overrides

Current Omarchy releases also include a user-level shell override file that can preserve machine-specific shell styling across theme changes.

Current release notes describe:

```
~/.config/omarchy/shell.toml
```

as a machine-level override merged over the active theme.

This can be useful for personal adjustments such as:

- font-related shell sizing;
- spacing;
- bar tweaks.

Because this is actively evolving, verify current documentation/source before relying on a specific key or schema.

13.43 — Theme vs Personal Override

A good model:

```
Theme
→ shared visual palette/style

Personal shell override
→ machine-specific adjustment
```

Example:

```
Catppuccin theme
+
slightly different personal bar spacing
```

You should not need to fork the theme just to tweak one machine-specific shell property.

13.44 — Consistency Across Development Tools

A coordinated developer theme can help:

- terminal syntax;
- Neovim;
- Git UI;
- shell;
- browser;
- notifications;

feel like one environment.

This reduces visual context switching.

But readability comes first.

13.45 — Check Contrast in the Terminal

When evaluating a theme, inspect:

```
normal text
comments
errors
warnings
Git diff
selected text
inactive pane
active pane
```

A beautiful screenshot can still be terrible for eight hours of coding.

13.46 — Check Contrast in Git Diffs

Open:

```
git diff
```

or:

```
lazygit
```

Check whether:

- additions are clearly visible;
- deletions are clearly visible;
- context remains readable.

Git diff readability is especially important in AI-assisted development.

13.47 — Check Neovim

Open a real source file.

Look at:

- comments;
- strings;
- keywords;
- current line;
- search highlights;
- diagnostics.

Do not judge a developer theme only from the desktop wallpaper.

13.48 — Check btop

Open:

```
Super + Ctrl + T
```

The current official themes manual explicitly includes btop among themed applications.

Check whether graphs and labels remain readable.

13.49 — Check the Omarchy Shell

Inspect:

- top bar;
- menus;
- notifications;
- OSD;
- lock screen.

The current theme should feel coherent across all of these.

13.50 — Practical Exercise: Compare Three Themes

Choose three current built-in themes.

For each:

1. switch theme;
2. open terminal;
3. open editor;
4. open btop;
5. open browser;
6. inspect top bar/menu;
7. evaluate readability.

Do not choose based only on wallpaper.

13.51 — Practical Exercise: Choose a Final Baseline

Pick one theme as your current workstation baseline.

Write down:

```
Theme:
Background:
Font:
Text size:
Monitor scale:
```

Example:

```
Theme: Catppuccin
Background: Waves
Font: JetBrainsMono Nerd Font
Text size: default
Monitor scale: 1
```

This becomes part of your reproducible workstation documentation.

13.52 — Practical Exercise: Change the Background Only

Keep the theme unchanged.

Press:

```
Super + Ctrl + Space
```

Choose a different background.

Observe:

```
theme colors remain
wallpaper changes
```

This reinforces the separation between theme and background.

13.53 — Practical Exercise: Inspect Fonts

Run:

```
omarchy font list
```

Then inspect:

```
Super + Space
→ Style
→ Font
```

If you want to experiment, install one alternative Nerd Font through:

```
Install
→ Style
→ Font
```

Try it temporarily.

Return to your preferred font afterward.

13.54 — Practical Exercise: Font Readability Test

Open a file containing:

```
0 O
1 l I
{} [] ()
!= == ===
=> ->
```

Compare two fonts.

Do not evaluate only the font's personality.

Evaluate actual code readability.

13.55 — Practical Exercise: Inspect Theme Structure

Run:

```
ls /usr/share/omarchy/themes
```

Choose your current theme:

```
ls /usr/share/omarchy/themes/YOUR_THEME
```

Inspect:

```
bat /usr/share/omarchy/themes/YOUR_THEME/colors.toml
```

Do not edit it.

The goal is to understand what drives the theme.

13.56 — Practical Exercise: Create a Test Theme Copy

Optional.

Copy a built-in theme into your user theme directory:

```
mkdir -p ~/.config/omarchy/themes
cp -r /usr/share/omarchy/themes/YOUR_BASE_THEME \
  ~/.config/omarchy/themes/stevinator-test
```

Inspect:

```
bat ~/.config/omarchy/themes/stevinator-test/colors.toml
```

Change one harmless color.

Then select the custom theme.

If you do not want to keep it, remove the user theme afterward.

13.57 — Practical Exercise: Add a Background

Find the active theme name.

Create its user background folder if needed:

```
mkdir -p ~/.config/omarchy/backgrounds/YOUR_THEME
```

Copy one appropriate image into it.

Then:

```
Super + Ctrl + Space
```

Verify that the new background appears.

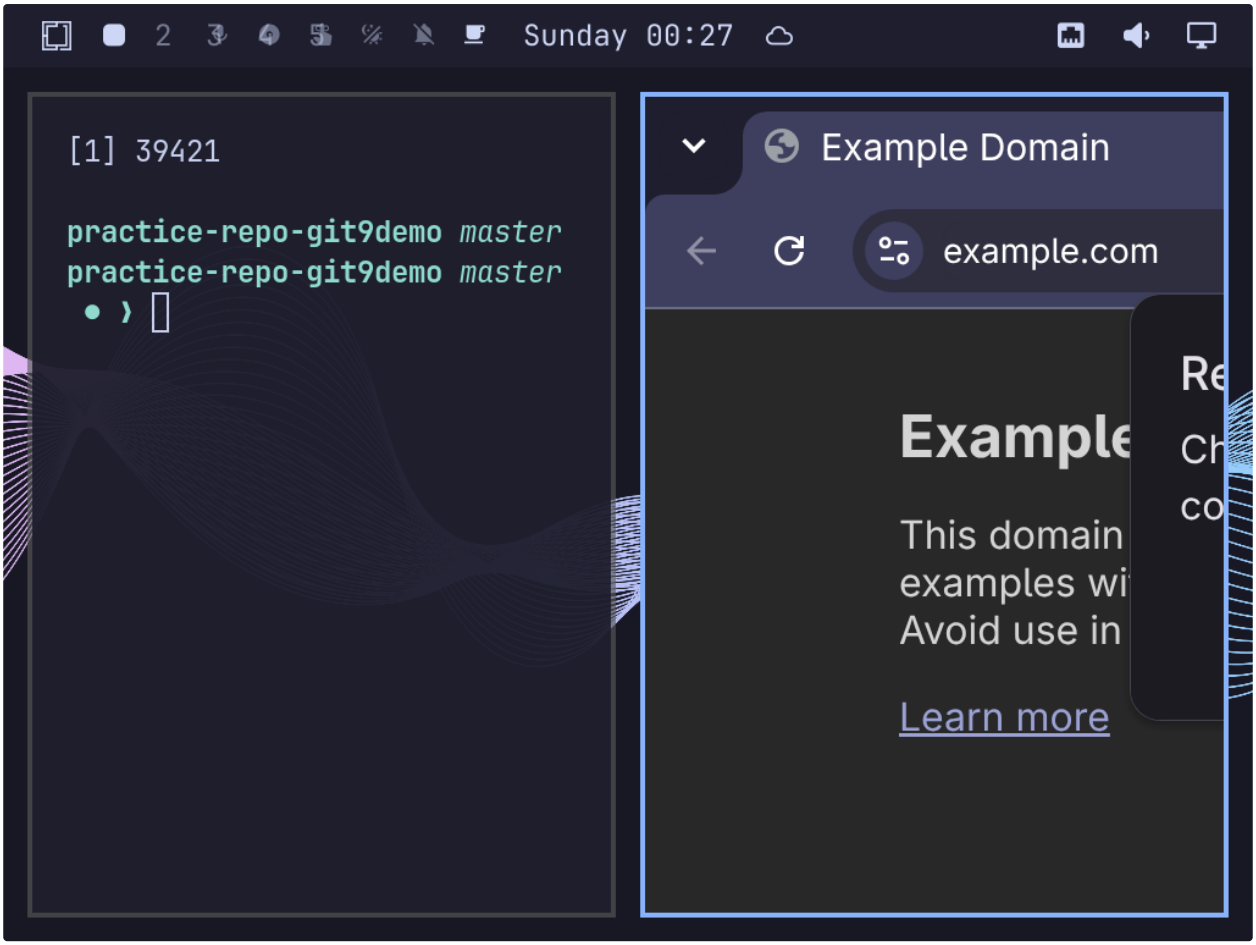
13.58 — Practical Exercise: Inspect Theme Integration

With your chosen theme active, open:

```
terminal
Neovim
btop
Chromium
Omarchy menu
```

Capture screenshots.

These become useful course assets demonstrating coordinated theme propagation.



Screenshot 13.58-A
coordinated terminal/editor/browser theming

13.59 — Common Visual Customization Mistakes

Mistake 1 — Customizing Every App Independently

You lose Omarchy's coordinated theme advantage.

Mistake 2 — Editing `/usr/share/omarchy/themes`

Those are package-owned themes.

Copy to:

```
~/.config/omarchy/themes
```

first.

Mistake 3 — Choosing a Theme Based Only on Wallpaper

Test code, diffs, terminal output, and notifications.

Mistake 4 — Installing Random Fonts Without Nerd Font Glyphs

You may break icons in terminal-oriented tools.

Mistake 5 — Fixing Tiny Text With Random Monitor Scaling

Understand:

```
font size
vs
text size
vs
monitor scale
```

Mistake 6 — Adding Huge Background Collections Everywhere

Keep your visual system intentional and manageable.

Mistake 7 — Treating Third-Party Themes as Automatically Safe

Current Omarchy filters executable theme content, but still use trusted sources and inspect external repositories.

Mistake 8 — Forking an Entire Theme to Change One Machine-Specific Shell Setting

Use user-level override mechanisms when available.

Mistake 9 — Over-Optimizing Aesthetics Before Ergonomics

This is a developer workstation.

Readability wins.

Mistake 10 — Forgetting to Document Your Chosen Visual Baseline

Theme/font/background choices are part of reproducibility.

13.60 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Themes

- [ ] I know how to open the theme selector.
- [ ] I understand that Omarchy themes style multiple applications.
- [ ] I can list themes from the current system.
- [ ] I know where built-in themes live.
- [ ] I know where personal themes live.

Backgrounds

- [ ] I know how to open the background selector.
- [ ] I understand that background and theme are separate.
- [ ] I know where personal theme backgrounds live.
- [ ] I can add a personal background.

Fonts

- [ ] I know the current Omarchy default is JetBrainsMono Nerd Font.
- [ ] I understand why Nerd Fonts matter.
- [ ] I can inspect available fonts.
- [ ] I know how to install additional supported fonts.
- [ ] I understand font size vs monitor scaling.

Custom Themes

- [ ] I understand the role of `colors.toml`.
- [ ] I know how a user theme is created.
- [ ] I understand the purpose of Aether.
- [ ] I know how unsupported apps can be themed with templates.
- [ ] I understand the current third-party theme safety model.

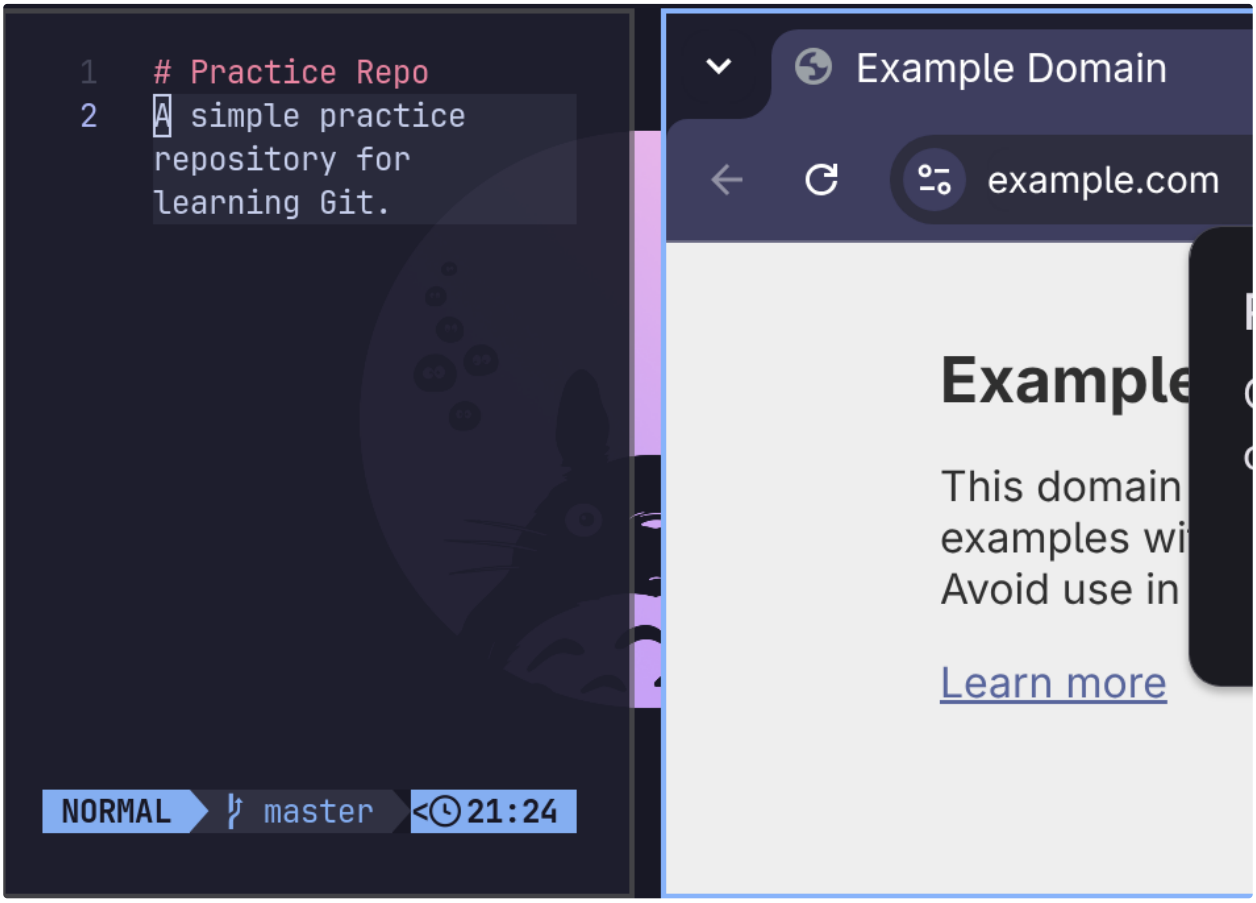
Design

- [ ] I prioritize readability over novelty.
- [ ] I evaluate themes using real code and Git diffs.
- [ ] I can document my final visual baseline.

13.61 — What You Can Now Do

After completing Module 13, you can now:

- choose an Omarchy theme intelligently;
- change backgrounds without changing the theme;
- select and install developer-friendly fonts;
- understand Nerd Font requirements;
- distinguish text size from monitor scaling;
- inspect how current themes are built;
- create your own theme safely;
- extend Omarchy's theming into unsupported applications;
- understand current third-party theme safety protections;
- create a coherent visual baseline for long development sessions.



Screenshot 13.61-A
final coherent workstation visual baseline

Most importantly:

> *Your workstation now looks intentional without becoming visually fragile or difficult to maintain.*

Module 13 Summary

Theme selector:

```
Super + Ctrl + Shift + Space
```

Background selector:

```
Super + Ctrl + Space
```

Theme CLI:

```
omarchy theme list
```

```
omarchy theme set <name>
```

Font discovery:

```
omarchy font list
```

Current default font:

```
JetBrainsMono Nerd Font
```

Built-in themes:

```
/usr/share/omarchy/themes
```

Personal themes:

```
~/.config/omarchy/themes
```

Personal backgrounds:

```
~/.config/omarchy/backgrounds/<theme>
```

Unsupported-app theme templates:

```
~/.config/omarchy/themed/*.tpl
```

Core theme file:

```
colors.toml
```

Theme model:

```
colors.toml
        ↓
generated coordinated configs
        ↓
shell
terminal
Hyprland
Neovim
btop
Chromium
supported editors
```

Visual sizing model:

```
whole interface too small
→ monitor scaling

text broadly too small
→ text/font sizing

one app only
→ app-specific setting
```

And the most important design rule:

> ***A good development theme disappears into the background and makes your work easier to read.***

MODULE 14

14

Plugins, Hooks, and Automation on Omarchy

You can now customize:

- Hyprland;
- keybindings;
- themes;
- fonts;
- backgrounds.

About This Module

You can now customize:

- Hyprland;
- keybindings;
- themes;
- fonts;
- backgrounds.

The next step is making Omarchy **react to events** and **extend itself**.

Current Omarchy 4 / Quattro provides two major extension mechanisms:

```
Hooks
→ run your scripts when specific Omarchy events occur

Shell plugins
→ extend or replace parts of the Omarchy desktop shell
```

These solve different problems.

A hook is ideal for:

```
"When X happens, run this script."
```

A shell plugin is ideal for:

```
"Add or replace an interactive piece of the desktop."
```

Examples:

```
post-update hook
→ notify me or run a health check

theme-set hook
→ update another application's theme

bar-widget plugin
→ add a custom widget to the top bar

panel plugin
→ add a custom floating panel

service plugin
→ run a headless shell service
```

This module is where your workstation begins to become programmable.

The goal is not to turn you into an Omarchy Shell/QML developer.

The goal is:

> ***Understand the extension model, automate simple events safely, install plugins carefully, and know where the security boundaries are.***

What You Will Learn

By the end of Module 14, you will understand:

- the difference between autostart, hooks, and plugins;
- where Omarchy hooks live;
- which current hook events Omarchy exposes;
- how hook scripts receive event arguments;
- how to install a hook;
- how to make a hook safe and non-disruptive;
- how the Omarchy Shell plugin architecture works at a high level;
- where built-in and third-party plugins live;
- how to list plugins;
- how to enable and disable plugins;
- how to add a plugin from Git;
- how plugin updates work;
- how to remove plugins;

- how to clone a built-in plugin before modifying it;
- what plugin manifests contain;
- the current plugin kinds;
- how to validate a plugin;
- how plugin state is stored in `shell.json`;
- why third-party plugins are security-sensitive;
- how to build a harmless automation exercise.

14.1 — Autostart vs Hooks vs Plugins

These three mechanisms are related but distinct.

Autostart

Use when:

> *Start this process every graphical login.*

Current location:

```
~/.config/hypr/autostart.lua
```

Example:

```
o.launch_on_start("my-service")
```

Hooks

Use when:

> *When this Omarchy event happens, run my script.*

Examples:

```
after boot
after update
after theme change
low battery
```

Plugins

Use when:

> *Extend or replace part of the Omarchy Shell.*

Examples:

```
top-bar widget
panel
overlay
menu
service
full replacement bar
```

14.2 — A Simple Decision Tree

Ask:

```
Should it run every login?
        ↓
yes
→ autostart

Should it run only when a specific event happens?
        ↓
yes
→ hook

Should it add/change interactive shell UI or shell services?
        ↓
yes
→ plugin
```

This distinction keeps configuration understandable.

14.3 — Current Hook Location

Current official Omarchy documentation stores user hooks under:

```
~/.config/omarchy/hooks/<event>.d/
```

Each event has its own directory.

Every executable file inside that directory runs when the event fires.

Conceptually:

```
~/.config/omarchy/hooks/
├── post-boot.d/
│   ├── 10-notify
│   └── 20-health-check
│
├── post-update.d/
│   └── 10-project-check
│
└── theme-set.d/
    └── 10-sync-my-app-theme
```

This directory-style model allows multiple independent automations to react to the same event.

14.4 — Current Official Hook Events

Current Quattro documentation lists these hook events:

```
post-boot
post-update
pre-refresh-pacman
theme-set
font-set
battery-low
```

Their current meanings are:

```
post-boot
→ after the desktop has started

post-update
→ during `omarchy update`, after packages and migrations

pre-refresh-pacman
→ before `omarchy refresh pacman` re-syncs package configuration

theme-set
→ after a theme change

font-set
→ after a font change

battery-low
→ when battery level becomes low
```

The exact set can evolve with Omarchy.

Inspect the current hook directories and documentation on your installed version.

14.5 — Event Arguments

Current official Omarchy documentation states that some hooks receive an argument.

Current examples:

```
theme-set
→ theme name in $1

font-set
→ font name in $1

battery-low
→ battery percentage in $1
```

That means a script can react differently depending on the event value.

Example:

```
#!/usr/bin/env bash

theme="$1"

printf 'Theme changed to: %s\n' "$theme"
```

14.6 — Inspect Existing Hook Samples

Current Omarchy hook directories include `.sample` files showing the intended structure.

Inspect:

```
find ~/.config/omarchy/hooks -maxdepth 2 -type f
```

or:

```
fd -H . ~/.config/omarchy/hooks
```

Read a sample:

```
bat ~/.config/omarchy/hooks/post-boot.d/*.sample
```

Use current shipped examples rather than copying an old internet snippet.

14.7 — Installing a Hook

Current official syntax:

```
omarchy hook install EVENT SCRIPT
```

Example:

```
omarchy hook install post-boot ~/my-hook
```

Current documentation says this copies the script into the correct hook location and makes it executable.

This is cleaner than manually guessing permissions and paths.

14.8 — Hook Script Basics

A hook is just an executable script.

Example:

```
#!/usr/bin/env bash

printf 'Omarchy boot hook ran at %s\n' "$(date)" >> "$HOME/.local/state/omarchy-hook.log"
```

Important parts:

```
shebang
→ tells Linux how to run the script

absolute/safe paths
→ reduce environment assumptions

logging
→ makes troubleshooting possible
```

14.9 — Make Hooks Fail Gracefully

A hook should not turn a minor automation failure into a broken workstation experience.

Prefer:

```
#!/usr/bin/env bash

if ! some-optional-command; then
  printf 'optional command failed\n' >> "$HOME/.local/state/my-hook.log"
  exit 0
fi
```

for non-critical automations.

The principle:

> ***A convenience hook should usually fail quietly and log clearly.***

14.10 — Avoid Interactive Commands in Hooks

Bad hook:

```
read -p "Continue?"
```

Why?

Hooks may run without a terminal attached.

Interactive prompts can hang or behave unexpectedly.

Prefer non-interactive commands.

14.11 — Avoid `sudo` in Automatic Hooks

A hook that fires automatically should not casually need root privileges.

Problems include:

- password prompts;
- hanging execution;
- unexpected privilege escalation;
- difficult debugging.

If automation genuinely requires privileged work, design it deliberately.

Do not sneak `sudo` into a convenience script.

14.12 — Log Useful Hook Output

A good pattern:

```
state_dir="$HOME/.local/state/my-hook"
mkdir -p "$state_dir"

{
  printf '=== %s ===\n' "$(date --iso-8601=seconds)"
  some-command
} >> "$state_dir/run.log" 2>&1
```

Now if something behaves strangely, you have evidence.

14.13 — Do Not Create Noisy Hooks

Bad:

```
post-boot
→ 25 notifications

post-update
→ 12 browser tabs

theme-set
→ rebuild half the workstation
```

Automation should reduce friction.

If it becomes more annoying than the manual action, remove it.

14.14 — Good Hook Use Cases

Useful examples:

```
post-boot
→ start a harmless user helper
→ run a quick environment sanity check
→ write a startup log

post-update
→ record current Omarchy version
→ run a non-destructive health check

theme-set
→ update an unsupported app theme

font-set
→ regenerate external editor config

battery-low
→ trigger an extra personal warning
```

14.15 — Poor Hook Use Cases

Avoid using hooks to:

- rewrite system files;
- reinstall large toolchains every boot;
- run destructive cleanup automatically;
- push Git repositories automatically;
- modify bootloader configuration;

- delete caches blindly;
- perform irreversible tasks without review.

The more dangerous the action, the less appropriate it is for implicit automation.

14.16 — Practical Hook Example: Post-Boot Log

Create:

```
mkdir -p ~/Projects/omarchy-hook-practice
nvim ~/Projects/omarchy-hook-practice/post-boot-log
```

Contents:

```
#!/usr/bin/env bash

state_dir="$HOME/.local/state/stevinator"
mkdir -p "$state_dir"

printf 'Omarchy desktop started: %s\n' "$(date --iso-8601=seconds)" \
  >> "$state_dir/post-boot.log"
```

Save.

Install:

```
omarchy hook install post-boot ~/Projects/omarchy-hook-practice/post-boot-log
```

This is intentionally harmless.

14.17 — Test Hook Logic Before Installing

Before installing:

```
bash ~/Projects/omarchy-hook-practice/post-boot-log
```

Then:

```
bat ~/.local/state/stevinator/post-boot.log
```

Only after it works manually should you connect it to an automatic event.

This is the same principle we use for keybindings:

```
test action first
→ wire automation second
```

14.18 — Theme Hook Example

A simple theme hook could log the chosen theme:

```
#!/usr/bin/env bash

theme="$1"

printf '%s %s\n' "$(date --iso-8601=seconds)" "$theme" \
  >> "$HOME/.local/state/theme-history.log"
```

When the theme changes, `$1` contains the current theme name according to the current official hook contract.

14.19 — Font Hook Example

Similarly:

```
#!/usr/bin/env bash

font="$1"

printf '%s %s\n' "$(date --iso-8601=seconds)" "$font" \
  >> "$HOME/.local/state/font-history.log"
```

This is a good exercise because it demonstrates event arguments without changing the system.

14.20 — Hooks and Git

If you create useful personal hooks, treat them as source code.

Put the source versions in your dotfiles repository later.

Do not rely only on:

```
the copy currently installed under ~/.config/omarchy/hooks
```

The reproducible source should live somewhere version-controlled.

14.21 — Omarchy Shell Plugins

Current Omarchy 4's desktop runs as a single long-lived Quickshell process:

```
omarchy-shell
```

The current official manual explains that much of what you see is implemented as plugins.

Examples include:

- the bar;
- drop-down panels;
- clipboard manager;
- emoji overlay;
- Omarchy menu;
- lock screen;
- polkit dialog;
- background services.

This means the shell itself is extensible.

14.22 — Built-In vs Third-Party Plugin Locations

Current official documentation defines:

```
Built-in plugins:
$OMARCHY_PATH/shell/plugins/
```

and user/third-party plugins:

```
~/.config/omarchy/plugins/
```

Typically:

```
$OMARCHY_PATH
→ /usr/share/omarchy
```

Built-ins belong to Omarchy.

User plugins belong to you.

14.23 — List Plugins

Current official command:

```
omarchy plugin list
```

This prints discovered plugins with information such as:

- id;
- enabled state;
- first-party vs third-party;
- kinds;
- display name.

For machine-readable output:

```
omarchy plugin list --json
```

Use this before changing plugin state.

14.24 — Plugin IDs

Current built-in plugin IDs use the reserved:

```
omarchy.
```

namespace.

Examples from current docs include:

```
omarchy.clock
omarchy.network
omarchy.notifications
```

Third-party plugins cannot claim the reserved built-in namespace.

This avoids collisions.

14.25 — Current Plugin Kinds

Current Omarchy plugin manifests can declare one or more kinds.

The current official plugin documentation lists:

```
bar-widget
panel
overlay
menu
service
bar
```

Conceptually:

```
bar-widget
→ one component in the active bar

panel
→ floating window/panel

overlay
→ fullscreen overlay

menu
→ summonable menu surface

service
→ headless singleton

bar
→ complete bar replacement
```

14.26 — Plugin Manifest

A third-party plugin is currently a Git repository with:

```
manifest.json
```

at its root.

A minimal conceptual manifest includes fields such as:

```
{
  "schemaVersion": 1,
  "id": "my.example.plugin",
  "name": "Example Plugin",
  "version": "1.0.0",
  "kinds": ["bar-widget"],
  "entryPoints": {
    "barWidget": "Widget.qml"
  }
}
```

Do not memorize the schema.

Use current official plugin documentation when creating one.

14.27 — Enable and Disable Plugins

Current official commands include:

```
omarchy plugin enable PLUGIN_ID
```

and:

```
omarchy plugin disable PLUGIN_ID
```

Examples:

```
omarchy plugin enable omarchy.tailscale
```

```
omarchy plugin disable omarchy.weather
```

You can also use:

```
Super + Space
→ Setup
→ Plugins
```

for plugin management.

14.28 — Plugin State Lives in `shell.json`

Current official documentation states that enabled/disabled shell state is stored in:

```
~/.config/omarchy/shell.json
```

This file controls things such as:

- bar selection;
- bar position;
- widget layout;
- enabled third-party plugins;
- disabled built-in plugins;
- idle/screen timing.

You normally do not need to hand-edit it for simple plugin management because the CLI/menu can update it.

14.29 — Third-Party Plugins Run Code

This is the single most important plugin security fact.

Current official Omarchy documentation explicitly warns that plugins run as:

> *arbitrary, unsandboxed code inside the long-running shell process.*

They have the same general user-level file/process access as the shell.

That means a plugin is **not just a visual theme**.

It is executable code.

14.30 — Plugin Security Model

Current Omarchy does apply capability-scoped interfaces to third-party plugins and protects some sensitive shell services.

But the official documentation still says:

```
plugin code runs with user-level access
```

and visual plugins share the QML object scene.

Therefore:

> ***Only install plugin repositories you are willing to run as your user account.***

14.31 — Review Before Enabling

Current plugin installation flow is deliberately cautious.

The official documentation states that adding a plugin:

- warns you that it runs code;
- shows the repository URL;
- asks for confirmation;
- clones and validates the manifest;
- does not run an install hook;
- does not ask for sudo.

Without explicit enablement, you can inspect the code before activating it.

This is the correct workflow.

14.32 — Add a Plugin from Git

Current official syntax:

```
omarchy plugin add REPOSITORY_URL
```

Example form:

```
omarchy plugin add https://github.com/example/omarchy-weather.git
```

Current documentation also supports:

```
--enable
```

when you deliberately want to enable it during installation.

For learning:

> *Add first, inspect second, enable third.*

14.33 — Where Third-Party Plugins Are Installed

Current official location:

```
~/.config/omarchy/plugins/<id>/
```

Inspect after adding:

```
ls ~/.config/omarchy/plugins
```

Then:

```
bat ~/.config/omarchy/plugins/PLUGIN_ID/manifest.json
```

Review the code before enabling.

14.34 — Plugin Updates

Current official command:

```
omarchy plugin update PLUGIN_ID
```

or all Git-managed plugins:

```
omarchy plugin update
```

Current documentation states that updates:

- fetch the current repository;
- show the diff;
- use a fast-forward update;
- refuse problematic local-change situations;
- validate the new revision;
- roll back if validation fails.

The important habit:

> *Read the diff before accepting code updates to a plugin.*

14.35 — Remove a Plugin

Current command:

```
omarchy plugin remove PLUGIN_ID
```

Current official documentation says removal first disables the plugin.

Git-managed plugin checkouts can then be removed.

For handmade plugin folders, current behavior is more cautious and can move them to a timestamped backup instead of blindly deleting them.

14.36 — Clone a Built-In Plugin Before Modifying It

Do not edit built-in plugin files under:

```
$OMARCHY_PATH
```

Current official workflow:

```
omarchy plugin clone omarchy.clock
```

This copies the built-in into:

```
~/.config/omarchy/plugins/<username>.clock/
```

renames it as your copy, enables it, and switches shell behavior to the clone.

This is the plugin equivalent of:

> ***Override Omarchy; do not overwrite Omarchy.***

14.37 — Edit the Clone

Current official command can also use:

```
omarchy plugin clone omarchy.clock --edit
```

to open the cloned plugin in your configured editor.

Changes under:

```
~/.config/omarchy/plugins/
```

are currently hot-reloaded by the shell.

That creates a quick edit/observe workflow.

14.38 — If the Clone Breaks

Current official behavior allows you to remove the clone:

```
omarchy plugin remove YOUR_ID
```

and return to the built-in plugin.

This makes experimentation significantly safer than editing package-owned shell code.

14.39 — Validate a Plugin

Current official command:

```
omarchy plugin validate ./my-plugin
```

The current validator checks areas including:

- schema version;
- required fields;
- reserved IDs;
- valid/safe entry-point paths;
- entry-point existence;
- required entry points for declared kinds;
- symlink restrictions.

Run validation before sharing or enabling a plugin you are developing.

14.40 — Plugin Development Language

Current Omarchy Shell plugins use:

```
QML
```

inside the Quickshell-based desktop.

A typical plugin therefore includes:

```
manifest.json
+
one or more .qml files
```

You do not need QML knowledge for normal Omarchy use.

Plugin development is an advanced track.

14.41 — Hot Reload

Current official documentation states that saving files anywhere under:

```
~/.config/omarchy/plugins/
```

automatically reloads plugin code.

That makes local plugin development interactive.

However:

> *A fast reload loop makes it easier to experiment, not safer to run untrusted code.*

Security rules still apply.

14.42 — Full Bar Plugins

A plugin of kind:

```
bar
```

replaces the entire active Omarchy bar.

Current official documentation says there is always exactly one active bar.

Enabling a full-bar plugin therefore replaces the current bar rather than adding another one.

This is a much bigger change than enabling one bar widget.

14.43 — Bar Widgets

A:

```
bar-widget
```

is a component placed into the active bar.

Current manifests may define a default section such as:

```
left
center
right
```

and whether multiple instances are allowed.

Current Omarchy also exposes dedicated bar-management commands, but those belong more to shell customization than this module's core goal.

14.44 — Plugins vs Hooks

A plugin should **not** be your default solution for simple automation.

Example:

```
After update, write version to a log
```

Use a hook.

Do not create a QML service plugin.

Conversely:

```
Add a new live top-bar widget
```

Use a plugin.

Do not continuously rebuild it through a shell hook.

Use the simplest correct mechanism.

14.45 — Plugins vs Autostart

If you merely need:

```
start syncthing wrapper every login
```

use autostart or an appropriate systemd user service.

Do not make a shell plugin just to launch a background process unless it genuinely needs shell/plugin integration.

14.46 — Practical Exercise: List Current Plugins

Run:

```
omarchy plugin list
```

Identify:

- one built-in bar widget;
- one panel;
- one service;
- one disabled plugin if present.

Then:

```
omarchy plugin list --json | head
```

to see that structured output is available.

14.47 — Practical Exercise: Use the Plugin Menu

Open:

```
Super + Space
→ Setup
→ Plugins
```

Inspect the available actions.

Current documentation includes:

```
Enable
Disable
Add
Clone
Remove
```

Do not change anything yet.

14.48 — Practical Exercise: Toggle a Non-Essential Built-In Plugin

Choose a harmless plugin that is not essential to your workflow.

Disable:

```
omarchy plugin disable PLUGIN_ID
```

Observe the shell.

Re-enable:

```
omarchy plugin enable PLUGIN_ID
```

The goal is to understand the plugin state model.

Do not disable critical lock/authentication functionality merely for practice.

14.49 — Practical Exercise: Inspect `shell.json`

Run:

```
bat ~/.config/omarchy/shell.json
```

Observe current sections such as:

```
bar
plugins
disabledPlugins
idle
```

Exact contents depend on your customizations.

Do not rewrite this file unless you understand the schema.

14.50 — Practical Exercise: Harmless Post-Boot Hook

Create and manually test the post-boot logging hook from earlier.

Install it:

```
omarchy hook install post-boot ~/Projects/omarchy-hook-practice/post-boot-log
```

After the next session startup, inspect:

```
bat ~/.local/state/stevinator/post-boot.log
```

This completes the module's core automation exercise.

14.51 — Practical Exercise: Theme Event Hook

Create a harmless script that records `$1`.

Install it for:

```
theme-set
```

Change theme once.

Inspect the resulting log.

Then return to your preferred theme.

This demonstrates event arguments safely.

14.52 — Practical Exercise: Inspect a Plugin Before Enabling

If you choose to install a third-party plugin for practice:

1. use a repository you intentionally selected;
2. add it without forcing automatic enablement;
3. inspect:
   - `manifest.json`;
   - QML files;
   - shell/process calls;
4. validate it;
5. only enable if you are comfortable running it.

Do not install arbitrary plugins merely to complete the exercise.

Inspection without installation is valid.

14.53 — Practical Exercise: Clone a Built-In Plugin

Optional advanced exercise.

Run:

```
omarchy plugin clone omarchy.clock --edit
```

Inspect:

```
manifest.json
QML files
```

Make no changes initially.

Observe where the clone was created.

Then remove your clone if you do not want to maintain it.

This demonstrates the correct way to customize a built-in plugin.

14.54 — Practical Exercise: Validate a Local Plugin Folder

If you cloned or created one:

```
omarchy plugin validate ~/.config/omarchy/plugins/YOUR_PLUGIN
```

Read validation output.

Do not interpret successful schema validation as a security audit.

It only means the plugin satisfies structural rules.

14.55 — Common Hook Mistakes

Mistake 1 — Putting an Event Script in Autostart

If it should only run after updates, use:

```
post-update
```

not login autostart.

Mistake 2 — Writing Interactive Hooks

Hooks should normally be non-interactive.

Mistake 3 — Using `sudo` in Automatic Hooks

Avoid unattended privilege escalation.

Mistake 4 — Failing Silently With No Log

If a hook matters, make troubleshooting possible.

Mistake 5 — Doing Destructive Cleanup Automatically

Automation should not casually remove data.

14.56 — Common Plugin Mistakes

Mistake 1 — Treating Plugins Like Themes

Plugins execute code.

They are not merely visual assets.

Mistake 2 — Editing Built-In Plugin Files

Clone first:

```
omarchy plugin clone ...
```

Mistake 3 — Enabling an Unknown Plugin Before Reading It

Review first.

Mistake 4 — Assuming Plugin Validation Means "Safe"

Validation checks structure, not intent.

Mistake 5 — Auto-Updating Plugins Without Reading Diffs

Current Omarchy exposes the diff for a reason.

Mistake 6 — Building a Plugin for a Problem a Hook Can Solve

Use the simplest extension mechanism.

Mistake 7 — Using a Plugin for a Normal Background Service

Use autostart/systemd where appropriate.

Mistake 8 — Forgetting That Plugins Share User-Level Access

A malicious plugin can potentially access files your user account can access.

14.57 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Architecture

- [ ] I understand the difference between autostart, hooks, and plugins.
- [ ] I know when each mechanism is appropriate.
- [ ] I understand that Omarchy Shell runs as a long-lived Quickshell process.

Hooks

- [ ] I know where hooks live.
- [ ] I know the currently documented hook events.
- [ ] I understand event arguments such as `$1`.
- [ ] I can install a hook with `omarchy hook install`.
- [ ] I know how to make a hook non-interactive and safe.
- [ ] I know how to log hook output.

Plugins

- [ ] I can list current plugins.
- [ ] I know where built-in plugins live.
- [ ] I know where user plugins live.
- [ ] I can enable and disable a plugin.
- [ ] I know how to add a plugin from Git.
- [ ] I understand plugin updates and removal.
- [ ] I know how to clone a built-in before modifying it.
- [ ] I understand the current plugin kinds.
- [ ] I can validate a plugin.

Security

- [ ] I understand that third-party plugins execute unsandboxed user-level code.
- [ ] I know to review plugin code before enabling it.
- [ ] I understand that validation is not a security review.

- [ ] I know not to grant hooks/plugins unnecessary `sudo` behavior.

14.58 — What You Can Now Do

After completing Module 14, you can now:

- automate Omarchy events safely;
- create post-boot and post-update workflows;
- react to theme/font/battery events;
- keep hook behavior logged and debuggable;
- inspect the Omarchy Shell plugin system;
- enable and disable shell components;
- install and update third-party plugins carefully;
- clone built-in plugins before customizing them;
- distinguish UI extensions from simple automation;
- reason about plugin security;
- build a workstation that reacts to your workflow without becoming fragile.

Most importantly:

> *You can now extend Omarchy without editing its source and automate it without hiding dangerous behavior.*

Module 14 Summary

Use:

```
autostart
→ every session

hook
→ event-driven script

plugin
→ shell/UI/service extension
```

Hook location:

```
~/.config/omarchy/hooks/<event>.d/
```

Current documented events:

```
post-boot
post-update
pre-refresh-pacman
theme-set
font-set
battery-low
```

Install hook:

```
omarchy hook install post-boot ~/my-hook
```

Plugin discovery:

```
omarchy plugin list
```

Plugin control:

```
omarchy plugin enable PLUGIN_ID
omarchy plugin disable PLUGIN_ID
```

Install:

```
omarchy plugin add REPOSITORY_URL
```

Update:

```
omarchy plugin update PLUGIN_ID
```

Remove:

```
omarchy plugin remove PLUGIN_ID
```

Clone built-in:

```
omarchy plugin clone omarchy.clock --edit
```

Validate:

```
omarchy plugin validate ./my-plugin
```

Plugin kinds:

```
bar-widget
panel
overlay
menu
service
bar
```

User plugins:

```
~/.config/omarchy/plugins/
```

Shell state:

```
~/.config/omarchy/shell.json
```

And the most important security rule:

> ***A third-party Omarchy plugin is executable code running as your user. Review it like code, not like a wallpaper.***

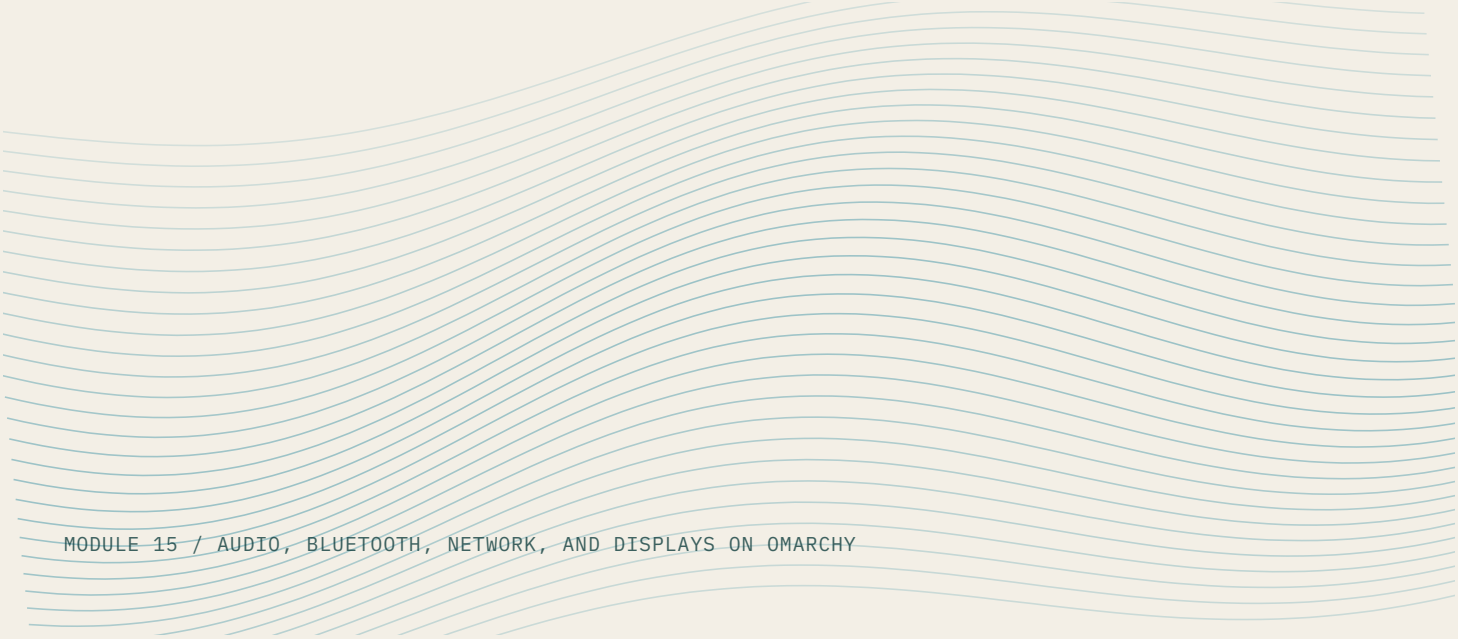
MODULE 15

15

Audio, Bluetooth, Network, and Displays on Omarchy

A development workstation is only useful if the hardware layer is reliable.

You need:



About This Module

A development workstation is only useful if the hardware layer is reliable.

You need:

- working audio;
- predictable Bluetooth;
- stable Wi-Fi/networking;
- correct monitor configuration;
- usable brightness and scaling.

Current Omarchy 4 / Quattro has moved a large part of this day-to-day hardware control into the **Omarchy Shell panels** and corresponding CLI groups.

Current official documentation provides dedicated panels for:

```
Audio
Network
Bluetooth
Display
Power
```

with direct shortcuts such as:

```
Super + Ctrl + A
Super + Ctrl + W
Super + Ctrl + B
Super + Ctrl + D
```

The goal of this module is not to turn you into an audio-stack or NetworkManager expert.

The goal is:

> ***Operate the hardware confidently, use the Omarchy layer first, restart only the affected subsystem when something breaks, and know when to inspect deeper.***

What You Will Learn

By the end of Module 15, you will understand:

- how the Omarchy Shell exposes hardware controls;
- how to use the audio panel;

- how to switch output devices;
- how to control microphone/input;
- how per-app volume works;
- how to restart audio without rebooting;
- how to inspect laptop speaker tuning;
- how to use the Bluetooth panel;
- how to pair, connect, disconnect, forget, and power Bluetooth;
- how to restart Bluetooth;
- how to use the network panel;
- how to inspect Wi-Fi signal and status;
- how to connect to networks;
- how to use DNS selection;
- how to use Wi-Fi QR sharing;
- how to inspect/change Wi-Fi band behavior;
- how to run network speed tests;
- how to restart networking;
- how to use the display panel;
- how brightness controls differ from monitor configuration;
- how to inspect text-size/display CLI controls;
- how to verify monitor state with `hyprctl`;
- how to distinguish display brightness, scale, resolution, refresh rate, and text size;
- why subsystem restart should come before rebooting.

15.1 — Omarchy's Hardware Control Philosophy

Current Omarchy 4 routes common desktop hardware tasks through the Omarchy Shell.

