BUILD THE ULTIMATE

AI / SOFTWARE-DEVELOPMENT WORKSTATION

with Omarchy

STEVINATOR HANDBOOK

Build the Ultimate AI/Software-Development Workstation with Omarchy

Volume I — The Foundation

Modules 00 through 12

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

0 From Bare Metal to a Ready Omarchy AI Development Workstation

1 How Omarchy Actually Works

2 Linux Concepts Developers Actually Need

3 Master the Omarchy Desktop

4 Terminal Mastery for Omarchy Developers

5 Tmux for Omarchy Developers

6 Development Environment Management with Omarchy and Mise

7 Git and GitHub on Omarchy

8 Editor Integration on Omarchy

9 AI Coding Agents in Omarchy

10 Building an AI Development Workflow on Omarchy

11 Hyprland Configuration the Omarchy Way

12 Custom Keybindings on Omarchy

MODULE 00

00

From Bare Metal to a Ready Omarchy AI Development Workstation

This module takes you from an existing computer — potentially one currently running Windows — to a working, updated, verified Omarchy installation ready to become an AI/software-development workstation.

We will not assume that you already know Linux, Arch Linux, UEFI, bootloaders, disk encryption, Hyprland, or Linux package management.

About This Module

This module takes you from an existing computer — potentially one currently running Windows — to a working, updated, verified Omarchy installation ready to become an AI/software-development workstation.

We will not assume that you already know Linux, Arch Linux, UEFI, bootloaders, disk encryption, Hyprland, or Linux package management.

By the end of Module 0, you will have:

- installed Omarchy;
- understood the important installation decisions;
- configured a safe dual-boot setup if you need Windows;
- verified your hardware and boot configuration;
- connected networking, audio, and Bluetooth;
- configured your display;
- performed your first safe update;
- configured your preferred browser, terminal, editor, and coding agent;
- authenticated with GitHub;
- prepared the foundation for development runtimes;
- verified that the system is healthy;
- understood how recovery works before you start customizing the machine.

This is our **known-good baseline**.

Everything we do in later modules builds on it.

Important: Version and Documentation Policy

Omarchy evolves quickly.

This handbook was developed and tested using **Omarchy 4.x / Quattro**, including a real workstation running Omarchy `4.0.3-1`, and was cross-checked against the current official Omarchy documentation in September 2026.

The goal of this handbook is **not** to hard-code one exact Omarchy release.

Instead, you will learn how to verify commands and system behavior on your own installation.

Whenever possible, this handbook prioritizes:

1. the current official Omarchy Manual;

2. the current official Omarchy GitHub repositories;
3. the behavior of the current Omarchy CLI itself;
4. upstream documentation for the underlying component when Omarchy's documentation does not cover something.

Third-party tutorials should not be treated as authoritative when they disagree with the current Omarchy installation or CLI.

0.1 — What You Are Actually Installing

What You'll Learn

Before installing anything, you should understand what Omarchy is and what makes it different from a traditional desktop Linux distribution.

Why This Matters

If you expect Omarchy to behave like Windows, macOS, Ubuntu, GNOME, or KDE, some of its design choices may initially feel strange.

They make much more sense when you understand the philosophy.

What Is Omarchy?

Omarchy is an opinionated Linux distribution built around:

- **Arch Linux** as the underlying operating system;
- **Hyprland** as the Wayland compositor/window manager;
- **Quickshell** as the foundation of the modern Omarchy desktop shell;
- a keyboard-driven workflow;
- terminal-based development tools;
- integrated themes;
- development tooling;
- AI coding-agent integration;
- configuration through plain files;
- a unified `omarchy` command-line interface.

You are therefore not simply installing "Arch with a theme."

Omarchy adds an opinionated system layer around Arch, including:

- application installation;
- hardware configuration;

- defaults;
- themes;
- updates;
- recovery;
- desktop controls;
- coding agents;
- development environments;
- shell extensions;
- system helpers.

One of the most important concepts throughout this handbook will be:

```
Arch Linux
    ↓
Omarchy system layer
    ↓
Hyprland + Omarchy Shell
    ↓
Your personal configuration
```

We will explore this properly later.

For now, just remember:

> ***Use Omarchy's own tools when Omarchy provides a tool for the job.***

Do not immediately copy generic Arch commands from random tutorials.

Learning Philosophy

This course follows a practical principle:

> ***Do it first. Understand it deeply afterwards.***

We will explain enough theory to make safe decisions during installation.

Later, in the dedicated Linux concepts module, we will connect everything together properly.

You do **not** need to understand the complete Linux boot process before installing Linux.

0.2 — Before You Touch the Computer

What You'll Learn

How to prepare before installing Omarchy.

Why This Matters

Operating-system installation is one of the rare computer tasks where selecting the wrong disk can destroy all data on that disk.

Do this preparation before opening your BIOS or booting the installer.

Step 1 — Decide What Kind of Installation You Want

There are three common scenarios.

OPTION A — DEDICATED OMARCHY COMPUTER

Omarchy receives the entire computer or entire installation drive.

This is the simplest setup.

```
Drive
└── Omarchy
```

Choose this when:

- you do not need Windows;
- the drive contains nothing you need;
- this machine will become a Linux workstation.

OPTION B — SEPARATE PHYSICAL DRIVES

Example:

```
NVMe 1
└── Windows

NVMe 2
└── Omarchy
```

This is often the cleanest dual-boot arrangement when the hardware allows it.

The operating systems remain physically separated.

OPTION C — WINDOWS AND OMARCHY ON THE SAME DRIVE

Example:

```
NVMe
├── Windows
├── Windows data
├── Unallocated free space
└── Omarchy
```

Omarchy's official installer supports a **Free space install** specifically for this situation.

We will cover it separately.

Step 2 — Back Up Important Files

Before modifying partitions or installing an operating system, make sure anything important exists somewhere else.

Examples:

- source code;
- SSH keys;
- documents;
- password databases;
- browser data;
- photos;
- project files;
- local databases;
- game saves;
- application configuration;
- encryption/recovery keys.

A backup on another partition of the **same physical drive** is not sufficient protection against selecting the wrong drive during installation.

Prefer:

- external drive;
- NAS;
- another computer;
- trusted cloud storage.

Step 3 — If Windows Is Installed, Check BitLocker

This is particularly important for dual boot.

On current Windows versions, this may appear under:

```
Settings
→ Privacy & Security
→ Device encryption
```

The exact Windows interface can change between releases and editions.

BEFORE DISABLING BITLOCKER

Make sure you have access to your Windows recovery information.

Do not begin repartitioning an encrypted Windows installation without understanding its current encryption state.

IMPORTANT

Turning the BitLocker switch off does not necessarily mean the drive is instantly decrypted.

Wait for Windows to complete the decryption process.

Step 4 — Identify Your Drives

Before installation, know what hardware is inside your machine.

Write down:

- number of SSDs/NVMe drives;
- approximate capacity of each;
- which drive contains Windows;
- which drive you intend to use for Omarchy.

Example:

```
Drive 1 — Samsung 980 Pro — 1 TB — Windows
Drive 2 — WD Black SN850X — 2 TB — Omarchy target
```

This simple note can prevent an expensive mistake.

0.3 — Hardware Preparation

Desktop or Laptop?

Omarchy can be used on both.

Some later controls are hardware-specific.

DESKTOP EXAMPLES

- external monitor configuration;
- DDC brightness;
- dedicated GPU;
- wired Ethernet;
- external keyboard/mouse.

LAPTOP EXAMPLES

- touchpad;
- touchscreen;
- internal display;
- hybrid GPU;
- suspend;
- hibernation;
- battery/power profiles;
- fingerprint reader.

We will separate desktop and laptop-specific instructions wherever necessary.

CPU and GPU

You do not need to manually install a separate operating system before installing Omarchy.

Because hardware support can change, this handbook will not maintain a static "supported GPU list."

Instead, use the current Omarchy documentation and installer when checking unusual or very new hardware.

Virtualization

If you intend to:

- run virtual machines;

- use certain containerized VM solutions;
- run a Windows VM inside Omarchy;

enable CPU virtualization in your firmware.

Depending on platform, the option may appear as:

```
Intel VT-x
Intel Virtualization Technology

or

AMD-V
SVM Mode
```

The exact name is motherboard-dependent.

0.4 — Understanding UEFI Before Installation

What You'll Learn

Enough firmware terminology to safely boot the Omarchy installer.

What Is UEFI?

UEFI is firmware that runs before your operating system.

Very simplified:

```
Power button
    ↓
UEFI firmware
    ↓
Bootloader
    ↓
Operating system
```

People still commonly call the firmware setup screen "the BIOS," even though modern systems generally use UEFI.

Throughout this handbook, we may say **BIOS/UEFI settings** when referring to that firmware configuration interface.

How to Enter UEFI Setup

The key varies by manufacturer.

Common keys include:

```
Delete
F2
F10
F12
Esc
```

Desktop motherboards commonly use `Delete` or `F2`.

Laptops vary significantly.

Do not guess when changing unknown firmware settings. Consult the motherboard or laptop manufacturer's documentation when necessary.

0.5 — Secure Boot and TPM

This is an area where following **current documentation** matters.

Because firmware implementations and Omarchy's boot architecture can change over time, always check the current Omarchy installation documentation before installing.

Secure Boot

Secure Boot verifies trusted boot components before they are allowed to execute.

For the current Omarchy installation process:

```
Secure Boot → Disable for installation
```

The option is normally under sections such as:

```
Boot
Security
Authentication
Secure Boot
```

The exact location depends on the computer.

TPM

TPM stands for **Trusted Platform Module**.

It is used by operating systems including Windows for security features such as device encryption and credential protection.

IMPORTANT FOR WINDOWS USERS

Do not randomly change TPM-related settings on an existing encrypted Windows machine without first ensuring you have your Windows recovery information.

Checkpoint

Before continuing:

- [ ] Important files are backed up.
- [ ] You know which drive will receive Omarchy.
- [ ] Windows BitLocker is handled if dual booting.
- [ ] You can enter UEFI.
- [ ] Secure Boot is configured according to the current Omarchy installation requirements.
- [ ] You have a wired or 2.4 GHz keyboard available if using encryption.

0.6 — Why You Should Have a Wired or 2.4 GHz Keyboard

This detail is easy to overlook.

Omarchy uses drive encryption by default.

During the earliest boot stage, Bluetooth is not yet available to enter the disk-unlock password.

The official Omarchy documentation therefore recommends using:

- a wired keyboard; or
- a keyboard with a 2.4 GHz USB receiver.

A Bluetooth-only keyboard cannot be relied upon for entering the encryption password during startup.

This is especially important for compact wireless keyboards.

0.7 — Download the Omarchy ISO

What Is an ISO?

An ISO is a disk image containing the operating-system installer.

The current official installation method for Omarchy is the Omarchy ISO.

Step 1 — Download from the Official Omarchy Site

Use the download provided by the official Omarchy project.

Avoid downloading installation images from:

- mirrors linked by random tutorials;
- file-sharing sites;
- forum attachments;
- unofficial repackaged images.

Step 2 — Optional but Recommended: Verify the Download

Omarchy publishes signatures for its ISO releases.

This allows you to verify that:

1. the ISO was signed by the expected Omarchy key;
2. the downloaded image was not modified.

For a beginner's first installation, do not paste an unfamiliar signature-verification command from an old guide. Follow the current official signature instructions and compare the fingerprint against the current official security documentation.

Screenshot 0.7-A
Official Omarchy download page

0.8 — Create the Bootable USB

You will normally need:

- an Omarchy ISO;
- a USB flash drive;

- approximately several minutes.

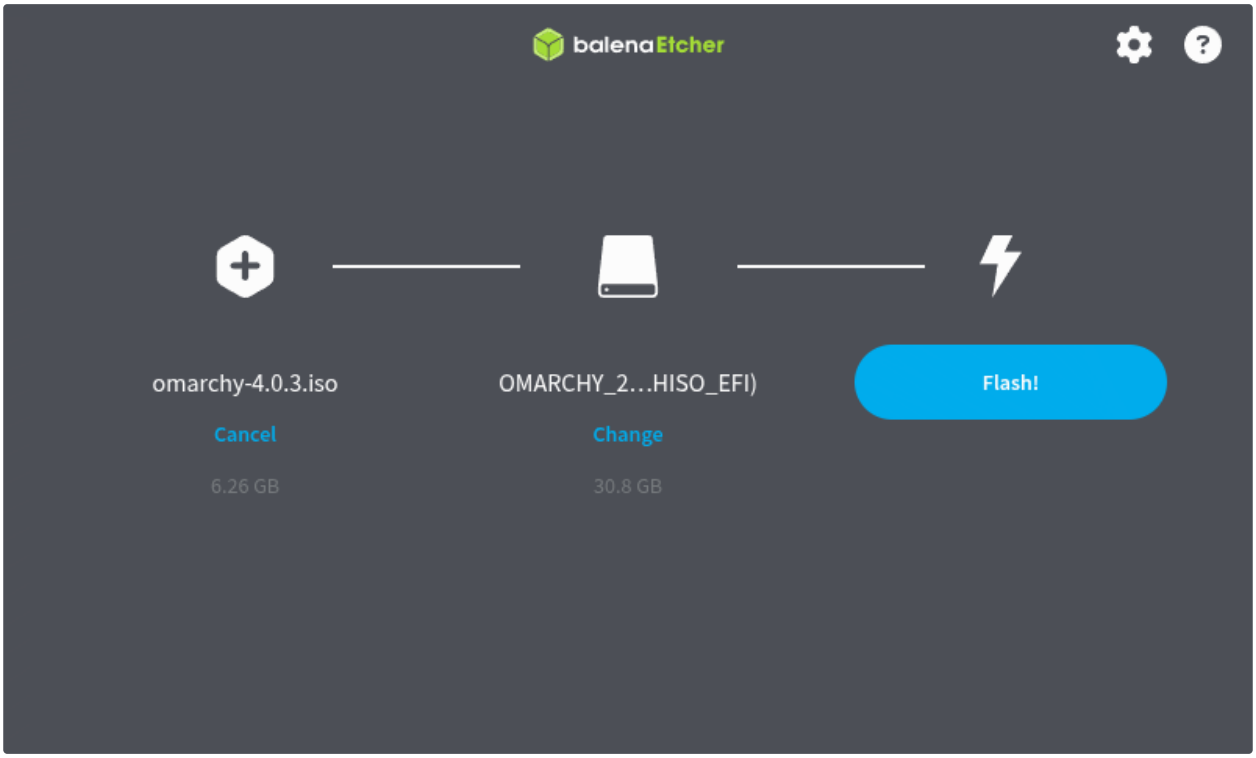
Everything currently on the USB may be erased.

From Windows or macOS

The current Omarchy installation guide recommends **balenaEtcher**.

The basic process is:

```
Select image
→ Select USB drive
→ Flash
→ Wait for verification
```



Screenshot 0.8-A
balenaEtcher with Omarchy ISO selected

From Linux

The current official Omarchy installation guide recommends **caligula** for Linux.

Follow the current installation instructions for caligula rather than copying an old package command from a tutorial.

Do Not Simply Copy the ISO File

This:

```
USB/
└── omarchy.iso
```

is not the same thing as creating a bootable installer.

The image-writing tool writes the ISO's filesystem and boot structure onto the USB device.

Checkpoint

You should now have:

- [ ] official Omarchy ISO;
- [ ] bootable USB;
- [ ] backups;
- [ ] known target drive;
- [ ] appropriate BIOS/UEFI settings.

0.9 — Boot from the USB

Restart the computer.

Open the firmware boot menu.

Common boot-menu keys include:

```
F8
F11
F12
Esc
```

The exact key is manufacturer-specific.

Choose the USB's **UEFI** boot entry when available.

Example:

```
UEFI: SanDisk USB
```

rather than a legacy/non-UEFI entry.