That includes:

```
Audio
Network
Bluetooth
Display
Power
```

This is intentional.

Instead of opening several separate TUIs or random vendor utilities, the workstation provides integrated shell panels.

Conceptually:

```
top bar / hotkey
        ↓
Omarchy Shell panel
        ↓
underlying Linux subsystem
```

This gives you a consistent user experience while still leaving the underlying Linux tools available when deeper troubleshooting is required.

15.2 — Current Hardware Panel Shortcuts

Current official Omarchy hotkeys:

```
Super + Ctrl + A
→ Audio panel

Super + Ctrl + W
→ Network panel

Super + Ctrl + B
→ Bluetooth panel

Super + Ctrl + D
→ Display panel

Super + Ctrl + P
→ Power panel
```

These panels are also accessible by clicking their corresponding icons in the top bar.

15.3 — The Top-Bar Hardware Widgets Are Interactive

Current official Omarchy top-bar documentation states that the hardware widgets are not just indicators.

They are controls.

Examples:

```
Audio icon
left-click
→ audio panel

right-click
→ mute

scroll
→ volume
```

```
Bluetooth icon
left-click
→ Bluetooth panel

right-click
→ toggle Bluetooth radio
```

```
Display icon
left-click
→ display panel

scroll
→ brightness
```

This is worth learning because it makes routine adjustments very fast.

15.4 — Use the Panel First

For normal daily tasks, prefer:

```
panel
```

over:

```
low-level command experimentation
```

Examples:

```
Need new audio output?
→ Audio panel

Need Wi-Fi network?
→ Network panel

Need to reconnect headphones?
→ Bluetooth panel

Need brightness?
→ Display panel
```

The lower-level tools remain valuable for diagnostics.

15.5 — Audio Panel

Open:

```
Super + Ctrl + A
```

Current official Omarchy documentation says the audio panel includes:

- master volume;
- output-device selection;
- per-application mixer.

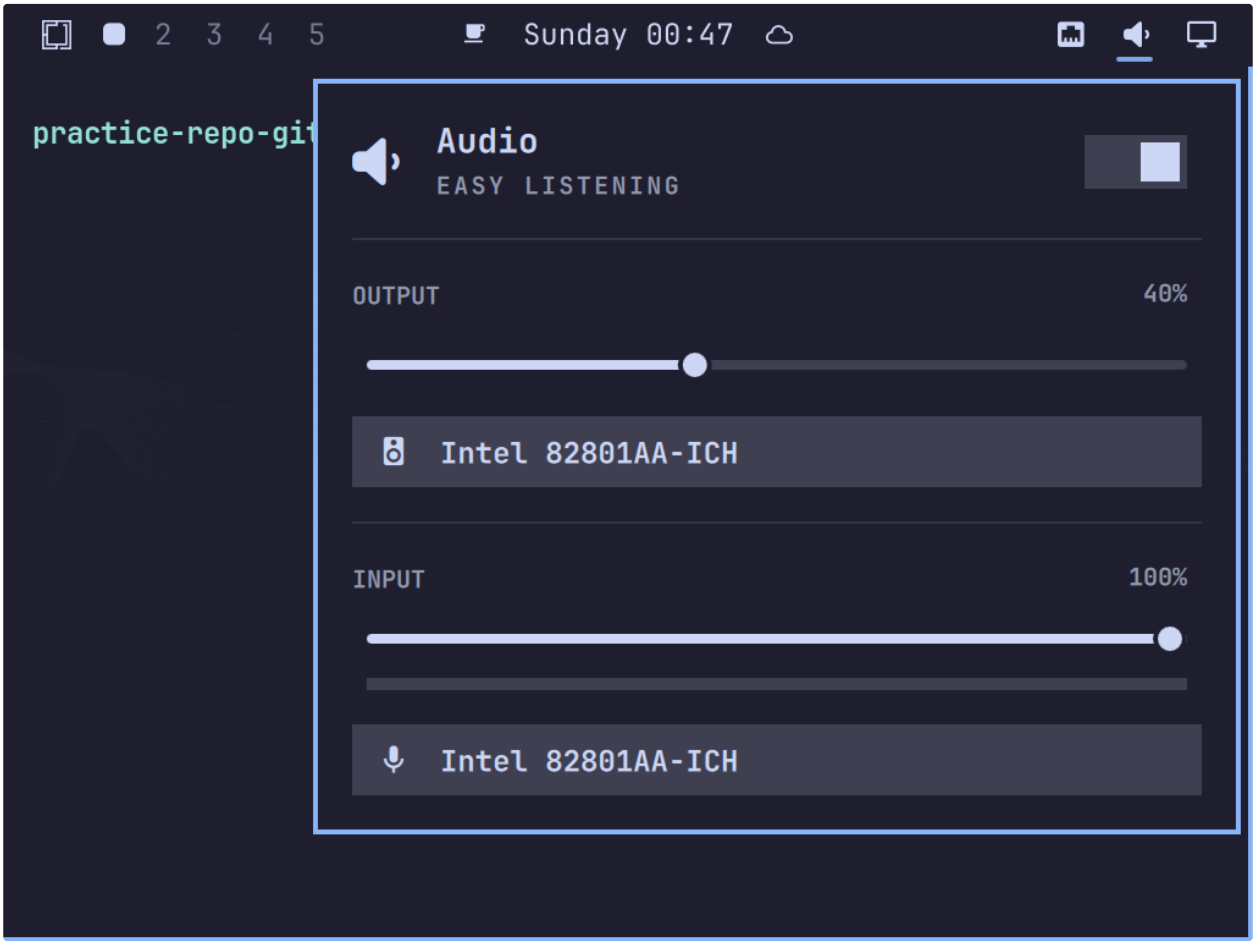
This means you can control both:

```
global audio
```

and:

```
individual application streams
```

from one panel.



Screenshot 15.5-A
Omarchy Audio panel with output picker and per-app mixer

15.6 — Master Volume

The master volume changes the current output level.

You may also use hardware media keys.

Current official hotkeys document:

```
Alt + Volume Up/Down
```

for precise 1% volume changes.

Normal hardware volume keys can perform the usual larger-step adjustments.

15.7 — Mute

The current top-bar audio widget supports right-click mute.

Hardware mute keys can also work where present.

The important distinction is:

```
output mute
vs
microphone mute
```

Do not confuse the two.

15.8 — Microphone / Input

Current Omarchy shell includes microphone/input controls.

If the microphone widget is enabled on the bar, current official docs state that it supports:

```
left click
→ mute microphone

middle click
→ audio panel

scroll
→ input volume
```

Even when the microphone widget is not visible, input control is available through the audio subsystem/CLI.

15.9 — Output Device Selection

The most common audio problem on a multi-device workstation is:

```
sound is playing
but from the wrong device
```

Examples:

- HDMI monitor;
- USB headset;
- Bluetooth headphones;
- built-in speakers;
- external speakers.

Current official troubleshooting guidance says the first thing to check is whether the correct output is selected in the audio panel.

Do that before assuming the audio stack is broken.

15.10 — Per-Application Mixing

Current Omarchy's audio panel includes a per-app mixer.

This lets you do things like:

```
Browser
→ 20%

Music player
→ 60%

Game/dev tool
→ 40%
```

without changing master volume.

This is especially useful during development when:

- a browser tab is noisy;
- a meeting app needs to stay audible;
- a local dev app produces media.

15.11 — Switching Output While Audio Is Playing

Current Quattro audio behavior is designed to preserve active playback when switching outputs.

This matters when moving between:

```
speakers
→ headset
→ monitor
```

The shell/CLI handles stream movement rather than forcing you to restart applications manually.

15.12 — Audio CLI

Current Omarchy CLI exposes an:

```
omarchy audio
```

group.

Inspect:

```
omarchy audio --help
```

This is the correct first discovery method because exact subcommands can evolve.

Current workstation behavior has included controls for:

- output;
- input;
- mute;
- source switching;
- output switching;
- tuning.

15.13 — Inspect Current Audio State

Start with:

```
omarchy audio --help
```

Then inspect relevant current subcommands.

Do not memorize old syntax if the installed help has changed.

The course rule remains:

> ***Use current CLI discovery before guessing subcommands.***

15.14 — Precise Volume Control

Current official Omarchy hotkeys include:

```
Alt + Volume Up
Alt + Volume Down
```

for 1% volume increments.

This is useful when:

```
normal volume steps feel too coarse
```

15.15 — Switch to the Next Audio Output

Current official hotkeys include:

```
Shift + Mute
```

to switch to the next audio output.

This is very convenient when you frequently alternate between:

```
headphones
speakers
monitor
```

15.16 — Audio Restart Before Reboot

Current official Omarchy troubleshooting guidance explicitly recommends restarting the affected subsystem before rebooting.

If sound suddenly stops:

```
do not reboot first
```

Use the Omarchy hardware restart workflow.

Current Omarchy restart commands include an audio restart helper.

Inspect:

```
omarchy restart --help
```

or:

```
omarchy restart audio
```

if supported on your installed version.

15.17 — Why Component Restart Is Better

A full reboot disrupts:

- Tmux sessions;
- development servers;
- applications;
- browser state;
- running jobs.

If the problem is only the audio subsystem, restarting only audio is much cheaper.

Use escalation:

```
select correct output
↓
restart audio
↓
inspect deeper
↓
reboot only if needed
```

15.18 — Laptop Speaker Tuning

Current official troubleshooting documentation states that some laptops can receive Omarchy speaker tuning.

Inspect current state with:

```
omarchy audio tuning status
```

Turn tuning off if you intentionally want raw output:

```
omarchy audio tuning off
```

This is laptop-specific.

Desktop systems may not use tuning at all.

15.19 — Do Not "Fix" Tuning That Is Working

If laptop speakers sound correct:

```
leave tuning alone
```

Do not disable it simply because you discovered the command.

This course repeatedly follows the rule:

> ***Do not change healthy settings for the sake of customization.***

15.20 — Bluetooth Panel

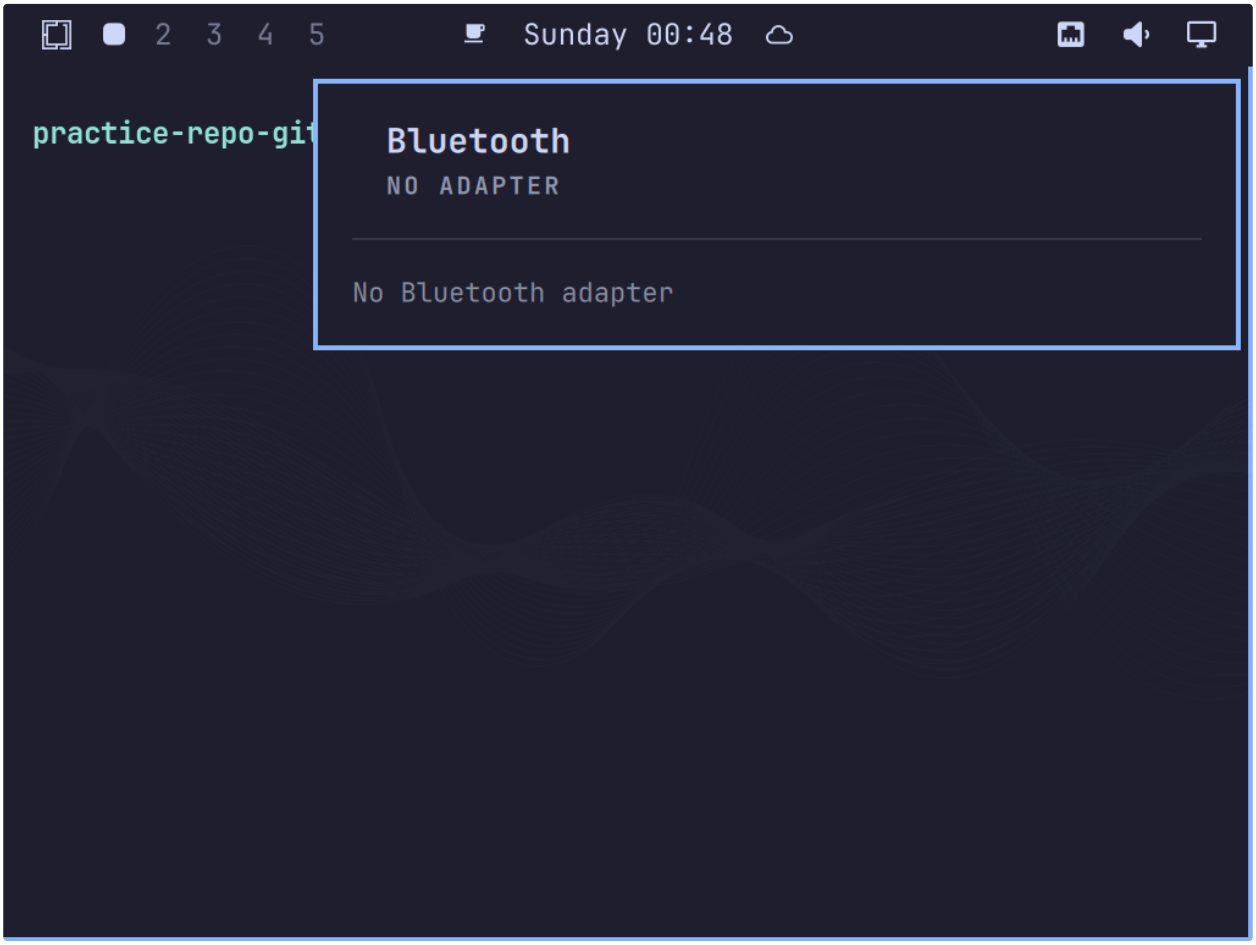
Open:

```
Super + Ctrl + B
```

Current official Omarchy top-bar documentation says the Bluetooth panel can show:

- known devices;
- connection state;
- battery levels where available;
- connect/disconnect controls.

This makes normal Bluetooth management a shell-level task.



Screenshot 15.20-A
Omarchy Bluetooth panel with paired device list

15.21 — Pairing a Device

Typical workflow:

```
1. put device into pairing mode
2. open Bluetooth panel
3. select device
4. pair
5. connect
```

The exact UI labels may vary slightly across current releases.

Use the panel first.

15.22 — Connect and Disconnect

For a previously paired device:

```
open Bluetooth panel
→ select device
→ connect/disconnect
```

This is faster than re-pairing.

If a device refuses to reconnect, try:

```
disconnect
→ reconnect
```

before forgetting it.

15.23 — Forget a Device

Forget/removal is appropriate when:

- pairing state is corrupted;
- device changed identity/firmware;
- you intentionally want to remove trust.

Do not forget a device every time it fails to connect once.

That creates extra pairing work unnecessarily.

15.24 — Bluetooth Radio Toggle

Current official top-bar behavior allows Bluetooth radio toggle from the Bluetooth widget.

Use this when you want:

```
Bluetooth fully off
```

rather than merely disconnecting one device.

15.25 — Bluetooth CLI

Current Omarchy CLI exposes:

```
omarchy bluetooth
```

Inspect:

```
omarchy bluetooth --help
```

Current workstation behavior has included controls for:

- power;
- pairing;
- connect;
- disconnect;
- forget.

Exact subcommands should be taken from the installed help.

15.26 — Bluetooth Power State

Current Omarchy Bluetooth commands support power control.

Typical capabilities include:

```
on
off
toggle
status/is-on
```

Inspect:

```
omarchy bluetooth --help
```

for the current spelling.

15.27 — Bluetooth State Persists Across Reboots

Current Omarchy behavior remembers Bluetooth power state across reboots.

That means if you intentionally turn Bluetooth off, do not be surprised when it remains off after restarting.

Always check power state before debugging pairing.

15.28 — Restart Bluetooth Before Reboot

Current official troubleshooting guidance says Bluetooth is one of the subsystems that should be restarted independently first.

If a headset worked earlier and suddenly will not reconnect:

```
1. verify Bluetooth is on
2. disconnect/reconnect device
3. restart Bluetooth subsystem
4. retry
5. reboot only later if necessary
```

15.29 — Network Panel

Open:

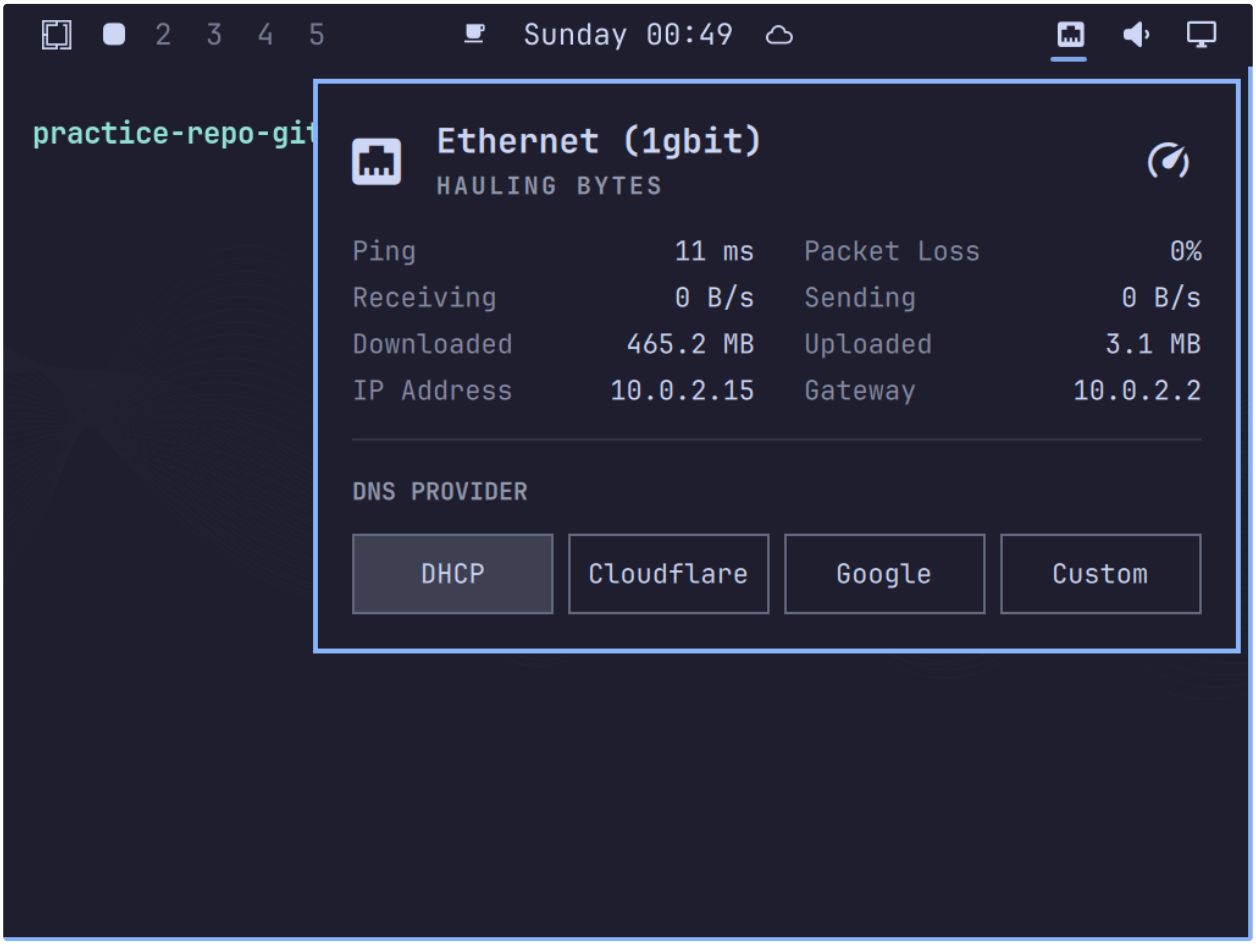
```
Super + Ctrl + W
```

Current official Omarchy top-bar documentation states that the network panel can:

- scan Wi-Fi networks;
- show signal strength;
- connect to networks;
- select DNS providers.

Current Quattro also includes additional network functionality such as:

- live up/down statistics;
- ping;
- speed testing;
- Wi-Fi QR sharing;
- Wi-Fi band selection.



Screenshot 15.29-A
Omarchy Network panel with Wi-Fi scan and signal strength

15.30 — Connect to Wi-Fi

Normal workflow:

```
open Network panel
→ choose network
→ enter password
→ connect
```

Current Omarchy 4 uses NetworkManager for Wi-Fi management.

This replaced older Omarchy generations that used a different stack.

That is a good example of why current-version documentation matters.

15.31 — Do Not Follow Old Omarchy Wi-Fi Tutorials Blindly

Older tutorials may mention tools that current Quattro no longer uses as the primary Wi-Fi interface.

Current Omarchy has moved network management into the Shell and NetworkManager-backed controls.

Therefore:

> ***Use the current Network panel and current `omarchy network` CLI before following legacy instructions.***

15.32 — Network CLI

Current Omarchy exposes:

```
omarchy network
```

Inspect:

```
omarchy network --help
```

Current workstation behavior includes capabilities such as:

- status;
- speed tests;
- band selection;
- Wi-Fi QR sharing;
- password display.

Use the installed help as the authoritative command map.

15.33 — Network Status

A useful current command is:

```
omarchy network status --verbose
```

This is useful for inspecting:

- connectivity;
- interface state;
- current network;
- other relevant connection details.

This has been tested on the workstation used in this course.

15.34 — Network Status as a Troubleshooting Baseline

When something says:

```
"Internet doesn't work"
```

first distinguish:

```
Wi-Fi not connected
vs
connected but no internet
vs
DNS issue
vs
remote service issue
```

Start with:

```
omarchy network status --verbose
```

Then test deeper only as needed.

15.35 — Signal Strength

The current network panel shows signal information.

Poor Wi-Fi performance may simply be:

```
weak signal
```

not a Linux configuration problem.

Before changing DNS, kernel settings, or drivers:

```
check signal
```

15.36 — Wi-Fi Band Selection

Current Quattro includes Wi-Fi band controls.

Current workstation behavior has included modes such as:

```
auto
2.4 GHz
5 GHz
6 GHz
```

depending on adapter/router support.

Inspect:

```
omarchy network --help
```

and the Network panel for current options.

15.37 — When Band Selection Helps

Possible reasons:

```
2.4 GHz
→ longer range, lower throughput, more interference

5 GHz
→ shorter range, higher throughput

6 GHz
→ newer hardware, cleaner spectrum, shorter range
```

These are general wireless characteristics.

Your router and client hardware determine what is actually available.

For most users:

```
Auto
```

is the correct default.

15.38 — Wi-Fi QR Sharing

Current Omarchy includes Wi-Fi QR sharing from the network panel/CLI.

This is useful when you want to connect a phone or another device without manually typing the network password.

Use the current panel or:

```
omarchy network --help
```

to discover the exact QR subcommand on your installed version.

15.39 — Wi-Fi Password Display

Current Omarchy network commands can expose the current Wi-Fi password where available.

Treat that as sensitive information.

Do not:

- paste it into public screenshots;
- expose it during recordings;
- include it in course assets.

When filming the course, blur/redact credentials.

15.40 — DNS Provider Selection

Current official top-bar documentation says the Network panel lets you choose a DNS provider.

DNS affects:

```
name resolution
```

not raw Wi-Fi signal strength.

If:

```
Wi-Fi signal is poor
```

changing DNS will not fix the radio link.

Use DNS settings for DNS-related reasons.

15.41 — Speed Test

Current Omarchy includes network speed-test capability.

Current workstation behavior has included separate:

```
download
upload
```

tests.

Inspect the current syntax:

```
omarchy network --help
```

Use this to distinguish:

```
network connected
```

from:

```
network performing well
```

15.42 — Ping

Current Quattro network panel includes ping information.

Ping helps evaluate:

- reachability;
- basic latency;
- packet-path responsiveness.

Do not treat one ping result as a complete network diagnosis.

Use it as one signal.

15.43 — Restart Wi-Fi Before Reboot

Current official troubleshooting guidance includes Wi-Fi among the subsystems that should be restarted independently first.

Escalation:

```
check connected network
↓
check signal
↓
restart Wi-Fi/network subsystem
↓
reconnect
↓
inspect deeper
↓
reboot only if needed
```

15.44 — Display Panel

Open:

```
Super + Ctrl + D
```

The display panel handles normal display controls.

Current top-bar documentation also states:

```
scroll on Display widget
→ brightness
```

This is distinct from persistent monitor mode configuration in:

```
~/.config/hypr/monitors.lua
```

15.45 — Brightness vs Monitor Configuration

These are different.

Brightness

```
how much light the panel emits
```

Monitor Configuration

```
resolution
refresh rate
scale
position
```

Do not edit:

```
monitors.lua
```

to solve a brightness issue.

15.46 — Hardware Brightness Keys

Current official hotkeys include:

```
Shift + Brightness Up
→ maximum brightness

Shift + Brightness Down
→ minimum brightness

Alt + Brightness Up/Down
→ precise 1% changes
```

Normal hardware brightness keys can use regular step sizes.

15.47 — Keyboard Backlight

Current official hotkeys also include keyboard-backlight controls when hardware supports them:

```
Keyboard Brightness Up/Down
Keyboard Backlight
```

Hardware availability varies.

A desktop keyboard may manage lighting through its own firmware instead.

15.48 — Brightness CLI

Current Omarchy exposes:

```
omarchy brightness
```

Inspect:

```
omarchy brightness --help
```

Current workstation behavior includes display brightness functionality including DDC for compatible external displays.

15.49 — External Monitor DDC Brightness

Some external monitors support:

```
DDC/CI
```

which allows software brightness control over the display connection.

Current Omarchy includes DDC-related brightness commands.

Inspect:

```
omarchy brightness display --help
```

or the current command tree.

Support depends on:

- monitor;
- cable;
- GPU;
- DDC settings.

Do not assume every external display supports software brightness control.

15.50 — Text Size

Current workstation behavior exposes:

```
omarchy display text size
```

for text-size adjustment.

Inspect:

```
omarchy display text size --help
```

This is different from:

```
monitor scale
```

and:

```
resolution
```

15.51 — Display Settings Hierarchy

Use this decision tree:

```
Image too dark?
→ brightness

Everything physically too small?
→ monitor scale

Text specifically too small?
→ text size/font settings

Image blurry/wrong geometry?
→ resolution

Motion not smooth enough?
→ refresh rate

Windows on wrong screen?
→ monitor positioning/workspace rules
```

This prevents changing the wrong layer.

15.52 — Persistent Monitor Configuration

For:

```
resolution
refresh rate
scale
position
```

use:

```
~/.config/hypr/monitors.lua
```

and the Module 11 workflow.

Inspect reality:

```
hyprctl monitors all
```

before changing anything.

15.53 — Verify Refresh Rate

A monitor can display an image correctly while still running at the wrong refresh rate.

Example:

```
144 Hz monitor
running at 60 Hz
```

The desktop works.

But it is not configured optimally.

Run:

```
hyprctl monitors all
```

and confirm the active mode.

This was a real issue during the workstation setup used for this course.

15.54 — Display Panel vs `monitors.lua`

Use:

```
Display panel
```

for routine interactive controls.

Use:

```
monitors.lua
```

for persistent hardware layout configuration.

Examples:

```
brightness now
→ panel

make DP-2 always 2560x1440@144
→ monitors.lua
```

15.55 — Laptop Display Toggle

Current official hotkeys include:

```
Super + Ctrl + Delete
```

for toggling the laptop display on/off.

Current Quattro also includes mirroring behavior:

```
Super + Ctrl + Alt + Delete
```

for toggling laptop display mirroring.

These are laptop-specific.

Desktop users can ignore them.

15.56 — Power Panel

Although this module focuses on audio/Bluetooth/network/display, the Power panel belongs in the same hardware-control family.

Open:

```
Super + Ctrl + P
```

Current official documentation says it provides:

- battery stats;
- power-profile switching;
- separate remembered AC/battery choices;
- system information.

Power profiles can affect performance and battery life.

15.57 — Power Profiles on Laptops

Typical profile categories can include:

```
power saver
balanced
performance
```

depending on hardware/support.

Use:

```
balanced
```

as the safe general default unless your workload or battery goals justify another profile.

15.58 — Hardware Troubleshooting Principle

Current official Omarchy troubleshooting guidance explicitly says:

> *Before rebooting, restart only the offending subsystem.*

This applies to:

```
Wi-Fi
Bluetooth
Audio
Trackpad
```

This should become your default mental model.

15.59 — The Hardware Recovery Ladder

Use:

```
1. verify current state
2. correct obvious selection/config
3. restart only affected subsystem
4. retry
5. inspect logs/services
6. reboot if needed
```

This is far better than:

```
problem
→ reboot
```

as the first response.

15.60 — Why Rebooting First Is Bad Troubleshooting

If you reboot immediately:

```
problem disappears
```

but you learned nothing.

You do not know whether the issue was:

- Bluetooth daemon;
- PipeWire/WirePlumber;
- NetworkManager;
- device state;
- application stream;
- compositor state.

A targeted restart teaches you which layer failed.

15.61 — Audio Troubleshooting Flow

If audio is missing:

```
1. check master mute
2. check correct output
3. check per-app stream volume
4. switch output away/back
5. restart audio subsystem
6. inspect deeper
```

For laptop speakers specifically:

```
check tuning state
```

if sound quality is the issue rather than total silence.

15.62 — Bluetooth Troubleshooting Flow

If a device will not connect:

```
1. confirm Bluetooth radio is on
2. verify device is paired
3. disconnect/reconnect
4. restart Bluetooth
5. forget/re-pair if necessary
6. inspect deeper
```

Do not start with re-pairing unless needed.

15.63 — Wi-Fi Troubleshooting Flow

If networking fails:

```
1. open network panel
2. check current SSID
3. check signal strength
4. run network status
5. reconnect
6. restart Wi-Fi/network
7. test connectivity
8. inspect deeper
```

Changing DNS should only happen when the symptoms support a DNS issue.

15.64 — Display Troubleshooting Flow

If display behavior looks wrong:

```
1. hyprctl monitors all
2. identify actual output
3. check mode/resolution/refresh
4. check scale
5. check monitors.lua
6. hyprctl configerrors
7. change one thing
8. verify again
```

Do not combine display configuration with unrelated theme/text-size changes.

15.65 — Trackpad Troubleshooting

Current official Omarchy troubleshooting also includes trackpad restart among common hardware restart actions.

Laptop users should remember:

```
trackpad stopped after suspend
```

does not necessarily mean:

```
reboot required
```

Try the current Omarchy trackpad restart helper first.

15.66 — Practical Exercise: Audio Panel

Open:

```
Super + Ctrl + A
```

Identify:

- current output;
- master volume;
- application streams.

If you have two outputs, switch between them.

Return to your preferred output.

15.67 — Practical Exercise: Audio Shortcut

Test:

```
Alt + Volume Up
Alt + Volume Down
```

Observe 1% changes.

If your hardware supports multiple outputs, test:

```
Shift + Mute
```

and observe output switching.

15.68 — Practical Exercise: Audio CLI Discovery

Run:

```
omarchy audio --help
```

Write down the current subcommands on your installed version.

If your machine exposes tuning:

```
omarchy audio tuning status
```

Do not change tuning unless you actually want to.

15.69 — Practical Exercise: Bluetooth Panel

Open:

```
Super + Ctrl + B
```

Identify:

- Bluetooth power state;
- paired devices;
- currently connected device;
- battery percentage if available.

If you have a device, disconnect/reconnect it.

15.70 — Practical Exercise: Bluetooth CLI Discovery

Run:

```
omarchy bluetooth --help
```

Identify the current commands for:

- power;

- connect;
- disconnect;
- pair;
- forget.

Do not forget a useful device just to complete the exercise.

15.71 — Practical Exercise: Network Panel

Open:

```
Super + Ctrl + W
```

Identify:

- current network;
- signal strength;
- DNS selection;
- any live network statistics;
- band control if available.

Do not disconnect if you are in the middle of important online work.

15.72 — Practical Exercise: Network Status

Run:

```
omarchy network status --verbose
```

Record:

```
connection state
active network
interface information
```

Then compare the CLI result with the panel.

15.73 — Practical Exercise: Speed Test

Inspect:

```
omarchy network --help
```

Run the current documented speed-test command.

Record:

```
download
upload
```

Do not treat one result as a permanent measurement.

Wireless speed fluctuates.

15.74 — Practical Exercise: Wi-Fi QR

If your current network CLI/panel supports it:

```
generate the Wi-Fi QR code
```

Do **not** publish a screenshot containing real credentials.

This exercise is optional.

15.75 — Practical Exercise: Display Panel

Open:

```
Super + Ctrl + D
```

Adjust brightness slightly.

Return to your preferred level.

Observe that this did not change:

- resolution;
- refresh rate;
- scale.

15.76 — Practical Exercise: Verify Monitor Mode

Run:

```
hyprctl monitors all
```

Record:

```
output
resolution
refresh
scale
position
```

Compare with:

```
bat ~/.config/hypr/monitors.lua
```

Confirm that persistent config and active reality make sense together.

15.77 — Practical Exercise: Text Size Discovery

Run:

```
omarchy display --help
```

Then:

```
omarchy display text size --help
```

Do not change it if your current text size is already correct.

The goal is knowing where the control lives.

15.78 — Practical Exercise: Restart One Subsystem

If one subsystem currently has a harmless issue, use that.

Otherwise simply inspect:

```
omarchy restart --help
```

Identify current helpers for:

- audio;
- Bluetooth;
- Wi-Fi;
- trackpad.

Do not restart hardware randomly in the middle of important work.

15.79 — Common Hardware-Control Mistakes

Mistake 1 — Rebooting First

Restart the affected subsystem first.

Mistake 2 — Assuming Audio Is Broken When the Wrong Output Is Selected

Check the audio panel.

Mistake 3 — Re-Pairing Bluetooth for Every Reconnect Failure

Try reconnect and subsystem restart first.

Mistake 4 — Changing DNS Because Wi-Fi Signal Is Weak

DNS does not improve radio strength.

Mistake 5 — Following Old Omarchy Networking Tutorials

Current Quattro uses the current shell/network stack and NetworkManager-backed controls.

Mistake 6 — Changing Monitor Scale to Fix Only Text Size

Use the correct display layer.

Mistake 7 — Assuming 144 Hz Is Active Because the Monitor Supports 144 Hz

Verify:

```
hyprctl monitors all
```

Mistake 8 — Assuming External Monitor Brightness Is Software-Controlled

DDC support varies by hardware.

Mistake 9 — Publishing Wi-Fi Password/QR Screenshots

Treat network credentials as secrets.

Mistake 10 — Treating Every Hardware Issue as a Kernel/driver problem

Start with current state, device selection, and subsystem restart.

15.80 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Audio

- [ ] I can open the audio panel.
- [ ] I can select the correct output.
- [ ] I understand per-app volume.
- [ ] I can distinguish microphone input from output audio.
- [ ] I know how to discover current audio CLI commands.
- [ ] I know to restart audio before rebooting.
- [ ] I know laptop speaker tuning may exist.

Bluetooth

- [ ] I can open the Bluetooth panel.
- [ ] I understand pair/connect/disconnect/forget.
- [ ] I can verify Bluetooth power state.
- [ ] I know to restart Bluetooth before rebooting.
- [ ] I know Bluetooth power state can persist across reboots.

Network

- [ ] I can open the Network panel.
- [ ] I can check signal strength.
- [ ] I can connect to Wi-Fi.
- [ ] I can run `omarchy network status --verbose`.
- [ ] I understand DNS vs Wi-Fi signal.
- [ ] I understand Wi-Fi band selection at a high level.
- [ ] I know about QR sharing and speed testing.
- [ ] I know to restart networking before rebooting.

Display

- [ ] I can open the Display panel.
- [ ] I understand brightness vs monitor configuration.
- [ ] I understand text size vs monitor scale.
- [ ] I can verify active monitor mode with `hyprctl monitors all`.
- [ ] I know where persistent monitor config lives.
- [ ] I understand DDC brightness may depend on hardware.

Troubleshooting

- [ ] I follow the subsystem-first restart model.
- [ ] I verify current state before changing configuration.
- [ ] I know when a full reboot is actually the later escalation step.

15.81 — What You Can Now Do

After completing Module 15, you can now:

- operate audio confidently;
- switch outputs without restarting applications;
- control per-app audio;
- manage microphones;
- pair and reconnect Bluetooth devices;
- troubleshoot Bluetooth without immediately re-pairing;
- connect and inspect Wi-Fi;
- use signal, DNS, band, speed, and QR features intelligently;
- control brightness;
- distinguish brightness, text size, scale, refresh rate, and resolution;
- verify display hardware state;
- restart individual hardware subsystems before rebooting;
- approach hardware troubleshooting methodically.

Most importantly:

> *You can now treat hardware control as part of the Omarchy workflow instead of a collection of unrelated Linux utilities.*

Module 15 Summary

Current panel shortcuts:

```
Super + Ctrl + A
→ Audio

Super + Ctrl + W
→ Network

Super + Ctrl + B
→ Bluetooth

Super + Ctrl + D
→ Display

Super + Ctrl + P
→ Power
```

Audio:

```
select output
per-app mixer
microphone/input
precise volume
output switching
subsystem restart
```

Bluetooth:

```
power
pair
connect
disconnect
forget
restart
```

Network:

```
Wi-Fi scan/connect
signal strength
DNS
band selection
QR sharing
speed test
status
restart
```

Display:

```
brightness
text size
persistent monitor mode
refresh rate
scale
position
```

Core verification:

```
omarchy audio --help
```

```
omarchy bluetooth --help
```

```
omarchy network --help
```

```
omarchy display --help
```

```
omarchy brightness --help
```

```
hyprctl monitors all
```

Hardware recovery ladder:

```
verify
↓
correct obvious selection
↓
restart affected subsystem
↓
retry
↓
inspect deeper
↓
reboot
```

And the most important rule:

> ***If one subsystem breaks, restart one subsystem — not the whole workstation.***

MODULE 16

16

System Toggles and Runtime Controls on Omarchy

You now know how to control the major hardware subsystems.

Next we focus on **runtime state**: the small but important workstation behaviors that change dynamically while you work.

About This Module

You now know how to control the major hardware subsystems.

Next we focus on **runtime state**: the small but important workstation behaviors that change dynamically while you work.

Examples include:

- idle/sleep behavior;
- "stay awake";
- night light;
- Do Not Disturb;
- bar visibility;
- screensaver;
- suspend;
- touchpad/touchscreen toggles;
- hybrid GPU mode on supported laptops.

Current Omarchy 4 / Quattro exposes these through a combination of:

```
Shell controls
+
omarchy toggle ...
+
hotkeys
```

This is a different category from permanent configuration.

A useful mental model is:

```
configuration
→ what the workstation should normally be

runtime toggle
→ what state the workstation should be in right now
```

The goal of this module is:

> ***Understand current runtime controls, inspect state before changing it, and avoid inventing commands that do not exist.***

That last point matters because Omarchy's CLI changes quickly, and several runtime controls live specifically under:

```
omarchy toggle
```

rather than as top-level commands.

What You Will Learn

By the end of Module 16, you will understand:

- the difference between persistent configuration and runtime toggles;
- how to discover current toggle commands;
- how idle behavior works at a high level;
- how "stay awake" relates to idle state;
- how to inspect current idle state;
- how night light works;
- how to inspect and toggle night light;
- how notification silencing / Do Not Disturb works;
- how to control bar visibility;
- how screensaver and suspend toggles fit into runtime control;
- how touchpad and touchscreen toggles work on supported hardware;
- how hybrid GPU toggles apply to supported laptops;
- why you should verify state rather than infer it from a tooltip;
- why `omarchy idle` is not the correct current top-level command;
- how runtime toggles complement shell panels and hotkeys.

16.1 — Runtime State vs Persistent Configuration

Compare these two tasks:

```
"Always run my monitor at 144 Hz."
```

That is persistent configuration.

Use:

```
~/.config/hypr/monitors.lua
```

Now compare:

```
"Keep the machine awake for the next hour."
```

That is runtime state.

Use a toggle.

This distinction keeps your configuration clean.

16.2 — Current Toggle Discovery

Current Omarchy exposes runtime controls under:

```
omarchy toggle
```

Start with:

```
omarchy toggle --help
```

Current versions include toggle families for areas such as:

```
idle
nightlight
notification silencing
bar
screensaver
suspend
touchpad
touchscreen
hybrid GPU
crash capture
```

The exact command names can evolve.

Use the installed help as the source of truth.

16.3 — Do Not Guess Top-Level Commands

A real lesson from the workstation setup:

```
omarchy idle
```

does not exist on the current system.

The current idle control is under:

```
omarchy toggle idle
```

This is a perfect example of why:

```
omarchy
omarchy commands
omarchy commands --all
omarchy <group> --help
```

matter.

16.4 — Idle Behavior

Idle behavior controls what happens after the machine receives no user input for some period.

Possible outcomes may include:

- dimming;
- locking;
- screensaver;
- suspend;
- other configured idle actions.

Current Omarchy exposes idle state through:

```
omarchy toggle idle
```

Inspect:

```
omarchy toggle idle --help
```

or current equivalent.

16.5 — Stay Awake

The user-facing concept:

```
Stay Awake
```

means:

> *temporarily prevent normal idle behavior from triggering.*

This is useful when:

- presenting;
- watching a long video;
- running a visible process you need to monitor;
- keeping a display awake temporarily.

It should not replace sane normal idle configuration.

16.6 — Inspect Idle State

On the workstation used during this course, current state inspection returned fields such as:

```
enabled: false
tooltip: Stay Awake
```

Interpret carefully.

If the toggle represents:

`Stay Awake`

then:

`enabled = false`

means the stay-awake override is **not active**.

Normal idle behavior remains allowed.

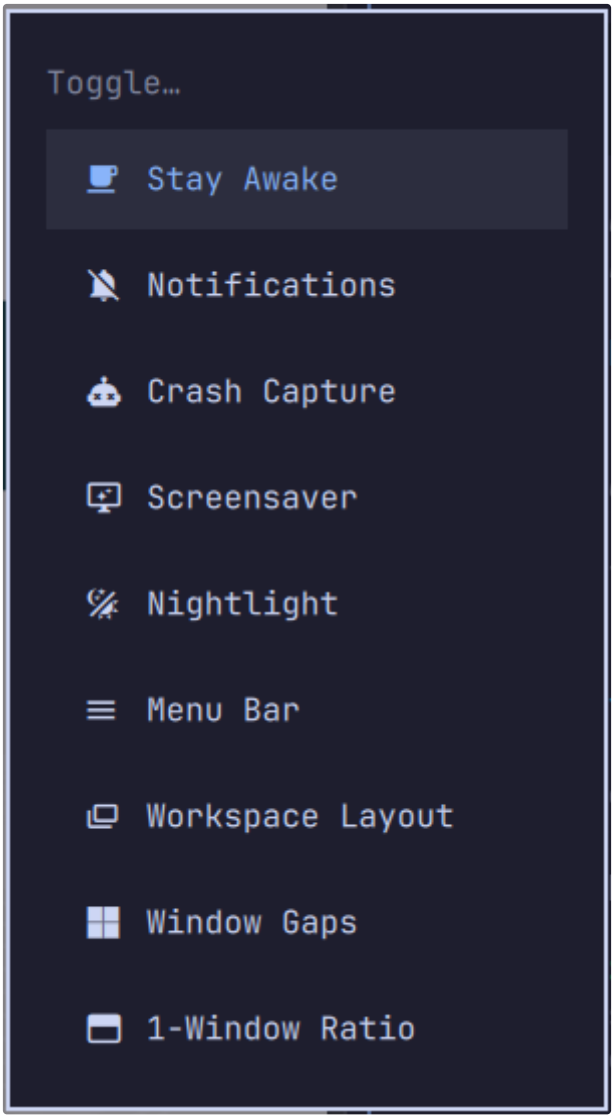
Do not confuse:

`idle enabled`

with:

`stay-awake enabled`

without reading the current command semantics.



Screenshot 16.6-A
Omarchy Shell Toggle menu with Stay Awake listed

16.7 — Verify Instead of Inferring

Whenever a toggle has unclear naming:

```
1. read --help
2. inspect status output
3. observe UI tooltip
4. change once
5. inspect status again
```

This is safer than guessing what `enabled=true` refers to.

16.8 — Night Light

Night light shifts the display toward a warmer color temperature.

Typical reasons:

- evening work;
- reduced blue-heavy light;
- visual comfort.

Current Omarchy exposes night light under:

```
omarchy toggle nightlight
```

Inspect:

```
omarchy toggle nightlight --help
```

16.9 — Current Night-Light Status

The workstation used in this course initially had night light off.

Toggling it produced state similar to:

```
enabled: true
temperature: 4000
```

This is a real current-system example.

Your current configured temperature may differ.

16.10 — When to Use Night Light

Useful:

```
late evening
dark room
extended reading
```

Less useful:

```
color-critical image/video work
design work requiring accurate white balance
```

If color accuracy matters, disable it temporarily.

16.11 — Night Light Is Runtime State, Not Theme

Changing theme:

```
dark → light
```

does not equal changing color temperature.

Theme controls styling.

Night light alters display color temperature.

Keep these layers separate.

16.12 — Do Not Hard-Code Temperature Without Checking Current Behavior

Different releases may expose temperature through:

- status;
- config;
- shell settings;
- current toggle implementation.

Use:

```
omarchy toggle nightlight --help
```

and current official docs.

Do not assume one old command forever.

16.13 — Notification Silencing / Do Not Disturb

Current Omarchy provides notification silencing as a runtime control.

The UI concept is similar to:

```
Do Not Disturb
```

Use it when:

- presenting;
- recording;
- deep work;
- focusing during coding;
- screen sharing.

16.14 — Notification Silencing Can Be Silent

A current-system lesson:

some toggle actions may print:

```
nothing
```

when successful.

Do not interpret no output as failure automatically.

After toggling:

- inspect status;
- observe shell UI;
- test a notification if appropriate.