If the USB Does Not Appear

Check:

1. Is the USB physically detected?
2. Was the ISO actually flashed rather than copied?
3. Is UEFI USB boot enabled?
4. Did you create the USB successfully?
5. Are Secure Boot settings compatible with the current Omarchy installer?
6. Can you try another USB port?

On desktop systems, rear motherboard USB ports are often a good troubleshooting choice.

0.10 — The Omarchy Installer

Once the USB boots, the Omarchy installer will guide you through configuration.

Read every screen.

Do not automatically press Enter through disk-selection screens.

Screenshot 0.10-A
Omarchy installer first screen

Screenshot 0.10-B

Keyboard/locale configuration

0.11 — Choose the Correct Installation Mode

This is the most important installation decision.

Path A — Full-Disk Installation

Choose this if the selected disk can be completely erased.

Conceptually:

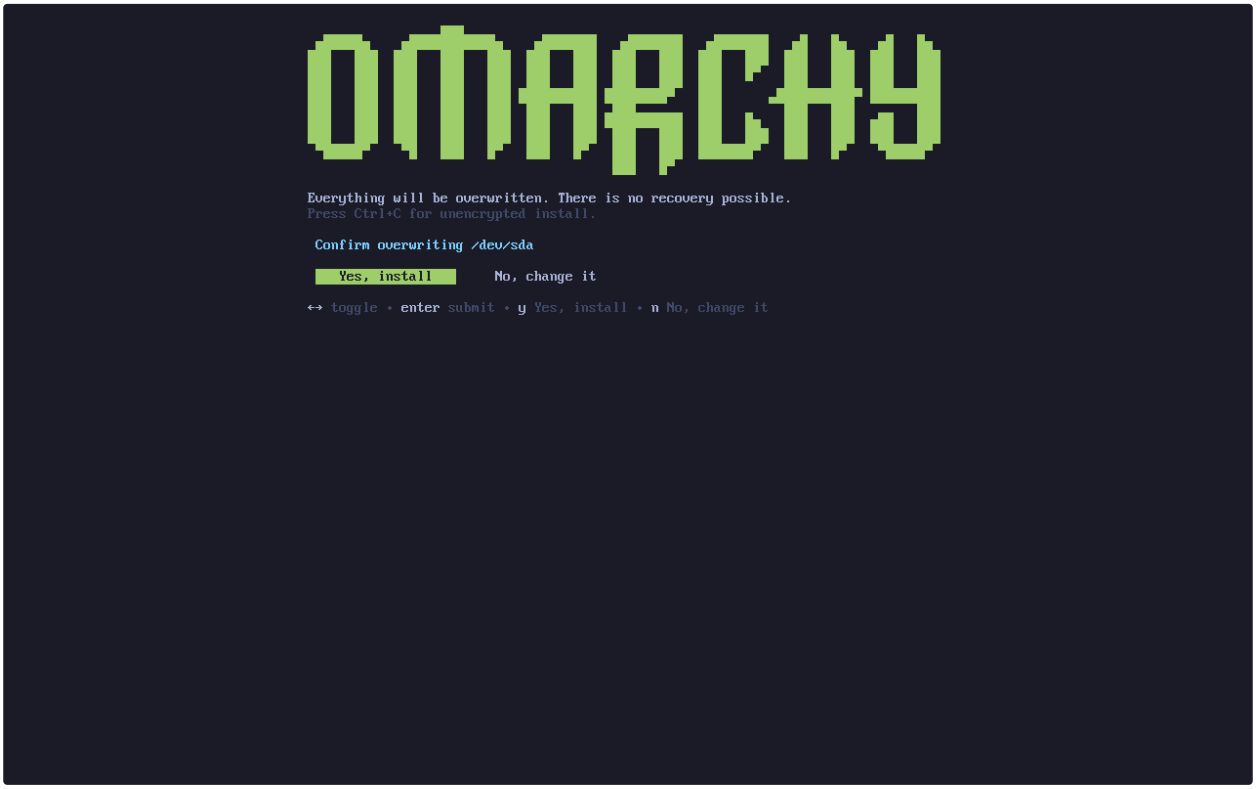
```
Before

Disk
├── Existing OS
├── Existing files
└── Existing partitions

After

Disk
└── Omarchy
```

STOP AND VERIFY

Before confirming:

- correct disk model;
- correct disk capacity;
- no important data on it.

Screenshot 0.11-A
Omarchy installer disk selection



Screenshot 0.11-B
Full-disk installation option

0.12 — Dual Boot with Windows

If you need Windows and Omarchy on the same computer, do not choose full-disk installation on the Windows drive.

The current Omarchy installer supports a **Free space install**.

Conceptually:

```
Disk

├── Windows partitions
├── Windows recovery partitions
└── Free / unallocated space
        ↓
     Omarchy
```

Preparing Free Space

Resize Windows from Windows before installation if necessary.

You want genuine:

`Unallocated space`

not simply an empty Windows folder.

Do not manually delete unfamiliar EFI or recovery partitions.

During Installation

After selecting the correct disk:

Choose:

`Free space install`

rather than:

`Full disk`

Confirm that the installer is using the intended free space.

Screenshot 0.12-A
Free space install option

0.13 — Disk Encryption

Omarchy installs with encryption by default.

What Encryption Protects

If the computer is stolen while powered off, the files on the encrypted drive cannot simply be read by removing the SSD and connecting it to another machine.

You unlock the encrypted system during boot using your password.

Should You Disable Encryption?

For the overwhelming majority of developer workstations:

No.

STEVINATOR RECOMMENDATION

> ***Use Omarchy's default encrypted installation unless you have a specific technical reason not to.***

Screenshot 0.13-A
Encryption option/confirmation

0.14 — Complete the Installation

Continue through the installer's personal settings.

These may include:

- keyboard;
- username;
- password;
- hostname;
- timezone.

Read the current prompts rather than relying on screenshots from an older installer release.

After confirmation, allow installation to finish.

Do not:

- remove power;
- reset the machine;
- unplug the target drive;
- interrupt installation.

When prompted to reboot:

1. complete the installation;
2. reboot;
3. remove the installer USB when appropriate.

Screenshot 0.14-A
Installation progress

0.15 — First Boot

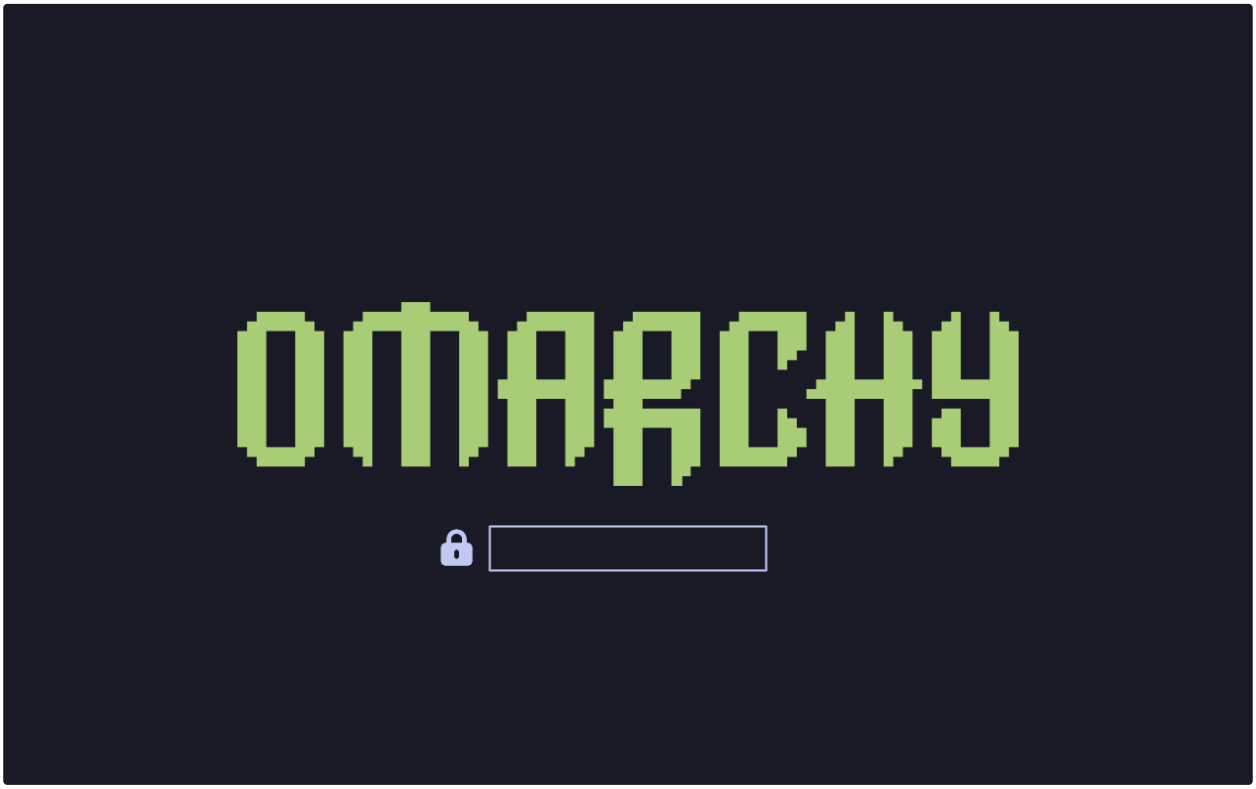
You should now reach your installed Omarchy system.

If encryption is enabled, the machine will ask for the disk-unlock password during boot.

This happens before the full desktop environment starts.

This is why your wired/2.4 GHz keyboard matters.

Screenshot 0.15-A
Limine boot screen



Screenshot 0.15-B
Disk decryption screen

What You Are Booting Through

On current standard Omarchy installations, Limine is the default bootloader.

A simplified modern Omarchy boot path looks like:

```
UEFI
  ↓
Limine
  ↓
Omarchy UKI
  ↓
Linux kernel
  ↓
Userspace
  ↓
Hyprland
  ↓
Omarchy Shell
```

Do not worry if most of those terms are unfamiliar.

We will explain them properly later.

0.16 — Your First Five Minutes in Omarchy

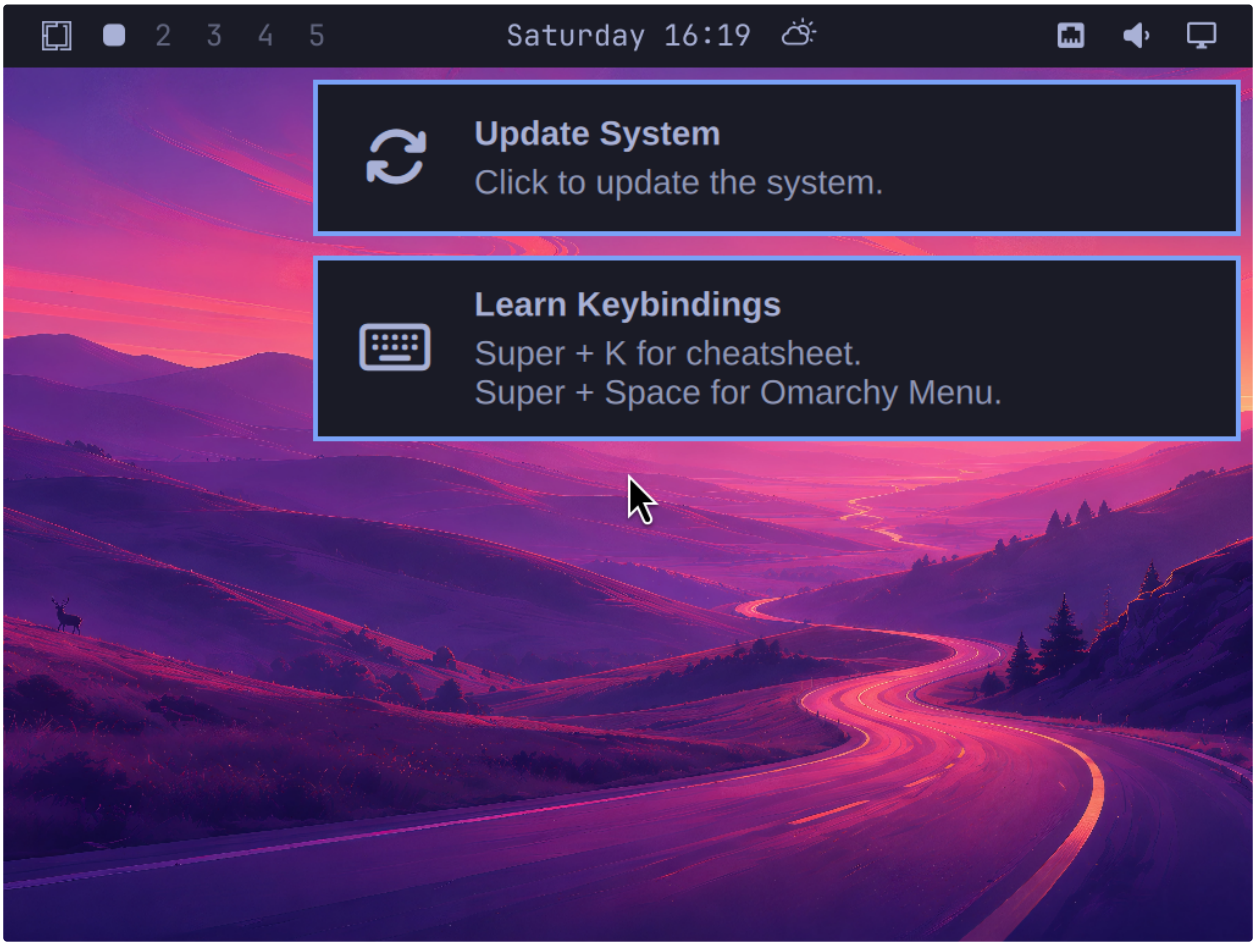
Omarchy is designed around the keyboard.

The first shortcut you should know is:

```
Super + Space
```

This opens the Omarchy Menu.

`Super` is normally the Windows key on a PC keyboard.



Screenshot 0.16-A
First Omarchy desktop

Open a Terminal

Press:

```
Super + Return
```

Open the Browser

Press:

```
Super + Shift + Return
```

You should see the browser tile alongside the terminal.

Show Available Keybindings

Press:

```
Super + K
```

Do not attempt to memorize dozens of shortcuts now.

Remember just this:

> ***When you forget a keybinding, ask Omarchy.***

0.17 — Verify the Omarchy Installation

Open a terminal.

Run:

```
omarchy version
```

IMPORTANT

Use:

```
omarchy version
```

not:

```
omarchy --version
```

Discover the CLI

Run:

```
omarchy
```

Then:

```
omarchy commands
```

For the complete command surface:

```
omarchy commands --all
```

This is one of the most important habits in the handbook.

Course Rule

Whenever documentation and your installed system appear to disagree:

1. check your version;
2. check the current official manual;
3. inspect the installed CLI;
4. verify actual system behavior.