Silent command success is a general Unix pattern.

16.15 — Do Not Invent Status Syntax

The exact status interface for notification silencing can differ.

Use:

```
omarchy toggle --help
```

and:

```
omarchy commands --all | grep -i notification
```

before using undocumented flags.

This is especially important because earlier assumptions about exact toggle subcommand syntax can easily become stale.

16.16 — Bar Visibility

Current Omarchy exposes bar state through runtime toggles.

Use cases:

- screen recording;
- distraction-free mode;
- temporary fullscreen presentation;
- testing shell layout.

Discover:

```
omarchy toggle --help
```

and current bar-related help.

16.17 — Bar Toggle vs Fullscreen

These solve different things.

```
Fullscreen
→ changes one window's layout

Bar toggle
→ changes shell/bar visibility
```

You may combine them for a clean presentation workspace.

16.18 — Screensaver

Current Omarchy includes screensaver runtime control.

The screensaver belongs to idle/runtime behavior.

Use the current:

```
omarchy toggle
```

command family to inspect it.

Do not confuse screensaver with:

- lock screen;
- suspend;
- idle override.

They can be related but are distinct.

16.19 — Suspend Toggle

Current Omarchy exposes suspend behavior as a toggle.

This can be useful when you temporarily want to prevent automatic suspend while keeping other idle behavior.

Example:

```
long compile
download
presentation
remote monitoring
```

Again:

> *Inspect current semantics before toggling.*

16.20 — Idle vs Suspend

These are related but not identical.

```
idle system
→ tracks inactivity / triggers actions

suspend
→ one possible power-state transition
```

You may want:

```
screen can lock
but machine should not suspend
```

during a long-running task.

That is why separate controls are useful.

16.21 — Touchpad Toggle

Current Omarchy includes a touchpad toggle on supported laptops.

Useful when:

- using an external mouse;
- typing causes accidental cursor movement;
- troubleshooting trackpad behavior.

Current restart and toggle command families are separate concepts:

```
toggle touchpad
→ enable/disable state

restart trackpad
→ recover subsystem/device behavior
```

Do not confuse them.

16.22 — Touchscreen Toggle

Supported touchscreen devices can also expose a runtime toggle.

This is useful when:

- touchscreen input is unwanted;
- external monitor workflow makes touch confusing;
- you are troubleshooting ghost touches.

Desktop users without touchscreen hardware can ignore this.

16.23 — Hybrid GPU Toggle

Current Omarchy includes hybrid-GPU-related toggle behavior on supported systems.

This primarily matters for laptops with combinations such as:

```
integrated GPU
+
discrete GPU
```

Possible goals:

- battery efficiency;
- performance mode;
- GPU switching.

Hardware support varies significantly.

Do not change hybrid-GPU settings just because the command exists.

16.24 — Hybrid GPU Changes Are Higher Risk Than Night Light

Compare:

```
night light
→ easy reversible display tint
```

with:

```
GPU mode
→ potentially affects rendering/session behavior
```

Treat GPU-mode changes more cautiously.

Check current official hardware guidance first.

16.25 — Crash Capture Toggle

Current Omarchy includes crash-capture runtime control.

This relates to the crash-diagnosis flow covered earlier.

When enabled, Omarchy can observe crashes through system crash data and surface notifications/AI-assisted diagnostics.

This is useful for debugging.

It can also generate diagnostic state you may not care about during certain workflows.

16.26 — Runtime Toggle Design Principle

A good toggle should answer:

```
"What temporary state do I want right now?"
```

Examples:

```
Stay awake
Night light on
Do Not Disturb
Bar hidden
Touchpad off
```

If you want the state permanently, look for the underlying persistent configuration instead.

16.27 — Shell UI vs CLI

Many runtime controls are available through the Omarchy Shell.

The CLI matters because it gives you:

- inspectable commands;
- scriptability;
- reproducibility;
- automation potential.

The shell UI matters because it gives you:

- visibility;
- quick control;
- current-state feedback.

Use both.

16.28 — Toggling From Scripts

Because runtime controls have CLI forms, they can be used from:

- keybindings;
- scripts;
- hooks;
- custom workflow helpers.

Example concept:

```
Presentation mode script:
→ silence notifications
→ stay awake
→ hide bar
→ maybe set brightness
```

This becomes possible because each behavior is exposed programmatically.

16.29 — Do Not Build "Presentation Mode" Yet

It is tempting to immediately automate several toggles together.

First understand each one independently.

Then, later:

```
binding
→ script
→ several verified toggle commands
```

becomes safe and maintainable.

16.30 — Inspect Before Automation

Before scripting a toggle:

```
omarchy toggle --help
```

Then test the exact command manually.

Only once you understand:

- command;
- resulting state;
- reversal;

should you automate it.

16.31 — Persistent State Can Differ by Toggle

Some runtime states may survive:

- session restart;

- reboot;

while others may reset.

Do not assume all toggles behave the same.

Verify current behavior if persistence matters.

16.32 — A Safe Toggle Testing Pattern

For any toggle:

```
1. inspect state
2. toggle once
3. inspect state again
4. observe UI behavior
5. toggle back
```

This gives you both semantics and reversibility.

16.33 — Practical Exercise: Discover Current Toggles

Run:

```
omarchy toggle --help
```

Write down all current toggle categories on your system.

Compare them with the categories in this module.

If your installed version includes something new, treat your installed CLI as current reality.

16.34 — Practical Exercise: Idle State

Run the current idle help:

```
omarchy toggle idle --help
```

Then inspect current state using the supported status form.

Record:

```
enabled
tooltip/label
other state fields
```

Determine whether stay-awake is active.

Do not change it until you understand the current semantics.

16.35 — Practical Exercise: Toggle Stay Awake

Toggle the current idle/stay-awake state once.

Inspect status again.

Observe:

```
what changed?
```

Then return it to your preferred normal state.

This exercise is about interpreting state, not just executing commands.

16.36 — Practical Exercise: Night Light

Inspect:

```
omarchy toggle nightlight --help
```

Then check status.

Toggle night light.

Observe the warmer display temperature.

Check status again.

Return to your preferred state.

16.37 — Practical Exercise: Notification Silencing

Discover the current notification-silencing command through:

```
omarchy toggle --help
```

Toggle it.

Observe shell state.

Then restore normal notifications.

If the command prints no output, verify via state/UI rather than assuming failure.

16.38 — Practical Exercise: Bar Toggle

Discover current bar toggle syntax.

Hide the bar briefly.

Observe:

- workspace behavior;
- fullscreen behavior;
- available screen area.

Restore it.

Do not leave it hidden accidentally before the next lesson recording.

16.39 — Practical Exercise: Suspend Control

Inspect current suspend toggle help.

Do not actually suspend the machine just for the exercise.

The goal is to understand:

```
automatic suspend behavior
```

and where the runtime control lives.

16.40 — Practical Exercise: Touchpad/Touchscreen

Laptop users:

inspect current toggle help for:

```
touchpad
touchscreen
```

If safe, toggle one and restore it.

Desktop users can skip this.

16.41 — Practical Exercise: Build a Runtime State Table

Create a small note:

```
Idle / Stay Awake:
Night Light:
Notifications:
Bar:
Screensaver:
Suspend:
Touchpad:
Touchscreen:
Hybrid GPU:
Crash Capture:
```

Fill in:

- current state;
- how to inspect;
- how to toggle;
- whether you care about it.

This becomes useful for the handbook and your future workstation setup.

16.42 — Runtime State and Screen Recording

Before recording a course video, you may want:

```
stay awake
notification silencing
clean bar state
controlled brightness
night light off for color consistency
```

This is a concrete real-world reason to understand runtime toggles.

16.43 — Runtime State and Deep Work

A focused coding session might use:

```
notifications silenced
stay awake off unless needed
night light based on time
normal bar
```

The correct state depends on context.

There is no single "power user" combination.

16.44 — Runtime State and Long Jobs

For a long build or test workload:

```
automatic suspend off
```

may be useful.

But:

```
screen lock
```

may still be desirable.

This is why individual controls matter.

16.45 — Runtime State and Security

Do not disable:

- lock behavior;
- suspend;
- notification privacy;

permanently without thinking about the physical-security consequences.

A workstation that never locks is convenient until you leave it unattended.

16.46 — Keep Security and Convenience Separate

Good:

```
temporarily stay awake during a presentation
```

Risky:

```
disable idle/locking forever because re-authentication is annoying
```

Use temporary controls for temporary needs.

16.47 — Use Tooltips as Signals, Not Documentation

Shell tooltips can help.

But for ambiguous state:

```
CLI help
+
status output
+
official docs
```

should decide semantics.

This prevents misunderstandings such as interpreting a label as the state itself.

16.48 — Silent Commands and Exit Codes

If a toggle prints nothing:

```
echo $?
```

can help determine whether the command succeeded.

Typical Unix convention:

```
0
→ success

non-zero
→ failure
```

Then inspect state afterward.

16.49 — Do Not Chain Unknown Toggles

Bad:

```
toggle-a && toggle-b && toggle-c && toggle-d
```

when you do not yet understand their current semantics.

First verify each command independently.

Automation comes later.

16.50 — Common Runtime-Control Mistakes

Mistake 1 — Running `omarchy idle`

Current idle control belongs under:

```
omarchy toggle idle
```

on the workstation used for this course.

Mistake 2 — Assuming `enabled=false` Means Idle Is Disabled

Read what the toggle represents.

It may represent:

```
Stay Awake
```

rather than raw idle capability.

Mistake 3 — Confusing Night Light With Theme

Night light changes color temperature.

Theme changes UI styling.

Mistake 4 — Assuming No Command Output Means Failure

Check state and exit code.

Mistake 5 — Hard-Coding Undocumented Toggle Flags

Use current `--help`.

Mistake 6 — Disabling Suspend Permanently for One Long Build

Use temporary runtime state for temporary needs.

Mistake 7 — Toggling Hybrid GPU Without a Reason

GPU state can be hardware-sensitive.

Mistake 8 — Building Complex Toggle Scripts Before Testing Each Action

Verify each component first.

Mistake 9 — Treating Tooltips as the Entire Contract

Use installed CLI and official docs for ambiguous behavior.

Mistake 10 — Forgetting to Restore Temporary State

After recording/presenting/testing, restore your normal workstation behavior.

16.51 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Concepts

- [ ] I understand persistent configuration vs runtime state.
- [ ] I know current runtime controls live under `omarchy toggle`.
- [ ] I know not to invent top-level commands.

Idle

- [ ] I understand the concept of stay-awake.
- [ ] I know how to inspect current idle-toggle semantics.
- [ ] I understand why state labels can be ambiguous.

Night Light

- [ ] I can inspect night-light state.
- [ ] I understand temperature at a high level.
- [ ] I know when night light is inappropriate.

Notifications

- [ ] I understand notification silencing / Do Not Disturb.
- [ ] I know a successful toggle may be silent.
- [ ] I know to verify state afterward.

Other Toggles

- [ ] I understand bar visibility.
- [ ] I understand screensaver vs suspend.
- [ ] I understand touchpad/touchscreen toggles.
- [ ] I understand hybrid GPU is hardware-specific.

- [ ] I understand crash capture at a high level.

Safety

- [ ] I test one toggle at a time.
- [ ] I know how to reverse a change.
- [ ] I avoid permanently weakening security for temporary convenience.

16.52 — What You Can Now Do

After completing Module 16, you can now:

- distinguish permanent configuration from temporary runtime state;
- inspect current Omarchy toggle capabilities;
- use stay-awake intentionally;
- control night light;
- silence notifications for focus/presentation;
- hide/show shell UI when needed;
- understand screensaver/suspend controls;
- manage laptop touch input state;
- reason about hybrid-GPU toggles;
- verify ambiguous toggle state instead of guessing;
- prepare the workstation for recording, presentation, deep work, or long jobs.

Most importantly:

> *You now control the workstation's temporary operating state without polluting permanent configuration.*

Module 16 Summary

Discover toggles:

```
omarchy toggle --help
```

Current categories can include:

```
idle
nightlight
notification silencing
bar
screensaver
suspend
touchpad
touchscreen
hybrid GPU
crash capture
```

Core model:

```
persistent preference
→ config file

temporary state
→ runtime toggle
```

Stay-awake model:

```
normal idle allowed
vs
temporary idle suppression
```

Night light:

```
theme
≠
color temperature
```

Silent command verification:

```
COMMAND
echo $?
```

Then inspect state.

Safe workflow:

```
inspect
↓
toggle once
↓
inspect again
↓
observe
↓
restore
```

And the most important lesson:

> ***Do not guess what a toggle means from its name alone. Verify the current installed behavior.***

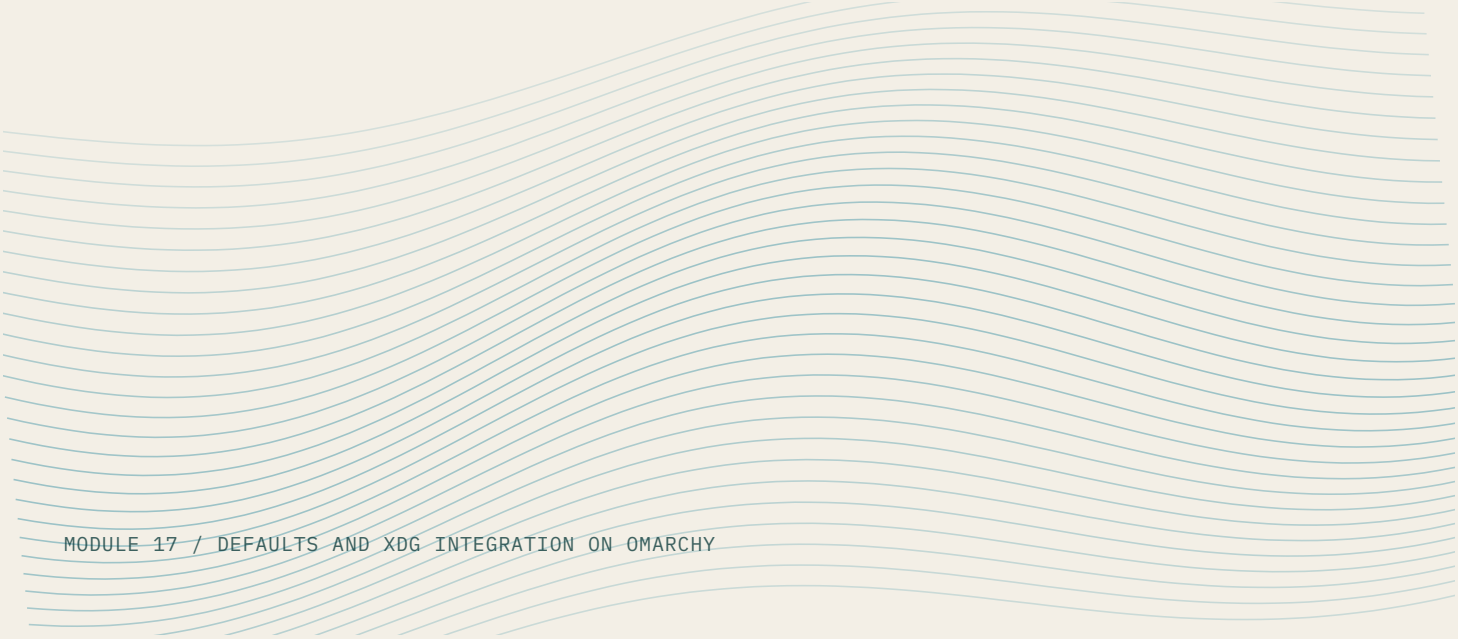
MODULE 17

17

Defaults and XDG Integration on Omarchy

You now know how to:

- customize the desktop;
- control hardware;
- change runtime state;
- add automation.



About This Module

You now know how to:

- customize the desktop;
- control hardware;
- change runtime state;
- add automation.

The next layer is deceptively important:

> ***What application does the system consider the default?***

When you click a link, which browser opens?

When Omarchy launches a terminal, which terminal opens?

When a configuration action requests an editor, which editor is used?

When a Linux application asks the desktop to open a file or URL, how does it know which application should handle it?

This is where Omarchy's own:

`Setup → Defaults`

system meets the Linux desktop standards commonly grouped under:

`XDG`

A correctly configured workstation should behave consistently.

Example:

```
Super + Shift + Return
→ preferred browser

click a URL in another application
→ same preferred browser

xdg-open https://example.com
→ same preferred browser
```

The goal of this module is:

> ***Understand Omarchy's default-app layer, understand enough XDG to diagnose inconsistencies, and avoid hard-coding application choices into random config files.***

What You Will Learn

By the end of Module 17, you will understand:

- Omarchy's current default categories;
- how to inspect current defaults;
- how to set the default browser;
- how to set the default terminal;
- how to set the default editor;
- how the default coding agent fits into the same concept;
- what XDG means in practical desktop use;
- what `.desktop` application IDs are;
- what MIME types are;
- what URL scheme handlers are;
- how `xdg-open` works conceptually;
- how `xdg-settings` fits into browser selection;
- how `xdg-mime` can inspect file-type associations;
- how `xdg-terminal-exec` fits into Omarchy's terminal default;
- why Omarchy defaults are preferable to editing keybindings directly;
- how to test whether a default actually works;
- where current Omarchy still has exceptions, such as file-manager behavior;
- how to diagnose a case where one launcher ignores your chosen default.

17.1 — What Is a "Default Application"?

A default application is the program the system chooses when another program says:

```
"Open this thing."
```

Examples:

```
Open this URL
→ browser

Open this directory
→ file manager

Edit this config
→ editor

Open a terminal
→ terminal emulator
```

The application requesting the action should ideally not need to know your personal preference.

It asks the desktop environment.

The desktop resolves the handler.

17.2 — Why Defaults Matter

Without a central default system:

```
browser shortcut
→ Chromium

Slack link
→ Firefox

terminal link
→ Chrome

GitHub CLI
→ Brave
```

That is inconsistent.

A good workstation aims for:

```
one declared default
→ many launch paths honor it
```

This improves:

- predictability;
- muscle memory;

- reproducibility;
- application interoperability.

17.3 — Current Omarchy Default Categories

Current Omarchy 4 / Quattro exposes first-class default selection for:

```
Browser
Terminal
Editor
Agent
```

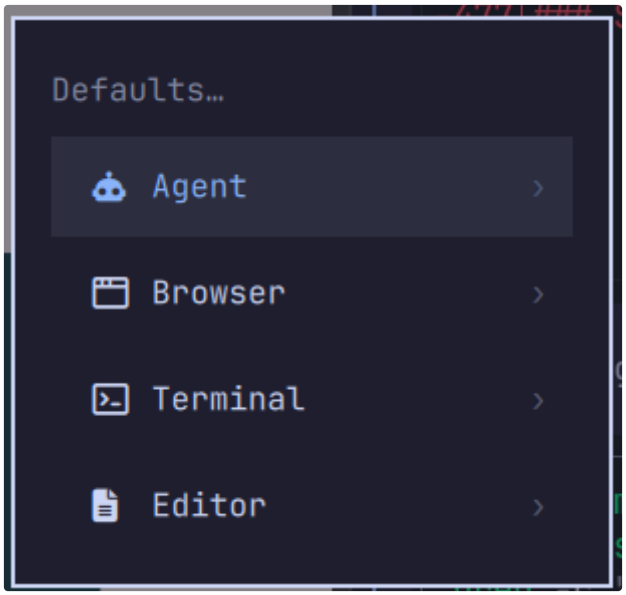
through:

```
Super + Space
→ Setup
→ Defaults
```

Current source and UI behavior treat these as explicit Omarchy default categories.

As of September 2026, the file manager is **not yet** an equivalent first-class Defaults category in the current Quattro source.

We will discuss that limitation later.



Screenshot 17.3-A
Setup → Defaults menu with Browser, Terminal, Editor, and Agent categories

17.4 — Current Course Workstation Defaults

The workstation used while building this course currently uses:

```
Browser
→ Chromium

Terminal
→ Ghostty

Editor
→ Neovim

Agent
→ OMP
```

These are workstation choices.

They are not mandatory course requirements.

Notice that current official Omarchy documentation describes:

```
Foot
```

as the fresh-install terminal default, while this course workstation has deliberately switched to:

```
Ghostty
```

That is exactly what the defaults layer is for.

17.5 — Inspect the Default Browser

Run:

```
omarchy default browser
```

Current official Omarchy browser documentation states that running this command with no browser argument prints the current default.

Example:

```
chromium
```

or:

```
firefox
```

depending on your machine.

17.6 — Set the Default Browser

Current official syntax:

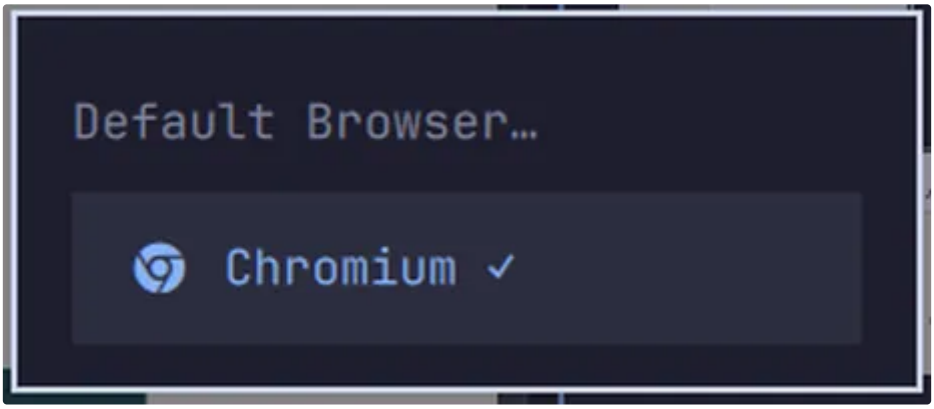
```
omarchy default browser firefox
```

or another installed supported browser.

The UI path:

```
Super + Space
→ Setup
→ Defaults
→ Browser
```

The menu currently lists installed browsers and marks the active one.



Screenshot 17.6-A
Browser default menu with active selection marked

17.7 — Installing a Browser Does Not Automatically Make It Default

This is an important current Omarchy behavior.

Suppose you install:

```
Firefox
```

through:

```
Install → Browser
```

That does not necessarily replace Chromium as the active default.

Installation and default selection are separate actions.

Workflow:

```
install application
↓
choose default
```

This is a good system-design pattern.

17.8 — Current Supported Browser Options

Current official browser documentation lists install options including:

```
Chromium
Chrome
Edge
Brave
Brave Origin
Firefox
Zen
```

The exact list may evolve.

Use:

```
Install → Browser
```

on the current system instead of treating a printed list as permanent.

17.9 — Browser Default and XDG

Current official Omarchy browser documentation explicitly states that:

```
omarchy default browser ...
```

sets the XDG handler.

That means it is intended to affect more than:

```
Omarchy's browser shortcut
```

It should also affect applications that ask the desktop to open a web link.

Conceptually:

```
Omarchy default browser
        ↓
XDG handler
        ↓
xdg-open / desktop apps / links
```

17.10 — Test the Browser Default

After changing the browser:

```
omarchy default browser
```

Then:

```
xdg-open https://example.com
```

Expected:

```
the selected default browser opens
```

Then test:

```
Super + Shift + Return
```

Expected:

```
same preferred browser family
```

If one path differs, investigate the launcher rather than assuming the default command failed.

17.11 — `xdg-settings`

The XDG utilities include:

```
xdg-settings
```

For browsers, a useful inspection command is:

```
xdg-settings get default-web-browser
```

This typically returns a desktop-file ID such as:

```
chromium.desktop
```

or:

```
firefox.desktop
```

This is lower-level than:

```
omarchy default browser
```

but valuable for diagnosis.

17.12 — What Is a `.desktop` ID?

Linux desktop applications commonly install metadata files ending in:

```
.desktop
```

Examples:

```
firefox.desktop
chromium.desktop
com.mitchellh.ghostty.desktop
```

These files describe things such as:

- application name;
- executable command;
- icon;
- supported MIME types;
- desktop integration.

The filename acts as an application identifier in many XDG workflows.

17.13 — Why Omarchy Can Say `ghostty` While XDG Says `com.mitchellh.ghostty.desktop`

Omarchy gives humans a simple command:

```
omarchy default terminal ghostty
```

Internally, current Quattro maps that to the desktop ID:

```
com.mitchellh.ghostty.desktop
```

This is normal.

Think of it as:

```
friendly Omarchy name
→ desktop application ID
```

17.14 — Default Terminal

Current official Omarchy terminal documentation says fresh Omarchy uses:

```
Foot
```

by default.

Current supported terminal options include:

```
Foot
Alacritty
Ghostty
Kitty
```

through the Omarchy terminal installation/default workflow.

17.15 — Inspect the Default Terminal

Run:

```
omarchy default terminal
```

On the course workstation:

```
ghostty
```

should be expected.

Your machine may show another value.

17.16 — Set the Default Terminal

Current syntax:

```
omarchy default terminal ghostty
```

or:

```
omarchy default terminal foot
```

or another supported terminal.

UI path:

```
Super + Space
→ Setup
→ Defaults
→ Terminal
```

17.17 — Current Terminal Default Uses `xdg-terminal-exec`

Current Quattro source explicitly describes:

```
omarchy default terminal
```

as setting the default used by:

```
xdg-terminal-exec
```

Current implementation writes:

```
~/.config/xdg-terminals.list
```

with the selected desktop application ID.

That is a very useful example of Omarchy integrating with a desktop standard rather than hard-coding every terminal launch.

17.18 — Inspect the XDG Terminal Preference

Run:

```
cat ~/.config/xdg-terminals.list
```

On a Ghostty-based workstation, the current file may contain:

```
com.mitchellh.ghostty.desktop
```

Current implementation comments describe the first valid entry as the preferred terminal.

17.19 — Inspect Through `xdg-terminal-exec`

Current Quattro terminal default code uses:

```
xdg-terminal-exec --print-id
```

to discover the active terminal desktop ID.

Run:

```
xdg-terminal-exec --print-id
```

if supported by your installed version.

Then compare it with:

```
omarchy default terminal
```

The values may be expressed differently but should correspond to the same application.

17.20 — Test the Terminal Default

After changing the default terminal:

```
Super + Return
```

should open the selected terminal.

Also test a workflow that uses terminal dispatch indirectly if available.

The important question is:

> *Does the launcher resolve the current default dynamically?*

17.21 — Default Editor

Current official Omarchy development documentation says Neovim ships as the default editor.

Alternative editor installation options currently include:

```
VS Code
Cursor
Zed
Sublime Text
Helix
Vim
Emacs
```

Current Omarchy exposes the system-wide editor choice under:

```
Setup
→ Defaults
→ Editor
```

17.22 — Inspect the Default Editor

Run:

```
omarchy default editor
```

if supported by the current installed CLI.

If exact CLI behavior differs, inspect:

```
omarchy default --help
```

and:

```
omarchy commands --all | grep -i editor
```

The installed CLI is the final source of truth.

17.23 — Set the Default Editor

Use the current Omarchy UI:

```
Super + Space
→ Setup
→ Defaults
→ Editor
```

Or use the corresponding current CLI command if available.

The goal is that Omarchy's config-edit actions can open your chosen editor without hard-coded per-feature changes.

17.24 — Default Agent

The agent default behaves conceptually like the other defaults, but it is not an XDG association.

Current Omarchy stores the chosen coding-agent name in:

```
~/.config/omarchy/defaults/agent
```

Set:

```
omarchy default agent omp
```

Inspect:

```
omarchy default agent
```

This is an Omarchy-specific default rather than a freedesktop desktop standard.

17.25 — XDG: What Does It Actually Mean?

"XDG" is commonly used to refer to desktop interoperability standards and tools from the freedesktop ecosystem.

For this course, you mainly need to understand three practical ideas:

```
1. application metadata
2. content-type handlers
3. standard launch/open behavior
```

You do **not** need to study the entire XDG specification.

17.26 — MIME Types

A MIME type identifies the kind of content being opened.

Examples:

```
text/plain
image/png
application/pdf
inode/directory
```

The desktop can associate a MIME type with a default `.desktop` application.

Conceptually:

```
application/pdf
→ PDF viewer.desktop
```

17.27 — URL Scheme Handlers

URLs also have scheme handlers.

Common schemes:

```
http
https
mailto
```

A desktop system can associate:

```
x-scheme-handler/https
```

with the default browser.

This is one reason browser selection affects links opened from many unrelated applications.

17.28 — `xdg-mime`

Use:

```
xdg-mime query default MIME_TYPE
```

Example:

```
xdg-mime query default text/plain
```

or:

```
xdg-mime query default inode/directory
```

This reveals the currently associated `.desktop` application.

17.29 — Inspect the Default Directory Handler

Run:

```
xdg-mime query default inode/directory
```

You may see a file-manager desktop ID.

This tells you the XDG association for directories.

However, current Omarchy has an important exception here.

17.30 — Current File-Manager Limitation in Omarchy Quattro

As of September 2026, current Omarchy Quattro does **not yet** expose:

```
File Manager
```

as a first-class:

```
Setup → Defaults
```

category equivalent to browser, terminal, editor, and agent.

A current open Omarchy issue documents that the default file-manager launcher still resolves directly to Nautilus rather than dynamically honoring the XDG directory handler in all Omarchy launch paths.

This means:

```
xdg-mime default org.kde.dolphin.desktop inode/directory
```

may successfully change the XDG directory handler while an Omarchy file-manager hotkey can still open Nautilus.

This is a current implementation limitation, not a general XDG rule.

17.31 — Why Current Limitations Matter in a Course

A bad handbook would say:

```
"Set XDG default and everything will always follow it."
```

That would be too broad.

A better handbook says:

```
XDG defines the association
but an application can bypass it by hard-coding a launcher
```

That distinction is extremely useful for troubleshooting.

17.32 — `xdg-open`

`xdg-open` asks the desktop environment to open a resource using its associated application.

Examples:

```
xdg-open https://example.com
```

```
xdg-open README.md
```

```
xdg-open .
```

Conceptually:

```
resource
→ determine type/scheme
→ resolve default handler
→ launch application
```

17.33 — `xdg-open` Is a Great Diagnostic Tool

Suppose:

```
clicking a URL in App A opens the wrong browser
```

Test:

```
xdg-open https://example.com
```

If `xdg-open` uses the correct browser:

```
XDG default is probably correct
```

The problematic application may be:

- using its own browser preference;
- hard-coding a browser;
- launching through another mechanism.

17.34 — Test a File Handler

Create:

```
echo "hello" > /tmp/xdg-test.txt
```

Then:

```
xdg-open /tmp/xdg-test.txt
```

Observe which application opens the file.

Query:

```
xdg-mime query default text/plain
```

The two should normally align.

17.35 — Changing MIME Associations Manually

Low-level syntax:

```
xdg-mime default APPLICATION.desktop MIME_TYPE
```

Example:

```
xdg-mime default some-editor.desktop text/plain
```

But:

> *Prefer Omarchy's own Defaults UI/CLI when Omarchy already exposes that category.*

Use raw XDG commands primarily when:

- diagnosing;
- handling unsupported categories;
- following upstream application guidance.

17.36 — Why Omarchy Defaults Are Better Than Editing Hotkeys

Suppose you dislike Chromium.

Bad approach:

```
edit browser hotkey
→ hard-code firefox
```

Now:

```
Super + Shift + Return
→ Firefox

but xdg-open
→ Chromium
```

You created inconsistency.

Better:

```
omarchy default browser firefox
```

Now Omarchy updates the intended system-level browser default integration.

17.37 — Why Omarchy Defaults Are Better Than Editing Internal Scripts

Do not edit:

```
/usr/share/omarchy/bin/...
```

just to change your default application.

Those files are package-owned.

Updates may replace them.

Use:

```
Setup → Defaults
```

or the public CLI.

17.38 — Environment Variables Are Not the Whole Story

Linux applications may sometimes inspect variables such as:

```
EDITOR
VISUAL
BROWSER
```

But modern graphical desktop default handling also uses XDG associations and `.desktop` metadata.

Do not assume:

```
export BROWSER=firefox
```

is universally sufficient.

Current Omarchy explicitly uses proper browser/XDG integration for its supported default workflow.

17.39 — `$EDITOR`

Command-line tools often use:

```
$EDITOR
```

to determine which editor to open.

Examples can include:

- Git;
- CLI configuration tools;
- text-mode programs.

Inspect:

```
echo "$EDITOR"
```

Also inspect:

```
echo "$VISUAL"
```

These are separate from MIME associations.

17.40 — Git Editor vs System Editor

Git can have its own configured editor.

Inspect:

```
git config --global core.editor
```

If empty, Git may fall back to environment/default behavior.

A system-wide Omarchy editor preference and a Git-specific override are not necessarily the same layer.

Understand which one you are changing.

17.41 — Application-Specific Defaults Can Override System Defaults

Some applications maintain their own preferences.

Example:

```
App setting:
"Open links in internal browser"
```

or:

```
terminal configured explicitly inside an IDE
```

In such cases:

```
system default
```

may be correct while the application intentionally overrides it.

This is not a desktop bug.

17.42 — Web Apps Are a Special Case

Current Omarchy uses Chromium-family browser integration for web-app behavior.

Current official browser documentation says Chromium-family browsers receive Omarchy-specific integration, and Omarchy web apps are designed around that system.

Therefore:

```
default browser
```

and:

```
web-app runtime
```

should not always be assumed to be identical concepts.

This is particularly relevant if you prefer Firefox or another non-Chromium browser for normal browsing.

17.43 — Do Not Hack Web-App Internals Just to Match the Browser

If current Omarchy's web-app launcher does not behave exactly like your preferred default browser:

```
do not immediately overwrite internal scripts
```

First determine:

- whether this is intended current behavior;
- whether the official implementation supports your browser;
- whether there is a current upstream solution.

Package-owned script modifications create maintenance debt.

17.44 — Current Browser Shortcut

Current official hotkeys include:

```
Super + Shift + Return
→ Browser
```

The browser launcher is intended to follow the chosen default browser.

Test it after changing defaults.

17.45 — Current Terminal Shortcut

Current official hotkeys include:

```
Super + Return
→ Terminal
```

Current official terminal docs say the binding dynamically follows the selected default terminal.

That means:

```
switch Foot → Ghostty
```

should not require rewriting the keybinding.

17.46 — Current Editor Shortcut

Current official hotkeys include:

```
Super + Shift + N
→ Editor
```

The selected default editor should be used by Omarchy's editor integration.

Again:

```
choose default
```

is better than:

```
rewrite hotkey
```

17.47 — The Best Default-App Test Matrix

When you change a default, test multiple paths.

Browser

```
omarchy default browser
xdg-settings get default-web-browser
xdg-open https://example.com
Super + Shift + Return
```

Terminal

```
omarchy default terminal
xdg-terminal-exec --print-id
Super + Return
```

Editor

```
Omarchy default/editor command
Super + Shift + N
Setup → Config
```

This catches inconsistent launchers.

17.48 — Diagnosing "Wrong Browser Opens"

Use:

```
1. omarchy default browser
2. xdg-settings get default-web-browser
3. xdg-open https://example.com
4. test Omarchy browser hotkey
5. test the specific problematic application
```

Now classify the problem.

17.49 — Browser Diagnosis Outcomes

CASE A

```
Omarchy default wrong
XDG wrong
```

Fix the default.

CASE B

```
Omarchy default correct
XDG correct
xdg-open correct
specific app wrong
```

The application likely has its own behavior.

CASE C

```
Omarchy browser shortcut wrong
xdg-open correct
```

Investigate the current Omarchy launcher/config, not XDG itself.

17.50 — Diagnosing "Wrong Terminal Opens"

Run:

```
omarchy default terminal
```

Then:

```
xdg-terminal-exec --print-id
```

Then:

```
Super + Return
```

If the selected app differs, inspect:

```
~/.config/xdg-terminals.list
```

and current Omarchy terminal documentation/source.

Do not hard-code the terminal inside personal keybindings unless you intentionally want a special non-default shortcut.

17.51 — Special Terminal Shortcuts Can Intentionally Ignore the General Default

Example:

```
Super + Return
→ default terminal

Super + Alt + Return
→ terminal + Tmux workflow
```

The second shortcut is a specific workflow rather than merely "open any terminal."

This is a valid distinction.

17.52 — Default vs Explicit Application Binding

Understand the difference:

```
"Open my default browser"
```

vs:

```
"Open Firefox specifically"
```

Both are valid.

Use default-resolution when your intention is generic.

Use an explicit app only when the application identity itself matters.

17.53 — Reproducibility and Defaults

Your workstation setup documentation should record:

```
browser
terminal
editor
agent
```

Example:

```
Browser: Chromium
Terminal: Ghostty
Editor: Neovim
Agent: OMP
```

Later, Module 22 will incorporate these into your reproducible workstation process.

17.54 — Practical Exercise: Record Current Defaults

Run:

```
omarchy default browser
```

```
omarchy default terminal
```

```
omarchy default agent
```

Inspect the current editor through the appropriate default command/menu.

Create:

```
Browser:
Terminal:
Editor:
Agent:
```

and fill it in.

17.55 — Practical Exercise: Inspect Browser XDG State

Run:

```
xdg-settings get default-web-browser
```

Compare with:

```
omarchy default browser
```

Example relationship:

```
Omarchy:
chromium

XDG:
chromium.desktop
```

The names differ in representation but refer to the same application.

17.56 — Practical Exercise: Test `xdg-open`

Run:

```
xdg-open https://example.com
```

Verify the selected browser.

Then:

```
Super + Shift + Return
```

Verify the normal Omarchy browser launcher.

Both should make sense relative to your chosen browser.

17.57 — Practical Exercise: Inspect Terminal XDG State

Run:

```
cat ~/.config/xdg-terminals.list
```

Then:

```
xdg-terminal-exec --print-id
```

if supported.

Compare with:

```
omarchy default terminal
```

Understand the friendly name vs `.desktop` ID relationship.

17.58 — Practical Exercise: Change Terminal Temporarily

If you have two supported terminals installed, switch:

```
omarchy default terminal OTHER_TERMINAL
```

Press:

```
Super + Return
```

Confirm.

Then switch back to your preferred terminal.

If only one terminal is installed, skip the change.

17.59 — Practical Exercise: Query MIME Handlers

Run:

```
xdg-mime query default text/plain
```

Then:

```
xdg-mime query default inode/directory
```

Record the returned desktop IDs.

You do not need to change them.

17.60 — Practical Exercise: Inspect a `.desktop` File

Suppose:

```
xdg-settings get default-web-browser
```

returns:

```
chromium.desktop
```

Find it:

```
fd chromium.desktop /usr/share/applications ~/.local/share/applications 2>/dev/null
```

Then inspect:

```
bat /usr/share/applications/chromium.desktop
```

if that is the resolved path.

Look for:

```
Name=
Exec=
MimeType=
```

Do not edit package-owned desktop files.

17.61 — Practical Exercise: Test Directory Handling

Run:

```
xdg-mime query default inode/directory
```

Then:

```
xdg-open ~
```

Observe the file manager.

Then test Omarchy's file-manager hotkey.

If they differ on a current release:

```
you have observed a launcher/XDG mismatch
```

rather than a mysterious Linux failure.

17.62 — Practical Exercise: Editor Integration

Open:

```
Super + Space
→ Setup
→ Defaults
→ Editor
```

Verify the selected editor.

Then launch:

```
Super + Shift + N
```

Finally open an Omarchy config through:

```
Setup → Config
```

Confirm that the default editor integration behaves consistently.

17.63 — Common Default/XDG Mistakes

Mistake 1 — Editing a Keybinding to Change the Default Browser

Use:

```
omarchy default browser ...
```

instead.

Mistake 2 — Assuming Installing an App Makes It Default

Installation and default selection are separate.

Mistake 3 — Confusing Omarchy Names with `.desktop` IDs

Example:

```
ghostty
```

vs:

```
com.mitchellh.ghostty.desktop
```

Mistake 4 — Assuming `$BROWSER` Controls Every Graphical Link

Modern desktop integration uses XDG handlers as well.

Mistake 5 — Assuming Every App Honors XDG

Applications can override or hard-code behavior.

Mistake 6 — Assuming File Manager Is Currently a First-Class Omarchy Default

As of September 2026, current Quattro still has a documented gap here.

Mistake 7 — Editing `/usr/share/applications/*.desktop` Directly

Package updates can replace those files.

Mistake 8 — Editing Omarchy Internal Launch Scripts

Use public defaults/configuration interfaces first.

Mistake 9 — Assuming Web Apps Must Use the Same Runtime as Your Normal Browser

Current Omarchy web-app integration has its own browser constraints.

Mistake 10 — Testing Only One Launch Path

A default is only useful if the launch paths you care about actually honor it.

17.64 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Omarchy Defaults

- [ ] I know the current first-class default categories.
- [ ] I can inspect the default browser.
- [ ] I can inspect the default terminal.
- [ ] I know where to change the default editor.
- [ ] I can inspect/change the default agent.
- [ ] I understand installation does not automatically imply default selection.

XDG Concepts

- [ ] I understand what a `.desktop` ID is.

- [ ] I understand MIME types at a practical level.
- [ ] I understand URL scheme handlers.
- [ ] I know what `xdg-open` does conceptually.
- [ ] I can inspect the default browser with `xdg-settings`.
- [ ] I can inspect MIME associations with `xdg-mime`.

Terminal Integration

- [ ] I understand `xdg-terminal-exec` at a high level.
- [ ] I know Omarchy writes `~/.config/xdg-terminals.list`.
- [ ] I understand the friendly-name vs desktop-ID mapping.

Troubleshooting

- [ ] I test more than one launch path.
- [ ] I understand that applications can override system defaults.
- [ ] I know current file-manager behavior is an Omarchy exception/limitation.
- [ ] I do not edit package-owned launchers merely to change a preference.

17.65 — What You Can Now Do

After completing Module 17, you can now:

- set Omarchy defaults cleanly;
- understand why links open in a particular browser;
- verify browser integration with XDG;
- understand terminal-default resolution;
- distinguish friendly application names from `.desktop` IDs;
- inspect MIME and directory handlers;
- diagnose applications that ignore defaults;
- avoid unnecessary keybinding hacks;
- understand current Omarchy file-manager limitations;
- document your preferred application stack reproducibly.

Most importantly:

> *You now understand the difference between "I bound a key to this app" and "the workstation considers this app the system default."*

Module 17 Summary

Current first-class Omarchy default categories:

```
Browser
Terminal
Editor
Agent
```

Browser:

```
omarchy default browser
omarchy default browser firefox
```

XDG browser check:

```
xdg-settings get default-web-browser
```

Open through desktop association:

```
xdg-open https://example.com
```

Terminal:

```
omarchy default terminal
omarchy default terminal ghostty
```

Current terminal preference file:

```
~/.config/xdg-terminals.list
```

Inspect:

```
xdg-terminal-exec --print-id
```

MIME handler inspection:

```
xdg-mime query default text/plain
xdg-mime query default inode/directory
```

Core model:

```
Omarchy preference
        ↓
desktop/XDG integration where applicable
        ↓
launcher/app
```

Current exception:

```
file manager
→ not yet equivalent first-class Defaults category
→ some Omarchy launch paths can still be hard-coded to Nautilus
```

And the most important rule:

> ***Change a default at the default layer. Do not patch every launcher individually.***

MODULE 18

18

Updates and Maintenance on Omarchy

You now have a customized, AI-native development workstation.

The next challenge is keeping it healthy.

About This Module

You now have a customized, AI-native development workstation.

The next challenge is keeping it healthy.

Omarchy is Arch-based, but current Omarchy 4 / Quattro deliberately does **not** treat system maintenance as:

```
sudo pacman -Syu
```

and nothing else.

Current Omarchy wraps system updates in its own update pipeline because an Omarchy update can involve more than package replacement.

The current official update process includes:

- package updates;
- Omarchy updates;
- per-user migrations;
- pre-update snapshot creation;
- update hooks;
- mise-managed development-tool updates;
- restart/reboot checks;
- update-log analysis;
- update-state refresh.

The blessed update command is:

```
omarchy update
```

The goal of this module is:

> ***Update the workstation through the supported path, understand what Omarchy is protecting for you, recover safely when an update goes wrong, and develop a sensible maintenance routine without turning the machine into a full-time Arch maintenance hobby.***

What You Will Learn

By the end of Module 18, you will understand:

- the supported Omarchy update path;
- why direct `pacman -Syu` is guarded;
- what happens during `omarchy update`;
- what Omarchy migrations are;
- why update snapshots matter;
- what snapshots do and do not protect;
- how the shell update indicator works;
- how to inspect your Omarchy version;
- the difference between stable, RC, edge, and dev channels;
- why most users should remain on stable;
- how firmware updates differ from normal package updates;
- what happens to mise-managed tools during an Omarchy update;
- why the update process checks free disk space;
- where update logs are written;
- when rebooting after an update is appropriate;
- how to think about orphan-package cleanup;
- how to handle a failed update without immediately reinstalling the machine;
- how to build a practical weekly/monthly maintenance routine.

18.1 — The Supported Update Command

Current official Omarchy CLI:

```
omarchy update
```

This is the primary supported system-update path.

You can also launch it from:

```
Super + Space
→ Update
→ Omarchy
```

or by clicking the current Omarchy update indicator in the top bar when an update is waiting.

18.2 — Why Omarchy Wraps the Update

On plain Arch, you may be used to:

```
sudo pacman -Syu
```

But Omarchy adds system-level behavior around the package transaction.

Current official Omarchy update documentation says the update flow owns:

```
package transaction
+
migrations
+
post-update hooks
+
update-state refresh
+
restart checks
```

That means bypassing Omarchy's updater can leave the package state updated while Omarchy-specific follow-up steps remain incomplete.

18.3 — Do Not Use `pacman -Syu` as Your Normal Omarchy Update

Current Omarchy actively guards direct full-system pacman upgrades.

If you run:

```
sudo pacman -Syu
```

the current update guard normally stops the transaction and tells you to use:

```
omarchy update
```

instead.