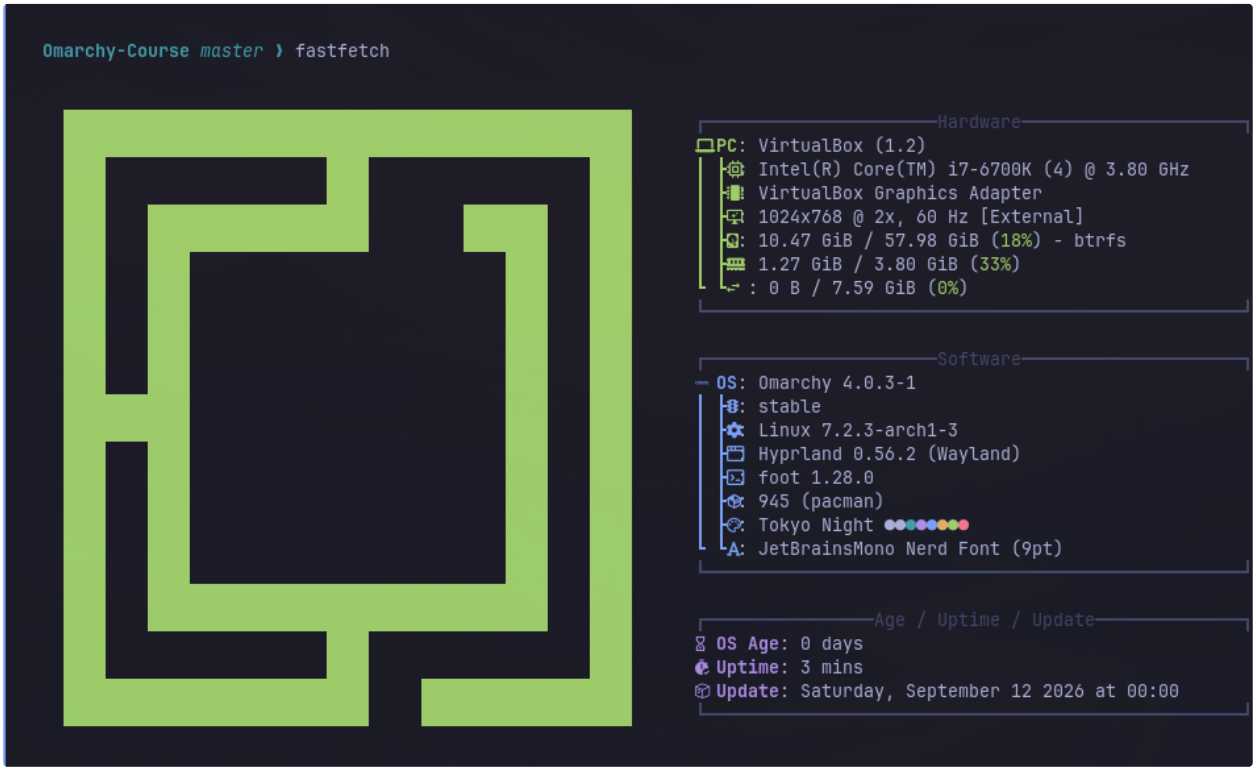
Never blindly assume.

0.18 — Inspect the Workstation

On current releases:

```
omarchy launch about
```

Use this as a quick visual health check.



Screenshot 0.18-A
Omarchy About screen

0.19 — Verify the Bootloader

Check:

```
bootctl status
```

Look for:

```
Current Boot Loader:
```

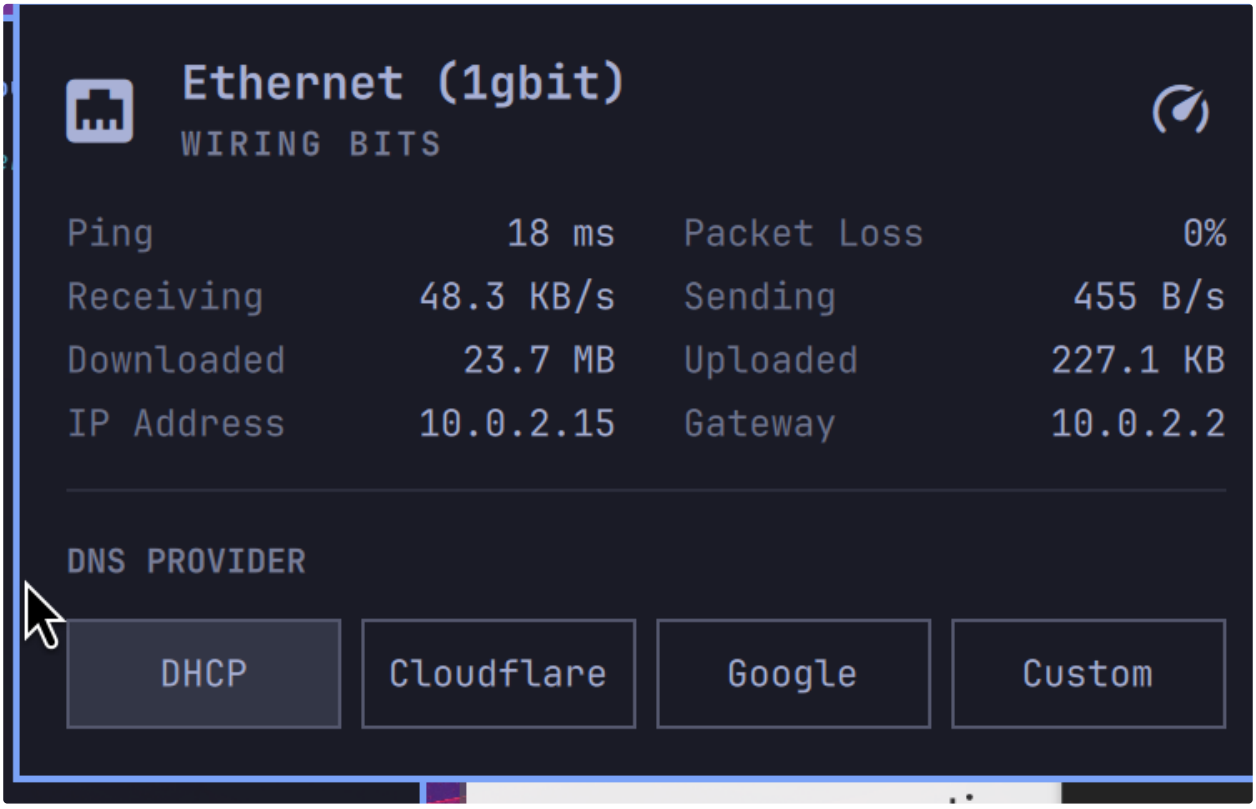
A standard current Omarchy installation should normally report Limine.

Example:

```
Current Boot Loader:
    Product: Limine
```

0.20 — Connect to the Network

Wi-Fi



Screenshot 0.20-A
Wi-Fi panel

Press:

```
Super + Ctrl + W
```

Select your network and connect.

Ethernet

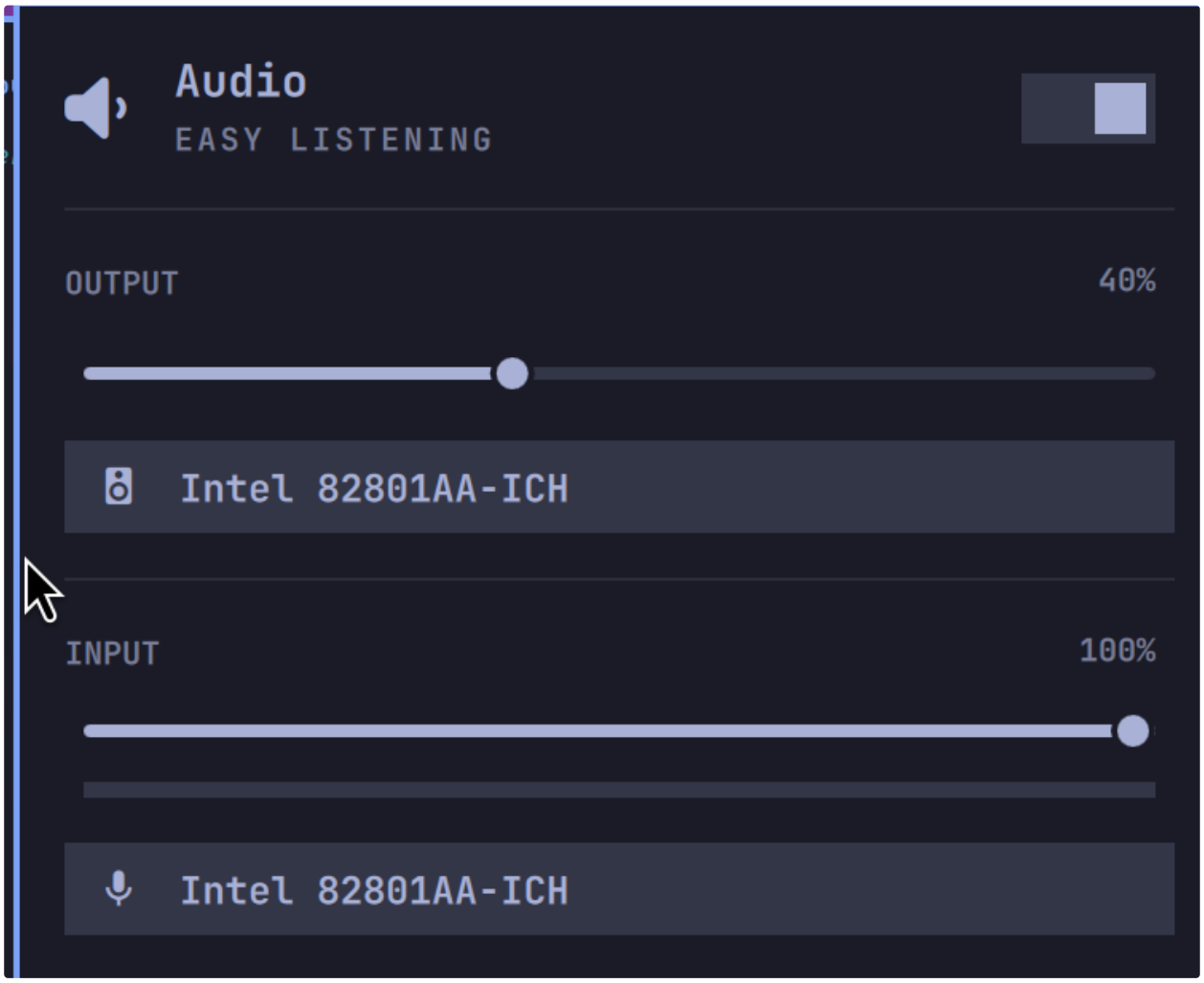
Connect the cable.

Verify from the Terminal

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

0.21 — Check Audio

Open:



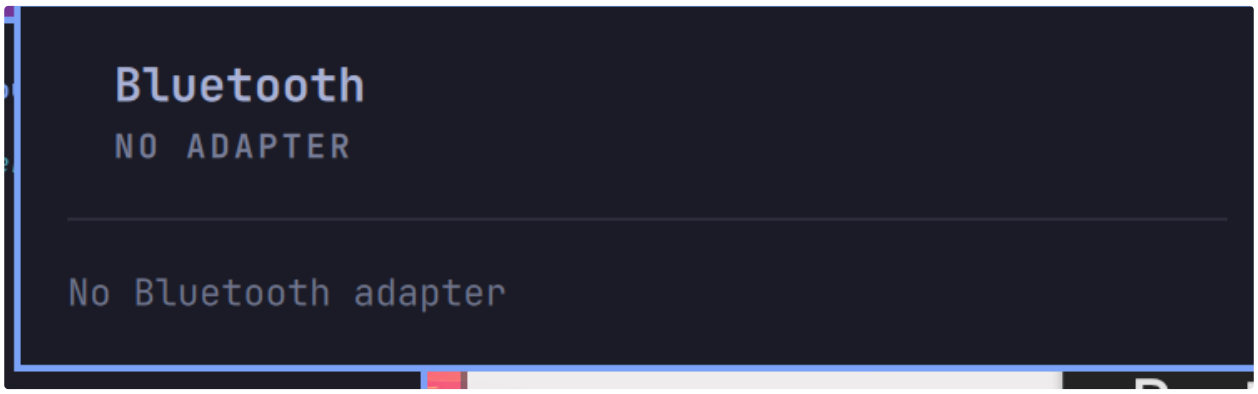
Screenshot 0.21-A
Audio panel

```
Super + Ctrl + A
```

Confirm the expected output device appears and sound works.

0.22 — Check Bluetooth

Open:



Screenshot 0.22-A
Bluetooth panel

```
Super + Ctrl + B
```

If you use Bluetooth:

1. ensure Bluetooth is powered on;
2. pair the device;
3. connect;
4. verify operation.

0.23 — Check Your Display Properly

Run:

```
hyprctl monitors all
```

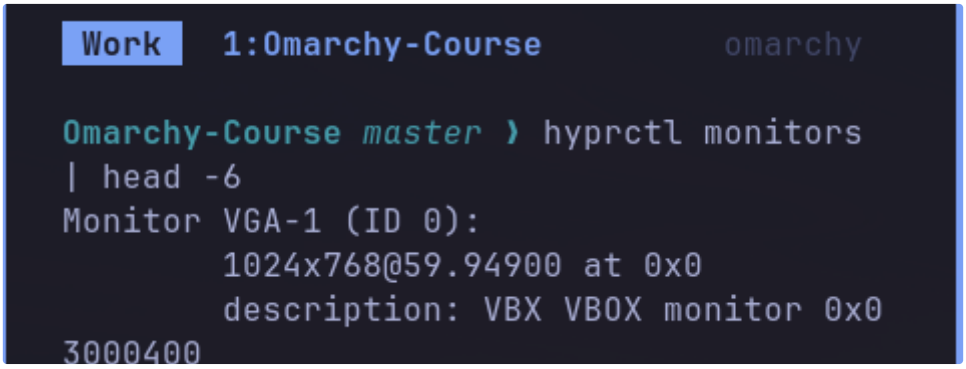
Look for:

- output name;
- current resolution;
- current refresh rate;
- available modes;
- scale.

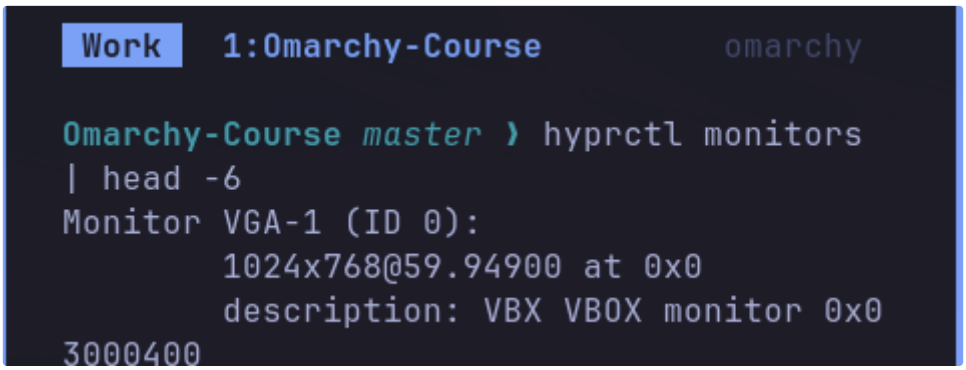
Example desired mode:

```
2560x1440@144
```

A monitor displaying an image does not mean it is running at its best settings.



Screenshot 0.23-A
Example 60 Hz configuration



Screenshot 0.23-B
Example 144 Hz configuration

0.24 — Check Keyboard and Mouse Behavior

Omarchy input customization lives in:

```
~/.config/hypr/input.lua
```

Verify:

- keyboard layout;
- Super key;
- mouse movement;
- scrolling;
- Caps Lock behavior;
- numpad if relevant.

0.25 — Compact and Unusual Keyboards

Before assuming an Omarchy shortcut is broken, verify what key event the keyboard actually sends.

Useful tool:

```
wev
```

LESSON

> ***Diagnose the input event before changing the operating system.***

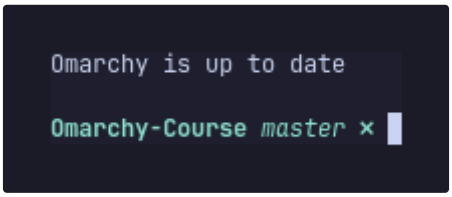
0.26 — Perform the First Omarchy Update

Check whether an update is available:

```
omarchy update available
```

Then perform the update through the Omarchy update workflow.

Prefer Omarchy's update tooling over blindly running generic Arch upgrade commands.



Screenshot 0.26-A
`omarchy update available`

0.27 — Understand the Automatic Snapshot

Omarchy creates system snapshots around updates.

But remember:

```
Snapshot ≠ personal backup
```

A system snapshot restores the system root state, not your personal `/home` files.

0.28 — Reboot After the Update if Required

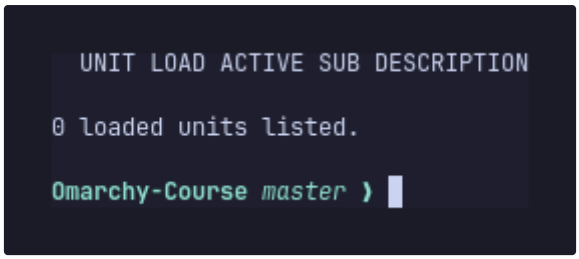
Check:

```
omarchy update restart
```

If Omarchy indicates a reboot is required, reboot.

0.29 — Verify System Health

```
systemctl --failed
```



Screenshot 0.29-A
`systemctl --failed`

Then:

```
systemctl --user --failed
```

A healthy baseline should ideally show no failed units.

0.30 — Do Not Panic Over Every Log Message

Later, when troubleshooting, we may use:

```
journalctl -p 3 -b
```

But logs require interpretation.

Use this methodology:

```
symptom
→ relevant component
→ service state
→ logs
→ reproduce
→ fix
```

not:

```
see red text
→ panic
```

0.31 — Choose Your Default Terminal

For this course our recommended AI/developer workstation uses:

Ghostty

but this is a recommendation, not a requirement.

Inspect the current choices with:

```
omarchy default terminal --help
```

0.32 — Choose Your Browser

Our baseline is:

Chromium

Inspect supported choices with:

```
omarchy default browser --help
```

0.33 — Choose Your Editor

Our baseline is:

```
Neovim
```

Inspect current choices:

```
omarchy default editor --help
```

0.34 — Choose Your Default Coding Agent

Inspect the available agents:

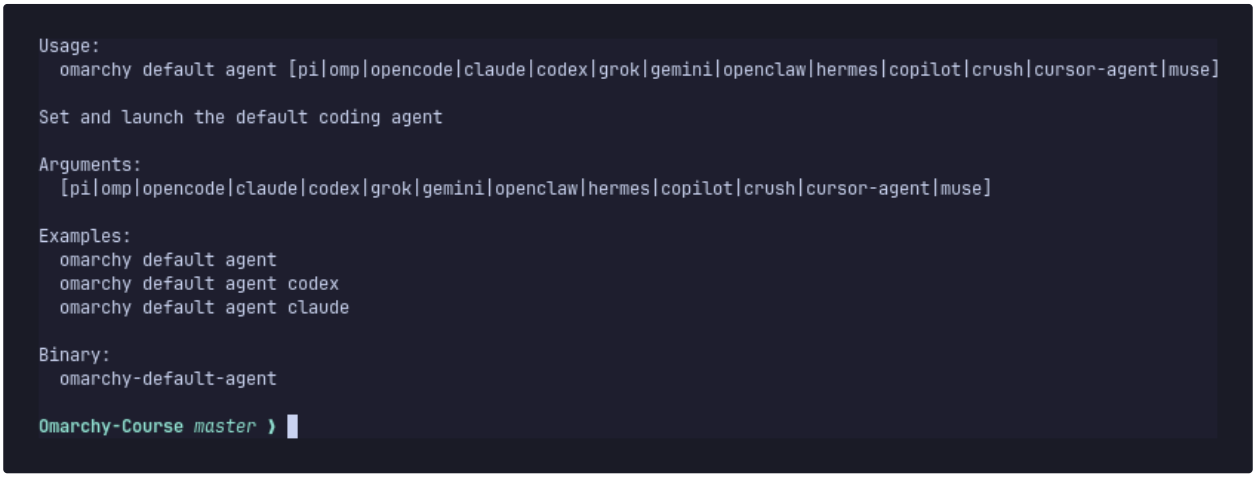
```
omarchy default agent --help
```

Stevinator Baseline

For the workstation used while building this handbook, the selected default coding agent is:

```
OMP
```

This course teaches the **Omarchy integration**, not one specific coding agent.



Screenshot 0.34-A
coding-agent selection

0.35 — Test the Coding-Agent Launcher

Current shortcut:

```
Super + Shift + Ctrl + A
```

The checkpoint is only:

> ***Can Omarchy launch your selected coding agent?***

0.36 — Git

Check:

```
git --version
```

0.37 — GitHub CLI

Check:

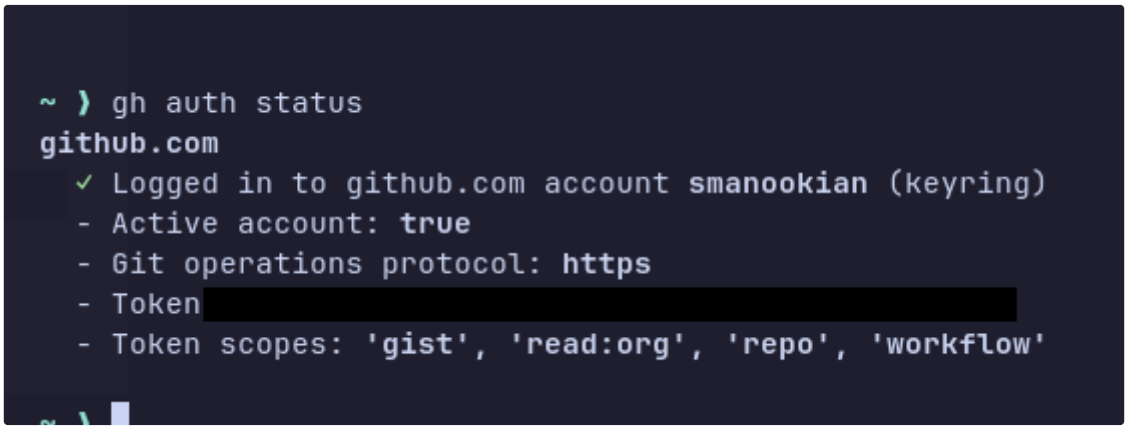
```
gh --version
```

Authenticate:

```
gh auth login
```

Verify:

```
gh auth status
```



Screenshot 0.37-A
Successful `gh auth status`

0.38 — Create Your Development Folder

```
mkdir -p ~/Projects
cd ~/Projects
pwd
```

Expected:

```
/home/YOUR_USERNAME/Projects
```

0.39 — Understand the Three Software Installation Layers

Layer 1 — System Packages

```
omarchy pkg
```

Layer 2 — Omarchy-Integrated Applications

```
omarchy install
```

Layer 3 — Developer Runtimes and Tools

```
mise
```

This separation keeps the workstation cleaner and more reproducible.

0.40 — Verify Mise

```
mise --version
```

The important concept:

```
System packages
       ≠
Development runtimes
```

0.41 — Establish the Runtime Baseline

Common environments:

- Node.js;
- Python;
- Bun.

Verify:

```
node --version
python --version
bun --version
```

If one is missing, do not randomly install it with `sudo pacman`.

0.42 — Why Project-Specific Versions Matter

Example:

```
Project A
requires Node 24

Project B
requires Node 26
```

With mise:

```
~/Projects/project-a/mise.toml
→ Node 24

~/Projects/project-b/mise.toml
→ Node 26
```

0.43 — First Manual Snapshot

Inspect:

Screenshot 0.43-A
Snapshot creation/update snapshot

```
omarchy snapshot --help
```

Then create a snapshot:

```
omarchy snapshot create
```

0.44 — What a Snapshot Will NOT Save

A snapshot does not replace:

- Git;
- cloud backup;
- external backup;
- dotfiles repository;
- source-control repositories.

0.45 — Know How to Reach Limine

You do **not** need to restore a snapshot now.

You only need to know:

> *"If an update breaks the system, I have a recovery path."*

0.46 — Security Baseline

A fresh Omarchy installation already includes important security decisions.

General rule:

> *Do not enable services you do not need.*

0.47 — Check the Firewall Conceptually

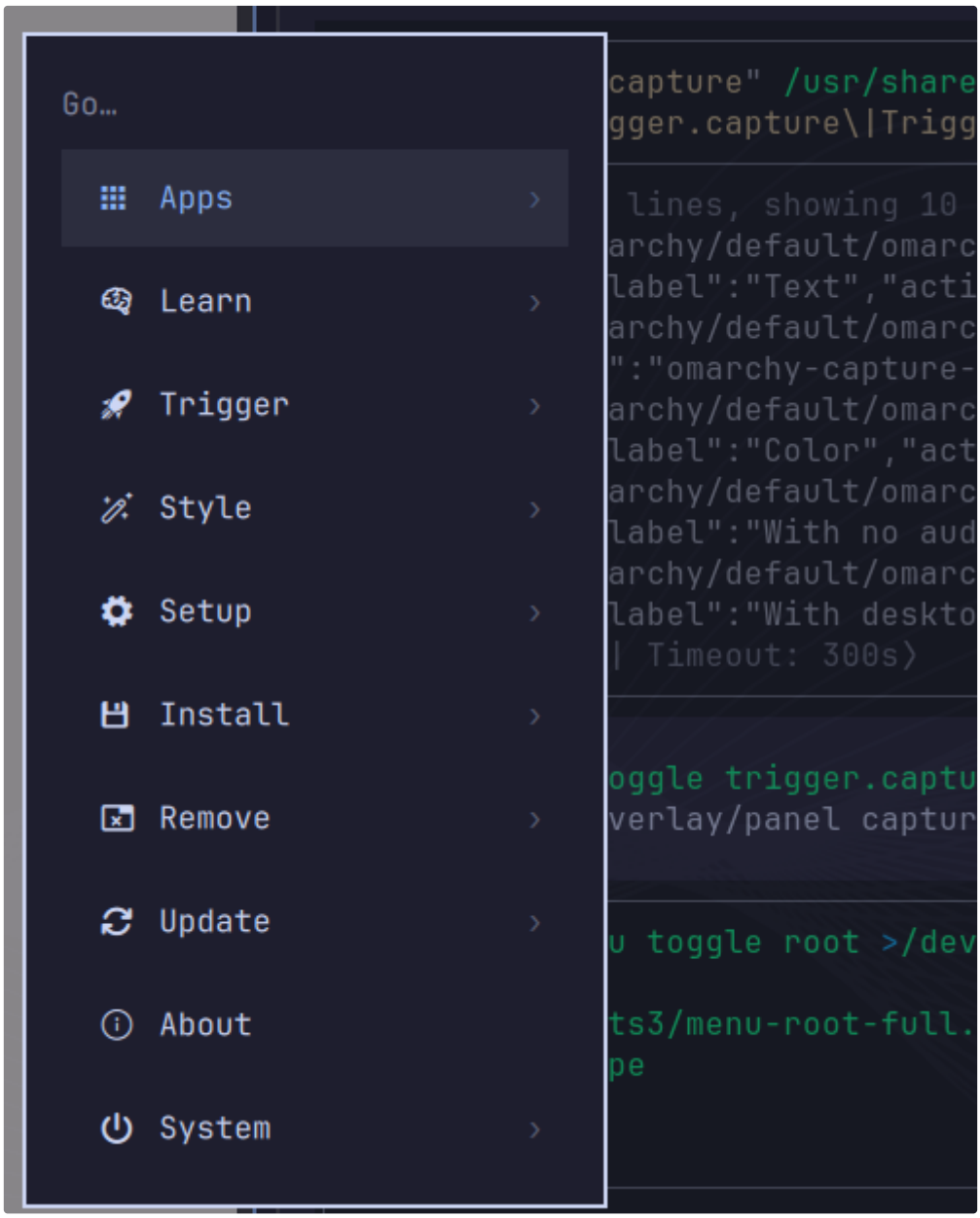
At this stage, the goal is simply to know:

```
Firewall exists
Incoming connections are restricted
SSH is not automatically exposed
```