This is intentional.

18.4 — Why the Guard Exists

The guard protects you from accidentally skipping:

- the pre-update snapshot;

- Omarchy migrations;
- configuration transitions;
- post-update hooks;
- restart handling;
- other Omarchy update coordination.

The important lesson is:

> ***Arch knowledge is useful on Omarchy, but Omarchy's supported maintenance workflow still comes first.***

18.5 — The High-Level Update Pipeline

Current official update-process documentation describes a flow conceptually like:

```
omarchy update
        ↓
check free disk space
        ↓
confirm update
        ↓
prune old package-cache versions
        ↓
create system snapshot
        ↓
temporarily inhibit sleep
        ↓
update packages
        ↓
run Omarchy migrations
        ↓
update mise-managed tools
        ↓
run post-update hook
        ↓
analyze update log
        ↓
refresh update status
        ↓
restore normal idle behavior
        ↓
restart/reboot checks
```

You do not need to memorize every internal helper.

You should understand what the wrapper is doing for you.

18.6 — Check the Current Version

Use:

```
omarchy version
```

This is the correct current public CLI form.

Do **not** assume:

```
omarchy --version
```

works.

That was one of the real mistakes encountered during the course setup.

18.7 — Version Verification After an Update

After updating:

```
omarchy version
```

Then, if a kernel update occurred:

```
uname -r
```

You may also check your core working state:

```
git --version
mise --version
```

and the specific runtimes you depend on.

You do not need to inventory the whole machine after every update.

18.8 — The Update Indicator

Current Omarchy Shell includes an update indicator near the clock.

The current official update-process documentation says the shell checks for Omarchy updates:

```
on shell startup
+
approximately every six hours
```

When an update is available, the update badge appears.

Clicking it launches the update flow in a terminal.

18.9 — You Do Not Need to Update Every Five Minutes

Omarchy is built on a rolling Linux base, but compulsive update checking adds little value.

A sensible workstation rhythm is:

```
when the shell shows an update
or
periodically when convenient
```

For a development machine, update when you have enough time to:

- let the update finish;
- reboot if required;
- validate your core workflow.

Do not start a major update two minutes before an important meeting.

18.10 — Free-Space Protection

Current official update-process documentation says:

```
omarchy update
```

checks free space on:

```
/
```

before proceeding.

The current threshold is:

```
10 GiB
```

If available root space is below that threshold, the updater aborts before the package transaction.

This helps prevent update failures caused by a nearly full system filesystem.

18.11 — Do Not Force Past Low Disk Space Casually

Current internals provide a force mechanism for exceptional cases.

That does not mean you should use it as the normal answer.

If the updater says:

```
not enough disk space
```

the correct response is:

```
investigate disk usage
```

not:

```
force the update
```

We will cover deeper storage troubleshooting later.

18.12 — Package Cache Maintenance

Current update logic trims the pacman package cache before creating the snapshot.

The current official process keeps:

```
two cached versions per package
```

through the package-cache pruning step.

This provides a balance:

```
retain downgrade options
+
avoid unlimited cache growth
```

You generally do not need to manually purge the whole pacman cache.

18.13 — Why Keeping Some Old Packages Is Useful

Suppose:

```
new package version
→ regression
```

Having an older package version cached can help with diagnosis or downgrade.

Deleting the entire cache after every update removes that safety net.

Omarchy's current maintenance process intentionally keeps a small history.

18.14 — Automatic Update Snapshots

Current official Omarchy snapshot documentation states that Omarchy creates a system snapshot automatically on every normal Omarchy update.

This gives you a rollback point from immediately before the update.

Conceptually:

```
working system
        ↓
snapshot
        ↓
update
        ↓
problem?
        ↓
boot previous snapshot
```

This is one of the strongest recovery features in Omarchy.

18.15 — Manual Snapshot Creation

Current official snapshot command is:

```
omarchy-snapshot create
```

Use this before a risky system-level experiment when you want an additional manual recovery point.

Examples:

- boot configuration changes;
- low-level graphics changes;
- risky package experiments;

- significant system modifications.

18.16 — Snapshots Are Not Backups of Your Personal Files

This distinction is critical.

Current official Omarchy documentation says snapshot restoration restores the:

```
root filesystem
```

but **not** your:

```
/home
```

That means snapshots are excellent for:

```
broken system update
```

but not for:

```
deleted project
lost documents
corrupted personal config
```

18.17 — Why `~/.config` Is Not Rolled Back

Your personal configuration normally lives under:

```
/home/YOUR_USER/.config
```

Since `/home` is not restored by the system snapshot:

```
~/.config
```

stays at its current state.

This can matter when rolling the system backward.

Example:

```
new app version
→ writes new config format

system snapshot restore
→ old app version returns

~/.config
→ still contains newer config
```

You may need to resolve that manually.

18.18 — Snapshot vs Dotfiles vs Backup

Use the correct recovery tool for the correct layer.

```
System snapshot
→ recover root/system update state

Dotfiles Git repository
→ recover intentional personal configuration

Project Git repositories
→ recover source-code history

Real file backup
→ recover personal data/files
```

No single one replaces the others.

18.19 — Booting a Snapshot

Current official Omarchy snapshot documentation says snapshots are bootable through:

```
Limine
```

On a normal Limine boot:

```
restart
→ select the appropriate snapshot
→ boot it
```

Snapshots are identified by date/version in the boot interface.

Screenshot 18.19-A
Limine boot menu with snapshot entries by date

18.20 — Direct Boot Changes Snapshot Access

Current Omarchy can be configured to boot directly into Omarchy rather than showing Limine.

If direct boot is enabled, getting to a snapshot requires entering the firmware/BIOS boot menu and choosing Limine.

This is a convenience/recovery trade-off.

18.21 — Snapshot Support Depends on the Boot Setup

Current official snapshot documentation says this bootable-snapshot feature is supported on Omarchy installations using:

```
Limine
```

and is not available in the same way on:

```
GRUB
systemd-boot
```

The course workstation uses Limine, so snapshot rollback is part of our expected recovery plan.

18.22 — Restoring a Snapshot

After booting a snapshot, current Omarchy can offer a notification to begin restoration.

The current manual also documents the restore helper:

```
omarchy-snapshot restore
```

Do not restore blindly.

First confirm:

- you booted the intended snapshot;
- the problem truly began after the later update;
- your personal `/home` state is understood.

18.23 — Omarchy Migrations

Current Omarchy updates can include:

```
migrations
```

A migration is a one-time transition that changes state package installation alone cannot safely own.

Examples can involve:

- user config;
- user systemd state;
- application preferences;
- session state;
- machine-level repair coordinated from the user update.

18.24 — Migrations Run Per User

Current official migration design stores completion state under:

```
~/.local/state/omarchy/migrations/
```

That means migration completion is tracked per user.

This is important on shared machines.

Each user gets the required user-level transitions.

18.25 — Normal Migration Command

The normal supported path is simply:

```
omarchy update
```

which runs pending migrations.

You normally do **not** need to run the migration engine manually.

For diagnosis, the current public migration helper is:

```
omarchy-migrate
```

and pending-state inspection supports:

```
omarchy-migrate --pending
```

Use this mainly when the official update flow or notification tells you migrations remain pending.

18.26 — Why Direct Pacman Updates Can Leave Pending Migrations

If a user deliberately bypasses the Omarchy pacman guard, package files may update without that user's normal Omarchy migration flow.

Current Quattro can detect pending per-user migrations at graphical login and notify the user.

The system does **not** silently run those migrations in the background.

That is a deliberate safety/design choice.

18.27 — Mise Is Part of the Blessed Update Path

Current official update-process documentation says:

```
mise-managed tools
```

are updated as part of the normal Omarchy update pipeline.

The current internal step uses:

```
mise up
```

with Omarchy's intended update policy.

This means your globally mise-managed development tools are part of routine workstation maintenance.

18.28 — Project Runtime Pinning Still Protects Projects

Suppose your global Node updates.

A project with:

```
mise.toml
```

requesting a specific version still has an explicit project environment.

This is another reason project-level version declarations matter.

System/workstation updates and project reproducibility are separate layers.

18.29 — Post-Update Hooks

Current Omarchy update flow runs:

```
post-update
```

hooks after package/migration work.

Your user hooks live under:

```
~/.config/omarchy/hooks/post-update.d/
```

as covered in Module 14.

This is where safe, non-destructive personal maintenance automation can plug into the update lifecycle.

18.30 — Keep Post-Update Hooks Lightweight

Good:

```
record version
run a quick status check
update harmless metadata
```

Risky:

```
rewrite boot config
delete large amounts of data
push repositories
modify unrelated system state
```

An update is already a complex event.

Do not turn it into an uncontrolled chain of personal scripts.

18.31 — Update Log

Current official update-process documentation says the update transcript is written to:

```
/tmp/omarchy-update.log
```

If an update fails:

```
less /tmp/omarchy-update.log
```

or:

```
bat /tmp/omarchy-update.log
```

can be useful.

This gives you real evidence instead of trying to remember terminal output.

18.32 — Update Log Analysis

Current Omarchy includes an internal update-log analysis step that checks for high-signal known failure patterns.

Current documentation specifically mentions initramfs-generation failures as one current class of analysis.

This is useful, but it does not replace reading the visible error when something goes wrong.

18.33 — Keep the Update Terminal Open When Something Fails

If the update reports an error:

```
do not immediately close the terminal
```

Capture:

- the visible error;
- the stage where it happened;
- whether packages finished;
- whether migrations ran;
- whether a reboot was requested.

Then inspect:

```
/tmp/omarchy-update.log
```

18.34 — Update Sleep Inhibition

Current Omarchy temporarily inhibits idle/sleep behavior during the update pipeline.

This reduces the chance that the workstation suspends halfway through a package transaction.

After the update, Omarchy restores the idle state it changed.

This connects directly to Module 16's runtime-control model.

18.35 — Restart Handling After Updates

Current official update process has a final restart/reboot stage.

The updater can detect when components require restart.

Current design includes:

- component restart markers;
- shell restart after every update;
- reboot prompting after important updates such as kernel/Hyprland cases.

Follow the updater's guidance.

18.36 — Do Not Reboot in the Middle of the Update Pipeline

Wait until:

```
package transaction
+
migrations
+
hooks
+
restart checks
```

finish.

If the updater asks for a reboot at the end:

```
reboot then
```

Interrupting package installation can create a much harder recovery problem.

18.37 — Kernel Updates

When the kernel updates, a reboot is generally required before the running system actually uses the new kernel.

Check before reboot:

```
uname -r
```

After reboot:

```
uname -r
```

You can compare if diagnosing whether the new kernel is active.

Normally, simply following Omarchy's restart prompt is enough.

18.38 — Firmware Updates Are Separate

Current official Omarchy update documentation explicitly keeps firmware updates separate from normal:

```
omarchy update
```

Use:

```
Super + Space
→ Update
→ Firmware
```

when you want to check hardware firmware.

The current firmware updater uses:

```
fwupd
```

and installs it if necessary.

18.39 — What Firmware Can Cover

Depending on hardware vendor support, LVFS/fwupd may provide firmware for:

- system BIOS/UEFI;
- SSDs;
- docks;
- peripherals;
- other supported hardware.

Not every device/vendor publishes firmware through Linux Vendor Firmware Service.

18.40 — Firmware Capability Does Not Mean an Update Is Available

A machine can support:

```
fwupd
```

while currently having:

```
no firmware update waiting
```

These are different facts.

This was another practical lesson from the course workstation.

Do not interpret:

```
firmware supported
```

as:

```
firmware outdated
```

18.41 — Firmware May Require Reboot

Many firmware updates are staged and applied during reboot.

That is normal.

Plan firmware maintenance when you can tolerate:

- a restart;
- potentially longer boot time;
- firmware update screens;
- temporary device unavailability.

Do not power off halfway through a firmware flash.

18.42 — Update Channels

Current Omarchy has four update channels:

```
stable
RC
edge
dev
```

They represent different levels of freshness and risk.

18.43 — Stable

Current official documentation says new installations begin on:

```
stable
```

Stable tracks official Omarchy releases.

Current Omarchy's stable Arch mirror intentionally lags the newest Arch packages by approximately:

```
one month
```

so Omarchy can catch incompatibilities/config-transition issues before those packages reach stable users.

For most users:

> ***Stable is the correct channel.***

18.44 — RC

The:

```
RC
```

channel is used for release-candidate testing before a major Omarchy release.

Use it when:

- you intentionally want to help test upcoming releases;
- you are comfortable reporting issues;
- you can recover from breakage.

Do not use RC merely because the name sounds newer.

18.45 — Edge

The:

```
edge
```

channel tracks more current Omarchy development packages and the newest Arch packages sooner.

Current official docs explicitly recommend it only for experienced Linux users who know how to recover from problems.

Edge means:

```
more freshness
+
more risk
```

18.46 — Dev

The:

```
dev
```

channel is for users working directly on Omarchy.

Current official docs describe it as linking the runtime to a Git checkout, typically in:

```
~/omarchy
```

and combining that with edge-style packages.

This is not a normal end-user channel.

18.47 — Change Channels Through Omarchy

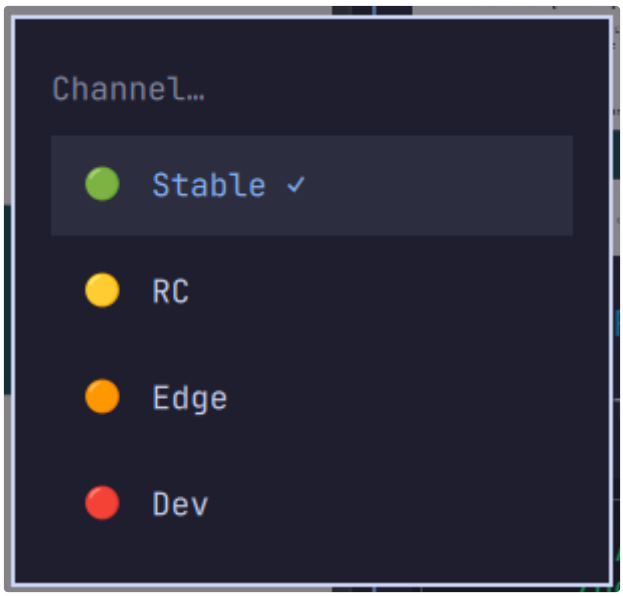
Current menu path:

```
Super + Space
→ Update
→ Channel
```

Current official docs also reference the underlying channel helper:

```
omarchy-channel-set <stable|rc|edge|dev>
```

For normal workstation use, the menu is preferable.



Screenshot 18.47-A
Update menu channel selection (Stable/RC/Edge/Dev)

18.48 — Do Not Channel-Hop Casually

Changing channels changes more than a label.

It can change:

- package repository selection;
- Omarchy package type;
- development checkout behavior;
- Arch package freshness.

Stay on stable unless you have a clear reason to participate in testing/development.

18.49 — Orphan Packages

An orphan package is generally a package that was installed as a dependency and is no longer required by another installed package.

Current Omarchy's update pipeline can identify orphan packages and prompt before removal.

Current unattended updates do **not** silently remove them.

That is a conservative and sensible design.

18.50 — Do Not Delete Packages Just Because They Are Called Orphans

Review the list.

Ask:

```
Do I recognize any of these?
Do I intentionally use one directly?
Was it manually installed in an unusual way?
```

Package metadata is useful, but human review still matters.

18.51 — Pacnew and Pacsave

Arch package management can create files such as:

```
.pacnew
.pacsave
```

when packaged configuration and local configuration differ.

Current Omarchy update-process documentation notes that broader handling of these still remains an area of concern.

For now, advanced users should understand the concept and inspect them when relevant.

We will revisit config-recovery/troubleshooting later rather than making beginners manually reconcile every package config after every update.

18.52 — A Sensible Maintenance Rhythm

For a normal stable-channel development workstation:

Regularly

- install Omarchy updates when indicated and convenient;
- reboot when the update requests it;
- verify your normal development workflow afterward.

Occasionally

- check firmware;
- review disk usage if root space becomes low;

- inspect orphan prompts during updates;
- verify snapshots/dotfiles/backups still form a coherent recovery plan.

Before Risky System Work

- create an extra manual snapshot;
- ensure personal data is backed up;
- commit important dotfile/project changes.

18.53 — Maintenance Is Not Constant Tinkering

A healthy maintenance philosophy is:

```
keep current
+
observe
+
recover cleanly
```

not:

```
run random cleanup commands
every day
```

Avoid maintenance cargo cults such as:

- constant cache deletion;
- forced package upgrades outside Omarchy;
- routine reinstalling;
- deleting logs because logs exist;
- rebooting after every small change.

18.54 — Pre-Update Checklist for Important Machines

Before a significant update:

```
1. finish/commit important work
2. ensure no critical long-running task is in progress
3. confirm adequate disk space
4. update when you have time to reboot/test
5. let Omarchy create its snapshot
```

You do **not** need this ceremony for every tiny update.

Use it when the machine is important and downtime matters.

18.55 — Post-Update Check

After update/reboot:

```
omarchy version
```

Then verify what you actually use:

```
terminal opens
network works
audio works
editor opens
Git works
agent launches
main project runtime works
```

Do not run a 200-command health audit unless something is wrong.

18.56 — Minimal Developer Verification

A practical quick check:

```
omarchy version
git --version
mise --version
```

Then:

```
cd ~/Projects/YOUR_MAIN_PROJECT
mise ls --current
git status
```

Launch your editor/agent and make sure your normal workflow is intact.

18.57 — If an Update Fails Before Package Changes

If the updater stops because of:

- insufficient free space;

- confirmation;
- a preflight issue;

the system may still be untouched.

Read the exact output.

Fix the cause.

Then rerun:

```
omarchy update
```

Do not escalate to snapshot recovery when nothing was changed.

18.58 — If Package Update Fails

Do not immediately:

```
reinstall Omarchy
```

Instead:

```
1. keep terminal output
2. inspect /tmp/omarchy-update.log
3. identify the package/error
4. check whether the updater provided a recovery action
5. consult current official docs/issues if necessary
```

Only then decide whether rollback is appropriate.

18.59 — If the System Boots but Something Is Broken

Use the smallest recovery scope.

Example:

```
audio broken
→ restart audio

shell broken
→ restart shell

one config broken
→ repair config

whole system broke after update
→ consider snapshot
```

Do not use the largest recovery tool first.

18.60 — If the System Does Not Boot After an Update

This is where the automatic pre-update snapshot becomes especially valuable.

Recovery concept:

```
firmware boot menu if necessary
↓
Limine
↓
select pre-update snapshot
↓
boot snapshot
↓
verify system
↓
restore if appropriate
```

Module 20 will go much deeper into boot/UKI/Limine recovery.

18.61 — `omarchy reinstall`

Current official update manual documents:

```
omarchy reinstall
```

as a broad recovery path if configuration/system state has become severely corrupted.

The current docs warn that reinstalling:

- reinstalls default Omarchy packages;
- returns the system to stable;
- can downgrade packages that are too new;
- resets Omarchy configuration.

This is destructive to custom Omarchy config.

18.62 — Reinstall Is Not Routine Maintenance

Do not use:

```
omarchy reinstall
```

as:

```
"maybe this will fix it"
```

for ordinary issues.

It is a broad recovery action.

Your hierarchy should be:

```
diagnose
↓
repair component
↓
repair specific config
↓
snapshot rollback if system update broke system
↓
reinstall only when justified
```

18.63 — Personal Config Before Reinstall

Because reinstall can reset Omarchy configuration, make sure intentional personal config is preserved elsewhere.

Examples:

```
dotfiles Git repo
backup
manual copy
```

This is another reason Module 22 matters.

18.64 — Practical Exercise: Inspect Version and Update Commands

Run:

```
omarchy version
```

Then:

```
omarchy update --help
```

if the installed CLI exposes help for it.

Also inspect:

```
omarchy commands --all | grep -i update
```

Record the current update-related public command surface.

18.65 — Practical Exercise: Check Update Indicator

Inspect the top bar.

If the update icon is visible:

```
an update is waiting
```

If it is not:

```
that does not mean the shell is broken
```

The indicator is intentionally conditional.

Do not force an update just for the exercise.

18.66 — Practical Exercise: Inspect Snapshot Support

Run:

```
command -v omarchy-snapshot
```

Then inspect current help if available:

```
omarchy-snapshot --help
```

Confirm that your system uses Limine:

```
bootctl status
```

may provide UEFI/boot context, but the actual bootloader should be verified from your installed system rather than assumed.

The course workstation has already verified Limine separately.

18.67 — Practical Exercise: Create a Manual Snapshot

If your system has the current snapshot setup configured:

```
omarchy-snapshot create
```

Record the created snapshot.

Do this once so snapshot creation is familiar before you ever need recovery.

If the command reports snapshot support is unavailable/unconfigured, do not improvise — investigate current official setup guidance.

18.68 — Practical Exercise: Inspect Migration State

Run:

```
omarchy-migrate --pending
```

Interpret correctly:

```
output
→ pending migrations exist

no output + non-zero exit
→ none pending
```

Check:

```
echo $?
```

immediately afterward if you want to inspect the exit status.

18.69 — Practical Exercise: Inspect Update Log Location

If you have run an update recently and the log still exists:

```
ls -lh /tmp/omarchy-update.log
```

Then:

```
less /tmp/omarchy-update.log
```

Do not worry if `/tmp` was cleared after reboot.

It is temporary storage.

18.70 — Practical Exercise: Check Firmware Through the Menu

Open:

```
Super + Space
→ Update
→ Firmware
```

Observe whether updates are available.

Do not install firmware in the middle of the lesson if you cannot safely reboot.

18.71 — Practical Exercise: Inspect the Current Channel

Open:

```
Super + Space
→ Update
→ Channel
```

Identify the active channel.

For this course baseline, use:

```
stable
```

unless you deliberately chose another channel for testing.

Do not switch channels merely to see what happens.

18.72 — Practical Exercise: Build Your Maintenance Checklist

Create a personal checklist:

```
Regular:
- Omarchy update when available/convenient
- reboot when requested
- verify main dev workflow

Monthly/occasional:
- firmware check
- root disk-space awareness
- backup/dotfiles sanity check

Before risky system changes:
- commit config
- manual snapshot
- confirm backup
```

Keep it simple enough that you will actually use it.

18.73 — Common Update and Maintenance Mistakes

Mistake 1 — Using `sudo pacman -Syu` as the Normal Update

Use:

```
omarchy update
```

Mistake 2 — Using `omarchy --version`

Current public command:

```
omarchy version
```

Mistake 3 — Assuming Snapshot Restores `/home`

It does not.

Use backups/Git for personal data and config.

Mistake 4 — Updating Immediately Before Important Work

Leave time for reboot and verification.

Mistake 5 — Ignoring Low-Disk-Space Protection

Fix storage pressure rather than forcing through it.

Mistake 6 — Clearing the Entire Package Cache Constantly

Omarchy already maintains a bounded cache and preserves downgrade usefulness.

Mistake 7 — Treating Firmware Capability as Pending Firmware

Check whether an actual update exists.

Mistake 8 — Switching to Edge Because "Newer Is Better"

Stable is the correct default for most users.

Mistake 9 — Reinstalling for a Single Broken Component

Use component-level repair first.

Mistake 10 — Forgetting Personal Config Before Broad Recovery

Snapshots/reinstall do not substitute for a dotfiles/backup strategy.

18.74 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Updates

- [ ] I know `omarchy update` is the supported update path.
- [ ] I understand why direct full-system pacman updates are guarded.
- [ ] I understand the high-level update pipeline.
- [ ] I know how to check the current Omarchy version.
- [ ] I understand the update indicator.

Snapshots

- [ ] I understand automatic pre-update snapshots.
- [ ] I know the manual snapshot command.
- [ ] I understand snapshot recovery through Limine.
- [ ] I know snapshots restore root, not `/home`.
- [ ] I understand snapshots do not replace backups or dotfiles Git.

Migrations

- [ ] I understand what Omarchy migrations are.
- [ ] I know they are tracked per user.
- [ ] I know the normal update runs them automatically.
- [ ] I can inspect pending migration state when needed.

Maintenance

- [ ] I understand the root free-space check.
- [ ] I know Omarchy manages the package cache.
- [ ] I know mise-managed tools are included in the update flow.
- [ ] I understand restart/reboot handling.
- [ ] I know firmware updates are separate.

Channels

- [ ] I understand stable, RC, edge, and dev.
- [ ] I know stable is the normal recommendation.
- [ ] I understand why channel switching changes risk.

Recovery

- [ ] I know where the update log is written.

- [ ] I use targeted repair before broad recovery.
- [ ] I understand when a snapshot is appropriate.
- [ ] I understand that `omarchy reinstall` is destructive to custom config.

18.75 — What You Can Now Do

After completing Module 18, you can now:

- update Omarchy through the supported pipeline;
- understand what happens during an update;
- avoid bypassing important Omarchy migrations/snapshots;
- verify the installed version;
- recover from a bad system update with snapshots;
- understand the limits of snapshot recovery;
- inspect pending migrations;
- maintain sensible disk/package-cache hygiene;
- update firmware deliberately;
- choose the appropriate release channel;
- diagnose failed updates from real logs;
- apply the smallest appropriate recovery action;
- maintain the workstation without turning maintenance into a hobby.

Most importantly:

> *You now understand that a reliable rolling-release workstation is not one that never changes — it is one that updates through a controlled path and has a tested recovery path when change goes wrong.*

Module 18 Summary

Supported update:

```
omarchy update
```

Version:

```
omarchy version
```

Do not use direct full-system pacman upgrades as the routine Omarchy update path.

Current update flow includes:

```
free-space check
↓
package-cache prune
↓
snapshot
↓
package updates
↓
migrations
↓
mise updates
↓
post-update hooks
↓
log analysis
↓
restart/reboot checks
```

Automatic snapshots:

```
before normal Omarchy updates
```

Manual snapshot:

```
omarchy-snapshot create
```

Restore path:

```
Limine
→ boot snapshot
→ verify
→ restore
```

Important limitation:

```
root filesystem
→ snapshot recovery

/home
→ not restored
```

Pending migrations:

```
omarchy-migrate --pending
```

Update log:

```
/tmp/omarchy-update.log
```

Channels:

```
stable
RC
edge
dev
```

Firmware:

```
Update
→ Firmware
```

Maintenance principle:

```
update through Omarchy
+
keep recovery ready
+
verify the workflow you actually use
```

And the most important rule:

> ***Never bypass a maintenance wrapper until you understand what the wrapper is doing for you.***

MODULE 19

19

Restart, Repair, and Recovery on Omarchy

A stable workstation is not one that never breaks.

It is one where you know:

About This Module

A stable workstation is not one that never breaks.

It is one where you know:

```
what broke
→ what layer it belongs to
→ what the smallest repair is
→ how far to escalate
```

This module teaches that escalation ladder.

Current Omarchy 4 / Quattro deliberately provides component-level restart helpers and targeted recovery tools so that a single problem does **not** automatically become:

```
reboot
```

or worse:

```
reinstall the whole system
```

Examples:

```
audio disappears
→ restart audio

Bluetooth headset stops reconnecting
→ restart Bluetooth

trackpad dies after suspend
→ restart trackpad

Omarchy Shell stops responding
→ restart shell

XCompose changes do not appear
→ restart XCompose

one config is broken
→ repair/refresh that config

an update breaks the system
→ boot a snapshot

everything is severely corrupted
→ broad reinstall as a last resort
```

The goal of this module is:

> ***Use the smallest recovery action that matches the broken layer.***

What You Will Learn

By the end of Module 19, you will understand:

- the difference between restart, refresh, reinstall, snapshot rollback, and reboot;
- how to discover the current Omarchy restart helpers;
- how to restart the Omarchy Shell;
- how to restart hardware subsystems;
- how to restart Tmux and terminal-related components carefully;
- how to refresh specific user configuration;
- how Omarchy backs up a config before replacing it;
- why `omarchy refresh hyprland` is broad and destructive to personal Hyprland config;
- why `omarchy reinstall configs` is a last-resort configuration reset;
- how `omarchy-debug` fits into escalation;

- how to recover from password lockout from a TTY;
- when snapshots are appropriate;
- when a reboot is appropriate;
- when a reinstall is appropriate;
- how to preserve evidence before changing the system;
- how to build a disciplined recovery ladder.

19.1 — The Recovery Hierarchy

Use this mental model:

```
1. inspect
2. retry/reselect
3. restart the affected component
4. repair the specific config
5. refresh the specific config
6. restart the session / reboot
7. snapshot rollback
8. broad config/package reinstall
```

The order matters.

Do not jump from:

```
small symptom
```

to:

```
large destructive action
```

without evidence.

19.2 — Restart Is Not Reinstall

These words mean different things.

Restart

```
stop/start a process or subsystem
```

State/config remains.

Refresh

```
replace one or more user config files with current shipped defaults
```

User customizations may be overwritten, usually with backup behavior depending on the command.

Reinstall

```
restore broad default package/config state
```

Potentially destructive to customization.

Snapshot Rollback

```
restore root filesystem to a previous system state
```

Does not restore `/home`.

19.3 — First Principle: Preserve Evidence

Before repairing something, collect enough evidence to know what happened.

Examples:

```
git diff
```

```
hyprctl configerrors
```

```
systemctl --failed
```

```
systemctl --user --failed
```

```
journalctl --user -b
```

```
journalctl -b
```

You do not need to run all of these every time.

Use the ones appropriate to the symptom.

19.4 — Why Evidence Comes Before Restart

A restart may make the symptom disappear.

That is good operationally.

But if the issue keeps returning, you still need to know:

```
what failed?
```

For recurring problems:

```
capture logs/state first
→ restart second
```

For a one-off harmless glitch:

```
restart immediately
```

may be perfectly reasonable.

19.5 — Discover Current Restart Helpers

Current Omarchy uses restart helpers extensively.

Start with:

```
omarchy restart --help
```

and:

```
omarchy commands --all | grep -i restart
```

Current Omarchy generations include restart helpers for categories such as:

```
app
audio
bluetooth
btop
helix
herdr
hyprctl
hyprsunset
opencode
shell
terminal
tmux
trackpad
wifi
xcompose
```

The exact list changes.

Treat your installed CLI as the current source of truth.

19.6 — Why Restart Helpers Are Valuable

Without helpers, you would need to know which low-level process/service controls each component.

Example:

```
audio
→ PipeWire?
→ WirePlumber?
→ user services?
→ stream restoration?
```

Omarchy can wrap the intended recovery flow.

That keeps the public workflow stable even if implementation details change.

19.7 — Hardware Restart Menu

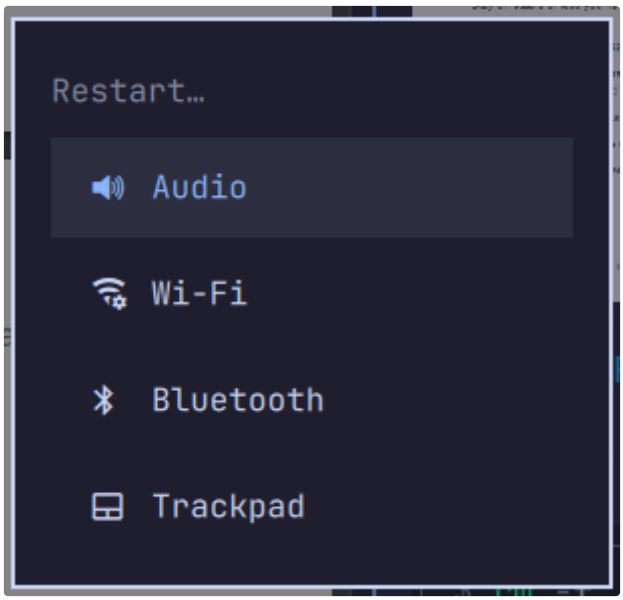
Current official troubleshooting documentation says common hardware recovery is available under:

```
Super + Space
→ Update
→ Hardware
```

Current documented hardware reloads include:

```
Wi-Fi
Bluetooth
Audio
Trackpad
```

This is the beginner-friendly recovery path.



Screenshot 19.7-A
Omarchy Hardware Restart menu (Wi-Fi, Bluetooth, Audio, Trackpad)

19.8 — Audio Restart

If audio disappears:

```
1. check output selection
2. check mute/volume
3. restart audio
4. retry playback
```

Current official troubleshooting guidance explicitly recommends restarting audio before rebooting.

Use the current menu or CLI.

Discover:

```
omarchy restart --help
```

Then use the current audio restart command.

19.9 — Bluetooth Restart

Typical symptom:

```
headset worked five minutes ago
now refuses to reconnect
```

Recovery:

```
verify Bluetooth power
↓
disconnect/reconnect
↓
restart Bluetooth
↓
retry
```

Forget/re-pair only if needed.

19.10 — Wi-Fi Restart

If Wi-Fi suddenly stops working:

```
check current network
↓
check signal/state
↓
restart Wi-Fi/network subsystem
↓
reconnect
```

Do not immediately start editing NetworkManager profiles or kernel modules.

19.11 — Trackpad Restart

Current official troubleshooting docs explicitly mention:

```
trackpad died after suspend
```

as a case where a subsystem restart often fixes the issue.

This is particularly useful on laptops.

19.12 — Omarchy Shell Restart

Current Omarchy Shell architecture states that the desktop shell runs as one long-lived Quickshell process.

It hosts:

- bar;
- panels;
- overlays;
- menus;
- services;
- plugins.

Current official shell architecture explicitly says to restart it with:

```
omarchy-restart-shell
```

The routed CLI may also expose:

```
omarchy restart shell
```

depending on current version.

Verify with:

```
omarchy restart --help
```

19.13 — When to Restart the Shell

Useful symptoms:

```
bar disappeared
panel will not open
plugin state is stale
shell UI is not responding
```

A shell restart is much smaller than:

```
restart Hyprland session
```

or:

```
reboot
```

19.14 — Shell IPC Check

Current Omarchy Shell exposes IPC.

A basic current check is:

```
omarchy-shell shell ping
```

If the shell is healthy, IPC should respond.

If it reports:

```
omarchy-shell is not running
```

or:

```
omarchy-shell is not responding
```

a shell restart is appropriate.

19.15 — Shell Plugin Problems

If the shell became unstable immediately after enabling a third-party plugin:

```
1. disable/remove the plugin if possible
2. restart shell
3. verify
```

Do not first reinstall the entire shell/system.

Plugin problems should be repaired at the plugin layer.

19.16 — XCompose Restart

Current official dotfiles docs say changes to:

```
~/.XCompose
```

require:

```
omarchy-restart-xcompose
```

to be picked up.

This is a good example of a targeted restart.

You changed one input-related user configuration.

Restart the one affected component.

19.17 — Tmux Restart

Tmux is persistent by design.

That means configuration/process state can remain alive longer than your terminal window.

If you changed Tmux configuration and the old session still behaves as before:

```
existing server/session may still have old state
```

Use current Omarchy Tmux restart helpers only when you intentionally want to restart that persistent environment.

WARNING

Restarting Tmux may terminate sessions/processes depending on current helper behavior.

Inspect:

```
omarchy restart --help
```

and:

```
type omarchy-restart-tmux
```

before using it during important work.

19.18 — Real Course Example: Stale Tmux Working Directory

During the real workstation setup, a persistent Tmux session kept the previous pane working directory.

The new terminal had:

```
~/Projects/projectinator
```

but the existing Tmux session retained older pane state.

The solution was not:

```
reboot Linux
```

It was to restart/recreate the Tmux session.

Persistent tools preserve state by design.

19.19 — Terminal Restart

A terminal restart helper can be useful when:

- terminal configuration changed;
- launch integration is stale;
- a terminal process/session needs a clean start.

But remember:

```
terminal
≠
shell
≠
Tmux
```

These are different layers.

A Ghostty problem should not automatically lead to restarting Tmux or the Omarchy Shell.

19.20 — Application Restart

Current Omarchy includes app-level restart helpers.

This is useful when one app is stuck while the rest of the system is healthy.

Use application-level recovery before:

```
session restart
```

or:

```
reboot
```

19.21 — Configuration Repair: Edit First

If you know exactly what broke:

```
I added one invalid binding
```

repair the file directly.

Example:

```
hyprctl configerrors
```

Then:

```
nvim ~/.config/hypr/bindings.lua
```

Undo/fix.

Save.

Verify:

```
hyprctl configerrors
```

This is better than refreshing all Hyprland config.

19.22 — Git Makes Config Repair Easier

If your dotfiles are version-controlled:

```
git diff
```

shows what changed.

Then:

```
git restore PATH
```

can restore one file.

Or revert one commit.

This is why Module 22's dotfiles workflow is a major recovery tool.

19.23 — `omarchy refresh config`

Current Omarchy includes a targeted config refresh mechanism.

The important current behavior is:

```
refresh specific user config
→ back up existing user version
→ copy current shipped config into place
```

This is useful when:

- the file is badly broken;
- current Omarchy changed the expected structure;
- you want a clean current baseline.

19.24 — When to Use Targeted Config Refresh

Good case:

```
one config file is corrupted
```

Bad case:

```
Bluetooth did not connect once
```

Use config refresh only when the evidence points to configuration.

19.25 — Compare Before Refreshing

Before replacing a config:

```
git diff
```

or:

```
cp PATH PATH.manual-backup
```

if needed.

Know which personal changes you are about to lose.

Even if Omarchy creates a backup, understanding the diff matters.

19.26 — `omarchy refresh hyprland`

This is a broad Hyprland recovery action.

Current Quattro source refreshes the user's Hyprland Lua configs including:

```
hyprland.lua
autostart.lua
bindings.lua
input.lua
looknfeel.lua
monitors.lua
```

This means it can wipe your personal Hyprland customizations from the active files.

Use it only when you intentionally want to reset that whole layer.

19.27 — Never Use `refresh hyprland` for One Shortcut

If:

```
one binding is wrong
```

repair:

```
bindings.lua
```

Do not reset:

- monitors;
- input;
- autostart;
- look-and-feel;

at the same time.

This is the central principle of scoped recovery.

19.28 — `omarchy reinstall configs`

Current Omarchy provides a broader configuration reinstall/reset.

This is destructive to personal Omarchy config.

Use when:

```
the user config layer is deeply corrupted
```

not:

```
one app is behaving strangely
```

19.29 — `omarchy reinstall`

Current official troubleshooting/update documentation describes broad reinstall as the later fallback when:

- rollback did not solve the issue;
- defaults/packages need to be restored;
- the system/config is severely damaged.

This is a high-impact recovery action.

Back up intentional config first.

19.30 — Broad Recovery Changes More Than the Symptom

A reinstall/reset may:

- reset Omarchy config;
- reinstall package state;
- return you to stable channel;
- downgrade packages that are too new;
- remove personal Omarchy configuration choices.

That may be exactly what you want.

But it should be deliberate.

19.31 — Snapshot Rollback

If the machine broke because of a system update:

```
snapshot rollback
```

may be more appropriate than reinstall.

Current official troubleshooting guidance says the first response to:

```
"I broke my system with an update"
```

is to try rolling back to the version before the recent update.

This is the right recovery layer.

19.32 — Snapshot Does Not Repair `/home`

Remember:

```
root filesystem
→ restored

/home
→ not restored
```

So if your personal Hyprland config caused the problem:

```
snapshot rollback may not change it
```

because:

```
~/.config
```

remains in `/home`.

This is crucial for diagnosis.

19.33 — System Problem vs User Config Problem

Ask:

```
Did the system break immediately after package/update change?
```

Possible snapshot candidate.

Or:

```
Did I just edit ~/.config/hypr/bindings.lua?
```

Personal config problem.

Snapshot is the wrong first tool.

19.34 — `omarchy-debug`

Current official troubleshooting documentation recommends:

```
omarchy-debug
```

when rollback does not solve a system problem and you need to share diagnostic information.

Use it to collect the current state rather than manually pasting random fragments.

Then provide that evidence when seeking help.

19.35 — Why Debug Bundles Matter

A useful diagnostic report can include current information such as:

- version;
- hardware;
- services;
- logs;
- config/system state.

That is much more useful than:

```
"It broke."
```

The purpose is evidence-based support.

19.36 — Do Not Publish Secrets in Debug Output Blindly

Before posting diagnostics publicly:

- inspect output;
- remove credentials;
- remove private hostnames/IPs if sensitive;
- remove private repository information if included.

Diagnostic convenience does not override privacy/security.

19.37 — TTY Recovery

If the graphical session or lock/login layer is unusable, Linux still provides virtual terminals.

Current official troubleshooting documentation specifically uses:

```
Ctrl + Alt + F2
```

as a recovery path.

This can give you a text login independent of the graphical session.

19.38 — Why TTY Is Powerful

From a TTY you can:

- log in;
- inspect logs;
- edit config;
- restart services;
- reset lockout state;
- reboot cleanly.

This means:

```
graphical desktop broken
```

does not automatically mean:

```
system inaccessible
```

19.39 — Password Lockout Recovery

Current official troubleshooting documentation says repeated failed password attempts can trigger:

```
faillock
```

lockout.

If locked out on the graphical screen:

```
Ctrl + Alt + F2
```

Log in as root where appropriate, then:

```
faillock --reset --user YOUR_USERNAME
```

Use your actual username.

This resets the authentication lockout.

19.40 — Do Not Confuse Lockout With Wrong Password

Before resetting faillock:

- verify keyboard layout;
- verify Caps Lock state;
- consider whether the password is actually wrong.

A reset does not fix a forgotten password.

19.41 — Systemd Failed Services

When a component is failing repeatedly:

```
systemctl --failed
```

shows failed system services.

For user services:

```
systemctl --user --failed
```

This is a useful health signal.

But:

> *A failed service entry does not automatically mean your current symptom comes from that service.*

Interpret it in context.

19.42 — Logs Need Interpretation

Use:

```
journalctl -b
```

or a relevant unit:

```
journalctl -u SERVICE -b
```

For user units:

```
journalctl --user -u SERVICE -b
```

Look for:

- timestamp matching the failure;
- repeated errors;
- exit status;
- dependency failure.

Do not panic because a log contains the word:

```
error
```

from an unrelated component.

19.43 — Coredumps

A crashed process may leave:

```
systemd-coredump
```

data.

Current Omarchy can surface process-crash notifications and hand diagnostics to the default AI agent.

This is useful when the problem is:

```
process crash
```

rather than:

```
configuration mistake
```

19.44 — AI-Assisted Repair

A good AI troubleshooting prompt:

```
Do not modify anything.

Inspect:
- current Omarchy version;
- failed system/user services;
- relevant logs from this boot;
- the config related to the symptom.

Explain the likely root cause and propose the smallest next action.
```

This follows the same evidence-first model as the rest of the course.

19.45 — AI Should Not Immediately "Fix Everything"

Avoid:

```
"My desktop is broken. Fix it."
```

especially in unattended mode.

The agent may change several layers at once.

Better:

```
inspect
→ explain
→ propose
→ approve
→ apply
```

19.46 — When a Reboot Is Appropriate

A reboot is reasonable when:

- kernel was updated;
- firmware requires it;
- session-wide state is corrupted;
- GPU/display driver state needs a full reset;
- targeted component restart failed;
- current updater explicitly requests reboot.

Reboot is not forbidden.

It is just not the first universal tool.

19.47 — When a Session Restart Is Enough

If:

- Hyprland/session state is broken;
- graphical environment is inconsistent;
- all desktop apps can be closed safely;

logging out/restarting the graphical session may be enough.

This is smaller than rebooting the OS.

19.48 — When Snapshot Rollback Is Appropriate

Use snapshot rollback when evidence points to:

```
system update caused the failure
```

Examples:

- system no longer boots after update;

- core package update caused major regression;
- current root filesystem state is unusable.

Do not use snapshot rollback for:

- deleted personal file;
- broken `~/.config`;
- bad Git change;
- one stuck Bluetooth device.

19.49 — When Reinstall Is Appropriate

Consider broad reinstall when:

- system/config state is severely damaged;
- targeted repair failed;
- snapshot rollback is unavailable or inappropriate;
- you intentionally want to return to defaults.

Before reinstalling:

```
preserve personal config/data
```

because broad recovery is destructive.

19.50 — The Recovery Decision Tree

Use:

```
What broke?
        ↓

One app?
→ restart app

Shell UI?
→ restart shell

Audio/Bluetooth/Wi-Fi/trackpad?
→ restart subsystem

One user config?
→ repair/refresh file

Whole Hyprland user config?
→ refresh Hyprland if intentional

Graphical session?
→ restart session

System after update?
→ snapshot rollback

Severely corrupted system/config?
→ reinstall
```

This is the most important diagram in this module.

19.51 — Practical Exercise: Discover Restart Commands

Run:

```
omarchy restart --help
```

Then:

```
omarchy commands --all | grep -i restart
```

Record the current helpers on your machine.

Do not rely on the course list if your current version differs.

19.52 — Practical Exercise: Shell Health

Run:

```
omarchy-shell shell ping
```

If healthy, record the response.

Then inspect:

```
command -v omarchy-restart-shell
```

Do not restart the shell during important recording/work unless you are prepared for the short UI interruption.

19.53 — Practical Exercise: Restart the Shell

When safe:

```
omarchy-restart-shell
```

Observe:

```
bar/panels briefly disappear
→ shell returns
```

Verify:

```
omarchy-shell shell ping
```

This makes shell recovery familiar before you need it urgently.

19.54 — Practical Exercise: XCompose Restart

Inspect:

```
command -v omarchy-restart-xcompose
```

If you intentionally modify:

```
~/.XCompose
```

run:

```
omarchy-restart-xcompose
```

Otherwise simply identify the helper.

19.55 — Practical Exercise: Failed Services

Run:

```
systemctl --failed
```

Then:

```
systemctl --user --failed
```

If entries exist:

```
do not repair them automatically
```

Ask:

```
Is this service related to an actual symptom?
```

This teaches interpretation rather than "zero errors at any cost."

19.56 — Practical Exercise: Config Error Recovery

Create a harmless temporary syntax error only if you are comfortable recovering it.