0.48 — Test the Core Daily Shortcuts

OMARCHY MENU

```
Super + Space
```



Screenshot 0.48-A

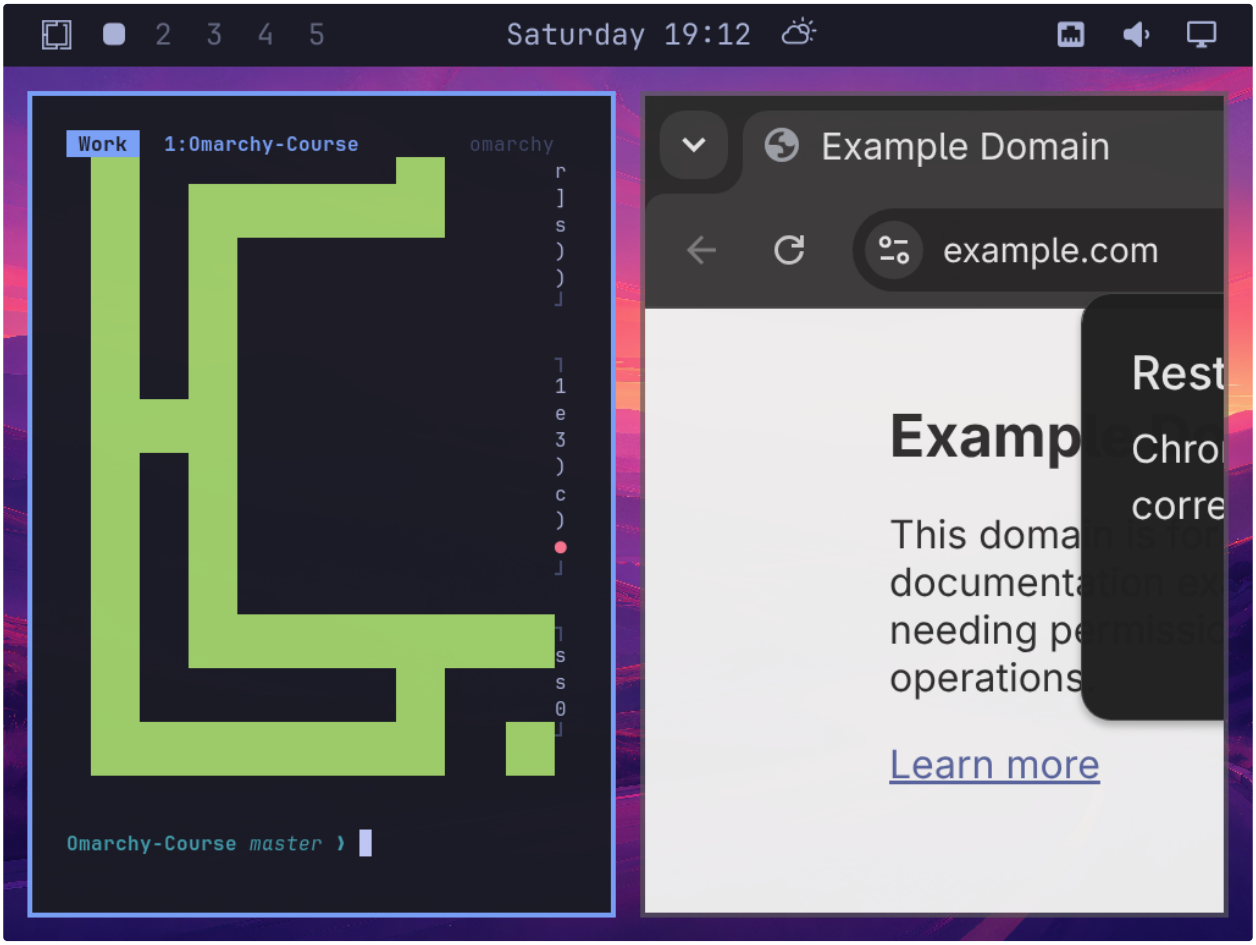
`Super + Space` Omarchy Menu

TERMINAL

`Super + Return`

BROWSER

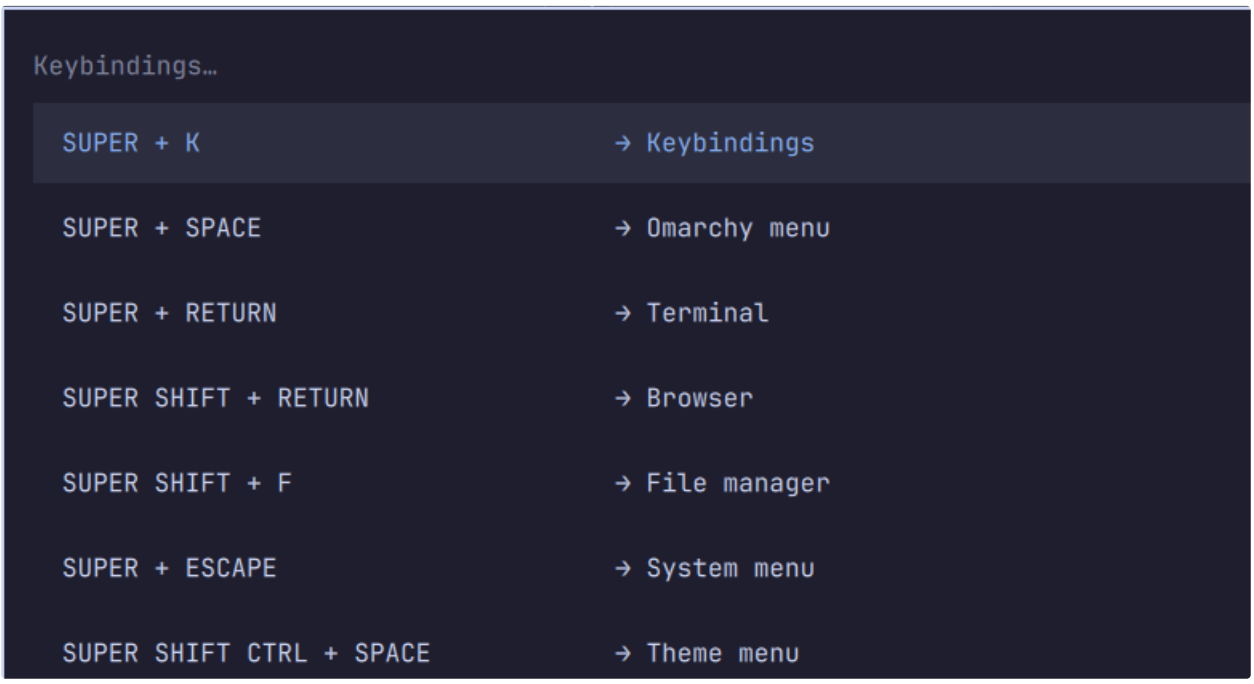
`Super + Shift + Return`



Screenshot 0.48-C
Terminal + browser tiling

KEYBINDING HELP

Super + K



Screenshot 0.48-B

`Super + K` keybinding search

AUDIO

```
Super + Ctrl + A
```

BLUETOOTH

```
Super + Ctrl + B
```

NETWORK

```
Super + Ctrl + W
```

DISPLAY

```
Super + Ctrl + D
```

Screenshot 0.48-G
Display panel

ACTIVITY MONITOR

```
Super + Ctrl + T
```

CAPTURE TOOLS

```
Super + Ctrl + C
```

EMOJI PICKER

```
Super + Ctrl + E
```

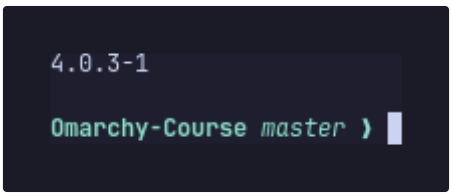
If one does not match your installation, inspect the actual binding with:

```
Super + K
```

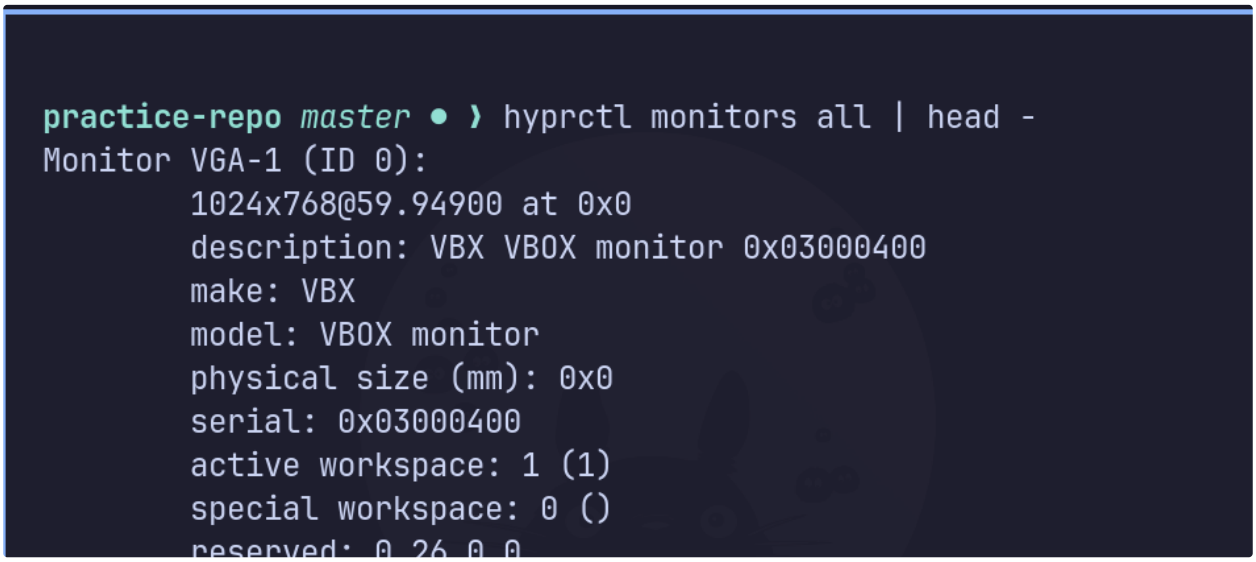
0.49 — Establish the Known-Good Baseline

Run:

```
omarchy version
systemctl --failed
systemctl --user --failed
omarchy network status --verbose
hyprctl monitors all
gh auth status
git --version
mise --version
node --version
python --version
bun --version
```



Screenshot 0.49-A
`omarchy version`



Screenshot 0.49-C
`hyprctl monitors all`

0.50 — Module 0 Practical Exercise

You should now be able to:

1. Open a terminal.
2. Find your Omarchy version.
3. Open the complete Omarchy command list.
4. Check your active network.
5. Identify your monitor.
6. Verify its refresh rate.
7. Check for failed services.
8. Check your GitHub authentication.
9. Check your development-runtime manager.
10. Open the Omarchy keybinding menu.

Suggested commands:

```
omarchy version
omarchy commands --all
omarchy network status --verbose
hyprctl monitors all
systemctl --failed
systemctl --user --failed
gh auth status
mise --version
```

Then:

```
Super + K
```

0.51 — Module 0 Checkpoint

Installation

- [ ] Omarchy boots successfully.
- [ ] The correct drive was used.
- [ ] Encryption works.
- [ ] I can enter my boot/decryption password.
- [ ] I know whether this is full-disk or dual-boot.

Windows / Dual Boot

If applicable:

- [ ] Windows still boots.
- [ ] BitLocker was handled according to current Omarchy requirements.
- [ ] I understand which partitions belong to which OS.

Firmware

- [ ] I know how to enter my UEFI/BIOS.
- [ ] I understand what Secure Boot is at a basic level.
- [ ] I know how to open the firmware boot menu.

Omarchy

- [ ] I know `Super + Space`.
- [ ] I know `Super + K`.
- [ ] I can open a terminal.
- [ ] I can open a browser.
- [ ] I know how to check `omarchy version`.
- [ ] I know how to discover Omarchy commands instead of guessing.

Hardware

- [ ] Wi-Fi/Ethernet works.
- [ ] Audio works.
- [ ] Bluetooth works if needed.
- [ ] Keyboard works.
- [ ] Mouse/trackpad works.
- [ ] Monitor resolution is correct.
- [ ] Monitor refresh rate is correct.

Development

- [ ] Git works.
- [ ] GitHub CLI works.
- [ ] GitHub authentication works.
- [ ] Mise works.
- [ ] I have a development-project directory.

- [ ] I understand that system packages and project runtimes are different things.

AI

- [ ] I understand that Omarchy supports a default coding agent.
- [ ] I have either chosen one or know where to choose one.
- [ ] I can launch the configured agent.

Recovery

- [ ] I know that Omarchy uses system snapshots.
- [ ] I understand that snapshots do not back up `/home`.
- [ ] I have established a known-good baseline.
- [ ] I know how to identify my bootloader.
- [ ] I understand that Limine is part of the normal current Omarchy recovery path.

0.52 — Common Installation Mistakes

Mistake 1 — Installing to the Wrong Drive

Know the drive model, capacity, and purpose before installation.

Mistake 2 — Leaving BitLocker Active for the Official Free-Space Dual-Boot Path

Check the current official Omarchy dual-boot documentation before proceeding.

Mistake 3 — Forgetting About Secure Boot

Always check the **current** installation requirements.

Mistake 4 — Using a Bluetooth-Only Keyboard

Keep a wired or 2.4 GHz keyboard available.

Mistake 5 — Assuming the Monitor Is Running at Maximum Refresh Rate

Verify:

```
hyprctl monitors all
```

Mistake 6 — Assuming a Shortcut from a Tutorial Must Be Correct

Use:

```
Super + K
```

or:

```
omarchy menu keybindings
```

Mistake 7 — Guessing Omarchy Commands

Use:

```
omarchy commands --all
```

and:

```
omarchy <command-group> --help
```

Mistake 8 — Treating Every Linux Tool as a Pacman Package Problem

Determine whether it belongs to:

```
system package
Omarchy-integrated application
mise-managed runtime/tool
```

Mistake 9 — Assuming the Bootloader

Verify:

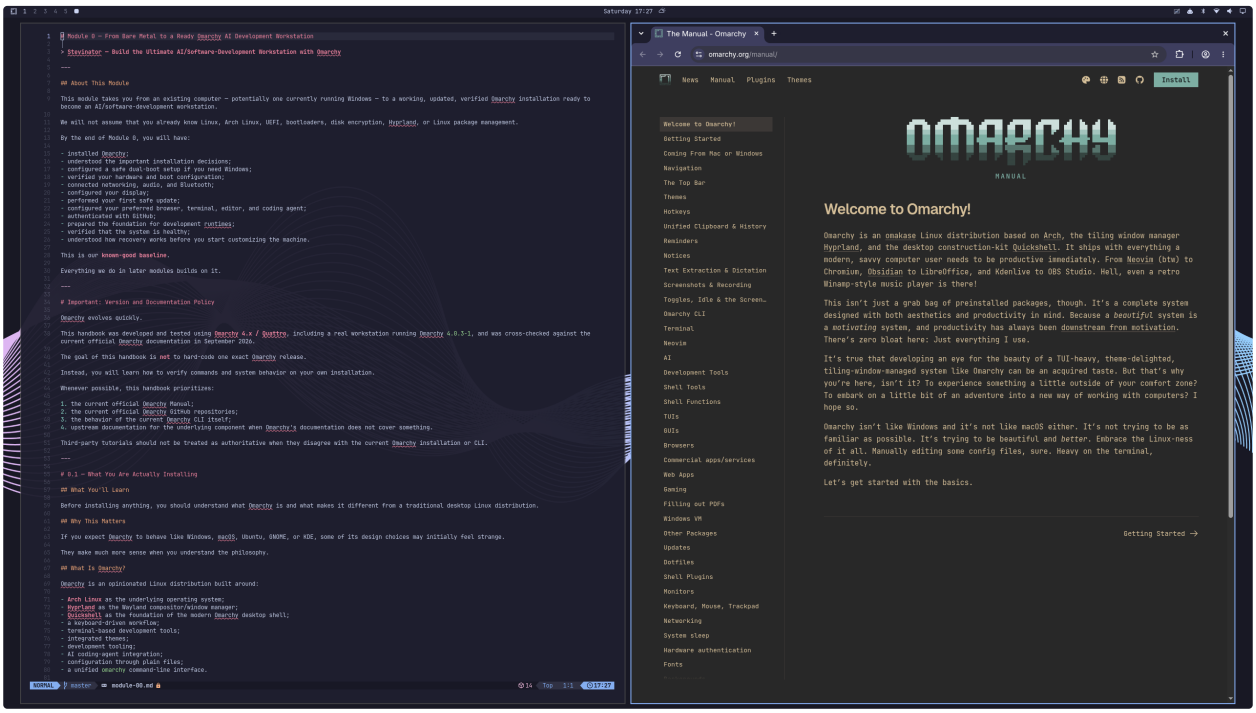
```
bootctl status
```

Mistake 10 — Thinking a Snapshot Is a Backup

It is not.

0.53 — What You Can Now Do

After completing Module 0, you can now:



Screenshot 0.53-A
final working workstation

- prepare a PC for Omarchy;
- distinguish full-disk and dual-boot installations;
- protect an existing Windows installation during planning;
- understand the role of BitLocker during dual boot;
- enter and configure UEFI;
- create the official Omarchy installer;
- install Omarchy;
- boot an encrypted Omarchy system;
- navigate the basic Omarchy desktop;
- discover commands and shortcuts;
- verify your Omarchy version;
- identify your bootloader;
- verify networking;
- verify audio;
- verify Bluetooth;
- inspect display resolution and refresh rate;
- perform a safe Omarchy update;

- understand automatic snapshots;
- recognize the difference between snapshots and backups;
- configure the foundations of a development workstation;
- authenticate with GitHub;
- understand why mise matters;
- select a coding agent;
- verify basic system health.

Most importantly:

> *You are no longer working on an unknown Linux installation.*

You now have a **known-good Omarchy baseline** from which we can deliberately build the AI/software-development workstation.

Module 0 Summary

The path we followed was:

```
Plan
  ↓
Back up
  ↓
Choose full-disk or dual-boot
  ↓
Prepare Windows if necessary
  ↓
Configure UEFI
  ↓
Create Omarchy USB
  ↓
Install
  ↓
Boot
  ↓
Verify
  ↓
Update
  ↓
Check hardware
  ↓
Configure core defaults
  ↓
Prepare Git/GitHub
  ↓
Prepare developer tooling
  ↓
Understand snapshots
  ↓
Verify system health
  ↓
Known-good baseline
```

MODULE 01

01

How Omarchy Actually Works

Module 0 gave you a working Omarchy workstation.

Module 1 gives you the mental model needed to understand what you installed.

About This Module

Module 0 gave you a working Omarchy workstation.

Module 1 gives you the mental model needed to understand what you installed.

This is important because Omarchy is not just "Arch Linux with Hyprland" and it is not a traditional desktop environment like Windows, macOS, GNOME, or KDE.

Omarchy combines several layers:

- Arch Linux
- Hyprland
- Quickshell
- Omarchy packages
- Omarchy configuration
- the Omarchy CLI
- your own user configuration

Once you understand where each layer begins and ends, customization becomes much safer.

You also stop treating every problem as:

> *"Linux is broken."*

Instead, you learn to ask:

> *"Which layer is responsible for this behavior?"*

That question will save you a huge amount of time later.

What You Will Learn

By the end of this module, you will understand:

- what Omarchy actually is;
- how Arch Linux fits underneath it;
- what Hyprland does;
- what Quickshell does;
- what the Omarchy shell is;
- what the `omarchy` CLI controls;

- where Omarchy's own files live;
- where your personal configuration lives;
- why you should not edit files under `/usr/share/omarchy`;
- how Omarchy loads defaults and personal overrides;
- how menus, keyboard shortcuts, and CLI commands relate to each other;
- how to inspect your installed system instead of guessing;
- why version-aware troubleshooting matters.