Safer alternative:

inspect your last real configuration change.

Run:

```
hyprctl configerrors
```

Then use:

```
git diff
```

inside your dotfiles repo if applicable.

The goal is to connect:

```
config error
→ specific file
→ specific repair
```

not to intentionally destabilize the desktop.

19.57 — Practical Exercise: Inspect Refresh Commands

Run:

```
omarchy commands --all | grep -i refresh
```

Identify:

- targeted config refresh;
- Hyprland refresh;
- other current refresh helpers.

Do not execute broad refresh commands for practice.

Discovery is enough.

19.58 — Practical Exercise: Review Snapshot Recovery Path

Do not restore anything.

Instead verify:

```
current bootloader
snapshot command availability
snapshot documentation
```

You already created/inspected snapshots in Module 18.

Mentally rehearse:

```
bad update
→ Limine
→ snapshot
→ verify
→ restore
```

19.59 — Practical Exercise: TTY Recovery

Do this only if comfortable.

Press:

```
Ctrl + Alt + F2
```

Observe the TTY login.

Return to the graphical session using the appropriate active VT key combination for your current system.

Exact VT placement can vary, so do not assume a hard-coded return key without checking.

This exercise is optional.

19.60 — Build Your Recovery Cheat Sheet

Create a short local note:

```
Shell:
omarchy-restart-shell

Hardware:
Super + Space → Update → Hardware

Config:
hyprctl configerrors
git diff
targeted refresh

System update failure:
Limine snapshot

Diagnostics:
omarchy-debug
systemctl --failed
systemctl --user --failed

Lockout:
Ctrl+Alt+F2
faillock --reset --user USER
```

Later this can become a course cheat sheet.

19.61 — Common Recovery Mistakes

Mistake 1 — Rebooting for Every Small Problem

Restart the affected component first.

Mistake 2 — Reinstalling for One Broken Config

Repair the file.

Mistake 3 — Refreshing All Hyprland Config for One Shortcut

Use the smallest scope.

Mistake 4 — Rolling Back a Snapshot for a `/home` Config Problem

Snapshots do not restore `/home`.

Mistake 5 — Restarting Tmux Without Realizing Processes May Be Lost

Inspect current sessions and helper behavior first.

Mistake 6 — Treating Any Failed Service as an Emergency

Correlate it with the symptom.

Mistake 7 — Posting `omarchy-debug` Output Publicly Without Reviewing It

Check for private information.

Mistake 8 — Letting an AI Agent Apply Broad Repair Immediately

Inspect and propose first.

Mistake 9 — Using Broad Reinstall Because It Is Easier Than Diagnosis

You lose useful state and learn nothing.

Mistake 10 — Trying to Repair Without Preserving Evidence

Capture logs/diffs/state before destructive actions.

19.62 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Restart

- [ ] I understand restart vs reboot.
- [ ] I can discover current Omarchy restart helpers.
- [ ] I know how to restart the shell.
- [ ] I know hardware subsystems can be restarted individually.
- [ ] I understand Tmux restart can affect persistent processes.

Config Repair

- [ ] I repair a specific config before refreshing broadly.
- [ ] I understand targeted config refresh.
- [ ] I understand `refresh hyprland` is broad.
- [ ] I understand config reinstall is destructive.

Diagnostics

- [ ] I can inspect failed system services.
- [ ] I can inspect failed user services.

- [ ] I understand logs require interpretation.
- [ ] I know what `omarchy-debug` is for.

Recovery

- [ ] I understand TTY recovery.
- [ ] I understand `faillock` reset at a high level.
- [ ] I understand when snapshot rollback fits.
- [ ] I know snapshots do not restore `/home`.
- [ ] I understand when reboot/reinstall becomes appropriate.

Strategy

- [ ] I use the smallest recovery scope.
- [ ] I preserve evidence before destructive repair.
- [ ] I can explain the recovery ladder from component restart to reinstall.

19.63 — What You Can Now Do

After completing Module 19, you can now:

- recover common hardware glitches without rebooting;
- restart the Omarchy Shell independently;
- understand persistent Tmux recovery;
- repair configuration mistakes surgically;
- use refresh commands appropriately;
- avoid destructive configuration resets;
- inspect failed services and logs intelligently;
- use TTY as a graphical-session escape hatch;
- recover authentication lockouts;
- use snapshots when system updates break the root filesystem;
- decide when reboot or reinstall is genuinely justified;
- use AI as a diagnostic assistant without giving it uncontrolled repair authority.

Most importantly:

> *You now have a recovery strategy based on layers and evidence, not superstition.*

Module 19 Summary

Recovery ladder:

```
inspect
↓
retry/reselect
↓
restart component
↓
repair specific config
↓
refresh specific config
↓
restart session/reboot
↓
snapshot rollback
↓
broad reinstall
```

Current shell recovery:

```
omarchy-shell shell ping
omarchy-restart-shell
```

Common hardware recovery:

```
Super + Space
→ Update
→ Hardware
→ Wi-Fi / Bluetooth / Audio / Trackpad
```

XCompose:

```
omarchy-restart-xcompose
```

Config diagnostics:

```
hyprctl configerrors
git diff
```

System diagnostics:

```
systemctl --failed
systemctl --user --failed
```

Debug bundle:

```
omarchy-debug
```

TTY recovery:

```
Ctrl + Alt + F2
```

Authentication lockout reset:

```
faillock --reset --user YOUR_USERNAME
```

System-update rollback:

```
Limine
→ pre-update snapshot
→ verify
→ restore
```

Critical snapshot limitation:

```
root
→ restored

/home
→ unchanged
```

And the most important rule:

> ***Repair the smallest broken layer first.***

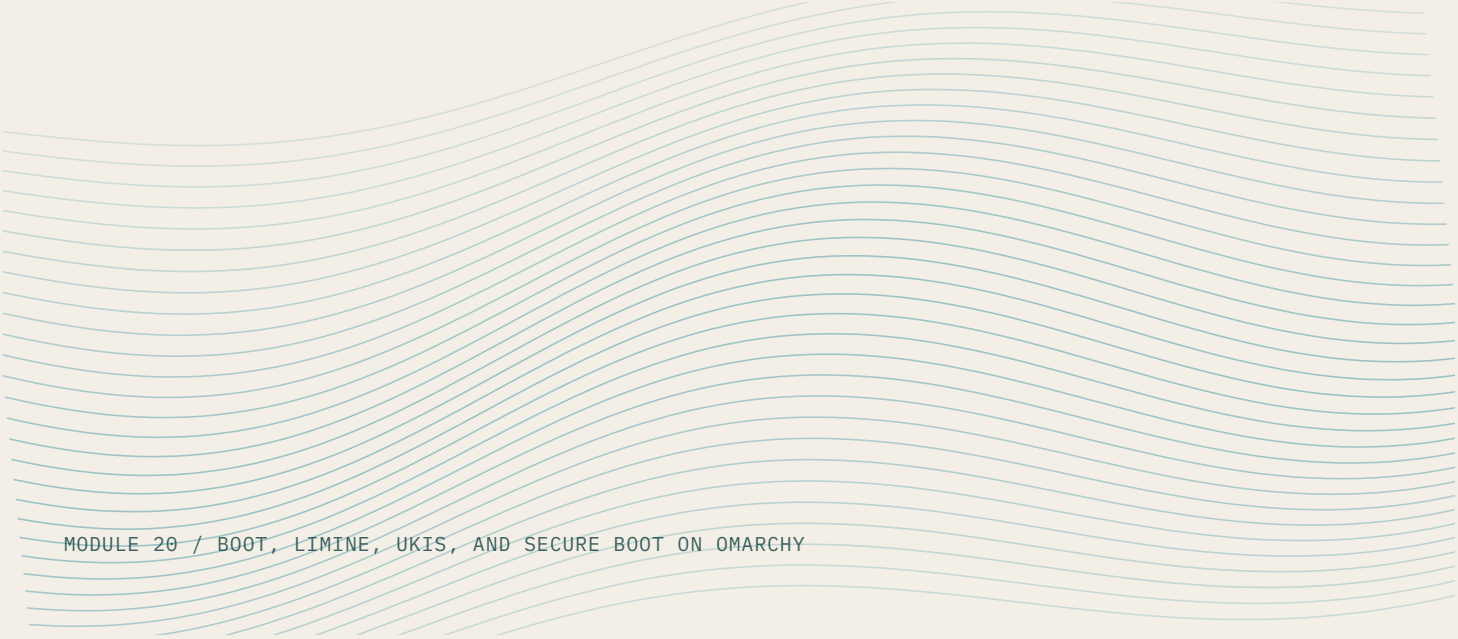
MODULE 20

20

Boot, Limine, UKIs, and Secure Boot on Omarchy

You now know how to update, restart, repair, and recover Omarchy.

The next step is understanding the layer beneath the desktop:



About This Module

You now know how to update, restart, repair, and recover Omarchy.

The next step is understanding the layer beneath the desktop:

```
firmware
→ bootloader
→ UKI
→ kernel
→ userspace
→ Omarchy session
```

This is one of the most important advanced modules in the course because boot problems can make an otherwise healthy system feel completely inaccessible.

The workstation used while building this course has already taught us one critical lesson:

> ***Never assume the bootloader. Verify it.***

During setup, GRUB was initially assumed.

The real machine was using:

```
Limine
```

with a Unified Kernel Image:

```
/boot/EFI/Linux/omarchy_linux.efi
```

and the active boot entry:

```
Omarchy.linux
```

The machine also had:

```
UEFI
TPM 2.0
Secure Boot disabled
```

This module explains what those pieces mean and how they fit together.

The goal is not to turn you into a firmware engineer.

The goal is:

> ***Understand the Omarchy boot chain well enough to verify it, troubleshoot it, recover snapshots, and make informed Secure Boot decisions.***

What You Will Learn

By the end of Module 20, you will understand:

- the Linux boot chain at a practical level;
- UEFI vs legacy BIOS;
- what a bootloader does;
- why Omarchy currently uses Limine in the supported setup;
- what a Unified Kernel Image is;
- how systemd-stub fits into a UKI;
- how to inspect UEFI boot state;
- how to verify the actual bootloader;
- how to identify the active UKI;
- how bootable snapshots relate to Limine;
- how Secure Boot fits into the chain;
- why Windows dual boot can complicate Secure Boot decisions;
- what TPM 2.0 does and does not mean;
- why "Secure Boot capable" is not the same as "Secure Boot enabled";
- why boot recovery should be prepared before boot experimentation;
- how to distinguish firmware, bootloader, UKI, kernel, and userspace failures.

20.1 — The Boot Chain

A simplified modern Omarchy boot sequence looks like:

```
Power on
↓
UEFI firmware
↓
boot entry
↓
Limine
↓
Unified Kernel Image
↓
Linux kernel + initramfs
↓
root filesystem
↓
systemd
↓
graphical session
↓
Hyprland + Omarchy Shell
```

Each layer has a different responsibility.

This matters because:

```
failure before kernel
≠
failure after kernel
```

The recovery approach depends on which layer failed.

20.2 — UEFI vs Legacy BIOS

Modern systems normally boot using:

```
UEFI
```

rather than legacy BIOS mode.

UEFI provides:

- standardized boot entries;
- EFI System Partition support;
- Secure Boot capability;
- firmware boot menus;

- direct loading of EFI binaries.

Current Omarchy installations should be treated as UEFI systems unless verified otherwise.

20.3 — Verify UEFI Mode

A useful command is:

```
bootctl status
```

Even if systemd-boot is not your bootloader, `bootctl status` can still expose useful firmware/EFI information.

You can also inspect:

```
ls /sys/firmware/efi
```

If this directory exists:

```
system is booted in UEFI mode
```

20.4 — Real Course Workstation Firmware

The workstation used in this course reported:

```
UEFI 2.70
```

This confirmed a modern UEFI boot environment.

The exact firmware version on your machine will differ.

Do not copy version numbers.

20.5 — What the Bootloader Does

The bootloader sits between firmware and the kernel.

Its job is to make boot choices and load the selected boot target.

Examples of bootloaders include:

```
Limine
GRUB
systemd-boot
```

Different Linux setups choose different ones.

Omarchy currently uses Limine in its supported modern setup.

20.6 — Verify the Bootloader Instead of Assuming

The workstation used in this course initially looked like it might be using GRUB.

After inspection, it was actually using:

```
Limine 12.8.0
```

This is a key course lesson.

Use evidence.

Do not infer the bootloader from:

- an old tutorial;
- what another Arch install used;
- what Windows dual boot normally looks like;
- what you vaguely remember selecting.

20.7 — Limine

Limine is the current bootloader used by the course workstation and the current Omarchy snapshot workflow.

Its responsibilities include:

- presenting boot entries;
- loading the Omarchy UKI;
- exposing bootable snapshots;
- providing a recovery path when a recent system update breaks root state.

This makes Limine important beyond normal boot.

It is part of recovery.

20.8 — Current Course Workstation Limine Version

The workstation used while developing this course had:

```
Limine 12.8.0
```

Treat this as a historical verification point, not a version requirement.

Your current Omarchy installation may use a newer version.

20.9 — EFI System Partition

UEFI boots EFI programs stored on an EFI System Partition.

You may see paths such as:

```
/boot/EFI/
```

A modern Linux boot installation often places EFI executables here.

On the course workstation:

```
/boot/EFI/Linux/omarchy_linux.efi
```

was the active Omarchy Unified Kernel Image.

20.10 — What Is a Unified Kernel Image?

A Unified Kernel Image, or:

```
UKI
```

bundles boot-critical components into one EFI executable.

Conceptually:

```
Linux kernel
+
initramfs
+
kernel command line
+
metadata
+
EFI stub
        ↓
one .efi file
```

Instead of firmware/bootloader loading several separate boot pieces, it can load one unified EFI image.

20.11 — Why UKIs Are Useful

A UKI can improve:

- boot-chain simplicity;
- signing;
- Secure Boot integration;
- reproducibility;
- measured boot workflows;
- bootloader interoperability.

For Omarchy learners, the main practical point is:

> *The thing Limine boots may be one `.efi` image containing the kernel and early userspace pieces.*

20.12 — systemd-stub

A UKI normally includes an EFI stub.

On the course workstation, inspection identified:

```
systemd-stub 261.2-1-arch
```

The stub is what allows the combined image to behave as an EFI executable.

It participates in handing control from firmware/bootloader into the Linux kernel.

20.13 — Active Omarchy Boot Entry

The workstation used in this course had an active entry:

```
Omarchy.linux
```

and the UKI:

```
/boot/EFI/Linux/omarchy_linux.efi
```

This gives us a concrete verified chain:

```
UEFI
↓
Limine
↓
Omarchy.linux
↓
omarchy_linux.efi
↓
kernel
```

20.14 — Inspect EFI Boot Entries

Use:

```
efibootmgr
```

if installed.

This can show:

- boot order;
- firmware boot entries;
- Windows Boot Manager;
- Limine/Omarchy-related entries.

Example concepts:

```
Boot0000* Windows Boot Manager
Boot0001* Limine
```

Exact numbering differs by machine.

20.15 — Boot Order

UEFI firmware stores a boot order.

Conceptually:

```
1. Limine
2. Windows Boot Manager
3. USB device
4. network boot
```

Your actual boot order is firmware-specific.

Do not change it unless you understand the consequence.

20.16 — Firmware Boot Menu

Most systems provide a one-time boot menu key.

Examples vary by vendor:

```
F8
F11
F12
Esc
```

This can be extremely useful.

It lets you choose:

- Limine;
- Windows Boot Manager;
- USB recovery media;

without permanently changing boot order.

20.17 — Dual Boot Architecture

In a Windows + Omarchy system, the firmware may contain separate boot entries:

```
Windows Boot Manager
Limine / Omarchy
```

The machine does not need GRUB merely because it dual boots.

This is exactly why bootloader assumptions are dangerous.

20.18 — Windows and Secure Boot

Modern Windows installations often expect Secure Boot availability, especially for some security features.

But:

```
Secure Boot capable
```

and:

```
Secure Boot currently enabled
```

are separate states.

The course workstation had:

```
Secure Boot disabled
```

during Omarchy installation/use.

20.19 — TPM 2.0

The course workstation had:

```
TPM 2.0 available
```

TPM is a hardware-backed security component used for things such as:

- key protection;
- measured boot;
- device security;
- Windows BitLocker-related workflows.

TPM presence does **not** automatically mean:

```
Secure Boot enabled
```

These technologies can work together, but they are not the same thing.

20.20 — Secure Boot

Secure Boot is a UEFI feature that verifies trusted signatures in the boot chain.

Conceptually:

```
firmware trusts key
↓
EFI executable signature verified
↓
boot continues
```

If an untrusted/unsigned boot image is encountered:

```
boot can be blocked
```

depending on configuration.

20.21 — Secure Boot Is Not "Windows Mode"

Secure Boot is a UEFI security technology.

It is not inherently Windows-only.

Linux can use Secure Boot too when the boot chain is configured and signed appropriately.

The challenge is not:

```
Linux cannot use Secure Boot
```

The challenge is:

```
the specific Linux boot chain must be prepared for it
```

20.22 — Do Not Enable Secure Boot Blindly

If Omarchy currently boots with Secure Boot disabled:

```
do not simply enable it in firmware
```

unless the current boot chain is prepared.

Possible outcome:

```
firmware rejects the Linux boot image
```

Then Omarchy will stop booting even though the installation itself is healthy.

20.23 — Prepare Recovery First

Before any Secure Boot experiment:

```
1. know how to enter firmware
2. know how to disable Secure Boot again
3. know the Limine entry
4. know the Windows Boot Manager entry
5. have recovery media available
6. confirm important data is backed up
```

Only then experiment.

20.24 — Windows BitLocker Consideration

If Windows uses BitLocker or device encryption, firmware/security changes can trigger recovery-key prompts.

Before changing:

- Secure Boot;
- TPM settings;
- boot order;
- firmware security settings;

make sure the Windows recovery key is available.

This is especially important on dual-boot machines.

20.25 — Fast Startup

Windows Fast Startup can complicate dual-boot disk access because Windows may leave filesystems in a hybrid hibernation state.

For shared filesystems between Windows and Linux, disabling Fast Startup is commonly recommended.

This is a dual-boot hygiene issue, not specifically a Limine issue.

20.26 — Secure Boot and Signed UKIs

A UKI architecture is well suited to Secure Boot because:

```
one EFI image
```

can be signed.

But successful Secure Boot still depends on:

- trusted keys;
- current Omarchy signing workflow;
- firmware trust configuration;
- bootloader/UKI integration.

Do not invent a signing procedure.

Use the current official Omarchy guidance for the current release.

20.27 — Why This Course Is Version-Aware Here

Boot/security tooling changes.

The exact supported Secure Boot workflow may evolve with:

- Omarchy releases;
- Limine changes;
- systemd-stub changes;
- kernel tooling;
- signing infrastructure.

Therefore this handbook teaches:

```
architecture
+
verification
+
recovery preparation
```

and tells learners to follow the current official Secure Boot procedure at execution time.

20.28 — Bootable Snapshots and Limine

Current Omarchy snapshot recovery integrates with Limine.

After an update snapshot is created, boot entries can expose historical root snapshots.

Conceptually:

```
normal Omarchy
snapshot before update A
snapshot before update B
```

This means the bootloader is part of the recovery system.

Screenshot 20.28-A
Limine boot menu with normal Omarchy entry and update snapshots

20.29 — Why Direct Boot Changes Recovery Convenience

If Omarchy is configured to boot directly without showing Limine:

```
normal startup becomes faster/cleaner
```

but snapshot access becomes less visible.

You may need the firmware boot menu to reach Limine.

This is a trade-off:

```
convenience
vs
visible recovery access
```

20.30 — Keep Recovery Discoverable

For a learning workstation, it is useful to know how to reach Limine.

Do not optimize away every recovery screen before you understand it.

A two-second convenience gain is not worth losing confidence during a broken update.

20.31 — Kernel

After the UKI loads, the Linux kernel takes control.

Verify the running kernel:

```
uname -r
```

The course workstation previously reported:

```
7.2.3-arch1-3
```

That is only a historical point from this build.

Your current kernel will change over time.

20.32 — Kernel on Disk vs Running Kernel

After a kernel package update:

```
new kernel may exist on disk
```

while:

```
old kernel is still running in memory
```

until reboot.

This is one reason Omarchy may request a reboot after updates.

20.33 — Initramfs

The initramfs is early userspace used during boot before the real root filesystem is fully available.

In a UKI, initramfs content is bundled into the unified image.

Problems here can prevent the root filesystem from mounting.

Symptoms may include:

- early boot failure;
- emergency shell;
- root-device errors.

This is earlier than Hyprland or Omarchy Shell.

20.34 — systemd

Once the kernel/initramfs hands off to the real root filesystem, systemd starts system services.

At this point:

```
bootloader already succeeded
kernel already succeeded
root filesystem mounted
```

If the machine reaches a TTY but not the graphical desktop:

```
the bootloader/UKI probably worked
```

The problem is likely later in userspace.

20.35 — Graphical Session

The graphical layer includes:

- user session;
- Hyprland;
- Omarchy Shell;
- user services.

If:

```
Ctrl + Alt + F2
```

gives a working TTY:

```
the machine booted significantly further than firmware/bootloader
```

This is a useful diagnostic boundary.

20.36 — Failure Classification

Use this rough map.

Firmware/Boot Entry Failure

Symptoms:

```
no Limine
wrong OS boots
firmware says no boot device
```

Investigate:

- firmware boot order;
- EFI entries;
- EFI partition.

Bootloader/UKI Failure

Symptoms:

```
Limine appears
Omarchy entry fails before kernel/userland
```

Investigate:

- UKI;
- snapshot;
- Secure Boot;
- boot entry.

Kernel/Initramfs Failure

Symptoms:

```
kernel starts
root mount/emergency failure
```

Investigate:

- kernel;
- initramfs;
- storage/root filesystem.

Userspace/Graphical Failure

Symptoms:

```
TTY works
desktop does not
```

Investigate:

- services;
- Hyprland;
- user config;
- shell.

20.37 — Why Failure Classification Saves Time

Without classification:

```
desktop won't start
→ reinstall bootloader?
```

That may be completely wrong.

If TTY works, the bootloader already did its job.

Likewise:

```
firmware cannot find Linux
→ edit Hyprland config?
```

That cannot help.

Always identify how far the boot got.

20.38 — Verify Boot State Before Changing Anything

Useful commands:

```
uname -r
```

```
bootctl status
```

```
efibootmgr
```

```
ls -lh /boot/EFI/Linux
```

```
find /boot -maxdepth 3 -type f | sort
```

Use only the commands that make sense for your system.

20.39 — Inspect the UKI

List:

```
ls -lh /boot/EFI/Linux
```

On the course workstation, the important file was:

```
omarchy_linux.efi
```

Do not rename/delete boot images casually.

20.40 — Do Not Manually Delete "Old-Looking" EFI Files

A file may be:

- active;
- recovery-related;
- snapshot-related;
- referenced by firmware/bootloader.

Boot storage is not a place for casual cleanup.

Use supported Omarchy maintenance tools.

20.41 — Boot Entry vs File

These are different.

```
UEFI boot entry
→ metadata stored in firmware NVRAM
```

```
EFI file
→ executable on EFI System Partition
```

The boot entry points toward a file/loader.

Deleting either layer can break boot.

20.42 — Limine Entry vs UKI Entry

Depending on current configuration, firmware may start:

```
Limine
```

and Limine then loads:

```
Omarchy UKI
```

Do not assume firmware directly loads the UKI.

Verify the actual chain.

20.43 — Practical Exercise: Verify UEFI

Run:

```
test -d /sys/firmware/efi && echo "UEFI mode"
```

Then:

```
bootctl status
```

Record:

```
firmware mode
Secure Boot state
TPM availability if shown
```

Do not change anything.

20.44 — Practical Exercise: Inspect EFI Entries

Run:

```
efibootmgr
```

Identify:

- current BootOrder;
- Windows Boot Manager if dual boot;
- Limine/Omarchy-related entry.

Take a screenshot for the course.

Do not modify BootOrder.

20.45 — Practical Exercise: Inspect Boot Files

Run:

```
find /boot/EFI -maxdepth 3 -type f | sort
```

Identify:

```
Omarchy UKI
Windows EFI loader
Limine EFI files
```

depending on your actual system.

Do not delete anything.

20.46 — Practical Exercise: Verify the Running Kernel

Run:

```
uname -r
```

Record the result.

Then compare after a future kernel update/reboot.

This helps explain:

```
installed kernel
vs
running kernel
```

20.47 — Practical Exercise: Verify Secure Boot State

Use:

```
bootctl status
```

or another current supported inspection tool.

Record:

```
Secure Boot:
enabled / disabled
```

Do not enable it as part of this exercise.

This module teaches understanding first.

20.48 — Practical Exercise: Verify TPM

Use your current firmware/system tooling to verify TPM 2.0.

Possible tools can include:

```
systemd-analyze has-tpm2
```

if available on the current system.

Record:

```
TPM 2.0 available?
```

Again:

```
TPM available
≠
Secure Boot enabled
```

20.49 — Practical Exercise: Snapshot Boot Path

Do not restore anything.

Restart when convenient and observe Limine.

Identify:

- normal Omarchy entry;
- available snapshot entries.

Then boot normally.

This gives you a visual memory of the recovery path.

20.50 — Practical Exercise: Identify Windows Boot Path

On dual-boot systems:

use the one-time firmware boot menu.

Identify:

```
Windows Boot Manager
Limine
```

Do not change permanent boot order.

The goal is knowing your escape paths.

20.51 — Practical Exercise: Build a Boot Recovery Note

Record:

```
Firmware entry key:
Boot menu key:
Omarchy bootloader:
Windows entry:
Secure Boot current state:
TPM:
UKI path:
Snapshot access path:
USB installer location:
Windows recovery key location:
```

Do not store the actual BitLocker recovery secret inside a public dotfiles repository.

Record where it is safely stored, not the secret itself.

20.52 — Before Enabling Secure Boot Later

Create this checklist:

```
[ ] current official Omarchy Secure Boot guide checked
[ ] important data backed up
[ ] Windows BitLocker recovery key available
[ ] firmware boot menu known
[ ] Secure Boot disable path known
[ ] Limine entry verified
[ ] Omarchy UKI verified
[ ] recovery USB available
```

Only then consider the change.

20.53 — Common Boot Mistakes

Mistake 1 — Assuming GRUB

Verify the actual bootloader.

The course workstation used Limine.

Mistake 2 — Enabling Secure Boot Without Preparing Linux

You can make a healthy Linux installation unbootable.

Mistake 3 — Confusing TPM With Secure Boot

They are different security technologies.

Mistake 4 — Changing TPM/Secure Boot With BitLocker Enabled and No Recovery Key

Windows may request recovery.

Mistake 5 — Deleting EFI Files to "Clean Up"

Boot files are not normal cache files.

Mistake 6 — Editing Hyprland to Fix a Firmware Boot Failure

Classify the failure layer first.

Mistake 7 — Reinstalling the Bootloader When TTY Already Works

If userspace is running, the bootloader likely succeeded.

Mistake 8 — Forgetting Snapshot Access After Enabling Direct Boot

Know how to reach Limine through firmware.

Mistake 9 — Assuming the New Kernel Is Running Immediately After Package Update

Reboot is required to load it.

Mistake 10 — Following an Old Secure Boot Tutorial Blindly

Use current official Omarchy guidance for the installed release.

20.54 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Boot Architecture

- [ ] I understand UEFI → bootloader → UKI → kernel → userspace.
- [ ] I understand what Limine does.
- [ ] I understand what a UKI is.
- [ ] I understand systemd-stub at a high level.
- [ ] I understand initramfs at a high level.

Verification

- [ ] I can verify UEFI mode.
- [ ] I can inspect EFI boot entries.

- [ ] I can inspect the Omarchy UKI path.
- [ ] I can check the running kernel.
- [ ] I can verify Secure Boot state.
- [ ] I understand TPM availability is separate.

Recovery

- [ ] I know how to reach Limine.
- [ ] I know how snapshots fit into boot recovery.
- [ ] I know how to use the firmware one-time boot menu.
- [ ] I know Windows Boot Manager may exist separately.

Secure Boot

- [ ] I understand Secure Boot conceptually.
- [ ] I know not to enable it blindly.
- [ ] I understand signed boot images are required.
- [ ] I understand Windows BitLocker can react to firmware/security changes.
- [ ] I know to use the current official Omarchy procedure before enabling it.

Troubleshooting

- [ ] I can distinguish firmware, bootloader, kernel, and userspace failures.
- [ ] I understand that a working TTY proves the machine booted well past the bootloader.
- [ ] I know not to apply desktop fixes to pre-kernel boot problems.

20.55 — What You Can Now Do

After completing Module 20, you can now:

- understand the full Omarchy boot chain;
- verify UEFI state;
- identify the real bootloader;
- inspect firmware boot entries;
- understand the role of Limine;
- identify the active UKI;
- understand how snapshots become bootable recovery points;
- distinguish TPM from Secure Boot;
- prepare safely for future Secure Boot changes;

- account for Windows/BitLocker in dual boot;
- classify boot failures by layer;
- choose recovery actions based on how far the machine actually booted.

Most importantly:

> *You no longer treat "the computer doesn't boot normally" as one giant problem. You can identify which layer failed.*

Module 20 Summary

Modern Omarchy boot chain:

```
Power
↓
UEFI
↓
Limine
↓
Unified Kernel Image
↓
kernel + initramfs
↓
systemd
↓
Hyprland
↓
Omarchy Shell
```

Course workstation verified:

```
UEFI 2.70
Limine 12.8.0
TPM 2.0 available
Secure Boot disabled
Omarchy.linux
/boot/EFI/Linux/omarchy_linux.efi
systemd-stub 261.2-1-arch
```

These are examples from one real workstation, not fixed requirements.

Useful inspection:

`bootctl status` `efibootmgr` `uname -r` `ls -lh /boot/EFI/Linux`

Failure classification:

```
firmware cannot find boot target
→ firmware/EFI layer

Limine appears but Omarchy target fails
→ bootloader/UKI layer

kernel starts but root fails
→ kernel/initramfs/storage layer

TTY works but desktop fails
→ userspace/session layer
```

Secure Boot:

```
capable
≠
enabled
```

TPM:

```
present
≠
Secure Boot enabled
```

Dual boot:

```
Limine
+
Windows Boot Manager
```

can coexist as separate UEFI entries.

And the most important rule:

> ***Verify the actual boot chain before trying to repair it.***

MODULE 21

21

Security and Hardening on Omarchy

You now understand the boot chain, updates, recovery, hardware, runtime controls, and the development stack.

The next question is:

About This Module

You now understand the boot chain, updates, recovery, hardware, runtime controls, and the development stack.

The next question is:

> ***How do we keep this workstation secure without turning it into an unusable fortress?***

Current Omarchy 4 / Quattro already makes several strong security choices by default.

The current official security documentation says the platform is designed around:

- mandatory full-disk encryption;
- a firewall enabled by default;
- SSH disabled until explicitly enabled;
- rate-limited SSH exposure when enabled;
- Docker firewall hardening;
- controlled package sources;
- frequent security updates through the rolling Arch base.

Omarchy also includes first-class setup paths for:

- fingerprint authentication;
- FIDO2 hardware authentication;
- SSHD;
- Docker privilege mode;
- password changes.

This module is deliberately practical.

We are not going to create a generic "Linux hardening checklist" full of cargo-cult commands.

The goal is:

> ***Understand the trust boundaries of the workstation, preserve the protections Omarchy already gives you, and tighten the system where your real risk justifies it.***

What You Will Learn

By the end of Module 21, you will understand:

- Omarchy's default security model;
- why full-disk encryption matters;
- what LUKS protects and what it does not;
- the difference between the disk-unlock password and user/sudo password;
- how the firewall behaves by default;
- why SSH is disabled by default;
- how to enable SSH intentionally;
- why SSH exposure changes your attack surface;
- password vs key-based SSH at a practical level;
- how FIDO2 authentication fits into sudo/polkit;
- how fingerprint authentication fits into lock screen and sudo;
- why Docker's `docker` group is effectively root-equivalent;
- why Omarchy defaults to sudo-controlled Docker access;
- how "Sudoless Docker" changes the security model;
- why plugins are executable code;
- why hooks and personal scripts are trust boundaries;
- why AI coding agents deserve the same security scrutiny as shell scripts;
- how project dependencies can become an attack vector;
- how browser extensions and third-party applications fit into workstation trust;
- why least privilege matters;
- how to inspect listening services;
- how to think about secrets;
- how to maintain a practical security checklist.

21.1 — Security Is About Layers

A useful workstation security model is:

```
Physical security
↓
Disk encryption
↓
Boot trust
↓
User authentication
↓
Firewall/network exposure
↓
Privilege boundaries
↓
Application trust
↓
Developer-tool trust
↓
Project/dependency trust
↓
Secrets/data handling
```

No single security feature solves everything.

For example:

```
LUKS
→ protects stolen powered-off storage
```

but it does not stop:

```
malicious code running while you are logged in
```

Likewise:

```
firewall
→ reduces network exposure
```

but does not stop:

```
a malicious plugin reading your home directory
```

Security must be layered.

21.2 — Omarchy's Current Default Security Baseline

Current official Omarchy security documentation says the default system includes:

```
mandatory full-disk encryption
firewall enabled
incoming traffic blocked by default
SSHD disabled
LocalSend exception
Docker firewall integration
current package updates
controlled default package sources
```

This is already a strong baseline.

Do not undo those protections casually.

21.3 — Full-Disk Encryption

Current Omarchy installs use:

```
LUKS
```

for full-disk encryption.

This protects data at rest when the computer is powered off.

If a laptop is stolen:

```
attacker removes drive
↓
reads raw storage
↓
encrypted data
```

without the unlock secret.

That is the main physical-data protection.

21.4 — What Disk Encryption Protects

LUKS helps protect:

- source code;

- SSH keys;
- browser profiles;
- documents;
- API credentials stored on disk;
- local databases;
- project history;
- personal configuration;

when the drive is not unlocked.

This is especially important for laptops.

21.5 — What Disk Encryption Does Not Protect

Once the machine is:

```
booted
+
unlocked
+
user logged in
```

the operating system must be able to read your data.

Therefore disk encryption does not stop:

- malicious software running as you;
- compromised browser extensions;
- untrusted shell plugins;
- rogue development dependencies;
- commands you intentionally run;
- someone using an already-unlocked workstation.

This distinction is essential.

21.6 — Two Important Passwords

Current official Omarchy security documentation distinguishes:

```
drive encryption password
```

and:

```
user/login/sudo password
```

They protect different layers.

Drive Encryption Password

Used at boot to unlock encrypted storage.

User Password

Used for:

- login;
- sudo;
- system authorization.

Do not assume changing one changes the other.

21.7 — Change Passwords Through Omarchy

Current official workflow:

```
Super + Space
→ Update
→ Password
```

Current categories include:

```
Drive Encryption
User
```

Use the supported UI/workflow rather than manually editing low-level authentication files unless you know exactly why.

21.8 — Strong Password Strategy

A useful local workstation password should be:

- long enough;
- difficult to guess;
- unique;
- memorable enough that you do not disable security out of frustration.

A passphrase is often easier to remember than a short complex password.

Do not reuse:

- GitHub password;
- email password;
- banking password;

as your local workstation password.

21.9 — Firewall

Current official Omarchy security documentation says the firewall is:

```
enabled by default
```

and incoming traffic is blocked except for intended exceptions.

The current documented default includes LocalSend on:

```
port 53317
```

SSHD is not exposed until you enable it.

This is a strong default.

21.10 — Why a Default-Deny Firewall Helps

A developer workstation can run many processes:

- dev servers;
- databases;
- containers;
- test services;
- local APIs.

Without care, some may listen on:

```
0.0.0.0
```

instead of:

```
127.0.0.1
```

A firewall reduces the chance that a local experiment becomes a network-exposed service.

21.11 — Inspect Listening Services

Use:

```
ss -tulpn
```

or for TCP:

```
ss -ltnp
```

This helps you see which programs are listening.

You may need appropriate permissions for some process details.

Ask:

```
What is listening?
On which address?
On which port?
Do I expect it?
```

21.12 — `127.0.0.1` vs `0.0.0.0`

This matters greatly.

```
127.0.0.1:3000
```

means:

```
local machine only
```

while:

```
0.0.0.0:3000
```

means:

```
all IPv4 interfaces
```

subject to firewall rules.

When running local dev servers, prefer local-only binding unless remote access is intentional.

21.13 — Dev Servers Are Part of Your Attack Surface

Examples:

`npm run dev` `python -m http.server` `uvicorn ...`

may bind differently depending on the tool/config.

Do not assume:

`"development server"`

means:

`"only accessible from my machine"`

Inspect the actual bind address.

21.14 — SSH Is Disabled by Default

Current official Omarchy security documentation states that:

`SSHD is off by default`

This is good.

If you do not need remote shell access:

> ***Leave it off.***

An unavailable network service has a very small attack surface.

21.15 — Enable SSH Intentionally

Current official menu path:

```
Super + Space
→ Setup
→ Security
→ SSHD
```

Current documentation says enabling SSH:

- enables the SSH daemon;
- opens port 22 in the firewall;
- applies rate limiting against brute-force attempts.

This is much better than enabling the service and forgetting firewall state.

21.16 — Enabling SSH Changes the Threat Model

Before SSH:

```
network attacker
→ port closed
```

After SSH:

```
network attacker
→ can reach SSH authentication
```

That does not mean SSH is unsafe.

It means:

> *You deliberately created a remote login surface.*

Treat it accordingly.

21.17 — SSH Keys

For remote administration, SSH keys are generally preferable to repeatedly typing remote account passwords.

A typical key workflow:

```
ssh-keygen
```

then install the public key on the remote machine.

Private key:

```
keep secret
```

Public key:

```
safe to distribute to servers you authorize
```

Do not upload private SSH keys to repositories.

21.18 — SSH Private Keys Are Credentials

Files under:

```
~/.ssh
```

can include extremely sensitive material.

Examples:

```
id_ed25519
config
known_hosts
authorized_keys
```

The private key is equivalent to a credential.

Protect it like a password.

21.19 — Passphrases on SSH Keys

A passphrase can protect a private SSH key if the key file is stolen.

The trade-off:

```
more security
vs
more interaction
```

An SSH agent can reduce repeated prompting while preserving key encryption.

For high-value accounts/servers, passphrases are strongly worth considering.

21.20 — `authorized_keys`

On a machine accepting SSH:

```
~/.ssh/authorized_keys
```

defines which public keys may log in under that account.

Review it periodically if you use SSH.

Unknown key:

```
potential unauthorized access
```

Do not leave old temporary keys forever.

21.21 — Current Omarchy Unattended Install SSH Behavior

Current official unattended-install documentation says that when:

```
authorized_keys
```

is supplied during imaging, Omarchy:

- installs those keys for the user;
- enables SSHD;
- opens the firewall appropriately.

Otherwise, stock Omarchy leaves SSH disabled and port 22 closed.

This shows the security model is explicit-access-first.

21.22 — Fingerprint Authentication

Current official Omarchy hardware-authentication workflow:

```
Super + Space
→ Setup
→ Security
→ Fingerprint
```

The setup installs required support, enrolls the fingerprint, verifies it, and integrates it with authentication.

Current documentation says fingerprints can be used for:

- lock-screen unlock;
- sudo;

- system authorization prompts.

21.23 — Fingerprint Is Convenience + Security

A fingerprint can make secure workflows more convenient.

That matters because:

```
security that is too annoying
→ users disable it
```

But remember:

```
fingerprint
≠
secret you can change
```

Biometrics have different security properties from passwords.

Use them as part of a layered authentication setup.

21.24 — Laptop Lid Behavior

Current official docs say that when the laptop lid is closed:

```
fingerprint prompt is skipped
```

so the system does not wait for an inaccessible sensor.

This is a good example of security integrated with usability.

21.25 — Fingerprint and `sudo`

If a fingerprint prompt appears during `sudo` but you cannot/want not to use the sensor, current official docs say:

```
Ctrl + C
```

can move you to the password path in relevant situations.

This is useful with external keyboards/docks.

21.26 — FIDO2 Authentication

Current official workflow:

```
Super + Space
→ Setup
→ Security
→ Fido2
```

FIDO2 hardware tokens can be configured for:

- sudo authentication;
- system authorization prompts.

Current documentation says FIDO2 does **not** currently replace lock-screen unlock in the same way fingerprint authentication can.

21.27 — FIDO2 Security Keys

Examples include hardware devices such as:

- YubiKey;
- compatible FIDO2 security keys.

The key proves possession of a hardware authenticator.

This can significantly strengthen privileged authentication.

21.28 — Do Not Lose Your Only Authentication Path

Before enabling hardware authentication:

```
make sure password fallback works
```

Do not design a system where losing one hardware token makes local administration impossible.

For important machines, maintain a deliberate recovery path.

21.29 — Hardware Auth Can Have Current-Version Bugs

Authentication stacks involve:

- PAM;

- polkit;
- fingerprint services;
- FIDO2;
- shell dialogs.

These integrations evolve.

Current Omarchy has had active work around combined fingerprint/FIDO2/polkit UX.

Therefore:

> *Test your real authentication flows after enabling hardware auth.*

Test:

- sudo;
- lock screen;
- system authorization prompt;
- external keyboard/docked usage.

21.30 — Docker and Privilege

Docker is one of the most commonly misunderstood workstation security boundaries.

Current official Omarchy development-tool documentation explicitly states:

> *Membership in the `docker` group is effectively passwordless root.*

Why?

Because a user who controls Docker can start a privileged-enough container and mount the host filesystem.

Conceptually:

```
docker access
→ mount /
→ modify host
→ root-equivalent control
```

21.31 — Current Omarchy Docker Default

Current Omarchy deliberately keeps the user:

```
out of the docker group
```

by default.

Therefore Docker CLI use normally requires authorization such as:

```
sudo docker ps
```

Current graphical Docker tooling asks for authorization when necessary.

This preserves a privilege boundary.

21.32 — "Sudoless Docker"

Current Omarchy provides:

```
Setup
→ Security
→ Sudoless Docker
```

for users who intentionally prefer convenience.

Current CLI helper:

```
omarchy-setup-security-sudoless-docker
```

adds the user to the Docker group after warning about the security trade-off.

21.33 — Should You Enable Sudoless Docker?

Ask:

```
Do I use Docker constantly?
Do I understand Docker-group root equivalence?
Do I run untrusted project scripts/dependencies?
Do AI coding agents execute commands unattended?
```

If you frequently run unknown code:

```
keeping Docker behind sudo
```

creates an additional barrier.

For this course baseline:

> ***Keep the default unless you have a concrete reason to change it.***

21.34 — AI Agents Make Docker Privilege More Important

Consider:

```
AI agent
→ runs project command
→ project command invokes docker
```

If your user has unrestricted Docker access:

```
agent-controlled process
→ potentially root-equivalent Docker control
```

This is one reason security boundaries matter more on AI-native workstations.

21.35 — `sudo` Is a Security Boundary

`sudo` means:

```
run command with elevated privileges
```

Do not treat:

```
sudo
```

as a magical prefix that fixes permission errors.

Before using sudo, ask:

```
Why does this command need root?
```

If the answer is unclear:

```
stop and inspect
```

21.36 — Never Use `sudo` for Personal Config Files

Your files under:

```
~/.config
~/Projects
~/.local
```

normally belong to your user.

If you repeatedly need sudo to edit your own project/config:

```
ownership may be wrong
```

Fix the ownership cause rather than normalizing root-owned user files.

21.37 — Check File Ownership

Use:

```
ls -l FILE
```

or:

```
stat FILE
```

If a file in your home directory is unexpectedly owned by:

```
root
```

ask how it became root-owned.

Do not blindly `sudo` around the problem forever.

21.38 — Plugins Are Executable Code

Current official Omarchy Shell documentation states that third-party shell plugins run as:

```
unsandboxed user-level code
```

inside the shell environment.

They can have the same user-level file/process access as your account.

This means:

> ***Installing a plugin is closer to installing software than installing a wallpaper.***

21.39 — Plugin Trust Checklist

Before enabling a third-party plugin:

```
[ ] repository is the intended one
[ ] maintainer/source is credible enough
[ ] manifest inspected
[ ] QML/JS/helper code inspected
[ ] shell/process execution understood
[ ] update diff reviewed
[ ] no unexplained credential access
```

Validation checks structure.

It does not prove benign behavior.

21.40 — Current Plugin Security Improvements Do Not Remove Trust

Current Quattro provides scoped interfaces and restricts direct access to some sensitive shell services.

That is good hardening.

But current official docs still state plugins retain user-level code execution capabilities.

Therefore:

```
reduced ambient API access
≠
sandboxed untrusted code
```

The trust boundary remains.

21.41 — Hooks Are Executable Code Too

User hooks under:

```
~/.config/omarchy/hooks
```

run automatically when events occur.

Examples:

```
post-update
post-boot
theme-set
```

A malicious or broken hook can repeatedly run without you manually starting it.

Therefore:

- version your hooks;
- keep them simple;
- avoid unnecessary sudo;
- inspect scripts before installing them.

21.42 — Autostart Is Another Trust Boundary

Anything in:

```
~/.config/hypr/autostart.lua
```

can run every graphical session.

That makes autostart a persistence mechanism.

When troubleshooting suspicious behavior:

```
review autostart
```

along with systemd user services and shell plugins.

21.43 — AI Coding Agents Are Powerful Local Programs

An AI agent may have permission to:

- read files;
- write files;
- execute commands;
- use network access;

- call Git;
- install dependencies;
- modify config.

That makes agent permissions part of workstation security.

Do not think of an agent as:

```
a chat box
```

Think of it as:

```
a programmable local operator
```

with whatever permissions you grant.

21.44 — Agent Permission Modes Matter

Different agents expose different approval modes.