1.1 — The Mental Model

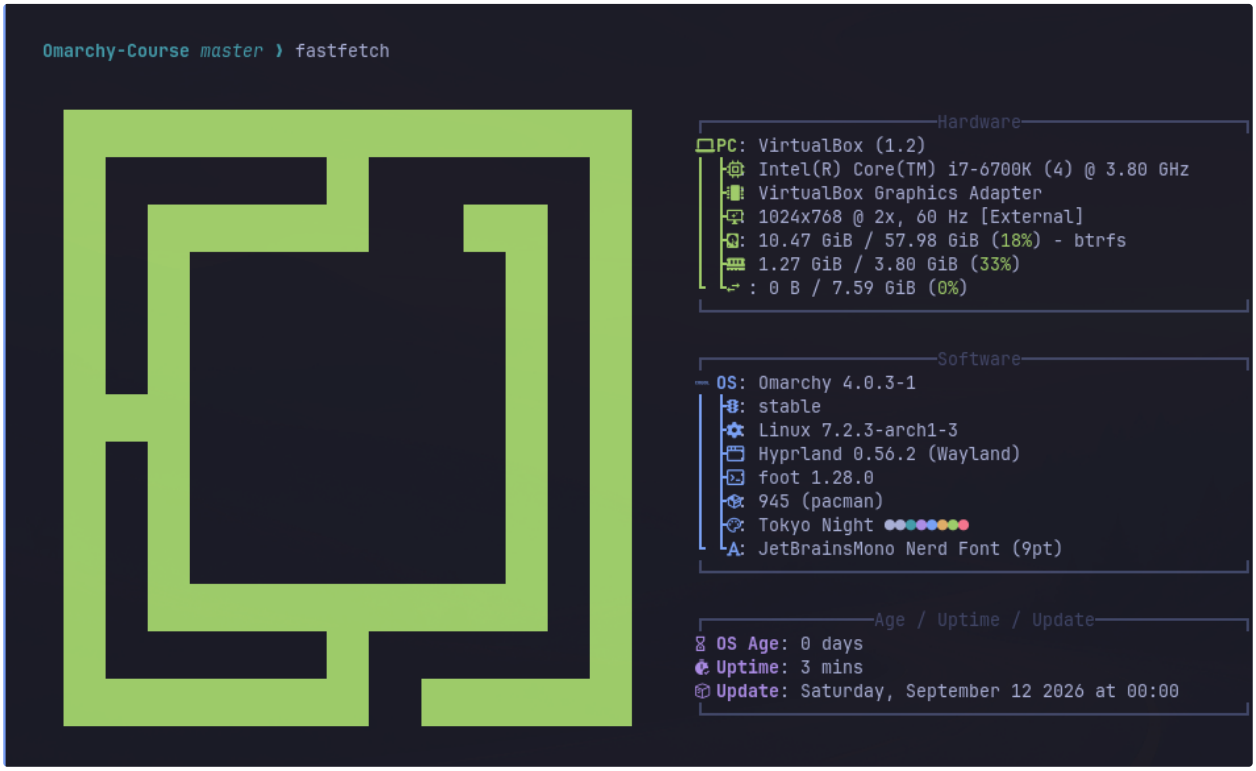
A useful simplified model of Omarchy looks like this:

```
Hardware
   ↓
UEFI / Firmware
   ↓
Linux kernel
   ↓
Arch Linux userspace
   ↓
Hyprland
   ↓
Omarchy Shell / Quickshell
   ↓
Omarchy tools and defaults
   ↓
Your personal configuration
   ↓
Your applications and development workflow
```

You do not need to understand every layer deeply yet.

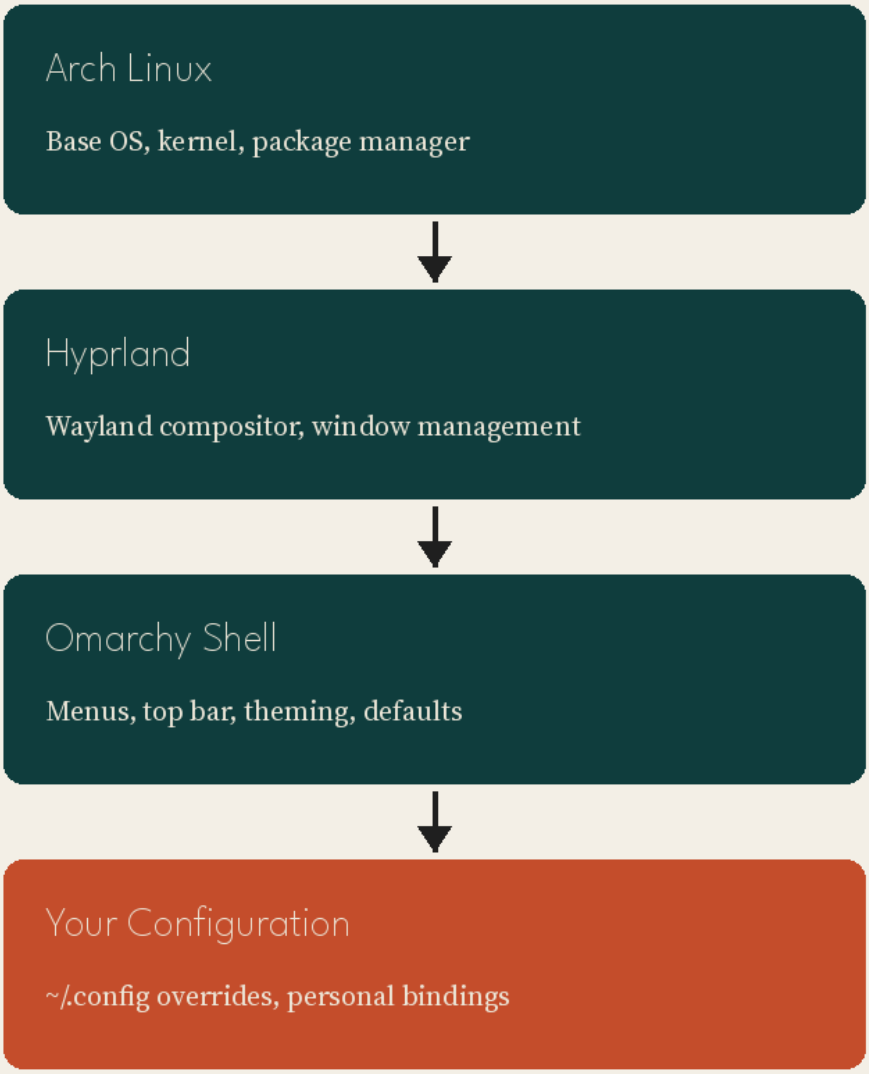
For now, the important point is:

> ***Omarchy is a complete system assembled from multiple Linux components, with its own opinionated configuration and tooling on top.***



Screenshot 1.1-A

Omarchy desktop overview



Screenshot 1.1-B
diagram showing Arch → Hyprland → Omarchy Shell → user configuration

1.2 — What Is Arch Linux?

What You'll Learn

What Arch Linux contributes to your Omarchy installation.

The Short Version

Arch Linux is the underlying Linux distribution.

It provides things such as:

- the Linux kernel;
- system libraries;
- package management;
- systemd;
- hardware support;
- filesystem layout;
- networking components;
- command-line tools.

Omarchy builds its own experience on top of this foundation.

Conceptually:

```
Arch Linux
    +
Omarchy configuration
    +
Hyprland
    +
Quickshell
    +
curated applications
    +
Omarchy tooling
```

Why This Matters

When following generic Arch Linux instructions, remember:

> *Omarchy may already have its own preferred way of doing the same task.*

For example, instead of immediately reaching for raw package-manager commands, Omarchy may expose a dedicated command or menu action.

This does not mean Arch documentation is useless.

It means you should first ask:

1. Does Omarchy already handle this?
2. Is there an `omarchy` command for it?
3. Is there an official Omarchy manual section?
4. Only then: do I need the underlying Arch approach?

1.3 — Omarchy Is Opinionated

The official Omarchy project describes itself as an **omakase Linux distribution**.

"Omakase" is a useful concept here.

It means the system makes many choices for you instead of presenting endless setup decisions.

Omarchy chooses and integrates:

- a tiling window manager;
- a shell;
- themes;
- keybindings;
- applications;
- development tools;
- workflows;
- helper commands.

This is deliberate.

The goal is not:

> *"Give the user a blank Arch installation and make them configure everything."*

The goal is closer to:

> *"Provide a highly curated workstation that can still be customized deeply."*

That makes Omarchy especially interesting for developers who want a powerful Linux workstation without spending weeks assembling every component manually.

1.4 — What Is Hyprland?

Short Definition

Hyprland is the Wayland compositor and tiling window manager used by Omarchy.

It controls major parts of how windows behave.

Examples include:

- window placement;
- tiling;
- floating;
- workspaces;
- focus;
- fullscreen behavior;
- monitor configuration;
- input configuration;
- window rules;
- keyboard bindings.

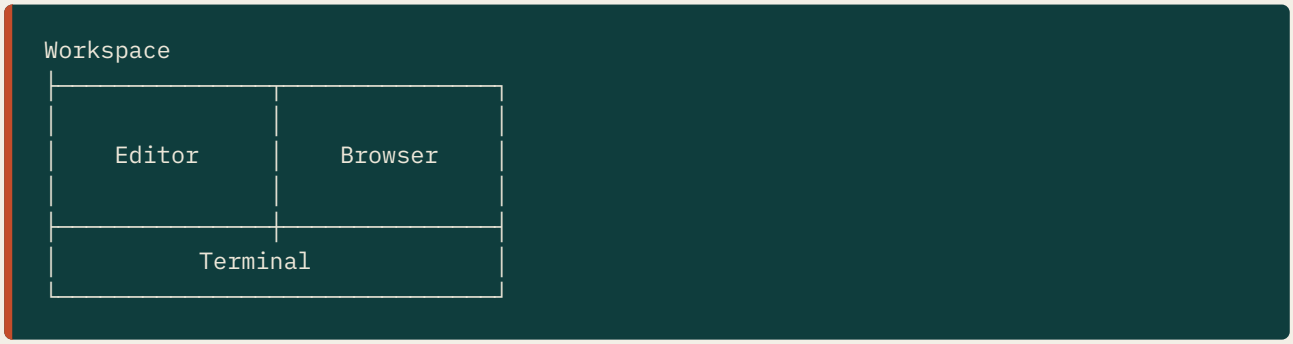
Traditional Desktop vs Tiling Workflow

A traditional desktop often encourages windows to overlap:

```
Desktop
├── Browser window
├── Terminal window
├── Editor window
└── Chat window
```

You manually resize and move them.

Hyprland usually tiles them automatically:



This is one reason Omarchy feels different immediately.

Important Distinction

Hyprland is **not the entire Omarchy desktop**.

Hyprland controls the compositor/window-management layer.

Omarchy adds its own shell and tools on top.

1.5 — What Is Wayland?

You will see the word **Wayland** frequently.

You do not need to learn the complete graphics stack now.

The simple explanation is:

> *Wayland is the modern display protocol used by Hyprland and many current Linux desktops.*

Applications talk to the compositor, and the compositor controls how they appear on screen.

Simplified:

Older Linux systems commonly used X11/Xorg.

Some applications may still use compatibility layers such as XWayland.

We will only go deeper when a real troubleshooting situation requires it.

1.6 — What Is Quickshell?

Omarchy 4 / Quattro introduced a major architectural change.

The current Omarchy desktop shell is built using **Quickshell**.

The Omarchy 4 release consolidated many previously separate desktop components into one integrated shell process.

This includes areas such as:

- the top bar;
- launcher;
- menus;
- notifications;
- on-screen displays;
- control panels;
- lock screen;
- policy authentication UI.

This architecture is one reason current Omarchy 4 documentation may differ significantly from older tutorials.

Why This Matters

If you find an older tutorial telling you to configure components such as:

- Waybar;
- Mako;
- SwayOSD;
- Hyprlock;
- Hypridle;

it may describe an older Omarchy architecture.

Current Omarchy 4 uses its Quickshell-based shell for many of those responsibilities.

This is exactly why this handbook prioritizes current official documentation.

1.7 — The Omarchy Shell

The Omarchy shell is the visible system layer surrounding your applications.

Examples include:

- top bar;
- Omarchy menu;
- Wi-Fi panel;
- Bluetooth panel;
- audio panel;
- display panel;
- notifications;
- lock screen;
- OSD elements;
- plugin-driven widgets.

You interact with it constantly, even though Hyprland is handling the windows underneath.

A useful distinction is:

```
Hyprland
→ manages windows, monitors, workspaces, focus

Omarchy Shell
→ menus, panels, bar, notifications, widgets, system UI
```

1.8 — The Omarchy CLI

One of the most important pieces of the system is:

```
omarchy
```

The official Omarchy manual describes the CLI as a command center exposing the same internal tooling used throughout the Omarchy experience.

This is particularly useful for:

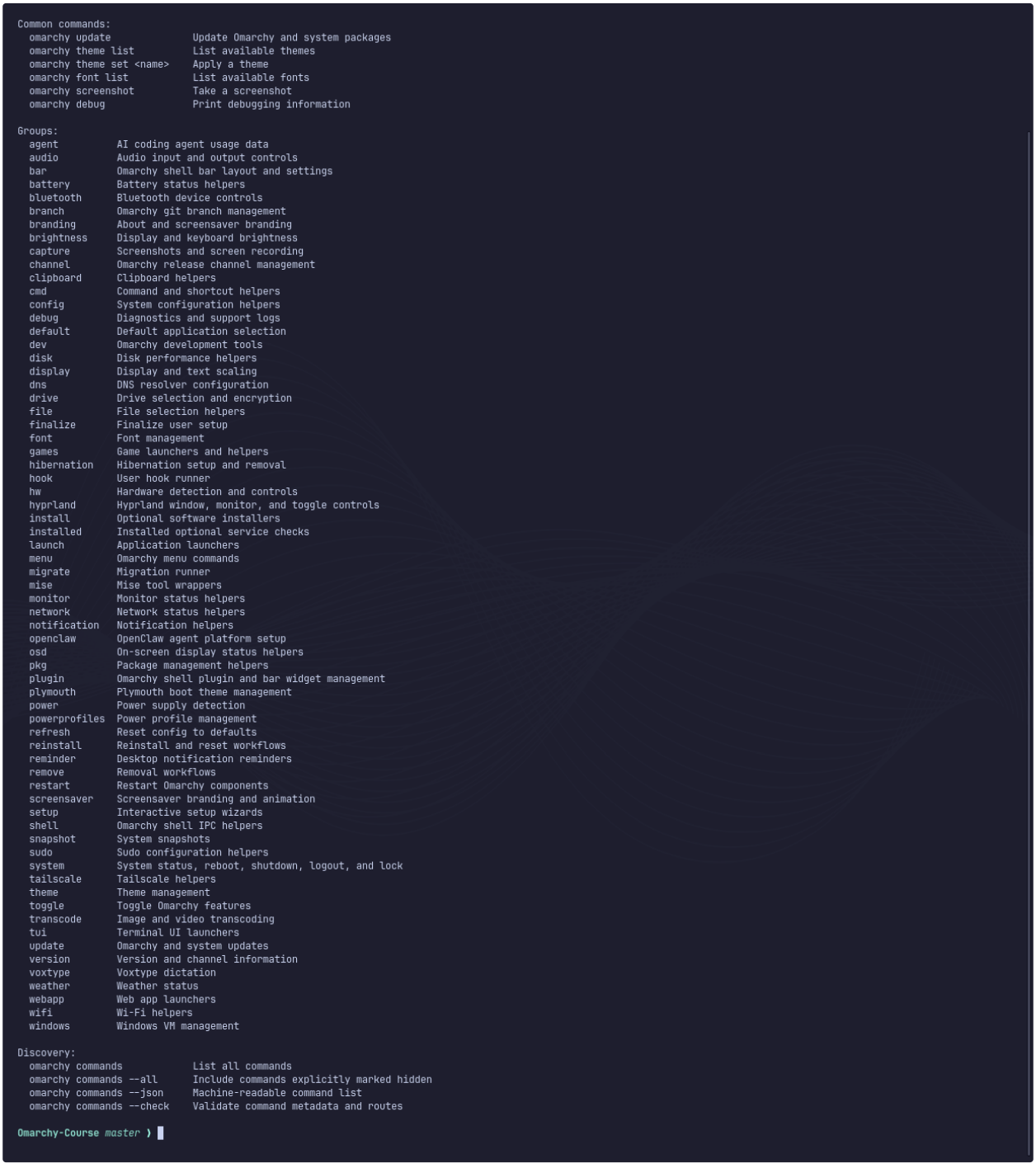
- developers;
- automation;
- troubleshooting;
- AI coding agents;
- scripting;
- reproducible workstation configuration.

Open the Command Center

Run:

```
omarchy
```

You should see the main command help.



Screenshot 1.8-B

`omarchy` command center output

Discover Common Commands

Run:

```
omarchy commands
```

Screenshot 1.8-A

`omarchy commands`

Discover Everything

Run:

```
omarchy commands --all
```

This can return a large list.

That is expected.

You are not supposed to memorize it.

Screenshot 1.8-C

`omarchy commands --all`

Get Help for a Group

Example:

```
omarchy theme --help
```

Another example:

```
omarchy restart --help
```

And for deeper command groups:

```
omarchy <group> <command> --help
```

1.9 — Why the CLI Matters for AI Development

One of the most interesting design aspects of Omarchy is that its graphical controls and command-line tools are closely related.

That makes it particularly suitable for AI-assisted workstation management.

For example, instead of telling an AI agent:

> *"Click the Wi-Fi icon, open some settings window, and try to figure out what is wrong."*

you may be able to ask it to inspect:

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

or discover the available system tools with:

```
omarchy commands --all
```

This gives coding agents a structured interface for inspecting and configuring the machine.

Later in the course, this becomes an important part of our AI-native workflow.

1.10 — Menu, Hotkeys, and CLI Are Different Interfaces to the Same System

You will often have multiple ways to perform the same task.

For example:

```
Keyboard shortcut
        ↓
Omarchy system action
```

or:

```
Super + Space
→ Menu
→ System action
```

or:

```
omarchy ...
```

The exact implementation varies by feature, but the important mental model is:

> ***Do not think of the Omarchy menu, keyboard shortcuts, and CLI as completely separate systems.***

They are different ways of interacting with the same workstation.

1.11 — Where Omarchy's Own Files Live

This is one of the most important lessons in the entire course.

Official Omarchy files primarily live under:

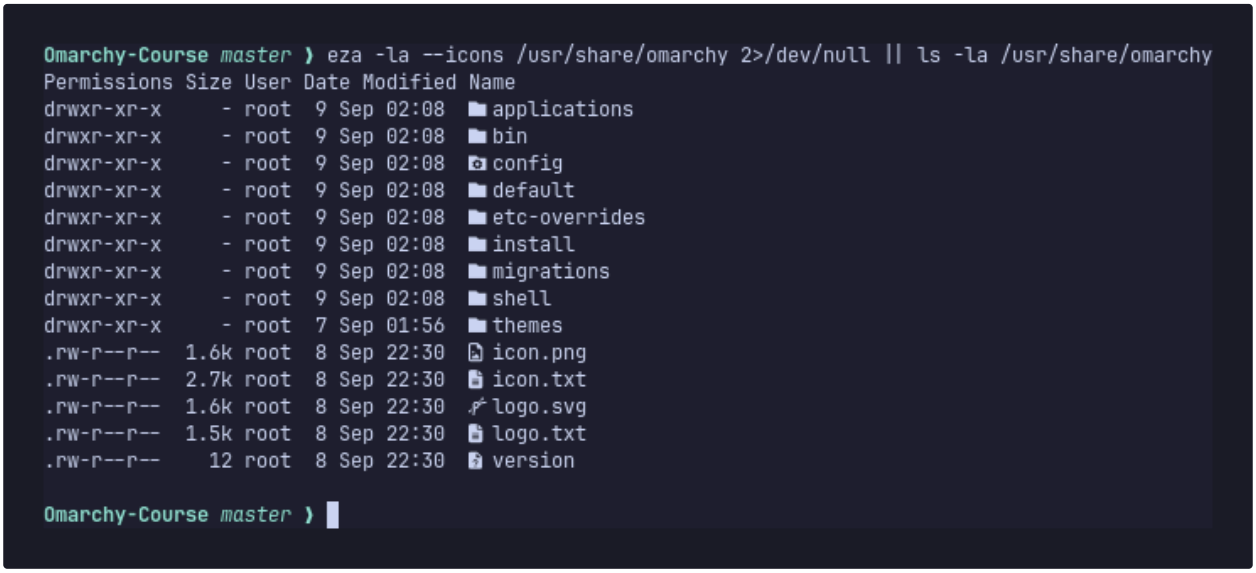
```
/usr/share/omarchy
```

These files belong to Omarchy itself.

The official dotfiles documentation explicitly advises users not to modify them directly.

Why?

Because package updates can replace them.



Screenshot 1.11-A

`/usr/share/omarchy` directory listing

Example

Suppose you change:

```
/usr/share/omarchy/default/...
```

It works perfectly.

Then you update Omarchy.

Your modification disappears.

That is not a bug.

You edited a file owned by the Omarchy package.

1.12 — Where Your Personal Configuration Lives

Your personal configuration primarily belongs under:

```
~/.config
```

The official Omarchy manual describes these dotfiles as **your files for your changes**.

Examples include:

```
~/.config/hypr/
~/.config/omarchy/
~/.config/mise/
```

These are the files we will customize throughout the course.

Screenshot 1.12-A
`~/.config/hypr` directory listing

Screenshot 1.12-B
`~/.config/omarchy` directory listing

Screenshot 1.11-B
comparison of `~/.config` and `/usr/share/omarchy`

1.13 — What Does `~` Mean?

You will constantly see paths beginning with:

```
~
```

The tilde means:

```
your home directory
```

For a user named:

```
alex
```

this:

```
~/.config
```

usually means:

```
/home/alex/.config
```

Check your own home directory:

```
echo "$HOME"
```

Then:

```
cd ~
pwd
```

1.14 — What Is a Dotfile?

A file or directory beginning with a period is normally hidden from ordinary directory listings.

Examples:

```
.config
.bashrc
.gitconfig
```

Hence the common Linux term:

> ***dotfiles***

Show hidden files with:

```
ls -la
```

Later you will learn more convenient Omarchy aliases and tools for this.

1.15 — Key Omarchy Configuration Files

Current Omarchy documentation identifies several important files under `~/.config`.

For Hyprland:

```
~/.config/hypr/hyprland.lua
```

Main Hyprland user configuration.

```
~/.config/hypr/bindings.lua
```

Personal keybindings and binding overrides.

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

Monitor resolution, scaling, and positioning.

```
~/.config/hypr/input.lua
```

Keyboard, mouse, and trackpad configuration.

```
~/.config/hypr/looknfeel.lua
```

Visual and layout behavior.

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

Extra processes started with your session.

For the Omarchy shell:

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

Controls shell behavior such as bar layout, widgets, screensaver, lock, and idle settings.

1.16 — Inspect Your Configuration

Run:

```
ls -la ~/.config/hypr
```

Then:

```
ls -la ~/.config/omarchy
```

Do not edit anything yet.

We are only learning the structure.

Try It Yourself

Find the monitor configuration:

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

Find the input configuration:

```
ls ~/.config/hypr/input.lua
```

Find the keybinding configuration:

```
ls ~/.config/hypr/bindings.lua
```

1.17 — Defaults First, Your Overrides After

Current Omarchy uses a layered configuration approach.

Conceptually:

```
Omarchy defaults
       ↓
Your configuration
       ↓
Final behavior
```

This is extremely useful.

It means you do **not** need to copy the entire Omarchy configuration into your home folder just to change one setting.

Instead, you change only what you want to override.

Example

Imagine Omarchy's default mouse sensitivity is:

```
0
```

You want:

```
-0.3
```

You do not need to rewrite the entire input configuration.

You add your override in your own configuration.

This keeps your personal setup small and allows Omarchy updates to continue improving its defaults.

1.18 — Real Example: Our Hyprland Load Order

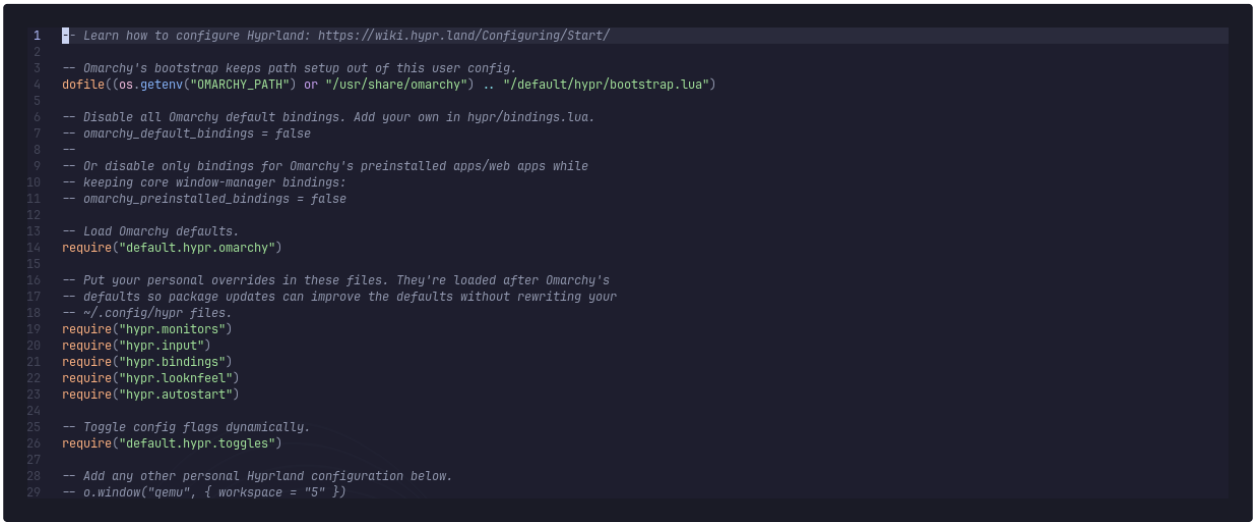
A current Omarchy user configuration may look conceptually like this:

```
dofile((os.getenv("OMARCHY_PATH") or "/usr/share/omarchy") .. "/default/hypr/
bootstrap.lua")

require("default.hypr.omarchy")

require("hypr.monitors")
require("hypr.input")
require("hypr.bindings")
require("hypr.looknfeel")
require("hypr.autostart")

require("default.hypr.toggles")
```

Notice the idea:

```
Omarchy defaults
        ↓
personal monitor settings
personal input settings
personal bindings
personal look and feel
personal autostart
```

This is the pattern we want.



```
-- Learn how to configure Hyprland: https://wiki.hypr.land/Configuring/Start/

-- Omarchy's bootstrap keeps path setup out of this user config.
dofile((os.getenv("OMARCHY_PATH") or "/usr/share/omarchy") .. "/default/hypr/bootstrap.lua")

-- Disable all Omarchy default bindings. Add your own in hypr/bindings.lua.
-- omarchy_default_bindings = false
--
-- Or disable only bindings for Omarchy's preinstalled apps/web apps while
-- keeping core window-manager bindings:
-- omarchy_preinstalled_bindings = false

-- Load Omarchy defaults.
require("default.hypr.omarchy")

-- Put your personal overrides in these files. They're loaded after Omarchy's
-- defaults so package updates can improve the defaults without rewriting your
-- ~/.config/hypr files.
require("hypr.monitors")
require("hypr.input")
require("hypr.bindings")
require("hypr.looknfeel")
require("hypr.autostart")

-- Toggle config flags dynamically.
require("default.hypr.toggles")

-- Add any other personal Hyprland configuration below.
-- o.window("qemu", { workspace = "5" })
```

Screenshot 1.18-A
`hyprland.lua` opened in editor

1.19 — Why You Should Not Copy Entire Defaults

A common beginner mistake is:

1. copy a large default config;
2. change two lines;
3. keep the entire copied file forever.

The problem is that your copy may stop benefiting from upstream improvements.

Instead, prefer:

> ***small intentional overrides***

This also makes your configuration easier to:

- understand;
- troubleshoot;
- back up;
- put in Git;
- reproduce on another machine.

Later, our dotfiles repository may contain only a handful of changes even though Omarchy itself contains thousands of lines of defaults.

That is a feature.

1.20 — Editing Configuration Through the Omarchy Menu

The current official documentation allows key configuration files to be opened directly from the Omarchy menu.

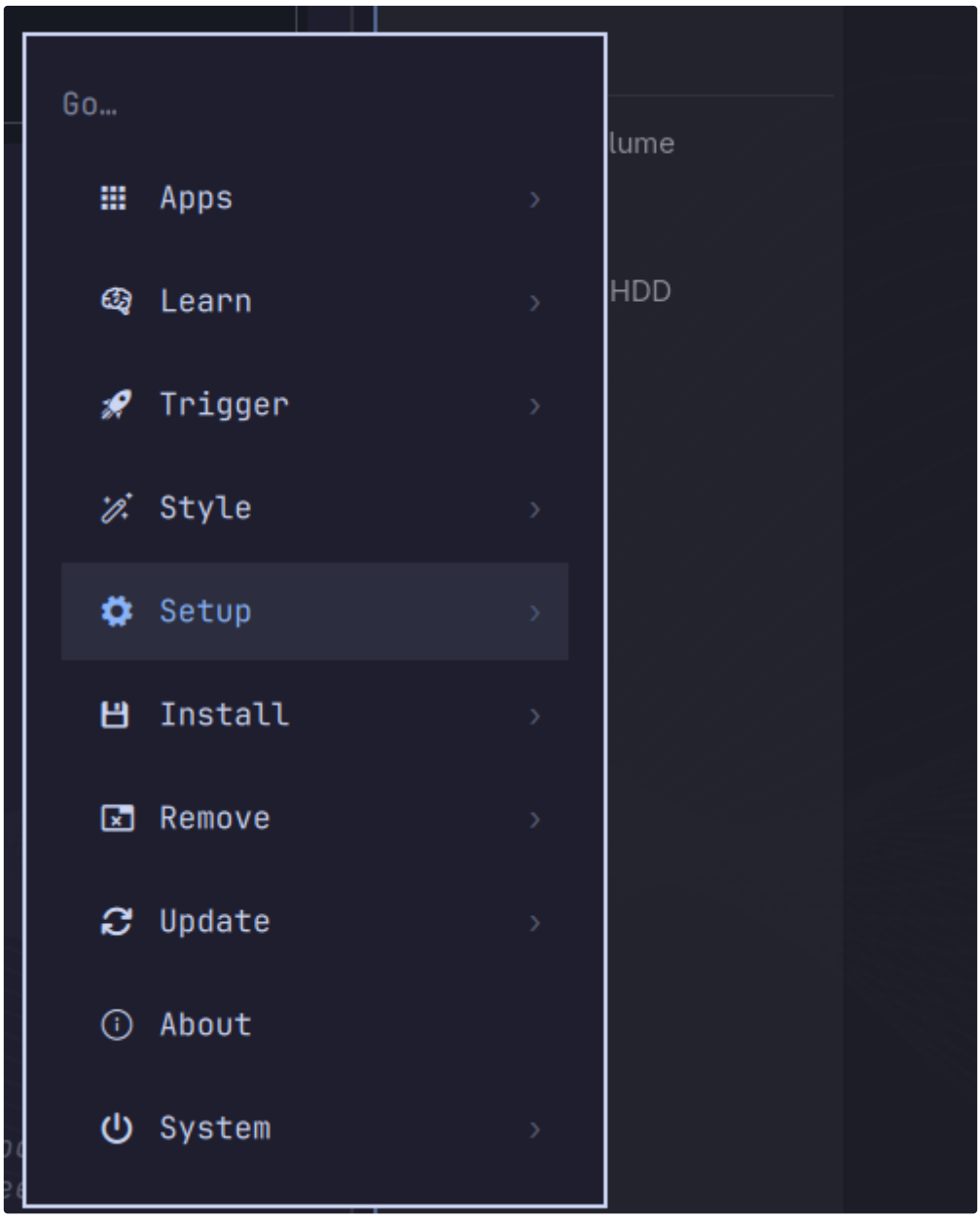
For example:

```
Super + Space
→ Setup
→ Monitors
```

or configuration-related menu entries for:

- keybindings;
- input;
- config files.