The stable principle is:

```
more automation
→ larger blast radius
```

Use broad unattended/yolo modes only when:

- repository is trusted;
- task is scoped;
- Git is clean;
- secrets are protected;
- system-level commands are not expected.

21.45 — Natural-Language Instructions Are Not a Sandbox

Prompt:

```
"Do not read secrets."
```

is useful guidance.

It is not technical isolation.

If the agent process can read:

```
~/.ssh
~/.config
.env
```

then it technically has access unless the tool/runtime prevents it.

Security should not depend only on a sentence in the prompt.

21.46 — Project Dependencies Are Code

Running:

`npm install` `pip install` `cargo build`

can execute code through:

- install scripts;
- build scripts;
- package hooks;
- compiled dependencies.

Do not assume:

`dependency declaration`

is passive data.

Untrusted repositories deserve caution.

21.47 — Clone First, Inspect Second, Execute Third

For unfamiliar repositories:

```
clone
↓
inspect README
↓
inspect package manifests
↓
inspect install scripts
↓
inspect mise.toml
↓
then install/run
```

This is why Module 6 taught:

> *Review project configuration before `mise install`.*

21.48 — Lockfiles Help Reproducibility, Not Trust

A lockfile helps ensure:

```
same dependency versions
```

It does not prove those dependencies are safe.

Security still requires:

- trusted sources;
- dependency review;
- updates;
- vulnerability awareness.

21.49 — AUR Packages

Current official Omarchy security documentation says the base installation relies primarily on:

- Arch core/extra/multilib;
- Omarchy's own repository.

Optional software may use the AUR.

AUR packages are community build recipes.

Treat them as third-party code.

Before installing:

- inspect package source/build instructions;
- understand origin;
- verify you actually need it.

21.50 — `curl | sh`

A command like:

```
curl https://example.com/install.sh | sh
```

downloads code and immediately executes it.

That is convenient.

It also removes the inspection step.

Prefer:

```
download
inspect
execute
```

for untrusted or high-impact scripts.

21.51 — Secrets

Developer workstations often contain:

- API keys;
- cloud credentials;
- database passwords;
- SSH keys;
- GitHub tokens;
- `.env` files;
- browser sessions.

The biggest practical security risk may not be someone "hacking Linux."

It may be:

```
accidentally exposing a secret
```

21.52 — Never Commit Secrets

Before committing:

```
git status
```

```
git diff --staged
```

Look for:

- `.env`;
- credentials;
- private keys;
- config files with tokens;
- copied production data.

Add appropriate entries to:

```
.gitignore
```

before the secret reaches Git history.

21.53 — `.gitignore` Does Not Remove Existing Secrets

If a secret was already committed:

```
adding it to .gitignore later
```

does not remove it from history.

Treat the secret as compromised.

Typical response:

```
rotate/revoke secret
↓
clean repository history if required
```

Do not rely on deletion alone.

21.54 — Environment Variables

Environment variables are convenient for secrets.

But remember:

- child processes may inherit them;
- crash logs may expose them in some cases;
- shell history can expose assignment commands;
- agent-run processes may inherit them.

Use environment-based secrets thoughtfully.

21.55 — `.env` Files

`.env` files are convenient for local development.

Good practice:

```
.env
→ ignored

.env.example
→ safe template
```

Do not put real production credentials in public examples.

21.56 — Shell History

Commands can contain secrets.

Bad:

```
curl -H "Authorization: Bearer SUPER_SECRET_TOKEN" ...
```

That may remain in shell history.

Prefer tools that read secrets from:

- environment;
- credential store;
- protected config;

- standard input;

rather than embedding them directly into commands.

21.57 — Browser Security

The browser is one of the highest-value applications on the workstation.

It may contain:

- logged-in email;
- cloud consoles;
- GitHub;
- password manager;
- banking;
- developer SaaS.

Use:

- automatic browser updates;
- minimal extensions;
- trusted extension sources;
- separate profiles where useful.

21.58 — Browser Extensions Are Code

A browser extension may have permission to:

- read page contents;
- modify pages;
- access tabs;
- read browsing activity.

Do not install random extensions because a blog recommended them.

Review permissions.

Remove extensions you no longer use.

21.59 — Password Managers

A dedicated password manager is generally preferable to password reuse or plain-text notes.

Your exact choice is personal.

Important goals:

- unique passwords;
- protected vault;
- strong account recovery;
- MFA where appropriate.

The course does not mandate one vendor.

21.60 — MFA for Developer Accounts

High-value accounts include:

- GitHub;
- email;
- cloud providers;
- package registries;
- domain registrars;
- production platforms.

Enable strong MFA.

Hardware security keys are especially valuable for high-impact accounts where supported.

21.61 — GitHub Authentication

GitHub CLI authentication can provide repository access.

Inspect:

```
gh auth status
```

Treat GitHub credentials as sensitive.

A compromised GitHub token/account can expose:

- private source code;
- actions/secrets;
- packages;
- organization resources.

21.62 — Principle of Least Privilege

Least privilege means:

> *Give a program/user only the access needed for the task.*

Examples:

```
Docker behind sudo
→ better privilege separation

SSHD off when unused
→ smaller attack surface

agent read-only during analysis
→ smaller blast radius

plugin disabled until reviewed
→ no execution before trust
```

This principle appears throughout Omarchy's current design.

21.63 — Security vs Convenience

Many security decisions are trade-offs.

Examples:

```
fingerprint
→ faster auth

FIDO2
→ stronger privileged auth

Sudoless Docker
→ faster Docker commands, weaker privilege boundary

SSHD
→ remote access, larger network surface

yolo agent mode
→ faster automation, larger blast radius
```

A good workstation makes those trade-offs consciously.

21.64 — Lock the Workstation

Current official hotkey:

```
Super + Ctrl + L
```

locks the workstation.

Use it whenever leaving the machine unattended.

Full-disk encryption does not help if:

```
the machine is already unlocked
```

and someone can walk up to it.

21.65 — Suspend Is Not the Same as Power-Off Protection

Depending on system state and threat model, a suspended machine may retain sensitive state in memory.

For ordinary users, screen lock + encrypted storage is still a strong baseline.

For very high-risk environments, shutdown/hibernate/advanced boot security may deserve separate consideration.

This course remains focused on practical developer-workstation security.

21.66 — Secure Boot

Module 20 covered Secure Boot architecture.

Security takeaway:

```
Secure Boot
→ strengthens boot-chain trust
```

but it must be configured correctly for the current Omarchy boot stack.

Do not enable it blindly.

Use the current supported procedure.

21.67 — TPM

TPM can support:

- key protection;
- measured boot;
- OS security features.

But:

```
TPM present
```

does not automatically harden every part of the workstation.

It is one building block.

21.68 — Security Updates

Current official Omarchy security docs emphasize that Arch's rolling base provides rapid access to security fixes.

On Omarchy, use:

```
omarchy update
```

rather than bypassing the supported update pipeline.

A secure workstation that never receives updates eventually stops being secure.

21.69 — Stable Channel and Security

Current Omarchy stable intentionally trades some package freshness for tested integration.

Do not assume:

```
edge
→ automatically more secure
```

because it is newer.

Security also includes:

- reliability;

- correct configuration;
- tested migrations.

For most users, stable remains the sensible security/operability balance.

21.70 — Check for Unexpected Services

Occasionally inspect:

```
systemctl --type=service --state=running
```

and:

```
systemctl --user --type=service --state=running
```

Do not try to disable everything unfamiliar.

Instead ask:

```
Do I recognize the service?
Why is it running?
Does it listen on the network?
Is it expected?
```

21.71 — Check Network Exposure

Useful:

```
ss -ltnup
```

Then correlate with firewall state/current services.

If you see:

```
0.0.0.0:PORT
```

ask whether that service is intentionally reachable.

This is especially important for:

- dev servers;
- databases;
- container ports.

21.72 — Localhost Development

For normal local development, prefer services to bind to:

```
127.0.0.1
```

or:

```
localhost
```

unless LAN/remote access is intentional.

Examples:

```
Postgres
Redis
local APIs
debug servers
```

A database intended only for local development usually does not need LAN exposure.

21.73 — Remote Development

Module 10 covered SSH port forwarding.

This is often safer than exposing dev services directly.

Instead of:

```
remote app listening publicly on port 3000
```

use:

```
SSH
→ forward remote 3000
→ localhost:3000
```

This keeps the service behind SSH authentication.

21.74 — Tailscale/VPN-Like Private Networking

Current unattended-install documentation supports Tailscale onboarding.

Private overlay networks can reduce the need to expose services publicly.

But:

```
private network
≠
no security needed
```

Devices inside the private network still need appropriate trust boundaries.

21.75 — Security Review After Installing New Software

After adding a major tool, ask:

```
Did it start a service?
Did it open a port?
Did it add autostart?
Did it add my user to a privileged group?
Did it install a browser extension?
Did it modify shell config?
```

This is a better hardening habit than running generic scanners blindly.

21.76 — Security Review After AI Tool Installation

For coding agents specifically, review:

- credential storage;
- API keys;
- file access;
- command execution;
- network access;
- MCP/plugin integrations;
- approval modes;
- shell integration.

An AI-native workstation has a broader automation surface than a traditional development laptop.

21.77 — Plugins, Connectors, and External Integrations

Anything connected to:

- GitHub;
- cloud accounts;
- Slack;
- databases;
- browsers;
- local filesystem;

inherits some trust.

Review:

```
what can it read?
what can it write?
what can it execute?
what credential does it use?
```

Do not grant broad scopes when narrow scopes work.

21.78 — Backups Are Security

Security is not only:

```
prevent unauthorized access
```

It also includes:

```
availability
+
recovery
```

Ransomware, hardware failure, accidental deletion, and bad automation all threaten data.

Your recovery layers:

```
Git
dotfiles
system snapshots
personal-data backup
```

are part of security.

21.79 — Snapshots Are Not Backups

Again:

```
Omarchy snapshot
→ root filesystem recovery

/home
→ not restored
```

Therefore backups for:

- personal files;
- source repositories not pushed;
- documents;
- local-only data;

still matter.

21.80 — Security Hardening Should Be Threat-Driven

Ask:

```
What am I protecting?
From whom?
What is the likely attack?
What is the impact?
```

Example:

LAPTOP DEVELOPER

High concern:

- theft;
- credential theft;

- browser compromise;
- malicious dependency;
- public Wi-Fi;
- accidental secret leak.

Useful controls:

- LUKS;
- lock screen;
- MFA;
- firewall;
- minimal SSH;
- safe Docker boundary;
- trusted dependencies;
- backups.

This is more valuable than random kernel-hardening flags you do not understand.

21.81 — Practical Exercise: Verify Encryption

Do **not** modify encryption.

Inspect current block devices with:

```
lsblk -f
```

Identify the encrypted/LUKS-backed structure.

The exact layout varies.

Goal:

```
understand that root storage is encrypted
```

not to reconfigure it.

21.82 — Practical Exercise: Inspect Listening Ports

Run:

```
ss -ltnup
```

Identify:

- localhost-only services;
- all-interface listeners;
- ports you recognize.

If process names require additional privileges, inspect carefully rather than immediately using sudo for every command.

21.83 — Practical Exercise: Check SSH State

Use the current Omarchy menu:

```
Setup
→ Security
→ SSHD
```

Determine whether SSH is enabled.

If you do not need SSH:

```
leave it disabled
```

If you do need it:

verify why and document the reason.

21.84 — Practical Exercise: Inspect SSH Keys

Run:

```
ls -la ~/.ssh
```

If you have keys:

```
ssh-keygen -lf ~/.ssh/id_ed25519.pub
```

or the appropriate public key.

Do not print private-key contents.

21.85 — Practical Exercise: Docker Privilege Check

If Docker is installed:

```
groups
```

Look for:

```
docker
```

If absent:

```
Omarchy's safer default privilege boundary remains
```

If present:

```
understand that your user is effectively Docker-root capable
```

Do not remove yourself impulsively if active workflows depend on it; plan the change.

21.86 — Practical Exercise: Fingerprint/FIDO2 Discovery

Open:

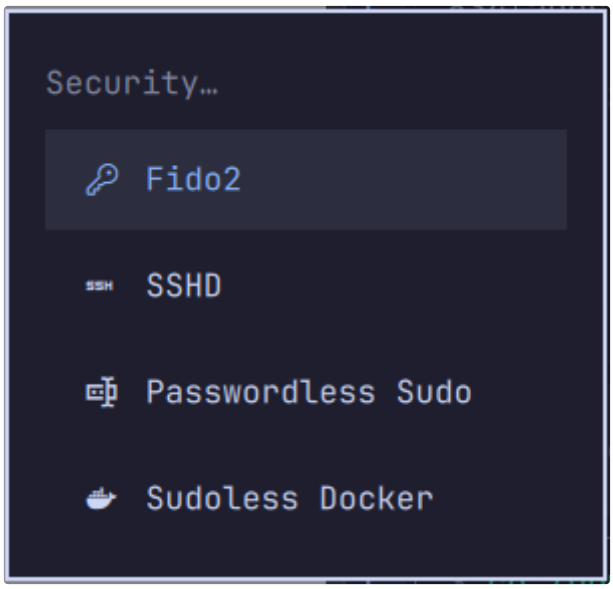
```
Super + Space
→ Setup
→ Security
```

Identify current options:

```
Fingerprint
Fido2
SSHD
Sudoless Docker
```

Your hardware may not support every option.

Do not enable hardware auth just to complete the exercise.



Screenshot 21.86-A
Setup → Security menu showing Fido2, SSHD, Passwordless Sudo, and Sudoless Docker options

21.87 — Practical Exercise: Plugin Review

Run:

```
omarchy plugin list
```

Identify third-party plugins.

For each:

```
Do I still use it?
Do I trust the source?
When was it last updated?
```

Remove or disable anything you no longer need.

21.88 — Practical Exercise: Autostart Review

Inspect:

```
bat ~/.config/hypr/autostart.lua
```

Ask:

```
Do I recognize every command?
Does each one need to start every session?
```

This is both performance maintenance and security review.

21.89 — Practical Exercise: Hook Review

Run:

```
fd -H . ~/.config/omarchy/hooks
```

Inspect executable personal hooks.

Ask:

```
What event triggers this?
What command does it run?
Does it use sudo?
Does it access secrets?
```

21.90 — Practical Exercise: Secret Scan Before Commit

From a project:

```
git status
```

Then:

```
git diff --staged
```

Look specifically for:

```
.env
token
secret
password
private_key
api_key
```

Do not rely only on automated secret scanners.

Human review remains valuable.

21.91 — Practical Exercise: Build Your Security Baseline

Write:

```
Disk encryption:
Firewall:
SSHD:
Docker group:
Fingerprint:
FIDO2:
Secure Boot:
TPM:
Main browser:
MFA enabled on GitHub:
Backups:
Third-party plugins:
```

Fill in your current state.

This creates a reproducible security baseline.

Do **not** put secrets themselves into the document.

21.92 — Common Security Mistakes

Mistake 1 — Disabling Security Because It Is Slightly Inconvenient

Use convenience tools like fingerprint/FIDO2 instead of removing boundaries blindly.

Mistake 2 — Enabling SSH "Just in Case"

If you do not use it, leave it off.

Mistake 3 — Joining the Docker Group Without Understanding Root Equivalence

Convenience changes privilege boundaries.

Mistake 4 — Treating Plugins as Visual Assets

They execute code.

Mistake 5 — Trusting AI Agent Instructions as a Security Sandbox

Natural language is not access control.

Mistake 6 — Running `curl | sh` From Unknown Sources

Inspect before executing.

Mistake 7 — Committing `.env` or Private Keys

Treat exposed credentials as compromised and rotate them.

Mistake 8 — Binding Dev Servers to `0.0.0.0` Without Intending Remote Access

Prefer localhost for local development.

Mistake 9 — Using `sudo` to Fix Every Permission Problem

Understand ownership and privilege first.

Mistake 10 — Hardening Random Things While Ignoring MFA, Backups, and Updates

Focus on realistic threats first.

21.93 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Physical/Data Security

- [ ] I understand what LUKS protects.
- [ ] I understand what LUKS does not protect.
- [ ] I know drive and user passwords are separate.
- [ ] I lock the workstation when unattended.

Network Security

- [ ] I understand the firewall is enabled by default.
- [ ] I know SSH is disabled by default.
- [ ] I understand enabling SSH changes the attack surface.
- [ ] I can inspect listening services.
- [ ] I understand localhost vs all-interface binding.

Authentication

- [ ] I understand fingerprint integration.
- [ ] I understand FIDO2 integration.

- [ ] I know hardware auth needs a recovery path.
- [ ] I understand TPM and Secure Boot are separate.

Privilege

- [ ] I understand why the Docker group is root-equivalent.
- [ ] I know Omarchy's safer default keeps Docker behind authorization.
- [ ] I understand `sudo` is a privilege boundary.
- [ ] I avoid root-owned personal files.

Code Trust

- [ ] I treat plugins as executable code.
- [ ] I treat hooks/autostart as persistence.
- [ ] I understand AI agents can have broad local authority.
- [ ] I inspect untrusted projects before running install/build scripts.
- [ ] I treat AUR packages as third-party code.

Secrets

- [ ] I do not commit secrets.
- [ ] I understand `.gitignore` does not erase Git history.
- [ ] I know shell history can expose secrets.
- [ ] I use strong MFA on important developer accounts.

Recovery

- [ ] I understand backups are part of security.
- [ ] I know snapshots do not replace backups.
- [ ] I can describe my current workstation security baseline.

21.94 — What You Can Now Do

After completing Module 21, you can now:

- reason about Omarchy security as layered trust boundaries;
- preserve the strong security defaults already provided;
- understand full-disk encryption;
- expose SSH only when needed;
- manage local authentication intelligently;

- use fingerprint/FIDO2 deliberately;
- understand Docker privilege risk;
- review third-party plugins and hooks as code;
- constrain AI agents more appropriately;
- inspect network exposure;
- protect developer secrets;
- reduce dependency/install-script risk;
- prioritize realistic security controls over random hardening.

Most importantly:

> *You can now make security decisions based on actual privileges and attack surfaces rather than vague "Linux is secure" assumptions.*

Module 21 Summary

Current Omarchy security baseline:

```
mandatory LUKS encryption
+
firewall enabled
+
incoming traffic blocked by default
+
SSHD off until enabled
+
Docker firewall hardening
+
controlled package sources
+
regular security updates
```

Authentication:

```
Drive password
→ unlock encrypted storage

User password
→ login / sudo / authorization

Fingerprint
→ lock screen / sudo / authorization

FIDO2
→ sudo / authorization
```

SSH:

```
unused
→ keep off

needed
→ Setup → Security → SSHD
→ firewall opened intentionally
→ rate limiting applied
```

Docker:

```
default
→ user not in docker group

docker group
→ effectively passwordless root
```

Code trust:

```
plugin
hook
autostart
AI agent
dependency install script
AUR package
```

are all executable-code trust decisions.

Network inspection:

```
ss -ltnup
```

Local-only development:

```
127.0.0.1
```

preferred unless remote exposure is intentional.

Secret hygiene:

```
never commit secrets
review staged diff
rotate leaked credentials
```

And the most important rule:

> ***Security hardening is not "disable everything." It is understanding which privileges and exposures your workflow actually needs.***

MODULE 22

22

Reproducible Workstation Setup on Omarchy

You now have a heavily customized Omarchy workstation.

The next question is:

About This Module

You now have a heavily customized Omarchy workstation.

The next question is:

> ***Could you rebuild it from scratch without relying on memory?***

If the answer is no, the workstation is not yet truly yours.

A reproducible workstation means that the important parts of your setup are captured in durable, inspectable files and procedures.

That includes:

- personal configuration;
- development runtimes;
- defaults;
- scripts;
- hooks;
- project conventions;
- keybindings;
- monitor/input settings;
- package/tool choices;
- recovery notes.

Current Omarchy already provides a clean foundation for this because your personal configuration lives primarily under:

```
~/.config
```

while Omarchy-owned files live under:

```
/usr/share/omarchy
```

Current official docs explicitly recommend backing up your dotfiles if you customize Omarchy heavily.

This module turns that recommendation into a complete workstation-rebuild strategy.

The goal is:

> ***A new machine should become your machine through a deliberate, documented process —not through weeks of rediscovering old tweaks.***

What You Will Learn

By the end of Module 22, you will understand:

- what "reproducible workstation" actually means;
- which Omarchy files are worth preserving;
- which files should not be committed;
- how to build a dotfiles repository;
- how to separate machine-specific from portable configuration;
- how to preserve your Hyprland setup safely;
- how to preserve Omarchy Shell configuration;
- how to preserve Bash, XCompose, hooks, scripts, and personal themes;
- how to capture default applications;
- how to capture mise-managed runtimes;
- how project-level `mise.toml` improves reproducibility;
- how to document packages and Omarchy-integrated app choices;
- how to bootstrap a new machine in layers;
- how to avoid committing secrets;
- how Git, snapshots, backups, and unattended installs solve different problems;
- how current Omarchy unattended installation can reproduce base machines;
- how to build a practical rebuild checklist;
- how to test whether your setup is actually reproducible.

22.1 — Reproducibility Is Not "Clone My Entire Home Directory"

A naive backup strategy is:

```
copy everything in /home
→ paste onto new machine
```

That can preserve a lot of state.

It can also preserve:

- caches;
- stale sessions;
- machine-specific files;
- old sockets;
- secrets;
- incompatible app state;
- junk.

A reproducible workstation should preserve **intentional configuration**, not accidental runtime debris.

22.2 — Four Recovery/Rebuild Layers

Think of your workstation as four separate layers.

```
1. System layer
2. User configuration layer
3. Development-tool layer
4. Project/data layer
```

Each should have its own recovery strategy.

22.3 — Layer 1: System

This includes:

- Omarchy installation;
- partitioning;
- bootloader;
- kernel;
- system packages;
- full-disk encryption;
- firmware/boot state.

Rebuild tool:

```
Omarchy installer
```

Recovery tool:

```
system snapshot
```

This layer should **not** be reconstructed from a copied root filesystem by default.

22.4 — Layer 2: User Configuration

This includes:

```
~/.config/hypr
~/.config/omarchy
~/.bashrc
~/.XCompose
~/.config/mise
personal scripts
hooks
themes
```

Rebuild tool:

```
dotfiles repository
```

Current official Omarchy docs explicitly treat `~/.config` as the user-owned configuration layer and `/usr/share/omarchy` as Omarchy-owned.

22.5 — Layer 3: Development Tools

This includes:

- Node;
- Python;
- Bun;
- other runtimes;
- CLI tools;
- GitHub CLI;
- project environment managers.

Rebuild tool:

```
mise
+
Omarchy install/pkg workflows
+
documented package list
```

Current Omarchy relies heavily on mise for development environments.

22.6 — Layer 4: Projects and Personal Data

This includes:

- source repositories;
- documents;
- notes;
- local databases;
- assets;
- personal media.

Rebuild/recovery tools:

```
Git remotes
+
backups
+
cloud/storage sync
```

This layer is not replaced by system snapshots or dotfiles.

22.7 — Dotfiles

"Dotfiles" usually refers to personal configuration files whose names often begin with:

```
.
```

Examples:

```
.bashrc
.config/
.XCompose
.gitconfig
```

Current Omarchy documentation explicitly refers to the user's configuration as dotfiles.

22.8 — What Should Go Into Your Dotfiles Repository?

For an Omarchy developer workstation, a strong baseline is:

```
hypr/
omarchy/
mise/
bash/
scripts/
git/
```

or a layout preserving real paths.

Important files include:

```
~/.config/hypr/hyprland.lua
~/.config/hypr/bindings.lua
~/.config/hypr/monitors.lua
~/.config/hypr/input.lua
~/.config/hypr/looknfeel.lua
~/.config/hypr/autostart.lua
~/.config/omarchy/shell.json
~/.bashrc
~/.XCompose
~/.config/mise/config.toml
```

Current Omarchy's official dotfiles documentation highlights these exact user-controlled files.

22.9 — Preserve Hyprland Configuration

Your Hyprland personal layer is one of the most valuable things to preserve.

Current key files:

```
hyprland.lua
bindings.lua
monitors.lua
input.lua
looknfeel.lua
autostart.lua
```

These define:

- monitor configuration;
- input behavior;
- shortcuts;
- autostart;
- appearance/layout overrides.

22.10 — Preserve Omarchy Shell Configuration

Current Omarchy Shell user state lives in:

```
~/.config/omarchy/shell.json
```

Current official docs identify this file as controlling shell behavior including:

- bar;
- widgets;
- idle/screen behavior;
- shell customization.

If you customize the bar or shell state, preserve it intentionally.

22.11 — Preserve Hooks

Current user hooks live under:

```
~/.config/omarchy/hooks
```

If you built:

- post-boot hooks;
- post-update hooks;
- theme hooks;

those are part of your workstation logic.

Version the scripts themselves.

Do not rely on memory.

22.12 — Preserve Personal Menu Extensions

Current Omarchy supports personal menu extensions in:

```
~/.config/omarchy/extensions/omarchy-menu.jsonc
```

Current official docs say this file can add or override Omarchy menu rows.

If you use custom menu items:

```
include this file
```

22.13 — Preserve Bash Customization

Current official Omarchy docs say personal:

- aliases;
- shell functions;
- exports;

belong in:

```
~/.bashrc
```

and that this file is not overwritten by Omarchy updates.

That makes it a natural dotfiles candidate.

22.14 — Preserve XCompose

If you customize:

```
~/.XCompose
```

include it.

This can contain:

- quick-access text;
- emoji sequences;
- name/email shortcuts;

- custom compose sequences.

Do not include sensitive data you would not want in your Git repository.

22.15 — Preserve Personal Scripts

A strong location is:

```
~/.local/bin
```

for your own executables.

Examples:

```
open-project-workspace
backup-dotfiles
record-workstation-state
development helpers
```

If a keybinding depends on a script:

```
the script is part of the keybinding
```

and must be reproducible too.

22.16 — Personal Themes and Backgrounds

If you created custom themes:

```
~/.config/omarchy/themes
```

and custom backgrounds:

```
~/.config/omarchy/backgrounds
```

consider versioning them.

Be careful with:

- large images;
- copyrighted assets;
- huge repositories.

For large wallpapers, a separate asset repository or backup may be more appropriate.

22.17 — Mise Global Configuration

Your global mise configuration usually lives under:

```
~/.config/mise/config.toml
```

This may define global tools such as:

```
Node
Python
Bun
```

The course workstation currently uses:

```
Node 26.8.1
Python 3.13.15
Bun 1.4.2
```

as global development tools.

Those exact versions are not permanent requirements.

The important thing is preserving the intended config.

22.18 — Project `mise.toml` Is More Important Than Global Versions

The workstation can change.

The project should remain reproducible.

Example:

```
global Node
→ 26

project
→ Node 24
```

The project should declare:

```
mise.toml
```

so a new machine can recreate the required runtime.

This is why project-level configuration beats relying on the developer's global environment.

22.19 — Do Not Use `.mise.toml` Unless Your Current Workflow Explicitly Requires It

A real course correction:

Current project setup uses:

```
mise.toml
```

not:

```
.mise.toml
```

This is another useful "verify current behavior" example.

22.20 — Capture Default Applications

Your workstation should document:

```
Browser
Terminal
Editor
Agent
```

Example from the course machine:

```
Browser: Chromium
Terminal: Ghostty
Editor: Neovim
Agent: OMP
```

These can change over time.

A rebuild script/checklist should set or verify them.

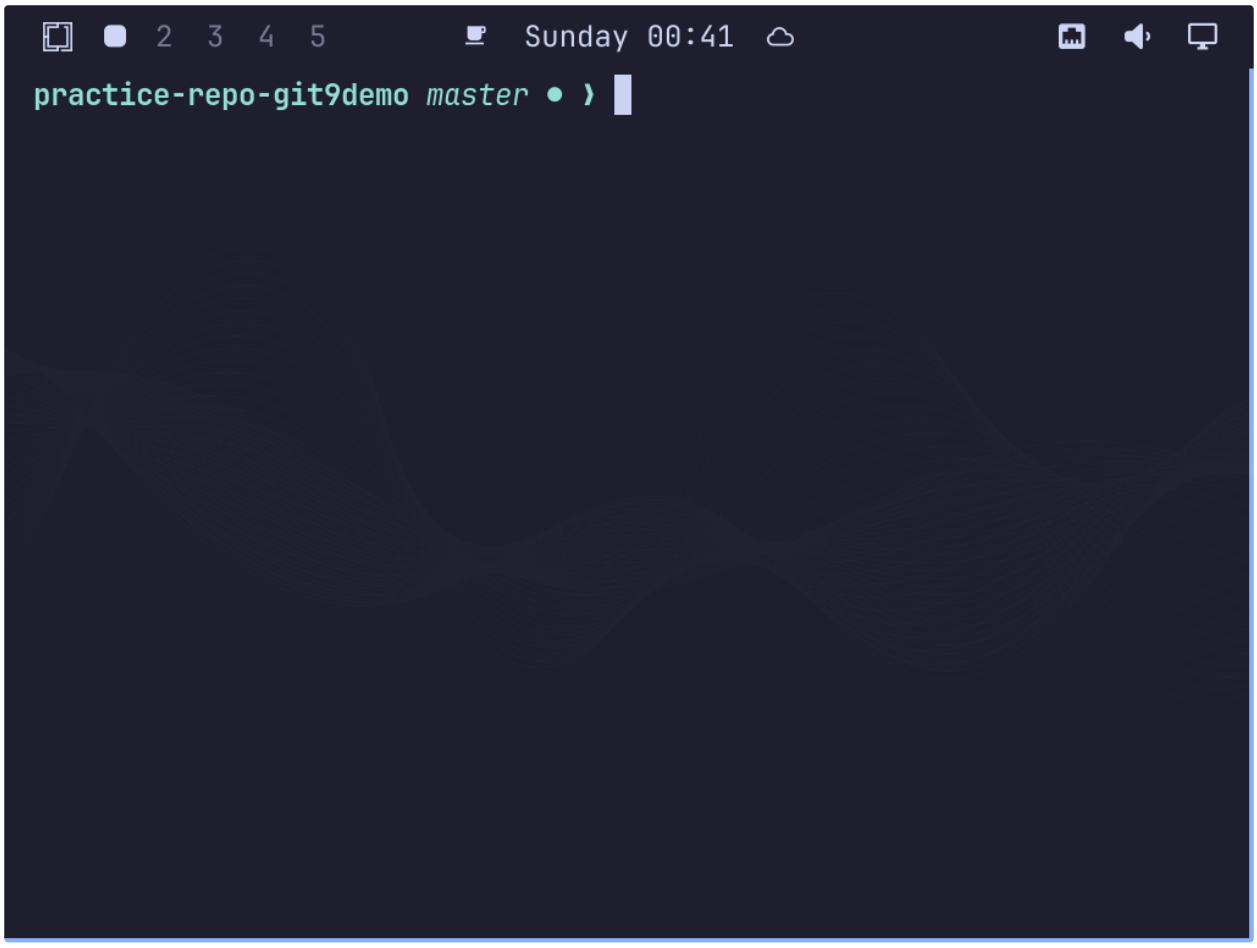
22.21 — Capture Important Omarchy Preferences

Examples:

```
Theme: Catppuccin
Background: Waves
Font: JetBrainsMono Nerd Font
Monitor: DP-2 2560x1440@144
Mouse sensitivity: -0.3
Mouse accel: flat
```

Do not hard-code another person's hardware identifiers.

Document **your machine**.



Screenshot 22.21-A
Catppuccin theme with Waves background applied

22.22 — Portable vs Machine-Specific Config

This distinction is one of the hardest parts of dotfiles.

Portable:

```
editor preference
theme
shell aliases
general keybindings
Git config
mise config
```

Machine-specific:

```
monitor output name
resolution
GPU-specific config
laptop touchpad settings
Bluetooth hardware state
```

Do not assume the same config should apply unchanged everywhere.

22.23 — One Repository Can Still Support Multiple Machines

You can organize machine-specific files explicitly.

Example:

```
dotfiles/
├── common/
├── desktop/
├── laptop/
└── install/
```

Then:

```
common
→ all machines

desktop
→ desktop-only monitor/input settings

laptop
→ laptop-only power/touchpad/display settings
```

This avoids fragile hostname conditionals everywhere.

22.24 — Alternative: Host-Aware Config

You may choose to branch logic based on:

```
hostname
```

Example conceptually:

```
if desktop
→ load desktop monitor config

if laptop
→ load laptop config
```

This can work.

But keep it readable.

If the logic becomes complex, separate files are better.

22.25 — Do Not Commit Generated Runtime State Blindly

Avoid committing things like:

- caches;
- sockets;
- logs;
- lock files;
- session databases;
- browser profiles;
- application caches;
- temporary files.

Version:

```
intent
```

not:

```
everything under ~/.config
```

22.26 — Do Not Commit Secrets

Never put secrets into a public dotfiles repository.

Common risks:

```
API keys
Wi-Fi passwords
SSH private keys
GitHub tokens
cloud credentials
password-manager exports
browser cookies
Tailscale auth keys
```

If a file mixes configuration and secrets:

```
split them
```

or use a private secret-management mechanism.

22.27 — Private Repository Is Not a Secret Vault

A private GitHub repository reduces exposure.

It does not make committed secrets risk-free.

Reasons:

- account compromise;
- accidental repository visibility change;
- collaborator access;
- local clone compromise;
- Git history persistence.

Do not normalize storing production credentials in Git simply because the repository is private.

22.28 — SSH Keys Should Not Be in Dotfiles Git

Do not commit:

```
~/.ssh/id_ed25519
```

or other private keys.

Public keys may be stored safely in appropriate contexts.

Private keys belong in protected credential storage/backup.

22.29 — Build a Dotfiles Repository

One simple approach:

```
mkdir -p ~/Projects/dotfiles
cd ~/Projects/dotfiles
git init
```

Then create a clean structure.

Example:

```
dotfiles/
├── config/
│   ├── hypr/
│   ├── omarchy/
│   └── mise/
├── home/
│   ├── .bashrc
│   └── .XCompose
├── bin/
└── README.md
```

This keeps the repository understandable.

22.30 — Do Not Immediately Symlink Everything

Symlink-based dotfile managers are useful.

But they introduce another abstraction.

For this course, first understand:

```
which files matter
```

Then choose whether to manage them with:

- manual copy;
- symlinks;

- GNU Stow;
- chezmoi;
- another tool.

The method is secondary.

22.31 — Simple Copy-Based Bootstrap

A beginner-friendly bootstrap can:

```
copy repo files into expected locations
```

Example concept:

```
mkdir -p ~/.config/hypr
cp config/hypr/*.lua ~/.config/hypr/
```

This is explicit and easy to debug.

Later, you can replace it with a more advanced dotfiles manager.

22.32 — Symlink-Based Bootstrap

Symlinks make the repository the live source of truth.

Conceptually:

```
~/.config/hypr/bindings.lua
→ ~/Projects/dotfiles/config/hypr/bindings.lua
```

Advantages:

- edit live config in repo;
- easy Git diff;
- no sync step.

Disadvantages:

- mistakes affect live config immediately;
- some tools dislike symlinks;
- path structure becomes more important.

Use deliberately.

22.33 — GNU Stow

GNU Stow is a popular dotfiles approach.

Structure:

```
dotfiles/
└── hypr/
    └── .config/
        └── hypr/
```

Then:

```
stow hypr
```

creates symlinks into the home directory.

This is useful, but not Omarchy-specific.

Do not add complexity unless it helps you.

22.34 — Chezmoi

Chezmoi can manage:

- templates;
- machine differences;
- secrets integration;
- multi-machine dotfiles.

It is powerful.

It is also more complex.

For a single Omarchy workstation:

```
plain Git repo
```

may be enough.

Choose the simplest tool that solves your real problem.

22.35 — GitHub Repository

After initial commit:

```
git add .
git commit -m "chore: bootstrap Omarchy dotfiles"
```

Create a private GitHub repository:

```
gh repo create omarchy-dotfiles --private --source=. --remote=origin --push
```

This gives you:

- remote history;
- off-machine copy;
- easy new-machine access.

Do not confuse this with a full personal-data backup.

22.36 — README as Rebuild Documentation

Your dotfiles repository should include:

```
README.md
```

with:

- supported machines;
- prerequisites;
- install order;
- manual steps;
- defaults;
- optional steps;
- secrets required but not stored;
- verification commands.

This is the human-readable source of truth.

22.37 — Example README Structure

```
## Omarchy Workstation

### Baseline

- Omarchy stable
- Ghostty
- Neovim
- Chromium
- OMP
- Catppuccin
- JetBrainsMono Nerd Font

### Install Order

1. Install Omarchy
2. Run Omarchy update
3. Clone this repo
4. Apply common configs
5. Apply machine-specific configs
6. Install development tools
7. Set defaults
8. Authenticate GitHub
9. Restore secrets manually
10. Verify workstation

### Machine-Specific

#### Desktop
- DP-2
- 2560x1440@144
- scale 1

### Secrets Not Stored Here

- SSH private keys
- API keys
- browser credentials
```

Simple, explicit, useful.

22.38 — Package Reproducibility

There are three major installation layers from Module 6:

```
omarchy pkg
omarchy install
mise
```

Your rebuild documentation should record which layer owns each tool.

Example:

```
system package
→ wev

Omarchy-integrated app
→ Ghostty

mise runtime
→ Node
```

This prevents reinstalling the same tool through conflicting layers.

22.39 — Build an Installation Manifest

Create:

`packages.md`

or:

`workstation.toml`

documenting intentional software.

Example:

```
Omarchy-integrated:
- Ghostty
- Chromium

System:
- wev

Mise global:
- node
- python
- bun

Optional:
- Docker: no
- Go: no
- Rust: no
```

This is more useful than dumping every installed package.

22.40 — Why Not Save the Entire `pacman -Q` List?

Because it contains:

- dependencies;
- base packages;
- transitive packages;
- Omarchy internals.

Reinstalling every package blindly can:

- fight current Omarchy defaults;
- reintroduce obsolete packages;
- preserve old dependency state.

Capture:

```
intentional additions
```

rather than the full package database.

22.41 — Record Manual Hardware Fixes

The course workstation has an important example:

NuPhyIO required a udev rule for the keyboard configuration tool.

That kind of machine-specific fix must be documented.

Otherwise, a rebuild produces:

```
everything works
except one mystery hardware utility
```

Document:

- what was changed;
- why;
- exact file path;
- verification command.

22.42 — Record Monitor Rules

Course workstation:

```
DP-2
2560x1440@144
scale 1
```

That belongs in machine-specific documentation.

On a new GPU/port, the output might become:

```
DP-1
```

so the rebuild process should verify:

```
hyprctl monitors all
```

before blindly applying the old rule.

22.43 — Record Keyboard-Specific Changes

The NuPhy Kick75 required Fn-layer work for a Print-key behavior.

That is not a generic Omarchy config.

Document it as:

```
hardware-specific setup
```

along with:

```
wev
```

verification.

This keeps portable and hardware-specific logic separate.

22.44 — Record Boot Facts, Not Secrets

Useful boot documentation:

```
bootloader: Limine
Secure Boot: disabled
TPM2: available
UKI path: /boot/EFI/Linux/omarchy_linux.efi
Windows dual boot: yes/no
```

Do not store:

- BitLocker recovery key;
- disk passphrase;
- private keys.

Record where secrets are stored safely.

22.45 — Build a Bootstrap Script

Once the manual process is understood, automate it.

Example high-level script:

```
#!/usr/bin/env bash
set -euo pipefail

echo "Applying Omarchy workstation configuration..."

## copy/symlink dotfiles
## install intentional packages
## install mise tools
## set defaults
## verify current state
```

Do not make the first version fully unattended.

Start with visible, reversible steps.

22.46 — Idempotence

A good bootstrap script should ideally be:

```
safe to run more than once
```

That property is called:

```
idempotence
```

Example:

Bad:

```
echo 'export FOO=bar' >> ~/.bashrc
```

every run duplicates the line.

Better:

```
copy a managed file
```

or test before adding.

22.47 — Fail Early

Use:

```
set -euo pipefail
```

carefully in scripts so unexpected errors stop the bootstrap.

But understand what those options do before relying on them.

For beginners, a readable sequence with explicit checks can be safer than clever shell automation.

22.48 — Interactive Bootstrap Is Fine

A rebuild script can ask:

```
Desktop or laptop?
Install Docker?
Set Ghostty as default?
Authenticate GitHub now?
```

Full zero-touch automation is not required for reproducibility.

The important part is:

```
repeatable decisions
```

22.49 — Verification Script

A separate:

```
verify.sh
```

is extremely useful.

It can check:

```
omarchy version
git --version
gh --version
mise --version
hyprctl monitors all
```

and verify expected config files exist.

A rebuild is not complete until it is verified.

22.50 — Verify Defaults

Your verification script can inspect:

```
omarchy default browser
omarchy default terminal
omarchy default editor
omarchy default agent
```

using current supported commands.

Do not assume install order automatically yields the desired defaults.

22.51 — Verify Development Runtimes

Example:

```
mise ls --current
```

For a project:

```
cd ~/Projects/projectinator
mise ls --current
```

This checks:

```
global workstation environment
+
project-specific environment
```

22.52 — Verify GitHub Authentication

Run:

```
gh auth status
```

Authentication is intentionally **not** stored in dotfiles.

The rebuild process should include:

```
authenticate account
```

as a manual secure step.

22.53 — Verify Keybindings

Open:

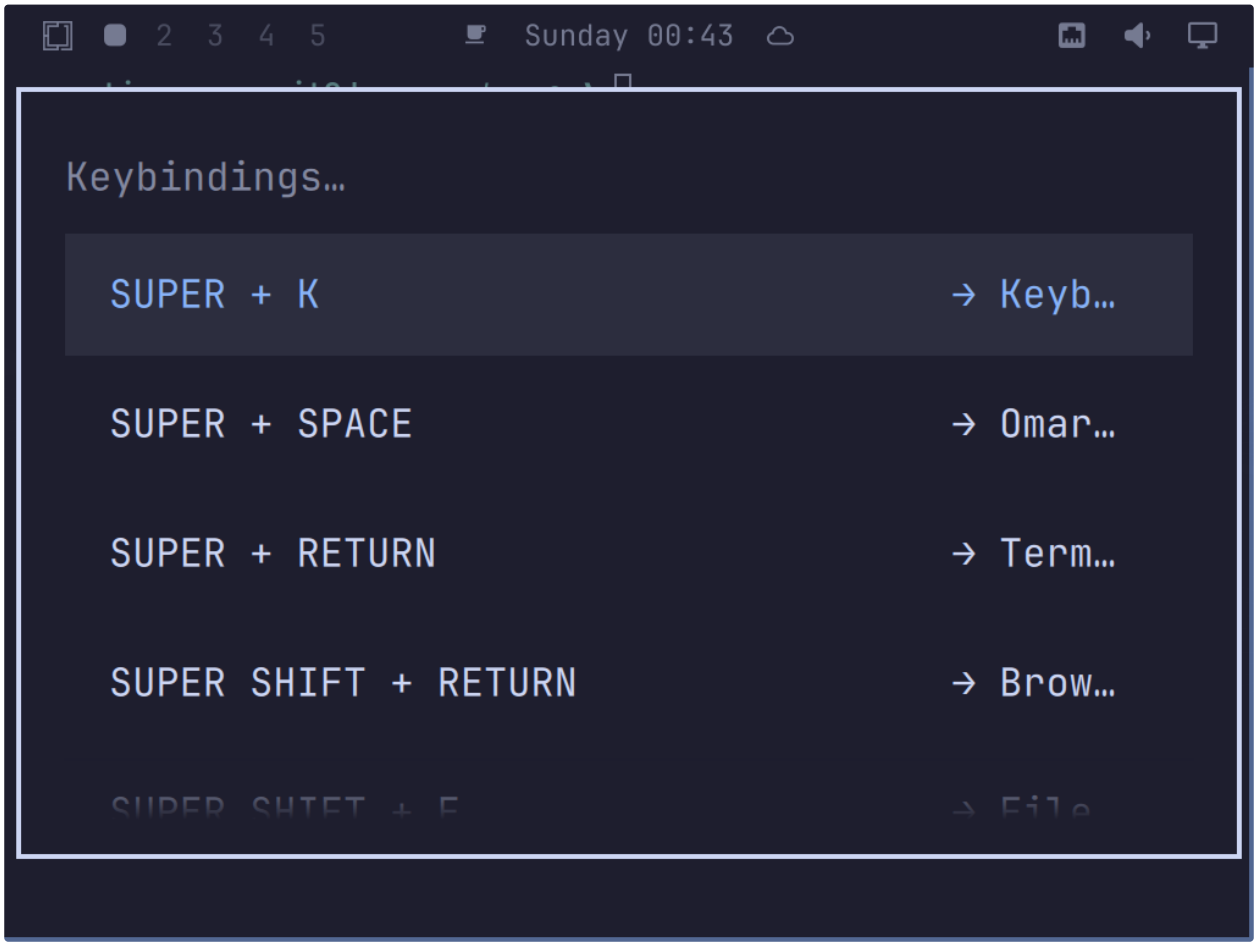
```
Super + K
```

or use current CLI output.

Check critical custom shortcuts:

```
Tmux launcher
project launcher
personal utilities
```

Do not test every shortcut manually unless necessary.



Screenshot 22.53-A
Omarchy keybindings cheatsheet overlay (Super + K)

22.54 — Verify Hardware

Run:

```
hyprctl monitors all
```

```
wev
```

if keyboard mapping matters.

Check:

- audio;
- Wi-Fi;
- Bluetooth;
- GPU;
- input.

This catches machine-specific rebuild gaps.

22.55 — Rebuild Order Matters

Use this order:

```
1. Install Omarchy
2. Update Omarchy
3. Verify boot/network
4. Clone dotfiles
5. Apply user config
6. Install intentional apps/packages
7. Restore mise tools
8. Set defaults
9. Authenticate external services
10. Apply machine-specific hardware config
11. Restore secrets
12. Clone projects
13. Verify
```

This avoids configuring tools that are not installed yet.

22.56 — Why Update Before Applying Old Dotfiles?

Your dotfiles should target:

```
current Omarchy
```

not the install ISO's potentially older state.

Workflow:

```
install
→ update
→ apply user config
```

reduces mismatch between:

- old shipped defaults;
- current user overrides.

22.57 — Current Omarchy Updates Can Change Config Expectations

Omarchy evolves.

A dotfiles repo from six months ago may contain:

- outdated syntax;
- obsolete keys;
- old helper names.

Therefore rebuild should include:

```
hyprctl configerrors
```

and current command discovery.

Do not treat reproducibility as:

```
freeze forever
```

It means:

```
make change explicit and repairable
```

22.58 — Version Your Dotfiles Changes

Before a system/customization experiment:

```
git status
git add .
git commit -m "chore: checkpoint workstation config"
```

Then if something breaks:

```
git diff
```

or:

```
git restore
```

becomes available.

This is stronger than remembering:

```
"I think I changed three files yesterday."
```

22.59 — Commit Small Logical Changes

Good:

```
feat: add tmux launcher binding
```

```
fix: set DP-2 to 144 Hz
```

```
chore: switch default terminal to Ghostty
```

Bad:

```
update everything
```

Small commits make recovery easier.

22.60 — Do Not Auto-Commit Runtime Noise

If some application rewrites config frequently:

```
git status
```

may become noisy.

Decide whether that file is:

- real user intent;
- generated state;
- machine-specific runtime state.

Ignore files that do not belong in reproducible configuration.

22.61 — System Snapshots and Dotfiles Complement Each Other

Current Omarchy snapshots restore root state but not `/home`. Therefore:

```
snapshot
→ system rollback

dotfiles Git
→ personal config rollback
```

Together they cover different failure domains.

Neither replaces real file backups.

22.62 — Personal Data Backup

Your workstation strategy should answer:

```
Where are non-Git personal files backed up?
```

Examples:

- external disk;
- NAS;
- encrypted cloud storage;
- sync service.

This course does not mandate one provider.

It does mandate that the answer exists.

22.63 — New Machine Drill

A workstation is not reproducible until you have tested the rebuild.

Best test:

```
VM
```

or:

```
spare machine
```

Try to recreate your environment using only:

- Omarchy installer;
- dotfiles repo;
- README;
- secure credentials;
- project remotes.

Every time you need to remember something undocumented:

```
add it to the rebuild process
```

22.64 — Rebuild Gaps Are Documentation Bugs

Suppose during rebuild you discover:

```
Ghostty theme wrong
```

Do not just fix it manually.

Ask:

```
Why was this not reproduced?
```

Then update:

- dotfiles;
- bootstrap;
- README;
- verification.

This turns each rebuild into an improvement cycle.

22.65 — Current Omarchy Unattended Installation

Current Omarchy supports unattended ISO installation.

The official docs say that if the installer finds a second drive labeled:

```
cidata
```

with supported configuration files, it can skip the setup wizard and complete installation automatically.

This is advanced but extremely relevant to reproducibility.

22.66 — Unattended Install Use Cases

Useful for:

```
VM templates
Packer builds
Proxmox dev VMs
disposable test machines
lab systems
repeatable workstation imaging
```

For one personal desktop:

```
manual install + dotfiles
```

may still be simpler.

22.67 — Current `cidata` Inputs

Current official unattended-install docs describe files such as:

```
user_configuration.json
user_credentials.json
user_full_name.txt
user_email_address.txt
user_encrypt_installation.txt
authorized_keys
tailscale_authkey
```

Some are optional.

These represent the same basic answers used by the normal installer.

22.68 — `authorized_keys` Has Security Consequences

Current official unattended docs say that when:

```
authorized_keys
```

is supplied, the install:

- places the keys for the user;
- enables SSHD;
- opens the firewall appropriately.

This is deliberate remote-access provisioning.

Do not include SSH keys casually in an image workflow.

22.69 — Tailscale Provisioning

Current unattended install also supports:

```
tailscale_authkey
```

for automatic Tailscale onboarding.

This is powerful for remote/disposable machines.

But the auth key is sensitive.

Treat it as a secret provisioning input, not dotfiles content.

22.70 — `defer-provisioning`

Current official unattended docs support an empty file:

```
defer-provisioning
```

instead of predefining personal credentials.

This is useful for:

```
prepare machine
→ hand it to another owner
→ owner creates user on first boot
```

This is a cleaner approach for reusable images.

22.71 — Unattended Install Is Not a Dotfiles Replacement

Unattended installer answers:

```
How do I install the base machine?
```

Dotfiles answer:

```
How do I make the user environment mine?
```

Mise answers:

```
How do I reproduce development runtimes?
```

Git/backups answer:

```
How do I recover projects/data?
```

These are complementary.

22.72 — Full Reproducibility Stack

A strong Omarchy workstation rebuild architecture:

```
Omarchy ISO / unattended install
        ↓
current Omarchy update
        ↓
dotfiles Git repo
        ↓
intentional packages/apps
        ↓
mise global tools
        ↓
defaults
        ↓
secure authentication
        ↓
machine-specific hardware config
        ↓
project Git clones
        ↓
project mise.toml
        ↓
verification
```

This is the target system.

22.73 — A Stevinator Rebuild Manifest

Create a file:

`WORKSTATION.md`

Suggested sections:

```
Hardware
Boot
Omarchy
Defaults
Theme
Hyprland
Input
Packages
Mise
GitHub
AI Agent
Hooks
Scripts
Security
Secrets
Projects
Verification
Recovery
```

This becomes the canonical rebuild worksheet.

22.74 — Example Workstation Manifest

```
## Workstation

### Hardware
- Desktop
- Radeon RX 6800 XT
- BenQ EX3203R on DP-2
- 2560x1440@144
- NuPhy Kick75

### Defaults
- Browser: Chromium
- Terminal: Ghostty
- Editor: Neovim
- Agent: OMP
```

```
### Theme
- Catppuccin
- JetBrainsMono Nerd Font

### Development
- Node: managed by mise
- Python: managed by mise
- Bun: managed by mise

### Security
- LUKS enabled
- Secure Boot currently disabled
- TPM2 available
- SSHD disabled unless needed

### Recovery
- Limine
- system snapshots
- dotfiles Git
- personal-data backup
```

Do not include passwords or private keys.

22.75 — Practical Exercise: Inventory Your Current Config

Run:

```
ls -la ~/.config/hypr
```

```
ls -la ~/.config/omarchy
```

```
ls -la ~/.config/mise
```

Identify files you intentionally changed.

Write them down.

22.76 — Practical Exercise: Create the Dotfiles Repo

```
mkdir -p ~/Projects/omarchy-dotfiles
cd ~/Projects/omarchy-dotfiles
git init
```

Create:

```
README.md
WORKSTATION.md
config/
home/
bin/
```

Do not copy secrets.

22.77 — Practical Exercise: Copy Hyprland Config

Example:

```
mkdir -p config/hypr
cp ~/.config/hypr/hyprland.lua config/hypr/
cp ~/.config/hypr/bindings.lua config/hypr/
cp ~/.config/hypr/monitors.lua config/hypr/
cp ~/.config/hypr/input.lua config/hypr/
cp ~/.config/hypr/looknfeel.lua config/hypr/
cp ~/.config/hypr/autostart.lua config/hypr/
```

Then:

```
git status
```

Review before adding.

22.78 — Practical Exercise: Copy Omarchy User Config

Copy only intentional user configuration.

Examples:

```
mkdir -p config/omarchy
cp ~/.config/omarchy/shell.json config/omarchy/
```

If you use:

- hooks;
- extensions;
- custom themes;

copy those intentionally too.

Do not blindly run:

```
cp -r ~/.config/omarchy/* ...
```

without reviewing what is there.

22.79 — Practical Exercise: Copy Bash/Mise

Example:

```
cp ~/.bashrc home/
```

```
mkdir -p config/mise
cp ~/.config/mise/config.toml config/mise/
```

If:

```
~/.XCompose
```

contains no sensitive personal values you do not want in Git:

```
cp ~/.XCompose home/
```

Otherwise sanitize it first.

22.80 — Practical Exercise: Add `.gitignore`

Create repository-level:

```
.gitignore
```

Include anything that should never enter the repo.

Example categories:

```
secrets
temporary files
editor swap files
generated caches
large private assets
```

Do not assume `.gitignore` protects files already committed.

22.81 — Practical Exercise: First Commit

Review:

```
git status
```

```
git diff --no-index /dev/null README.md
```

Then:

```
git add .
git diff --staged
```

Inspect carefully for secrets.

Commit:

```
git commit -m "chore: bootstrap Omarchy workstation config"
```

22.82 — Practical Exercise: Push Private Repo

Using your GitHub account:

```
gh repo create omarchy-dotfiles --private --source=. --remote=origin --push
```

If that repository name already exists, choose another.

Verify:

```
git remote -v
```

22.83 — Practical Exercise: Create `WORKSTATION.md`

Document:

```
current Omarchy version
hardware
monitor
defaults
theme
font
mise tools
custom bindings
hooks
security state
bootloader
recovery
```

Do not record secrets.

22.84 — Practical Exercise: Create `bootstrap.sh`

Start simple.

Example structure:

```
#!/usr/bin/env bash
set -e

echo "Omarchy workstation bootstrap"

mkdir -p "$HOME/.config/hypr"
mkdir -p "$HOME/.config/omarchy"
mkdir -p "$HOME/.config/mise"

## copy configs here
## install tools here
## set defaults here
```

Do not automate authentication secrets.

22.85 — Practical Exercise: Create `verify.sh`

Example:

```
#!/usr/bin/env bash

echo "Omarchy:"
omarchy version

echo
echo "Git:"
git --version

echo
echo "GitHub CLI:"
gh --version

echo
echo "Mise:"
mise --version

echo
echo "Monitors:"
hyprctl monitors all

echo
echo "Current defaults:"
omarchy default browser
omarchy default terminal
omarchy default agent
```

Adapt to the current installed CLI.

22.86 — Practical Exercise: Rebuild in a VM

Optional but strongly recommended.

Install Omarchy in a VM.

Then try:

```
1. update
2. clone dotfiles
3. run bootstrap
4. authenticate GitHub
5. run verify
```

Note every missing step.

Update documentation.

This is the real reproducibility test.

22.87 — Common Reproducibility Mistakes

Mistake 1 — Backing Up All of `~/.config` Blindly

Preserve intentional config, not every runtime artifact.

Mistake 2 — Forgetting Scripts Referenced by Keybindings

A config file can depend on external code.

Mistake 3 — Committing Secrets to a Private Repo

Private Git is not a secret vault.

Mistake 4 — Treating System Snapshots as Personal Backups

Snapshots do not restore `/home`.

Mistake 5 — Capturing Every Installed Package

Record intentional additions, not the whole dependency graph.

Mistake 6 — Applying Desktop Monitor Rules to a Laptop

Separate machine-specific config.

Mistake 7 — Automating Before Understanding the Manual Process

Automation should encode a known-good process.

Mistake 8 — Never Testing the Rebuild

Untested reproducibility is only a theory.

Mistake 9 — Forgetting External Authentication

GitHub, SSH, cloud accounts, and AI credentials require secure reprovisioning.

Mistake 10 — Letting the Dotfiles Repo Drift Away From the Live Machine

Commit intentional changes as they happen.

22.88 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Architecture

- [ ] I understand the system, user-config, dev-tool, and project/data layers.
- [ ] I understand which recovery mechanism belongs to each layer.
- [ ] I understand dotfiles are not full backups.

Dotfiles

- [ ] I know which Omarchy/Hyprland files matter.
- [ ] I preserve shell, Bash, XCompose, hooks, and scripts intentionally.
- [ ] I understand portable vs machine-specific config.
- [ ] I avoid committing runtime noise.

Development

- [ ] I preserve global mise configuration.
- [ ] I understand project `mise.toml` matters more than global versions.
- [ ] I document intentional package/tool ownership.

Security

- [ ] I do not commit secrets.
- [ ] I understand private Git is not a secret manager.
- [ ] I do not commit SSH private keys.
- [ ] I keep secure authentication as a separate rebuild step.

Automation

- [ ] I understand bootstrap scripts should encode a known-good manual process.
- [ ] I understand idempotence at a high level.
- [ ] I have a verification step.
- [ ] I know how to test rebuilds in a VM/spare machine.

Unattended Installation

- [ ] I understand what `cidata` does.

- [ ] I understand unattended installation reproduces the base system, not the whole user environment.
- [ ] I understand `authorized_keys` enables remote-access provisioning.
- [ ] I understand Tailscale auth keys are secrets.
- [ ] I understand `defer-provisioning`.

22.89 — What You Can Now Do

After completing Module 22, you can now:

- turn your Omarchy customization into a reproducible workstation;
- version your personal configuration;
- separate portable and machine-specific settings;
- preserve Hyprland and Omarchy Shell customizations safely;
- reproduce development runtimes with mise;
- document intentional applications and packages;
- avoid secret leakage in dotfiles;
- rebuild defaults deliberately;
- create bootstrap and verification scripts;
- use GitHub as a remote dotfiles history;
- understand how unattended Omarchy installation fits into larger deployment workflows;
- test your setup on a fresh machine instead of trusting memory.

Most importantly:

> *You no longer own one carefully tweaked machine. You own a reproducible workstation design.*

Module 22 Summary

Reproducibility layers:

```
System
→ Omarchy installer / snapshots

User config
→ dotfiles Git

Development tools
→ mise + documented install layers

Projects/data
→ Git remotes + backups
```

Important current Omarchy user config:

```
~/.config/hypr/
~/.config/omarchy/
~/.config/mise/
~/.bashrc
~/.XCompose
~/.local/bin
```

Important principle:

```
/usr/share/omarchy
→ Omarchy-owned

~/.config
→ user-owned
```

Project runtimes:

```
mise.toml
```

Rebuild order:

```
install
↓
update
↓
clone dotfiles
↓
apply config
↓
install tools
↓
restore mise
↓
set defaults
↓
authenticate
↓
apply machine-specific hardware config
↓
clone projects
↓
verify
```

Advanced base-system automation:

```
Omarchy ISO
+
cidata
→ unattended install
```

Critical rule:

```
dotfiles
≠
secrets
≠
personal data backup
≠
system snapshot
```

And the most important lesson:

> ***If rebuilding the workstation requires memory, it is not yet reproducible.***

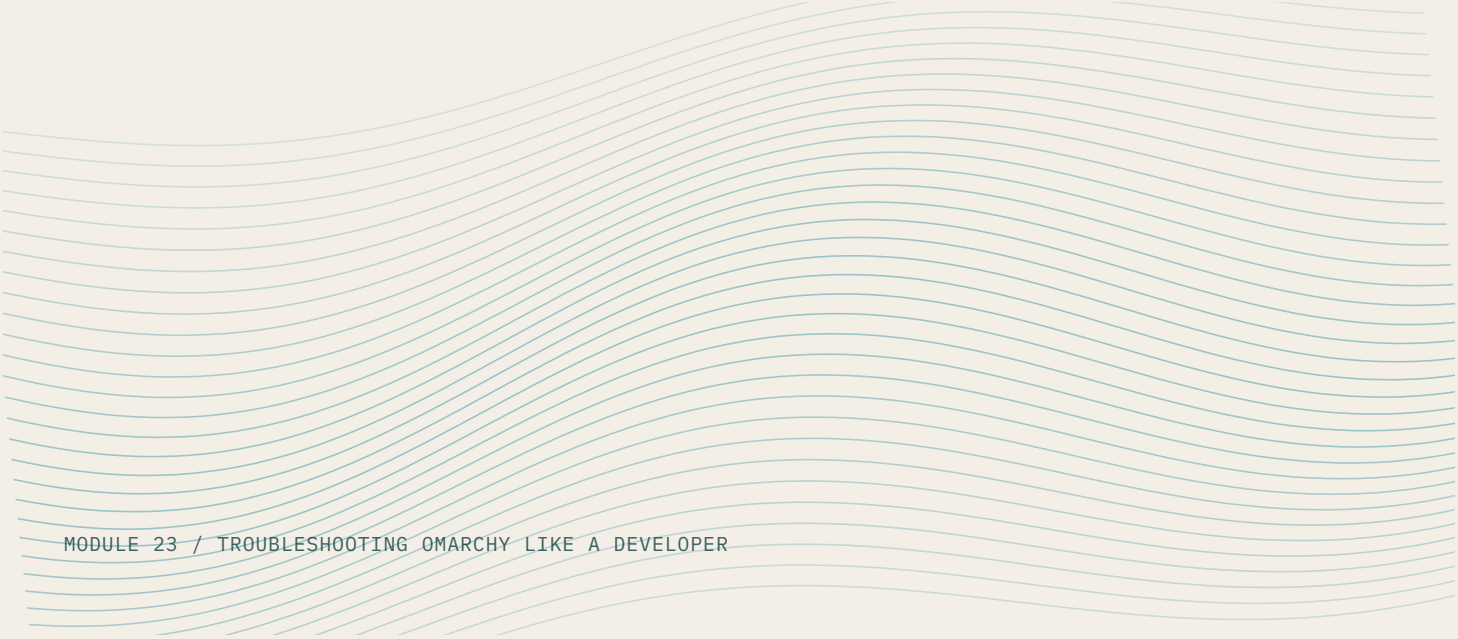
MODULE 23

23

Troubleshooting Omarchy Like a Developer

This is the troubleshooting module.

Everything you have learned so far now comes together here.



About This Module

This is the troubleshooting module.

Everything you have learned so far now comes together here.

A weak troubleshooting process looks like:

```
something feels wrong
→ guess
→ change three unrelated settings
→ restart random services
→ reboot
→ maybe reinstall
```

A strong troubleshooting process looks like:

```
observe
→ classify
→ reproduce
→ inspect evidence
→ isolate the layer
→ make one change
→ verify
→ document
```

This is especially important on Omarchy because the workstation spans several layers:

```
UEFI / Limine / UKI
systemd / services
Hyprland
Omarchy Shell
terminal / Tmux
mise
Git
AI agents
network/audio/Bluetooth
user config
```

Many problems that *look* similar belong to completely different layers.

For example:

```
"terminal shortcut does nothing"
```

could be:

- keyboard firmware;
- `wev` input mismatch;
- Hyprland binding collision;
- wrong terminal default;
- command missing from PATH;
- a stale Tmux/session state issue.

Likewise:

```
"Omarchy broke after an update"
```

could mean:

- package update failure;
- pending migration;
- user config mismatch;
- one failed shell plugin;
- kernel/boot problem;
- hardware service glitch.

The goal of this module is:

> ***Teach a repeatable debugging method that works across the whole Omarchy workstation.***

What You Will Learn

By the end of Module 23, you will be able to:

- troubleshoot from evidence instead of guesses;
- classify a failure by layer;
- use Omarchy's current troubleshooting/recovery tools correctly;
- inspect failed services;
- inspect logs without overreacting;
- use `hyprctl configerrors`;
- use `wev`;
- inspect window classes and monitor state;

- diagnose terminal/Tmux issues;
- diagnose mise/runtime issues;
- diagnose Git/GitHub issues;
- diagnose AI-agent launch/config issues;
- diagnose audio/Bluetooth/Wi-Fi issues;
- diagnose update/migration problems;
- distinguish user-config failure from system failure;
- know when to restart, refresh, rollback, or reinstall;
- use `omarchy-debug`;
- preserve useful evidence before changing the system;
- avoid common Omarchy-specific mistakes documented throughout this course.

23.1 — The Troubleshooting Mindset

The most important rule:

> ***Do not solve the symptom before you understand the layer.***

Example:

```
browser opens wrong
```

Could be:

```
Omarchy default
XDG handler
application-specific override
hard-coded launcher
```

If you edit a keybinding before checking the default handler, you may "fix" the wrong layer.

23.2 — The Developer Debug Loop

Use this sequence:

```
1. What exactly is wrong?
2. Can I reproduce it?
3. What changed recently?
4. Which layer owns this behavior?
5. What evidence can confirm that?
6. What is the smallest reversible test?
7. Did the result match the hypothesis?
8. What did I learn?
```

This is normal software debugging applied to the workstation itself.

23.3 — Define the Symptom Precisely

Bad:

```
"Omarchy is broken."
```

Better:

```
"Super+Return produces no terminal window after switching the default terminal."
```

Even better:

```
"Super+Return does nothing, but `ghostty` launches normally from an existing terminal,
and `hyprctl configerrors` is empty."
```

The third version already eliminates several possibilities.

23.4 — Ask: What Changed?

Before debugging, identify recent changes.

Examples:

```
Omarchy update
kernel update
new plugin
new Hyprland binding
new monitor
new keyboard firmware
new mise.toml
new AI-agent config
new browser default
```

Recent change does not always equal root cause.

But it is a strong clue.

23.5 — One Change at a Time

Do not do this:

```
edit bindings.lua
change terminal default
restart shell
delete config
reinstall terminal
```

Then test.

You have destroyed the ability to learn which action mattered.

Instead:

```
one hypothesis
→ one change
→ one test
```

23.6 — Capture Before Repair

Useful evidence can include:

`omarchy version` `git diff` `hyprctl configerrors` `systemctl --failed`

`systemctl --user --failed` `journalctl -b` `journalctl --user -b`

`omarchy-debug`

Use only what is relevant.

Do not produce huge log dumps by default.

23.7 — The Layer Model

Use this troubleshooting map:

```
Layer 1 — Firmware / Boot
Layer 2 — Kernel / systemd
Layer 3 — Hardware services
Layer 4 — Hyprland
Layer 5 — Omarchy Shell
Layer 6 — Terminal / Tmux
Layer 7 — Development environment
Layer 8 — Application / project
Layer 9 — Network / remote service
```

Each layer has different tools.

23.8 — Layer 1: Firmware / Boot

Typical symptoms:

```
wrong OS boots
Limine missing
no boot entry
Secure Boot rejects image
kernel never starts
```

Useful evidence:

`bootctl status` `efibootmgr` `ls -lh /boot/EFI/Linux`

Recovery may involve:

- firmware boot menu;

- Limine;
- bootable snapshot;
- current official boot documentation.

Do not troubleshoot this layer with Hyprland commands.

23.9 — Layer 2: Kernel / systemd

Typical symptoms:

```
boot reaches Linux but service fails
TTY works but graphical session does not
user service repeatedly crashes
```

Useful:

```
systemctl --failed
```

```
systemctl --user --failed
```

```
journalctl -b
```

```
journalctl --user -b
```

If TTY works:

```
firmware + bootloader + kernel progressed significantly
```

Focus later in the boot chain.

23.10 — Layer 3: Hardware Services

Typical symptoms:

```
audio vanished
Bluetooth stopped reconnecting
Wi-Fi disappeared
trackpad died after suspend
```

Current official Omarchy troubleshooting guidance explicitly recommends restarting the offending subsystem before rebooting.

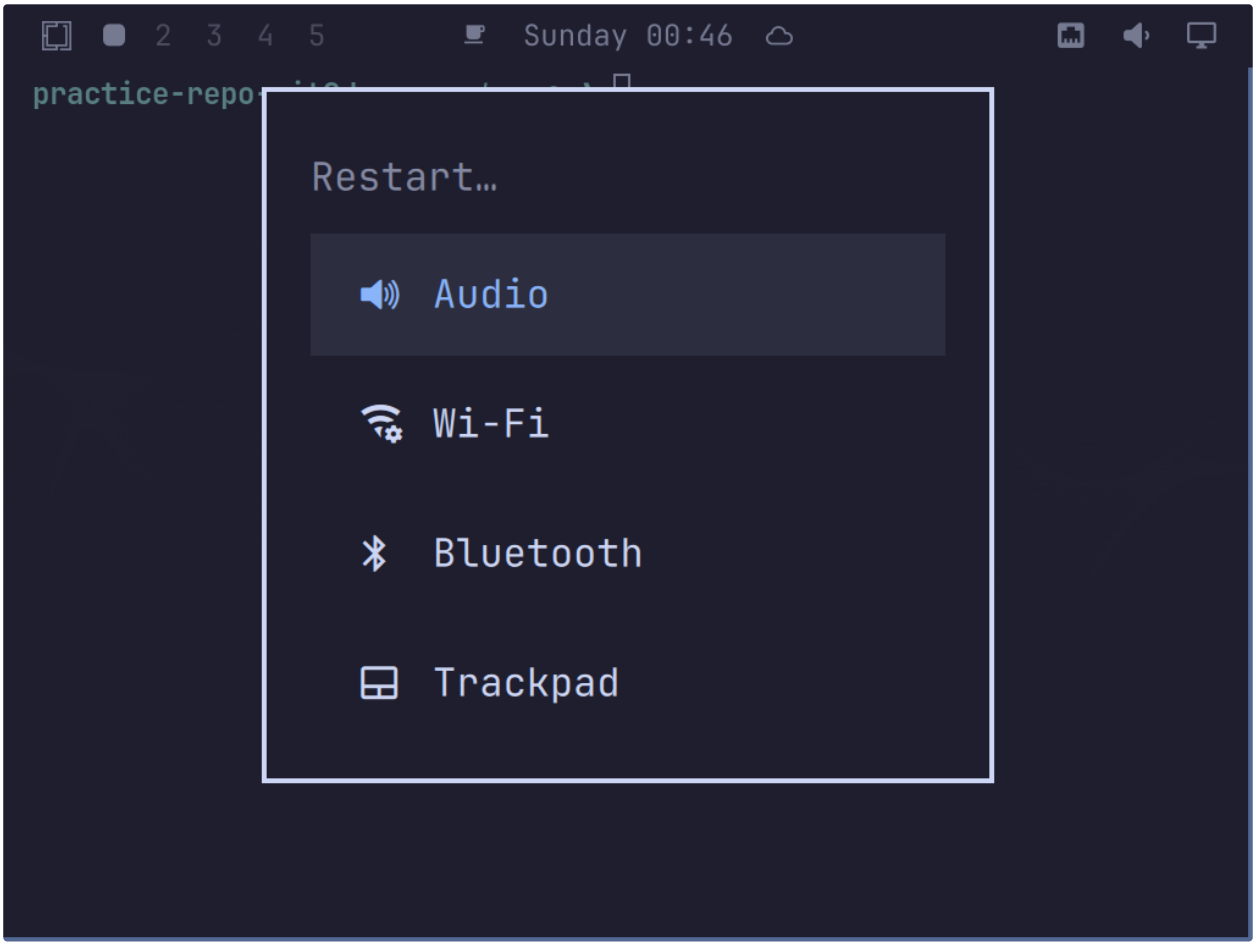
Use:

```
Super + Space
→ Update
→ Hardware
```

for:

- Wi-Fi;
- Bluetooth;
- Audio;
- Trackpad.

This is a high-value first action.



Screenshot 23.10-A
Omarchy Hardware panel with Wi-Fi, Bluetooth, Audio, and Trackpad restart options

23.11 — Layer 4: Hyprland

Typical symptoms:

```
keybinding fails
window rule wrong
monitor mode wrong
input behavior wrong
layout strange
```

Core tools:

```
hyprctl configerrors
```

```
hyprctl monitors all
```

```
hyprctl activewindow
```

```
hyprctl workspaces
```

```
wev
```

These should be second nature by now.

23.12 — Layer 5: Omarchy Shell

Typical symptoms:

```
bar missing
panel does not open
plugin stale
menu not responding
notification layer broken
```

Check:

```
omarchy-shell shell ping
```

Then restart the shell if appropriate:

```
omarchy-restart-shell
```

or the current routed restart command.

Do not restart the entire graphical session first.

23.13 — Layer 6: Terminal / Tmux

Typical symptoms:

```
wrong terminal opens
Tmux session in wrong directory
old config still active
pane layout unexpected
session persists after closing terminal
```

Remember:

```
terminal
≠
shell
≠
Tmux
```

Persistent Tmux state is often the reason "new settings" appear ignored.

23.14 — Layer 7: Development Environment

Typical symptoms:

```
wrong Node version
wrong Python version
command exists globally but not project
mise configuration ignored
PATH confusion
```

Core tools:

```
mise ls --current
```

```
mise config ls
```

```
type COMMAND
```

```
command -v COMMAND
```

```
pwd
```

Project environment problems are often simpler than they look.

23.15 — Layer 8: Application / Project

Typical symptoms:

```
tests fail
app crashes
Git diff strange
browser app behaves differently from system default
AI agent changed too much
```

Use project evidence:

```
git status
git diff
git log --oneline
```

and application-specific logs.

Do not blame Omarchy automatically.

23.16 — Layer 9: Network / Remote Service

Typical symptoms:

```
internet unavailable
GitHub push fails
API unreachable
remote server offline
```

First distinguish:

```
local network failure
vs
DNS
vs
remote service outage
vs
credentials
```

Use:

```
omarchy network status --verbose
```

and appropriate network tests.

23.17 — Current Official Troubleshooting Escalation

Current Omarchy troubleshooting guidance for a broken update is:

```
1. rollback the system to the previous version
2. if that fails, run `omarchy-debug`
3. if all else fails, reinstall default configs/packages
```

The current broad reinstall helper is:

```
omarchy-reinstall
```

This is a last resort.

Do not skip straight to step 3.

23.18 — `omarchy-debug`

Use:

```
omarchy-debug
```

when:

- the problem is system-wide;
- basic diagnosis is inconclusive;
- you need to share structured evidence;
- a rollback did not solve it.

This is better than posting random screenshots of terminal errors.

23.19 — Read Debug Output Before Sharing

Before posting diagnostics publicly:

check for:

- usernames;
- hostnames;
- network information;
- private repository paths;

- credentials;
- tokens.

Troubleshooting output can contain sensitive context.

23.20 — Failed Services

Run:

```
systemctl --failed
```

and:

```
systemctl --user --failed
```

A clean list is nice.

A non-empty list is not automatically an emergency.

Ask:

```
Is this failed service related to the symptom?
Did it fail once during boot and recover?
Is it an optional service?
```

Interpretation matters.

23.21 — Log Triage

Use a narrow time/context window.

Better:

```
journalctl --user -u SOME_SERVICE -b
```

than:

```
journalctl
```

for everything.

Useful patterns:

```
current boot
specific unit
recent timestamp
repeatable error
```

Noise is not evidence.

23.22 — Crash vs Config Failure

A process crash may produce:

```
coredump
```

A config error may produce:

```
parse/load error
```

These need different tools.

Config:

```
hyprctl configerrors
```

Crash:

```
systemd-coredump / journal evidence
```

Current Omarchy can also surface crash diagnostics to the configured AI agent.

23.23 — AI-Assisted Troubleshooting

A useful prompt:

```
Do not modify anything.

Inspect the current system state relevant to this problem.

I want:
1. likely layer
2. supporting evidence
3. one smallest next diagnostic step
4. no changes yet
```

This keeps the AI in analysis mode.

23.24 — Do Not Let AI Troubleshooting Become AI Chaos

Bad:

```
"Fix Omarchy."
```

Better:

```
"Inspect why Super+Return no longer launches the current default terminal. Do not edit files."
```

Scope is security and quality control.

23.25 — Troubleshooting Keyboard Shortcuts

If a shortcut fails:

```
1. Does the physical key produce the expected event?
2. Is the combination already bound?
3. Did bindings.lua load?
4. Is the target command available?
5. Does the target work manually?
```

Tools:

```
wev
```

```
Super + K
```

```
hyprctl configerrors
```

```
type COMMAND
```

This is the complete chain.

23.26 — Common Mistake: Wrong Keybinding Assumptions

During the course, several shortcut assumptions were wrong before checking the real config.

Examples included:

- incorrect resize shortcuts;
- initially wrong clipboard shortcut;
- guessed scratchpad bindings;
- unverified equal-split reset.

Lesson:

> ***Never invent Omarchy hotkeys from memory. Check `Super + K` or the current official hotkey docs.***

23.27 — Current Correct Shortcut Discovery

Use:

```
Super + K
```

and current CLI/manual.

This matters because Omarchy evolves quickly.

A shortcut from an old version can be completely valid historically and wrong today.

23.28 — Troubleshooting Clipboard

If clipboard manager behavior differs from expectation:

verify:

```
Super + V
→ paste

Super + Ctrl + V
→ clipboard manager
```

on the course workstation/current documented baseline.

Do not confuse direct paste with history UI.

23.29 — Troubleshooting Scratchpad

Current course workstation baseline:

```
Super + S
→ toggle scratchpad

Super + Alt + S
→ send to scratchpad
```

Verify with:

```
Super + K
```

before assuming current-release consistency.

23.30 — Troubleshooting Monitor Problems

Run:

```
hyprctl monitors all
```

Confirm:

- output name;
- resolution;
- refresh;
- scale;
- position.

Then inspect:

```
bat ~/.config/hypr/monitors.lua
```

Common real issue:

```
monitor supports 144 Hz
but system runs 60 Hz
```

A working image does not prove correct mode.

23.31 — Apps Too Large

Current official Omarchy troubleshooting documentation specifically addresses oversized apps.

Current guidance says Omarchy assumes a high-resolution scaling baseline and points to:

```
~/.config/hypr/monitors.lua
```

for GDK scaling adjustment when using a 1x display.

The lesson is:

```
one scale layer can affect some GUI apps differently from Hyprland monitor scale
```

Do not randomly resize every application individually before checking the system scaling model.

23.32 — Caps Lock "Not Working"

Current official Omarchy troubleshooting documentation explains that Caps Lock may be assigned to XCompose behavior by default.

If you want normal Caps Lock, inspect:

```
~/.config/hypr/input.lua
```

Current course machine chose to clear the inherited keyboard options with:

```
kb_options = ""
```

because normal Caps Lock was preferred.

Your setup may differ.

23.33 — `wev` Is the Hardware Truth

If:

```
Fn + key
```

does not trigger expected behavior:

```
wev
```

tells you what the compositor actually receives.

This was important with the NuPhy compact keyboard in the course workstation.

If the keyboard firmware sends the wrong event, Hyprland cannot bind the label printed on the key.

23.34 — Troubleshooting Window Rules

Focus the target app:

```
hyprctl activewindow
```

Inspect:

```
class
title
workspace
```

Do not guess from app name.

Real course example:

```
Ghostty class
→ com.mitchellh.ghostty
```

23.35 — Troubleshooting Hyprland Config

First:

```
hyprctl configerrors
```

If there is an error:

```
fix that error before doing anything else
```

Then inspect recent edits.

Do not reset all Hyprland config for one syntax mistake.

23.36 — `omarchy refresh hyprland` Warning

Current official source shows:

```
omarchy refresh hyprland
```

overwrites the user's current Hyprland Lua files including:

```
autostart.lua
bindings.lua
input.lua
looknfeel.lua
hyprland.lua
monitors.lua
```

Use it only as intentional broad recovery.

23.37 — `omarchy reinstall configs` Warning

Current source labels:

```
omarchy-reinstall-configs
```

as destructive.

It replays shipped user defaults over the home directory config surface.

This is not a normal first troubleshooting step.

23.38 — Troubleshooting Terminal Defaults

If the wrong terminal opens:

```
omarchy default terminal
```

Then:

```
cat ~/.config/xdg-terminals.list
```

Then:

```
xdg-terminal-exec --print-id
```

if available.

Then test:

```
Super + Return
```

Do not hard-code Ghostty into unrelated bindings to hide a default-resolution problem.

23.39 — Real Course Mistake: Assumed Foot

Earlier, Foot was assumed as the active terminal.

The actual workstation used:

```
Ghostty
```

Lesson:

```
official default
≠
current user's selected default
```

Inspect the machine.

23.40 — Troubleshooting Tmux

If the terminal opens in the correct directory but Tmux shows an old directory:

```
existing Tmux session preserved state
```

Check:

```
tmux ls
```

Persistent sessions retain pane working directories.

Create/restart the session if you intentionally want new initial state.

23.41 — Do Not Confuse Tmux Prefix Documentation

Tmux configuration can differ from generic Tmux tutorials.

Earlier course work used generic:

```
Ctrl+B
```

assumptions.

Current official Omarchy documentation has described a custom primary prefix such as:

```
Ctrl+Space
```

with compatibility behavior.

Always inspect the current Omarchy Tmux setup before teaching shortcuts.

23.42 — Tmux Layout Helpers

Current Omarchy documentation includes helpers such as:

```
tdl
tds
tdlm
tsl
```

If one fails:

```
type tdl
```

```
type tds
```

Then inspect required dependency commands.

Do not assume the layout helper itself is broken.

23.43 — Troubleshooting `tds`

Current docs describe `tds` as depending on:

```
hunk diff --watch
```

If one pane says:

```
command not found
```

check:

```
type hunk
```

The issue may simply be a missing dependency.

23.44 — Real Course Mistake: Installing Hunk Through the Wrong Layer

Earlier assumption:

```
omarchy pkg add hunk
```

was wrong for the intended current setup.

The course resolved Hunk through:

```
mise
```

Lesson:

```
missing command
→ first determine the correct installation layer
```

Do not reflexively install every CLI with pacman.

23.45 — Troubleshooting mise

Start:

```
mise --version
```

Then:

```
mise ls --current
```

Then:

```
mise config ls
```

Then inspect:

```
mise.toml
```

for the project.

Common problem:

```
wrong file name
```

23.46 — Real Course Mistake: `.mise.toml`

Earlier we incorrectly used:

```
.mise.toml
```

The current course/project workflow uses:

```
mise.toml
```

Lesson:

> *Do not let old tool conventions override current documented behavior.*

23.47 — Project Runtime vs System Runtime

Example:

```
system Python
→ 3.14

mise global Python
→ 3.13

project Python
→ another version
```

This is not necessarily a conflict.

Ask:

```
which layer is this command running in?
```

Use:

```
which python
python --version
mise ls --current
```

23.48 — Troubleshooting PATH

If:

```
command exists
but shell says not found
```

check:

```
type COMMAND
```

```
command -v COMMAND
```

```
echo "$PATH"
```

Then ask:

```
Is it installed?
Is the runtime activated?
Is this interactive shell different from Hyprland launch environment?
```

23.49 — Troubleshooting Git

Start with:

```
git status
```

Then:

```
git remote -v
```

Then:

```
git branch --show-current
```

Then:

```
git log --oneline -5
```

This answers:

```
where am I?
what branch?
what changes?
what remote?
```

before blaming GitHub.

23.50 — Troubleshooting GitHub CLI

Run:

```
gh auth status
```

Then:

```
git remote -v
```

Remember:

```
Git authentication
GitHub CLI authentication
remote URL method
```

can interact but are not identical concepts.

23.51 — HTTPS vs SSH Git Remotes

If push fails:

```
git remote -v
```

Check whether the remote uses:

```
https://
```

or:

```
git@github.com:
```

Troubleshoot the authentication method actually in use.

Do not debug SSH keys for an HTTPS remote.

23.52 — Troubleshooting AI Agent Launch

Current Omarchy agents can be lazy-installed/managed.

If the default agent fails:

```
omarchy default agent
```

Then:

```
type AGENT_COMMAND
```

Then:

```
command -v AGENT_COMMAND
```

Then launch it directly.

This distinguishes:

```
default mapping problem
vs
agent installation problem
vs
agent runtime problem
```

23.53 — Real Course Mistake: `omarchy agent --pick`

Earlier expectation:

```
chooser opens
```

did not match the real workstation; it launched OMP.

Lesson:

```
CLI name that sounds obvious
≠
guaranteed semantics
```

Use:

```
omarchy agent --help
```

and current official docs.

23.54 — Troubleshooting Agent Permissions

If an agent behaves more aggressively than expected:

verify the actual agent's permission mode.

Real course correction for OMP:

```
always-ask
→ reads auto, writes+exec prompt

write
→ reads+writes auto, exec prompt

yolo
→ everything auto
```

Earlier assumptions about `write` were wrong.

Lesson:

> ***Security-critical permission semantics must be verified from the actual tool, not guessed from a mode name.***

23.55 — Troubleshooting Agent Working Directory

Current Omarchy AI docs have documented a home-launch fallback into:

```
~/Work
```

when launching from `$HOME`.

But your project workflow may use:

```
~/Projects
```

Always verify:

```
pwd
```

before asking an agent to modify a project.

Wrong working directory can become a serious mistake.

23.56 — AI Agent Changed Too Much

First:

```
git status
```

Then:

```
git diff
```

Do not ask the agent:

```
"What did you change?"
```

before inspecting Git.

The diff is the ground truth.

23.57 — Troubleshooting Audio

Current official flow:

```
1. check master mute
2. check output device
3. check per-app volume
4. switch output
5. restart audio
6. deeper diagnostics
```

Current official troubleshooting docs specifically say external speakers are often simply not selected as the primary output.

23.58 — Laptop Speakers Sound Wrong

Current official current command:

```
omarchy audio tuning status
```

If tuning is active and you intentionally want raw output:

```
omarchy audio tuning off
```

Do not turn it off merely because you found the command.

23.59 — Troubleshooting Bluetooth

Flow:

```
radio on?
↓
device paired?
↓
disconnect/reconnect
↓
restart Bluetooth
↓
forget/re-pair only if necessary
```

Avoid re-pairing as the first response.

23.60 — Troubleshooting Wi-Fi

Start:

```
omarchy network status --verbose
```

Then inspect:

- SSID;
- signal;
- interface;
- connectivity.

Ask:

```
radio problem?
association problem?
DNS?
remote site?
```

Do not change DNS because signal is weak.

23.61 — Troubleshooting Update Problems

Current supported update command:

```
omarchy update
```

If it fails:

```
keep terminal open
```

Inspect:

```
/tmp/omarchy-update.log
```

Current update architecture records the transcript there.

23.62 — Pending Migrations

Current migration state:

```
~/.local/state/omarchy/migrations/
```

Current diagnostic:

```
omarchy-migrate --pending
```

If it prints pending names:

```
migration state is incomplete
```

Do not mark them complete manually unless you are following current official migration guidance.

23.63 — Real Course Mistake: `omarchy --version`

Incorrect:

```
omarchy --version
```

Current correct public command:

```
omarchy version
```

This belongs in the course FAQ because it illustrates a general pattern:

```
don't assume conventional CLI flags
```

Use Omarchy command discovery.

23.64 — Command Discovery Is a Troubleshooting Tool

Use:

```
omarchy
```

```
omarchy commands
```

```
omarchy commands --all
```

```
omarchy GROUP --help
```

Before saying:

```
"Omarchy doesn't support this."
```

or:

```
"this command should work."
```

discover the current command surface.

23.65 — Real Course Mistake: `omarchy idle`

There is no current top-level:

```
omarchy idle
```

on the course workstation.

Current idle control lives under:

```
omarchy toggle idle
```

Lesson:

```
correct concept
+
wrong command path
=
still wrong
```

23.66 — Runtime Toggle Ambiguity

If status says:

```
enabled: false
```

you must know what is being enabled.

Example:

```
Stay Awake override
```

rather than:

```
idle system itself
```

Verify:

```
help
status
UI
before/after change
```

Do not infer semantics from one boolean.

23.67 — Silent Commands

Some current runtime toggle commands may print no output on success.

Use:

```
echo $?
```

then inspect state.

No stdout does not mean failure.

23.68 — Troubleshooting Defaults / XDG

Wrong browser:

```
omarchy default browser
xdg-settings get default-web-browser
xdg-open https://example.com
```

Wrong terminal:

```
omarchy default terminal
cat ~/.config/xdg-terminals.list
xdg-terminal-exec --print-id
```

Test the problematic application last.

23.69 — Current File-Manager Exception

A system XDG directory handler may differ from an Omarchy file-manager launcher.

If:

```
xdg-open ~
```

opens one file manager but the Omarchy file-manager hotkey opens another:

```
do not assume XDG failed
```

The launcher may currently be hard-coded.

This is a good example of layer isolation.

23.70 — Troubleshooting Security Lockout

Current official Omarchy troubleshooting docs say repeated failed authentication can trigger:

```
faillock
```

If locked out on the graphical screen:

```
Ctrl + Alt + F2
```

Then, with appropriate root access:

```
faillock --reset --user YOUR_USERNAME
```

This resets lockout state.

It does not recover a forgotten password.

23.71 — Check Keyboard Layout Before Password Panic

If password suddenly "stops working":

check:

- Caps Lock;
- keyboard layout;
- Fn layer;
- physical keyboard.

The wrong layout can make a correct password appear wrong.

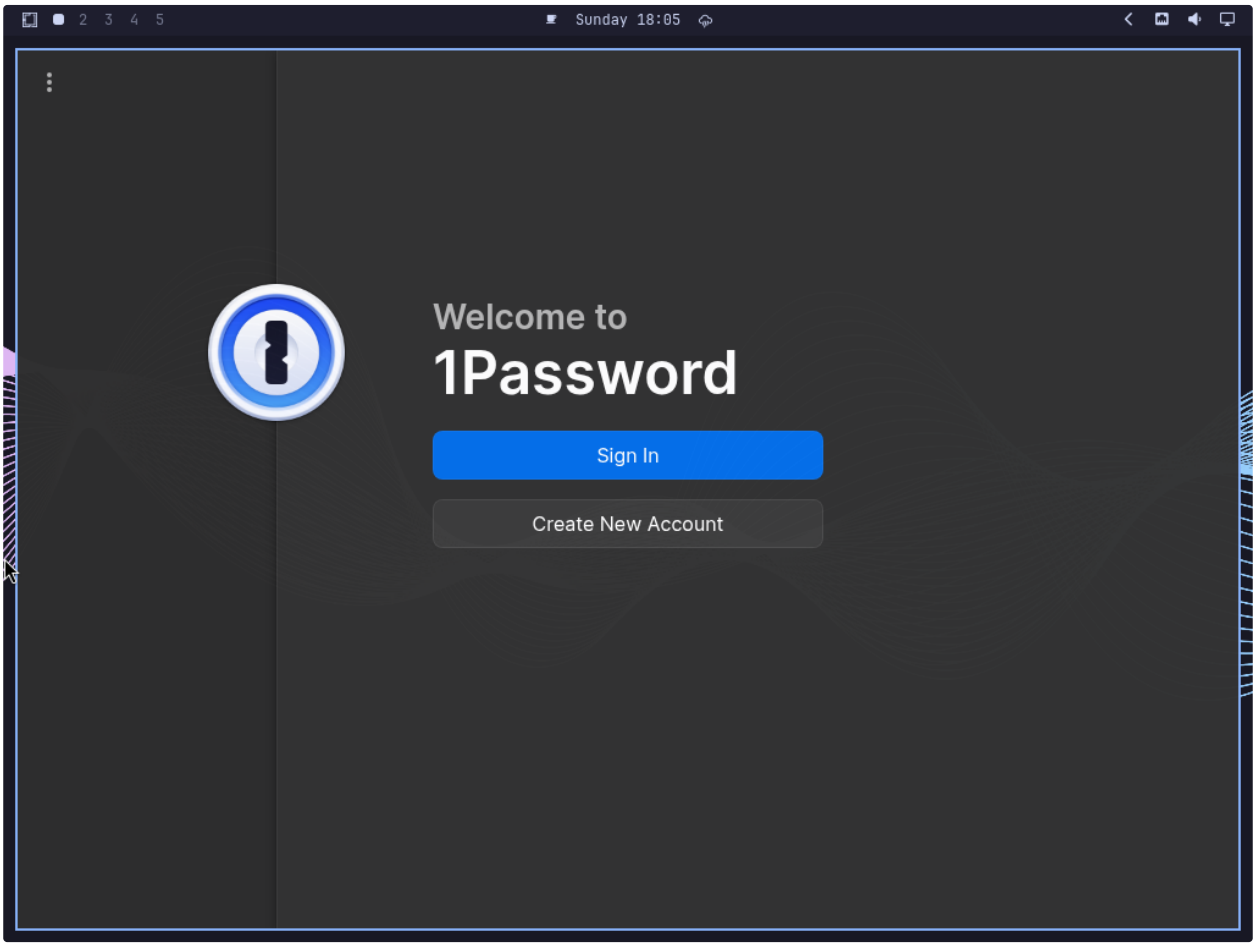
23.72 — 1Password Authorization Prompt Issues

Current official troubleshooting documentation specifically notes that 1Password authorization prompts may fail to appear if:

- hardware acceleration is disabled in 1Password;
- 1Password has not been launched since boot.

This is application-specific.

Use the official current troubleshooting note rather than debugging polkit blindly.



Screenshot 23.72-A
1Password installed and running (sign-in screen)

23.73 — Troubleshooting Plugins

If shell behavior changed after adding a plugin:

```
omarchy plugin list
```

Identify third-party plugins.

Then:

```
disable suspect plugin
restart shell
test
```

Plugin validation does not guarantee runtime correctness.

23.74 — Troubleshooting Hooks

If a problem occurs after every:

```
boot
update
theme change
```

inspect:

```
fd -H . ~/.config/omarchy/hooks
```

A hook may be repeatedly reintroducing the state.

Automatic code deserves suspicion when failures are event-triggered.

23.75 — Troubleshooting Autostart

If an app/service appears every login:

```
bat ~/.config/hypr/autostart.lua
```

Check systemd user services too.

Do not assume Omarchy itself launched it.

23.76 — Troubleshooting Personal Config vs Omarchy Defaults

Ask:

```
Does the problem disappear with current shipped config?
```

Before replacing anything:

```
compare
```

Read Omarchy defaults under:

```
/usr/share/omarchy
```

Do not edit them.

Use them as a reference.

23.77 — Refresh One Config Before Reinstalling Everything

If a user config is truly unrecoverable, use a targeted refresh.

Broad hierarchy:

```
edit/undo
↓
Git restore
↓
targeted refresh config
↓
refresh subsystem config
↓
reinstall configs
```

This preserves as much intentional customization as possible.

23.78 — Snapshot Rollback Decision

Use snapshot rollback when:

```
system update changed root state
```

Do not use it for:

```
~/.config problem
```

because current Omarchy snapshots do not restore `/home`.

This distinction prevents useless rollback attempts.

23.79 — Snapshot Did Not Fix the Problem

If rollback does not fix it, ask:

```
Was the problem in /home?
Was it a hardware issue?
Was it firmware?
Was it external?
```

Then run:

```
omarchy-debug
```

if the system-level cause remains unclear.

23.80 — When Reinstall Is Appropriate

Broad reinstall is appropriate when:

- default configs/packages are severely corrupted;
- rollback is insufficient/inapplicable;
- you intentionally want a clean current baseline.

It is not appropriate because:

```
one panel did not open once
```

23.81 — The "Do Not Panic" Rule for Logs

Logs can contain:

```
errors
warnings
coredumps
failed transient services
```

on an otherwise functional machine.

Ask:

```
Is this current?
Is this related?
Is it repeatable?
Does the service remain failed?
```

The course workstation had historical errors/coredumps while core services remained healthy.

Context matters.

23.82 — Troubleshooting Checklist: Shortcut

```
[ ] Super+K current binding
[ ] wev expected key event
[ ] hyprctl configerrors
[ ] command exists
[ ] command runs manually
[ ] environment/PATH correct
```

23.83 — Troubleshooting Checklist: Monitor

```
[ ] hyprctl monitors all
[ ] real output name
[ ] resolution
[ ] refresh
[ ] scale
[ ] monitors.lua
[ ] configerrors
```

23.84 — Troubleshooting Checklist: Terminal/Tmux

```
[ ] default terminal
[ ] terminal executable
[ ] xdg-terminal-exec mapping
[ ] terminal launches manually
[ ] tmux ls
[ ] current session cwd
[ ] helper exists with type
```

23.85 — Troubleshooting Checklist: Runtime

```
[ ] pwd
[ ] mise --version
[ ] mise ls --current
[ ] mise config ls
[ ] mise.toml
[ ] command -v runtime
[ ] runtime --version
```

23.86 — Troubleshooting Checklist: Git/GitHub

```
[ ] git status
[ ] branch
[ ] remote -v
[ ] gh auth status
[ ] HTTPS vs SSH
[ ] exact push error
```

23.87 — Troubleshooting Checklist: AI Agent

```
[ ] default agent
[ ] command exists
[ ] direct launch works
[ ] current directory
[ ] permission mode
[ ] Git clean state
[ ] exact agent error
```

23.88 — Troubleshooting Checklist: Hardware

```
[ ] current panel state
[ ] correct device selected
[ ] subsystem restart
[ ] retry
[ ] deeper logs only if still failing
```

23.89 — Troubleshooting Checklist: Update

```
[ ] exact update stage
[ ] /tmp/omarchy-update.log
[ ] pending migrations
[ ] reboot requested?
[ ] system still boots?
[ ] pre-update snapshot available?
```

23.90 — Practical Exercise: Create a Diagnostic Snapshot of the Workstation

Run:

```
omarchy version
```

```
systemctl --failed
```

```
systemctl --user --failed
```

```
hyprctl configerrors
```

```
omarchy network status --verbose
```

```
git --version
mise --version
```

Record only the important results.

This becomes your known-good baseline.

23.91 — Practical Exercise: Diagnose a Fake Shortcut Failure

Imagine:

```
Super + Alt + P does nothing
```

Walk the chain:

```
Super+K
↓
wev
↓
bindings.lua
↓
configerrors
↓
type target-command
↓
manual target-command
```

Write down which step would identify each possible fault.

23.92 — Practical Exercise: Diagnose a Fake Runtime Failure

Scenario:

```
Project expects Node 24
but `node --version` returns 26
```

Run:

```
pwd
mise ls --current
mise config ls
bat mise.toml
which node
node --version
```

Determine whether:

- project config is missing;
- project config is not trusted/loaded;
- you are in the wrong directory;
- command resolution is wrong.

23.93 — Practical Exercise: Diagnose a Fake Update Failure

Scenario:

`Omarchy update failed`

Do not reinstall.

Use:

```
visible terminal error
↓
/tmp/omarchy-update.log
↓
omarchy-migrate --pending
↓
system boot state
↓
snapshot availability
```

Build a written recovery plan.

23.94 — Practical Exercise: Build Your Own Troubleshooting Template

Create:

`TROUBLESHOOTING.md`

with:

```
### Symptom

### Reproduction

### Recent Changes

### Layer

### Evidence

### Hypothesis

### Smallest Test

### Result

### Fix

### Verification

### Lesson
```

This is how developers debug systems professionally.

23.95 — Common Troubleshooting Mistakes

Mistake 1 — Guessing Current Omarchy Behavior

Use current docs and installed commands.

Mistake 2 — Fixing Multiple Layers at Once

Change one variable.

Mistake 3 — Rebooting Before Learning Anything

Restart the smallest affected component.

Mistake 4 — Using Broad Config Refresh for One Typo

Repair the specific file.

Mistake 5 — Treating Every Log Error as Relevant

Correlate with symptom/time.

Mistake 6 — Assuming "Default" Means "Current User Choice"

Inspect current configured defaults.

Mistake 7 — Assuming Generic Linux Tutorial Commands Match Omarchy

Omarchy wraps many workflows.

Mistake 8 — Assuming an Agent's Explanation Is Evidence

Verify with system/Git output.

Mistake 9 — Treating Snapshot as `/home` Backup

It is not.

Mistake 10 — Failing to Document the Fix

A solved problem without a recorded lesson becomes a repeated problem.

23.96 — Omarchy "Verify, Don't Assume" FAQ

This section preserves the most useful mistakes encountered during the real course build.

`omarchy --version` Does Not Work

Use:

```
omarchy version
```

The Course Initially Assumed Foot

Actual course workstation:

```
Ghostty
```

Check:

```
omarchy default terminal
```

Resize Shortcut Guess Was Wrong

Do not rely on remembered/generic Hyprland shortcuts.

Use:

```
Super + K
```

Clipboard Shortcut Was Initially Misidentified

Current course baseline:

```
Super + V
→ paste

Super + Ctrl + V
→ clipboard manager
```

Verify current release.

Scratchpad Shortcuts Were Initially Guessed

Current course baseline:

```
Super + S
→ toggle scratchpad

Super + Alt + S
→ send to scratchpad
```

Verify current release.

`omarchy pkg add hunk` Was the Wrong Layer

Current course setup uses:

```
mise
```

for Hunk.

`.mise.toml` Was Wrong for the Current Course Workflow

Use:

```
mise.toml
```

`omarchy agent --pick` Did Not Behave as Expected

It launched the current agent rather than presenting the assumed chooser.

Read:

```
omarchy agent --help
```

Equal-Split Reset Was Guessed and Failed

Do not publish an unverified shortcut merely because generic Tmux/Hyprland conventions suggest one.

OMP `write` Permission Semantics Were Initially Misstated

Current verified course semantics:

```
always-ask
→ reads auto, writes+exec prompt

write
→ reads+writes auto, exec prompt

yolo
→ all auto
```

Permission modes are security-sensitive. Verify them.

`omarchy idle` Does Not Exist on the Course Workstation

Use the current:

```
omarchy toggle idle
```

family.

GRUB Was Assumed but the System Actually Used Limine

Verify:

```
bootloader
UKI
EFI entries
```

Never assume.

23.97 — Checkpoint

Before moving on, you should be able to answer **yes** to the following.

Method

- [ ] I define symptoms precisely.
- [ ] I identify recent changes.
- [ ] I classify the owning layer.
- [ ] I preserve evidence.
- [ ] I make one change at a time.
- [ ] I verify the result.

System

- [ ] I can inspect failed system/user services.
- [ ] I understand log triage.
- [ ] I understand `omarchy-debug`.
- [ ] I know when snapshot rollback fits.

Desktop

- [ ] I can diagnose Hyprland config errors.
- [ ] I can inspect monitor/window state.
- [ ] I can verify actual key events with `wev`.
- [ ] I know how to recover shell/plugins.

Development

- [ ] I can diagnose terminal/Tmux state.
- [ ] I can diagnose mise/runtime issues.
- [ ] I can diagnose Git/GitHub auth/remotes.
- [ ] I can diagnose AI-agent launch/permission/directory issues.

Hardware

- [ ] I restart audio/Bluetooth/Wi-Fi/trackpad before reboot.
- [ ] I distinguish device selection from subsystem failure.

Recovery

- [ ] I understand targeted config repair.
- [ ] I understand broad refresh/reinstall is destructive.
- [ ] I understand snapshots do not restore `/home`.
- [ ] I know when TTY recovery is appropriate.

Philosophy

- [ ] I verify current Omarchy behavior instead of assuming generic Linux behavior.
- [ ] I document solved problems for future use.

23.98 — What You Can Now Do

After completing Module 23, you can now:

- debug Omarchy like a software system;
- classify failures by layer;
- use evidence instead of guesses;
- recover common desktop/hardware issues quickly;
- distinguish shell, compositor, terminal, runtime, and application failures;
- diagnose current Omarchy CLI mismatches;
- diagnose development-environment issues;
- diagnose agent/Git/project state safely;
- decide between restart, refresh, rollback, and reinstall;
- preserve useful troubleshooting evidence;
- build your own troubleshooting knowledge base;
- avoid repeating the exact mistakes discovered during this course.

Most importantly:

> *You now know how to solve workstation problems without making the workstation harder to understand.*

Module 23 Summary

Troubleshooting loop:

```
observe
↓
classify
↓
reproduce
↓
inspect
↓
hypothesize
↓
smallest test
↓
verify
↓
document
```

Core system tools:

```
omarchy version
systemctl --failed
systemctl --user --failed
journalctl
omarchy-debug
```

Hyprland:

```
hyprctl configerrors
hyprctl monitors all
hyprctl activewindow
wev
```

Development:

```
pwd
type COMMAND
command -v COMMAND
mise ls --current
git status
git diff
gh auth status
```

Update diagnosis:

```
/tmp/omarchy-update.log
omarchy-migrate --pending
snapshot
```

Current recovery hierarchy:

```
restart component
↓
repair file
↓
targeted refresh
↓
session/reboot
↓
snapshot rollback
↓
broad reinstall
```

And the most important rule:

> ***If you do not know which layer is broken, you are not ready to apply a broad fix.***

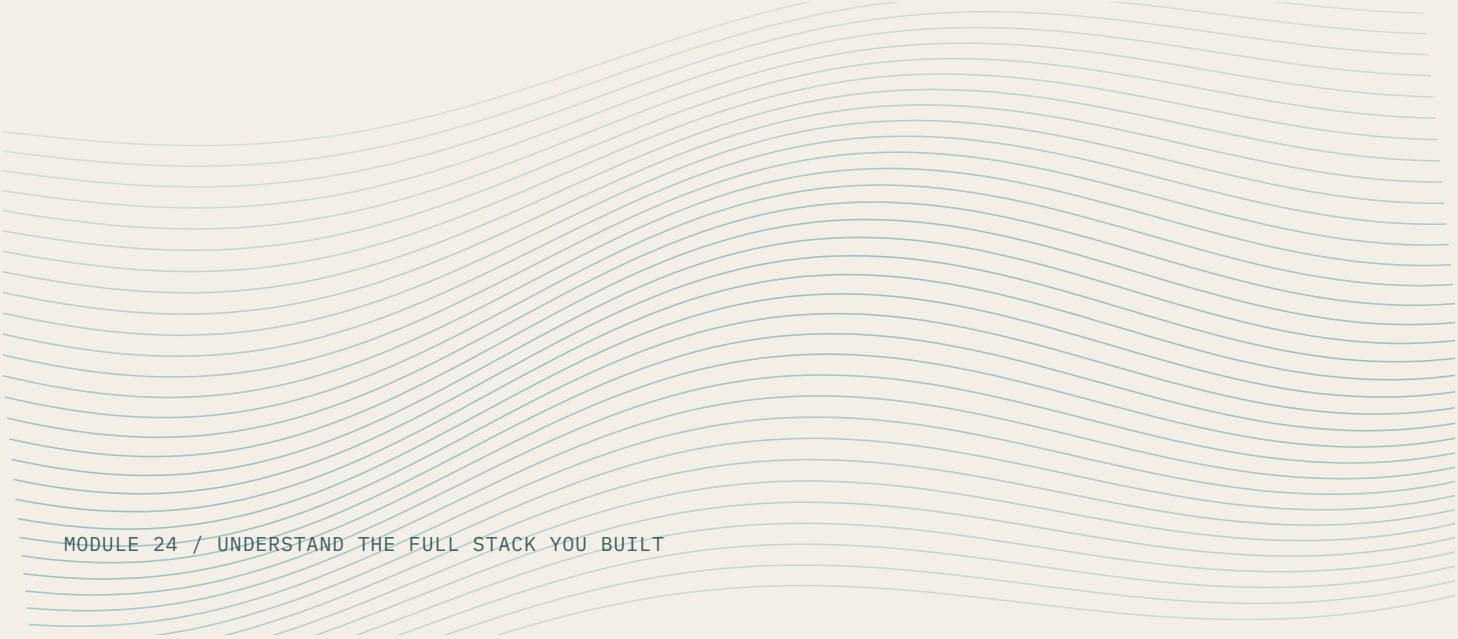
MODULE 24

24

Understand the Full Stack You Built

You have reached the final core module.

By now, you can:



About This Module

You have reached the final core module.

By now, you can:

- install Omarchy;
- navigate Hyprland;
- use the terminal;
- work in Tmux;
- manage runtimes with mise;
- use Git and GitHub;
- integrate coding agents;
- customize Hyprland;
- build keybindings;
- manage themes;
- automate with hooks/plugins;
- control hardware;
- manage defaults;
- update and recover the system;
- understand the boot chain;
- harden the workstation;
- reproduce your setup;
- troubleshoot it methodically.

This module answers one final question:

> ***What did we actually build?***

A power user should not see the workstation as a bag of commands.

You should see a stack.

Each layer has:

- a responsibility;
- an owner;
- an interface;
- a recovery strategy.

Once you understand that stack, unfamiliar problems become easier because you can place them somewhere.

The complete system is roughly:

```
Hardware
↓
UEFI firmware
↓
Limine
↓
Unified Kernel Image
↓
Linux kernel
↓
systemd
↓
Wayland
↓
Hyprland
↓
Omarchy Shell
↓
Ghostty
↓
Bash
↓
Tmux
↓
Neovim / editor
↓
mise
↓
Git / GitHub
↓
AI coding agent
↓
project
```

Not every interaction follows this exact vertical path, but it is the right mental model.

The goal of this module is:

> ***Turn everything you learned into one coherent architecture you can reason about, rebuild, debug, and extend.***

What You Will Learn

By the end of Module 24, you will be able to:

- explain the full Omarchy workstation stack;
- identify which layer owns a behavior;
- distinguish Linux, Hyprland, and Omarchy responsibilities;
- understand how the boot stack becomes the desktop stack;
- understand how the desktop becomes the development stack;
- understand how configuration flows through the system;
- understand how defaults and XDG connect applications;
- understand how Git, mise, Tmux, and AI agents work together;
- understand how recovery differs by layer;
- explain the workstation architecture to another developer;
- make future changes without losing the architectural model.

24.1 — The Complete Workstation

The system you built can be grouped into six major layers:

```
1. Hardware + firmware
2. Operating system + boot
3. Graphical desktop
4. Shell + terminal environment
5. Development environment
6. Project + AI workflow
```

Each has its own job.

24.2 — Layer 1: Hardware

Examples:

```
CPU
GPU
RAM
NVMe/SSD
monitor
keyboard
mouse
Wi-Fi
Bluetooth
audio hardware
TPM
```

The hardware determines what the operating system can work with.

Examples:

```
144 Hz monitor
→ compositor must request 144 Hz

compact keyboard
→ firmware may hide keys behind Fn layer

hybrid-GPU laptop
→ runtime GPU controls may exist

TPM 2.0
→ hardware-backed security capabilities
```

Software cannot make unsupported hardware capabilities appear.

24.3 — Firmware

The firmware is normally:

```
UEFI
```

It initializes hardware and chooses what EFI program to boot.

Useful responsibilities:

```
boot order
Secure Boot
TPM configuration
virtualization settings
hardware initialization
firmware boot menu
```

The firmware exists before Linux.

That distinction matters.

24.4 — Layer 2: Boot

The current Omarchy architecture uses a modern UEFI boot path.

Conceptually:

```
UEFI
↓
Limine
↓
Omarchy UKI
↓
Linux kernel
```

On the course workstation, this was verified as:

```
Limine
Omarchy.linux
/boot/EFI/Linux/omarchy_linux.efi
```

The key lesson was not the exact filename.

The lesson was:

> *Verify the actual boot chain.*

24.5 — Limine

Limine is more than a simple launcher.

In the current Omarchy recovery model, it also exposes:

```
bootable system snapshots
```

That means the bootloader is part of the recovery architecture.

Conceptually:

```
normal boot
or
previous system snapshot
```

can be selected before the Linux system fully starts.

24.6 — Unified Kernel Image

The UKI bundles important boot components into one EFI image.

Conceptually:

```
EFI stub
kernel
initramfs
kernel command line
metadata
```

become:

```
one bootable .efi image
```

This is a clean modern boot model and integrates naturally with boot signing.

24.7 — Linux Kernel

Once control passes to the kernel, Linux manages:

- CPU scheduling;
- memory;
- devices;
- filesystems;
- networking;
- process isolation;
- drivers.

Check the running kernel:

```
uname -r
```

The kernel does not know what:

```
Super + Space
```

means.

That behavior lives much higher in the stack.

24.8 — systemd

After early boot, systemd manages services and session infrastructure.

Examples:

```
NetworkManager
Bluetooth
PipeWire-related services
login/session services
user services
```

Troubleshooting tools:

```
systemctl
systemctl --user
journalctl
```

If a service is dead:

```
Hyprland config is usually not the first place to look.
```

24.9 — Linux Filesystem Model

The important high-level split:

```
/
→ system/root filesystem

/home
→ personal data/config
```

This matters strongly for recovery.

Current Omarchy snapshots restore:

```
root
```

but not:

```
/home
```

That is why:

```
snapshot
≠
dotfiles backup
```

24.10 — Package-Owned vs User-Owned Files

Current Omarchy follows an important configuration boundary:

```
/usr/share/omarchy
→ Omarchy-owned

~/.config
→ user-owned
```

This one distinction explains a huge part of the workstation architecture.

You normally:

```
read /usr/share/omarchy
override in ~/.config
```

You do not maintain a private fork of Omarchy by editing package-owned defaults.

24.11 — Layer 3: Wayland

Wayland is the graphical display protocol architecture used by the current desktop stack.

At a high level:

```
applications
↔
compositor
```

rather than applications talking to a traditional X server.

You do not need to become a Wayland developer.

You do need to know:

```
wev
```

is useful because input events are happening in a Wayland environment.

24.12 — Hyprland

Hyprland is the compositor/window manager.

It owns behavior such as:

- workspaces;
- window focus;
- tiling;
- floating;
- monitors;
- input;
- window rules;
- keybindings.

Your personal config lives under:

```
~/.config/hypr
```

Examples:

```
monitors.lua
input.lua
bindings.lua
looknfeel.lua
autostart.lua
```

24.13 — Hyprland Is Not Omarchy

This distinction is essential.

Hyprland provides the underlying compositor.

Omarchy provides:

- curated defaults;
- helper APIs;
- menu integration;
- shell UI;
- installation flows;
- restart/refresh commands;
- development setup;
- update/recovery behavior.

Conceptually:

```
Hyprland
→ engine

Omarchy
→ integrated workstation system around it
```

24.14 — Omarchy's Hyprland Layering

Current Quattro config follows the general pattern:

```
bootstrap
↓
Omarchy defaults
↓
your monitor config
↓
your input config
↓
your bindings
↓
your look/feel
↓
your autostart
↓
dynamic toggles
```

This is why small personal overrides can survive alongside evolving upstream defaults.

24.15 — Omarchy Shell

The Omarchy Shell is the interactive desktop shell layer.

It provides things such as:

- top bar;
- system panels;
- menu;
- notifications;
- overlays;
- clipboard UI;
- plugins.

Current architecture runs it as a long-lived Quickshell process.

This is distinct from Hyprland.

If:

```
bar is broken
```

but windows/workspaces still work:

```
Hyprland may be fine.
Omarchy Shell may be the broken layer.
```

24.16 — Plugins

Shell plugins extend this layer.

Current plugin types include concepts such as:

```
bar widget
panel
overlay
menu
service
bar
```

Third-party plugins execute code.

Therefore:

```
shell customization
```

is also a security boundary.

24.17 — Runtime Toggles

Current Omarchy exposes temporary state through:

```
omarchy toggle
```

Examples:

```
idle
night light
bar
notifications
suspend
touchpad
```

These are not the same as persistent config.

The architecture separates:

```
desired normal configuration
```

from:

```
temporary current state
```

24.18 — Layer 4: Terminal

The terminal emulator provides the graphical terminal window.

On the course workstation:

```
Ghostty
```

is selected.

A fresh Omarchy install may use a different current default.

The important abstraction is:

```
default terminal
```

not:

```
hard-code Ghostty everywhere
```

24.19 — XDG Terminal Integration

Current Omarchy terminal defaults integrate with:

```
xdg-terminal-exec
```

This lets generic terminal launchers ask:

```
"What terminal does this user prefer?"
```

rather than embedding an application name.

This is clean system design.

24.20 — Bash

Inside the terminal, Bash provides the shell.

Bash owns:

- shell command parsing;
- variables;
- aliases;
- functions;
- pipelines;
- redirection;
- environment.

Personal shell configuration belongs in:

```
~/.bashrc
```

The shell is not the terminal.

A useful distinction:

```
Ghostty
→ terminal emulator

Bash
→ command shell
```

24.21 — Shell Tools

Omarchy ships enhanced terminal tools such as current documented:

```
fzf
zoxide
eza
bat
rg
fd
```

These improve navigation and search.

Examples:

```
rg "pattern"
```

```
fd file
```

```
Ctrl + R
→ fuzzy shell history via fzf
```

These tools improve speed without changing the architecture.

24.22 — Tmux

Tmux sits inside the terminal/shell workflow and provides persistence.

It owns:

- sessions;
- windows;
- panes;
- detached processes;
- development layouts.

Conceptually:

```
terminal window can disappear
↓
Tmux session continues
```

This is why Tmux is ideal for:

- long-running development;
- SSH;
- multi-pane work;
- AI-agent layouts.

24.23 — Tmux State Is Independent

An important course lesson:

```
new terminal current directory
```

does not automatically alter:

```
existing Tmux pane current directory
```

Tmux persists state.

Once you understand that architectural boundary, this behavior is obvious rather than mysterious.

24.24 — Editor

Your editor is another replaceable layer.

Current course workstation:

```
Neovim
```

But Omarchy can use other editors.

The architectural goal is:

```
default editor
```

so Omarchy's workflows do not require rewriting every launch path.

The project should not depend on one human editor.

24.25 — Layer 5: Development Environment

Now we move from workstation mechanics into software-development mechanics.

Core tools:

```
mise
Git
GitHub CLI
editor
Tmux
coding agent
```

Each owns a different responsibility.

24.26 — Mise

Mise manages development tools and runtime versions.

It answers:

```
Which Node version?
Which Python version?
Which Bun version?
Which CLI tool version?
```

The key architectural separation:

```
system packages
≠
developer runtimes
```

This avoids polluting system-level package management with every project runtime.

24.27 — Global vs Project Runtime

Example:

```
global Node
→ workstation convenience

project Node
→ project requirement
```

The project should declare what it needs.

Current course convention:

```
mise.toml
```

Project configuration wins when inside that project environment.

24.28 — Why This Matters for AI

If an AI agent runs:

```
npm test
```

it should inherit the same project runtime environment as the developer.

A declared project environment reduces:

```
works for developer
fails for agent
```

and:

```
works on machine A
fails on machine B
```

24.29 — Git

Git is the change-control layer.

It owns:

- working tree;
- staging;
- commits;
- branches;
- history;
- worktrees.

Git is one of the most important safety boundaries in an AI-native workflow.

Before AI work:

```
git status
```

After AI work:

```
git diff
```

Git tells you what actually changed.

24.30 — GitHub

GitHub is the remote collaboration/backup layer for Git repositories.

GitHub is not Git.

Conceptually:

```
Git
→ local version control system

GitHub
→ remote hosting/collaboration service
```

`gh` is an interface to GitHub.

`git` remains the repository engine.

24.31 — Branches and Worktrees

Branches provide logical history separation.

Worktrees provide:

```
multiple filesystem checkouts
```

for separate branches at the same time.

This is especially useful for parallel AI tasks.

Conceptually:

```
Agent A
→ worktree A

Agent B
→ worktree B
```

instead of both writing into one directory.

24.32 — Layer 6: AI Coding Agent

The AI coding agent is a development assistant with local capabilities.

Depending on the selected agent and permission mode, it may:

- read;
- write;
- execute;
- search;
- use network tools;

- interact with Git;
- inspect system state.

That makes it:

```
a local operator
```

not merely a chat interface.

24.33 — Agent Neutrality

The workstation architecture should not depend on:

- one vendor;
- one model;
- one pricing plan.

The stable interface is:

```
project
+
terminal
+
Git
+
runtime
+
agent
```

Agents can change.

The project remains.

24.34 — `AGENTS.md`

`AGENTS.md` is durable project-level agent context.

It can describe:

- architecture;
- commands;
- constraints;
- test procedures;
- coding rules.

This is better than repeating important project instructions in every AI session.

It is also better than expecting an AI session history to become project documentation.

24.35 — AI Session vs Project Knowledge

An AI session is temporary.

Project documentation is durable.

Useful durable locations:

```
README
AGENTS.md
ADR
issue
code comments
tests
commit history
```

If a decision matters to future developers:

```
write it into the project
```

not only into the agent conversation.

24.36 — The AI-Native Development Loop

The stack now supports:

```
project
↓
mise environment
↓
Git clean state
↓
Tmux workspace
↓
editor + agent + terminal
↓
analyze
↓
implement
↓
test
↓
git diff
↓
review
↓
commit
↓
push
```

This is the practical outcome of the entire course.

24.37 — Full Input Path Example

What happens when you press:

```
Super + Alt + Return
```

on the course workstation?

Conceptually:

```
physical keyboard
↓
keyboard firmware/Fn layer
↓
Linux input subsystem
↓
Wayland
↓
Hyprland binding
↓
Omarchy action
↓
default/specific terminal launcher
↓
terminal
↓
Tmux workflow
```

A problem anywhere in that chain can make the shortcut "not work."

This is why debugging by layer is so powerful.

24.38 — Full Browser Launch Path

When an application opens a URL:

```
application
↓
XDG open/default logic
↓
desktop application ID
↓
selected browser
```

But an application can override this and hard-code its own browser behavior.

Therefore:

```
system default correct
```

does not guarantee:

```
every application follows it
```

Architecture gives you the right place to inspect.

24.39 — Full Monitor Configuration Path

Conceptually:

```
monitor hardware
↓
GPU/driver
↓
kernel DRM stack
↓
Wayland compositor
↓
Hyprland monitor config
↓
Omarchy user override
```

If:

```
display physically supports 144 Hz
```

but Hyprland is configured at:

```
60 Hz
```

the hardware capability exists, but the compositor is not using it.

Again:

```
layer matters.
```

24.40 — Full Audio Path

High level:

```
application
↓
audio stream
↓
PipeWire/session management
↓
selected output
↓
audio device
```

Omarchy's panel wraps the common controls.

If:

```
wrong output selected
```

you do not need to debug the kernel.

24.41 — Full Network Path

High level:

```
application
↓
DNS / sockets
↓
NetworkManager
↓
network interface
↓
Wi-Fi/Ethernet
↓
router
↓
internet
↓
remote service
```

A failed GitHub request could exist at several places in this path.

Start local, then move outward.

24.42 — Full Update Path

Current Omarchy update architecture conceptually:

```
omarchy update
↓
preflight
↓
snapshot
↓
package update
↓
migrations
↓
mise updates
↓
hooks
↓
restart checks
↓
current system
```

This is why bypassing the wrapper is architecturally significant.

It skips coordinated layers.

24.43 — Full Recovery Path

The full recovery strategy is layered:

```
app problem
→ restart app

shell problem
→ restart shell

hardware service
→ restart subsystem

user config
→ Git restore / targeted refresh

session problem
→ restart session

system update problem
→ boot snapshot

severe system/config corruption
→ reinstall
```

The correct recovery action is determined by ownership.

24.44 — Configuration Ownership Map

Use this reference.

```
Hyprland
→ ~/.config/hypr

Omarchy Shell
→ ~/.config/omarchy/shell.json

Bash
→ ~/.bashrc

Mise
→ ~/.config/mise
→ project mise.toml

XCompose
→ ~/.XCompose

Omarchy-owned defaults
→ /usr/share/omarchy
```

This is one of the most useful maps in the course.

24.45 — State Ownership Map

Not everything is configuration.

Examples:

```
Tmux session
→ runtime state

Git working tree
→ project state

notification silence
→ runtime toggle

Bluetooth pairing
→ hardware/service state

system snapshot
→ filesystem recovery state

browser session
→ application state
```

This prevents treating every state problem as a config-file problem.

24.46 — Tool Selection Map

Question:

```
What should I use?
```

Answer:

```
system package
→ omarchy pkg

Omarchy-integrated app/setup
→ omarchy install / menu

developer runtime/CLI
→ mise

project history
→ Git

remote repo
→ GitHub

persistent terminal workspace
→ Tmux

project AI assistance
→ coding agent
```

This is the workstation's operational taxonomy.

24.47 — Why the Taxonomy Matters

Without clear tool ownership:

```
install Node with pacman
install another Node with curl
install third Node with mise
```

or:

```
edit Omarchy defaults directly
override in user config
and also create a script doing the same thing
```

Complexity becomes self-created.

A coherent workstation has one preferred layer for each responsibility.

24.48 — The "One Owner" Rule

Whenever possible:

> *One behavior should have one primary owner.*

Examples:

```
default browser
→ Omarchy default/XDG

project Node version
→ mise.toml

window binding
→ bindings.lua

monitor refresh
→ monitors.lua

project history
→ Git

system update
→ omarchy update
```

This reduces ambiguity.

24.49 — When Two Layers Intentionally Interact

Some behaviors genuinely cross layers.

Example:

```
Omarchy default terminal
↓
xdg-terminal-exec
↓
Ghostty
```

Another:

```
Hyprland binding
↓
Omarchy command
↓
Tmux layout helper
```

Cross-layer behavior is fine when the interface is intentional.

The problem is accidental duplication.

24.50 — Security Boundaries in the Stack

Security-relevant boundaries include:

```
firmware Secure Boot
disk encryption
user authentication
sudo
Docker group
firewall
SSH
plugin execution
hook execution
agent execution
project dependencies
Git secrets
```

Notice how security is distributed across the entire architecture.

It is not one "security module" sitting beside everything else.

24.51 — Reproducibility Boundaries

Reproducibility is also distributed:

```
Omarchy installer
→ base system

dotfiles Git
→ user config

mise
→ dev tools

project mise.toml
→ project runtime

GitHub
→ project remote

WORKSTATION.md
→ human procedure
```

The stack is reproducible because each layer has a source of truth.

24.52 — Observability Boundaries

Useful diagnostics by layer:

```
Boot
→ bootctl / efibootmgr

System
→ systemctl / journalctl

Hyprland
→ hyprctl

Input
→ wev

Shell
→ omarchy-shell

Terminal command resolution
→ type / command -v

Mise
→ mise ls --current

Git
→ git status / git diff

Network
→ omarchy network status --verbose
```

A good power user knows which tool sees which layer.

24.53 — "Current Reality" Beats Memory

A recurring theme throughout the course:

```
official default
≠
your current selection

old documentation
≠
current release

generic Linux behavior
≠
Omarchy wrapper behavior

tool name
≠
obvious semantics
```

Examples from the course:

- Foot vs Ghostty;
- GRUB vs Limine;
- `.mise.toml` vs `mise.toml`;
- `omarchy --version` vs `omarchy version`;
- `omarchy idle` vs `omarchy toggle idle`;
- OMP permission semantics.

This lesson is foundational.

24.54 — The Best Question When Something Changes

Ask:

> ***Which layer changed?***

If Omarchy updates:

```
package defaults?
CLI?
shell?
migration?
```

If a project changes:

```
runtime?
dependencies?
Git state?
```

If hardware changes:

```
monitor output name?
input event?
driver?
```

The architectural model gives you a way to reason about change.

24.55 — The Best Question Before Customizing

Ask:

> ***Which layer should own this customization?***

Examples:

```
new keybinding
→ Hyprland user binding

new developer runtime
→ mise

new session startup helper
→ autostart or systemd user service

run action after update
→ hook

new top-bar widget
→ plugin

default browser
→ Omarchy default/XDG
```

This prevents hacks.

24.56 — The Best Question Before Installing

Ask:

> *Which installation layer should own this tool?*

Use the Module 6 decision tree.

```
system package
Omarchy-integrated install
mise
```

A clean workstation does not install the same tool through three unrelated mechanisms.

24.57 — The Best Question Before Recovery

Ask:

> *Which state do I want to restore?*

Examples:

```
one file
→ Git restore

personal config
→ dotfiles Git

root system state
→ snapshot

project source
→ project Git

personal documents
→ backup
```

Recovery only works well if the recovery tool matches the damaged state.

24.58 — The Best Question Before Giving AI Permissions

Ask:

> *What layer can this agent change?*

If it can:

```
write project files
```

Git is the main recovery boundary.

If it can:

```
sudo
```

the blast radius becomes system-wide.

If it can:

```
control Docker
```

it may effectively have root-equivalent power.

Architecture makes permission decisions clearer.

24.59 — Practical Exercise: Draw Your Own Stack

Without copying this handbook, draw:

```
your current workstation stack
```

Include:

- hardware;
- boot;
- compositor;
- shell;
- terminal;
- Tmux;
- editor;
- mise;
- Git;
- agent;
- project.

If you cannot explain one box:

```
review that module.
```

24.60 — Practical Exercise: Map Five Behaviors

For each behavior, identify the owner.

EXAMPLE 1

```
Super+Return opens Ghostty
```

Possible ownership chain:

```
Hyprland binding
→ Omarchy terminal default
→ xdg-terminal-exec
→ Ghostty
```

EXAMPLE 2

```
Node 24 inside project
```

Owner:

```
project mise.toml
```

EXAMPLE 3

```
144 Hz monitor
```

Owner:

```
Hyprland monitors.lua
```

EXAMPLE 4

```
bar widget
```

Owner:

```
Omarchy Shell plugin/state
```

EXAMPLE 5

```
system before update
```

Recovery owner:

```
snapshot
```

Do five more yourself.

24.61 — Practical Exercise: Map Your Configuration

Create a table:

```
Behavior | File/Tool | Layer | Recovery
```

Examples:

```
Monitor | monitors.lua | Hyprland | Git restore
Theme | omarchy theme | Omarchy | reset/select theme
Node | mise.toml | Project | Git restore
Shell bar | shell.json | Omarchy Shell | dotfiles restore
System update | omarchy update | OS | snapshot
```

This becomes a powerful reference.

24.62 — Practical Exercise: Map Your Development Loop

Write your actual workflow:

```
open project
↓
check Git
↓
check runtime
↓
start Tmux
↓
open editor
↓
launch agent
↓
analyze
↓
implement
↓
test
↓
review diff
↓
commit
↓
push
```

If your real workflow differs, document the real one.

The goal is not to imitate the course mechanically.

24.63 — Practical Exercise: Map Your Recovery Stack

Write:

```
Config mistake
→ ?

System update break
→ ?

Deleted project file
→ ?

Lost laptop
→ ?

Wrong runtime version
→ ?

Shell plugin crash
→ ?
```

Expected answers should involve different layers.

This proves you understand recovery ownership.

24.64 — Practical Exercise: Explain the Workstation to Someone Else

Explain in five minutes:

```
What is Omarchy?
What does Hyprland do?
What does the Omarchy Shell do?
Why use Tmux?
Why use mise?
Why is Git essential for AI coding?
Why does the bootloader matter?
```

Teaching is one of the best tests of understanding.

24.65 — Practical Exercise: Verify Current Reality

Run:

```
omarchy version
```

`uname -r` `omarchy default browser` `omarchy default terminal`

`omarchy default agent` `mise --version` `git --version` `hyprctl monitors all`

Then update:

`WORKSTATION.md`

with any changed facts.

24.66 — Practical Exercise: Identify One Unnecessary Complexity

Look at your setup and ask:

`What did I customize that I no longer need?`

Possible examples:

- unused binding;
- old script;
- stale plugin;
- obsolete global runtime;
- old workaround;
- duplicate config.

Remove complexity deliberately.

A mature workstation often becomes simpler over time.

24.67 — Avoid Configuration Archaeology

Bad future:

```
three years later
→ no idea why this script runs
→ afraid to delete it
```

Better:

```
small configs
clear comments
small Git commits
documented rationale
```

Maintainability applies to your workstation too.

24.68 — Your Workstation Is Software

Treat it like software.

It has:

- dependencies;
- configuration;
- runtime state;
- updates;
- interfaces;
- bugs;
- recovery;
- version control;
- documentation.

That is why developer practices work so well here.

24.69 — Your Workstation Is Also Infrastructure

It is also infrastructure.

It hosts:

- credentials;
- development runtimes;
- repositories;
- local services;
- remote-access tools;
- AI agents.

That means:

```
security
recovery
observability
reproducibility
```

matter just as much as appearance.

24.70 — Your Workstation Is Not the Project

Keep the final separation clear.

```
workstation
→ enables development

project
→ contains the software
```

Do not let a project depend on invisible workstation magic.

A strong project remains reproducible through:

```
Git
mise.toml
documentation
tests
```

24.71 — The Ultimate Architecture Principle

If you remember only one architectural principle:

> ***Make state explicit and put it in the layer that owns it.***

Examples:

```
project runtime
→ mise.toml

keybinding
→ bindings.lua

monitor
→ monitors.lua

system default browser
→ Omarchy/XDG default

project rules
→ AGENTS.md

system update
→ omarchy update

personal config history
→ dotfiles Git
```

This single habit prevents enormous complexity.

24.72 — The Ultimate Troubleshooting Principle

If you remember only one troubleshooting principle:

> ***Find the owning layer before changing state.***

Do not:

```
fix by superstition
```

Do:

```
inspect
classify
test
verify
```

24.73 — The Ultimate AI Principle

If you remember only one AI-workflow principle:

> ***AI accelerates the development loop; it does not replace Git, tests, review, or project documentation.***

The agent is one component.

The engineering system is larger.

24.74 — The Ultimate Omarchy Principle

If you remember only one Omarchy customization principle:

> ***Override Omarchy; do not overwrite Omarchy.***

Use:

```
~/.config
```

for your personal changes.

Let:

```
/usr/share/omarchy
```

remain package-owned.

This preserves upgradeability.

24.75 — Checkpoint

Before finishing the core curriculum, you should be able to answer **yes** to the following.

Boot

- [ ] I understand UEFI → Limine → UKI → kernel.
- [ ] I understand the role of snapshots.
- [ ] I can classify boot vs userspace failures.

Desktop

- [ ] I understand Wayland and Hyprland at a practical level.
- [ ] I understand Omarchy Shell is separate from Hyprland.
- [ ] I know where personal desktop config belongs.

Terminal

- [ ] I understand terminal vs shell vs Tmux.
- [ ] I understand persistent Tmux state.
- [ ] I understand default terminal integration.

Development

- [ ] I understand mise's responsibility.
- [ ] I understand Git vs GitHub.
- [ ] I understand branches/worktrees.
- [ ] I understand the AI agent as a local operator.

Configuration

- [ ] I can identify the owner of monitor/input/binding/theme/default/runtime settings.
- [ ] I understand package-owned vs user-owned files.
- [ ] I understand runtime state vs persistent config.

Security

- [ ] I understand where privilege boundaries appear.
- [ ] I understand plugins/hooks/agents execute code.
- [ ] I understand Docker privilege risk.

Recovery

- [ ] I understand component restart vs config restore vs snapshot vs backup.
- [ ] I know `/home` is not restored by system snapshots.
- [ ] I can choose the correct recovery layer.

Reproducibility

- [ ] I understand installer + dotfiles + mise + Git + backup as separate pieces.
- [ ] I can explain how I would rebuild the workstation.

Architecture

- [ ] I can explain the complete workstation stack without reading the diagram.

24.76 — What You Can Now Do

After completing Module 24, you can now:

- reason about Omarchy as a complete system;
- explain which component owns which behavior;
- understand how the machine boots into the development environment;
- understand the distinction between Linux, Hyprland, and Omarchy;
- understand how shell, terminal, Tmux, editor, mise, Git, and AI fit together;
- place configuration in the correct layer;
- place recovery in the correct layer;
- diagnose problems more quickly;
- keep the workstation reproducible;
- evolve the system without losing architectural clarity.

Most importantly:

> *You are no longer just using an Omarchy installation. You understand the workstation you built.*

Module 24 Summary

Full stack:

```
Hardware
↓
UEFI
↓
Limine
↓
UKI
↓
Kernel
↓
systemd
↓
Wayland
↓
Hyprland
↓
Omarchy Shell
↓
Terminal
↓
Bash
↓
Tmux
↓
Editor
↓
mise
↓
Git
↓
AI Agent
↓
Project
```

Core ownership:

```
Omarchy internals
→ /usr/share/omarchy

Personal config
→ ~/.config

Project runtime
→ mise.toml

Project change history
→ Git

Remote collaboration
→ GitHub

System rollback
→ snapshot

Personal config rollback
→ dotfiles Git

Personal data recovery
→ backup
```

Core engineering model:

```
explicit state
+
one clear owner
+
current verification
+
small reversible changes
```

And the final core-course lesson:

> ***The power of Omarchy is not that everything is customized. The power is that the whole workstation becomes understandable.***