One benefit of using Omarchy's integrated configuration actions is that required reload/restart behavior can be handled automatically after the editor exits.



Screenshot 1.20-A
Omarchy menu configuration section

1.21 — Editing Configuration Manually

You can also edit files directly.

For example:

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

or:

```
nvim ~/.config/hypr/bindings.lua
```

Later we will learn:

- which file to edit;
- how to reload changes;
- how to recover from errors.

For now:

> ***Know where the file is before editing it.***

1.22 — Do Not Edit `/usr/share/omarchy` for Normal Customization

This deserves its own rule.

Course Rule

> ***Treat `/usr/share/omarchy` as reference material, not your personal configuration folder.***

It is perfectly useful to **read** those files.

For example:

```
rg "workspace" /usr/share/omarchy/default/hypr
```

This can help you understand how Omarchy implements something.

But do not normally save your personal modifications there.

1.23 — Reading Defaults Is Extremely Useful

Although you should not usually edit Omarchy's own files, reading them is one of the best ways to learn the system.

Example:

```
grep -R "SUPER + S" /usr/share/omarchy/default/hypr 2>/dev/null
```

This might help you discover where a shortcut is defined.

A better modern search tool is often:

```
rg "SUPER \+ S" /usr/share/omarchy
```

We will learn `rg` properly in the terminal module.

1.24 — What Happens During Updates?

Because Omarchy's core files are package-managed:

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

can change when Omarchy updates.

Your personal files:

```
~/.config
```

are intended to hold your overrides.

This separation allows:

```
new Omarchy defaults
        +
your customizations
```

to coexist.

This is one of the reasons we strongly avoid hacking directly inside `/usr/share/omarchy`.

1.25 — What If You Break Your Personal Configuration?

Omarchy provides reset and refresh mechanisms.

For example, current Omarchy includes commands for refreshing specific configuration areas and reinstalling user configs.

However, some of these operations are destructive.

Example:

```
omarchy reinstall configs
```

can reset user configuration.

Do **not** run destructive recovery commands casually.

Our later recovery module will teach this hierarchy:

```
inspect
   ↓
fix the specific file
   ↓
restart/reload component
   ↓
refresh specific config
   ↓
restore/reinstall only if necessary
```

1.26 — Your Shell Configuration

Not all personal configuration belongs under `~/.config`.

For Bash, personal shell additions commonly live in:

```
~/.bashrc
```

The official Omarchy dotfiles documentation recommends putting your own:

- aliases;
- functions;
- exports;

there.

Later we will inspect how Omarchy's shell defaults are loaded and where your own additions belong.

1.27 — Autostart

If you want your own application or service to start with the desktop session, Omarchy provides a user autostart configuration.

The key file is:

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

A current Omarchy-style example is:

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

You do not need to use this yet.

The important point is:

> *Omarchy provides a supported place for your own session startup commands.*

1.28 — Hooks

Omarchy also supports hooks for system events.

Current documented hook events include areas such as:

- post-boot;
- post-update;
- theme changes;
- font changes;
- low battery.

User hooks live under:

```
~/.config/omarchy/hooks/
```

This is different from autostart.

Conceptually:

```
autostart
→ run when the desktop session starts

hook
→ run when a specific Omarchy event happens
```

We will build one later.

1.29 — Shell Plugins

Omarchy 4's shell also has a plugin architecture.

Plugins can extend the desktop shell with:

- widgets;

- panels;
- services;
- UI components.

This is much more powerful than a simple shell hook.

Conceptually:

```
Hook
→ event-driven script

Plugin
→ extension to the Omarchy shell
```

We will cover both in a dedicated module.

1.30 — Why Version Awareness Matters

Omarchy changes quickly.

A tutorial written for an older release may reference:

- different shortcuts;
- different shell components;
- removed applications;
- old configuration formats;
- renamed commands;
- tools no longer used by current Omarchy.

For example, Omarchy 4's Quickshell architecture replaced multiple previously separate desktop components.

Therefore our rule is:

> ***Never treat a tutorial's behavior as more authoritative than the current installed system and current official documentation.***

1.31 — Verify the Installed Version

Run:

```
omarchy version
```

Write down the result.

Example:

```
4.x.x
```

This handbook is intentionally not locked to one patch version.

But knowing your own version is essential when troubleshooting.

Screenshot 1.31-A
`omarchy version`

1.32 — Use the Current CLI as Documentation

Suppose this handbook says a command exists.

But your system reports:

```
Unknown Omarchy command
```

Do not keep retrying it.

Run:

```
omarchy commands --all
```

Search for the capability.

Example:

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

or:

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

Your installed command surface is often the fastest way to see what your current version supports.

1.33 — Use `--help` Before Guessing Syntax

Example:

```
omarchy network --help
```

Maybe the command group supports:

```
status
speedtest
band
qr
```

The exact syntax matters.

This is safer than guessing:

```
omarchy network test
```

and assuming something is broken when that command never existed.

1.34 — Real-World Mistakes That Became Course Lessons

While building this workstation, several assumptions turned out to be wrong.

These are valuable because they demonstrate why verification matters.

Examples included:

- assuming `omarchy --version` when the real command was `omarchy version`;
- assuming shortcut combinations from older expectations;
- assuming clipboard history used a different shortcut;

- assuming a command group existed when it did not;
- assuming a package should be installed through the system package layer when it belonged in mise;
- assuming a project config was named `.mise.toml` instead of the actual `mise.toml`;
- assuming the active bootloader instead of checking it.

These are not just mistakes.

They reveal an important engineering principle:

> ***Observe the real system before changing the real system.***

1.35 — A Better Troubleshooting Habit

Bad troubleshooting:

```
Something did not work.
↓
Search random blog.
↓
Paste five commands.
↓
System changes.
↓
Still do not know what was wrong.
```

Better troubleshooting:

```
Something did not work.
↓
Identify the affected component.
↓
Check current Omarchy version.
↓
Inspect current CLI/help.
↓
Inspect configuration/state.
↓
Reproduce the problem.
↓
Make one targeted change.
↓
Verify.
```

This methodology will appear throughout the handbook.

1.36 — Practical Exercise: Map Your Own System

Open a terminal.

Run:

```
omarchy version
```

Then:

```
omarchy commands
```

Then:

```
ls -la ~/.config/hypr
```

Then:

```
ls -la ~/.config/omarchy
```

Then:

```
ls -ld /usr/share/omarchy
```

Finally:

```
echo "$HOME"
```

What You Should Be Able to Explain

After running those commands, answer:

1. What version of Omarchy am I running?
2. Where are my Hyprland user configs?
3. Where is my Omarchy shell config?
4. Where are Omarchy's package-owned files?
5. What is my home directory?
6. Which location should contain my own customizations?

1.37 — Practical Exercise: Inspect Without Editing

Let's inspect one user configuration file:

```
cat ~/.config/hypr/hyprland.lua
```

Then inspect the Omarchy defaults directory:

```
ls /usr/share/omarchy/default/hypr
```

Do **not** modify anything.

The purpose is to recognize the separation:

```
my configuration
vs
Omarchy implementation/defaults
```

1.38 — Practical Exercise: Discover a Command

Imagine you want to know what network tools Omarchy provides.

Instead of searching the web first:

```
omarchy network --help
```

Then:

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

Now imagine you want to know about restart helpers:

```
omarchy restart --help
```

This is the habit we want to develop.

1.39 — Checkpoint

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

Omarchy Architecture

- [ ] I understand that Omarchy is based on Arch Linux.
- [ ] I know that Hyprland handles window/compositor behavior.
- [ ] I know that the Omarchy shell is a separate system layer.
- [ ] I know that current Omarchy 4 uses Quickshell for the desktop shell.
- [ ] I understand that Omarchy adds its own tooling and conventions on top.

Configuration

- [ ] I know that my personal configuration primarily belongs under `~/.config`.
- [ ] I know that Omarchy's own package-managed files live under `/usr/share/omarchy`.
- [ ] I understand why I should not normally edit `/usr/share/omarchy`.
- [ ] I understand the idea of defaults plus overrides.
- [ ] I can locate `bindings.lua`, `monitors.lua`, and `input.lua`.

CLI

- [ ] I know how to run `omarchy`.
- [ ] I know how to run `omarchy commands`.
- [ ] I know how to run `omarchy commands --all`.
- [ ] I know how to inspect a command group with `--help`.
- [ ] I understand that I should discover syntax instead of guessing.

Version Awareness

- [ ] I can check my Omarchy version.
- [ ] I know that older tutorials may describe outdated behavior.
- [ ] I understand that current official documentation and my actual installed system should take priority.

1.40 — What You Can Now Do

You can now:

- explain what Omarchy is at a high level;
- distinguish Omarchy from Arch Linux;
- explain Hyprland's role;
- explain the Omarchy shell's role;

- understand why Omarchy 4's Quickshell architecture matters;
- find Omarchy's own files;
- find your personal configuration;
- inspect configuration safely;
- discover available Omarchy commands;
- use built-in help;
- verify the installed version;
- avoid blindly following old tutorials;
- understand how personal overrides survive while upstream defaults continue evolving.

Module 1 Summary

The core mental model is:

```
Arch Linux
    ↓
Hyprland
    ↓
Omarchy Shell / Quickshell
    ↓
Omarchy tools and defaults
    ↓
Your configuration
    ↓
Your workflow
```

For configuration:

```
/usr/share/omarchy
→ Omarchy-owned
→ read/reference
→ updated by packages
→ do not normally modify
```

```
~/.config
→ user-owned
→ customize here
→ preserve your intentional overrides
```

For command discovery:

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

And the most important lesson of this module:

> ***Understand which layer owns the behavior before trying to change it.***

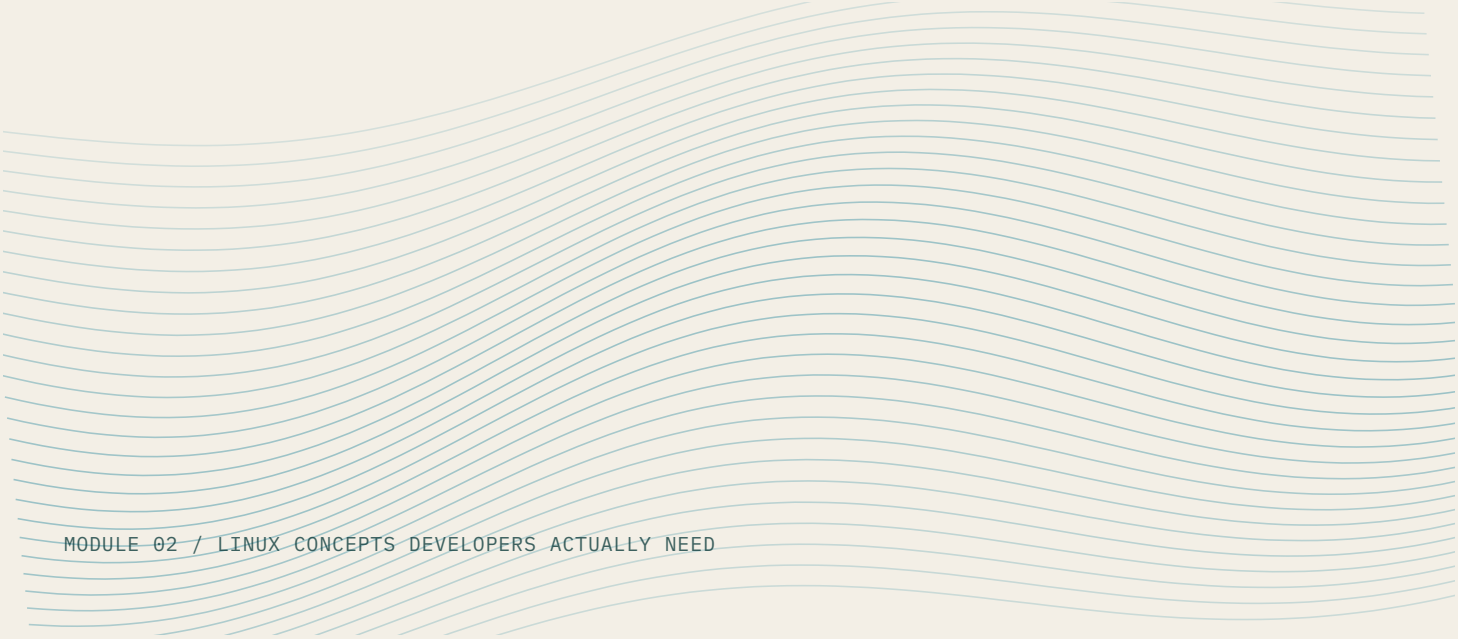
MODULE 02

02

Linux Concepts Developers Actually Need

You now have:

- a working Omarchy installation;
- a basic understanding of how Omarchy is layered;
- a known-good workstation baseline.



About This Module

You now have:

- a working Omarchy installation;
- a basic understanding of how Omarchy is layered;
- a known-good workstation baseline.

Before we go deeper into the terminal, development tooling, AI agents, and system customization, you need a small amount of Linux foundation.

This is **not** a generic Linux administration course.

We are going to learn only the concepts that make the rest of the Omarchy workstation easier to understand.

The goal is not to memorize every directory under `/`, every `systemd` command, or every permission bit.

The goal is to understand enough that commands stop looking like magic.

By the end of this module, you should be able to look at something like:

```
systemctl --user status
```

or:

```
echo "$PATH"
```

or:

```
ls -la ~/.config
```

and understand what you are actually asking the system to do.

What You Will Learn

By the end of Module 2, you will understand:

- what the Linux kernel is;
- what "userspace" means;
- the most important Linux filesystem locations;
- what your home directory is;
- absolute and relative paths;

- files, directories, hidden files, and symlinks;
- users and groups;
- permissions and ownership;
- what `sudo` actually does;
- processes and services;
- system-level vs user-level services;
- what `systemd` does;
- what an environment variable is;
- what `$PATH` is;
- how the shell finds commands;
- the difference between system packages and development runtimes;
- why Omarchy uses both pacman and mise;
- what UEFI, Limine, UKI, and the kernel do during boot;
- how Wayland and Hyprland fit into the desktop stack.

2.1 — Linux Is More Than the Kernel

People commonly use the word **Linux** to refer to an entire operating system.

Technically, Linux is the kernel.

The system you use contains many layers:

```
Hardware
   ↓
Linux kernel
   ↓
system libraries and system services
   ↓
shell + tools
   ↓
Hyprland + Omarchy Shell
   ↓
applications
```

The kernel is responsible for core responsibilities such as:

- CPU scheduling;
- memory management;
- hardware/device access;
- filesystems;

- networking;
- process isolation.

Most programs you interact with do not talk directly to hardware.

They ask the operating system to do things through the kernel.

2.2 — Kernel vs Userspace

This distinction will appear frequently in Linux documentation.

Kernel Space

The kernel runs with very high privileges.

It controls things such as:

- hardware;
- memory;
- process scheduling;
- filesystems;
- drivers.

Userspace

Applications and most services run in **userspace**.

Examples include:

- Ghostty;
- Chromium;
- Neovim;
- Git;
- Hyprland;
- Omarchy Shell;
- your coding agent.

Simplified:

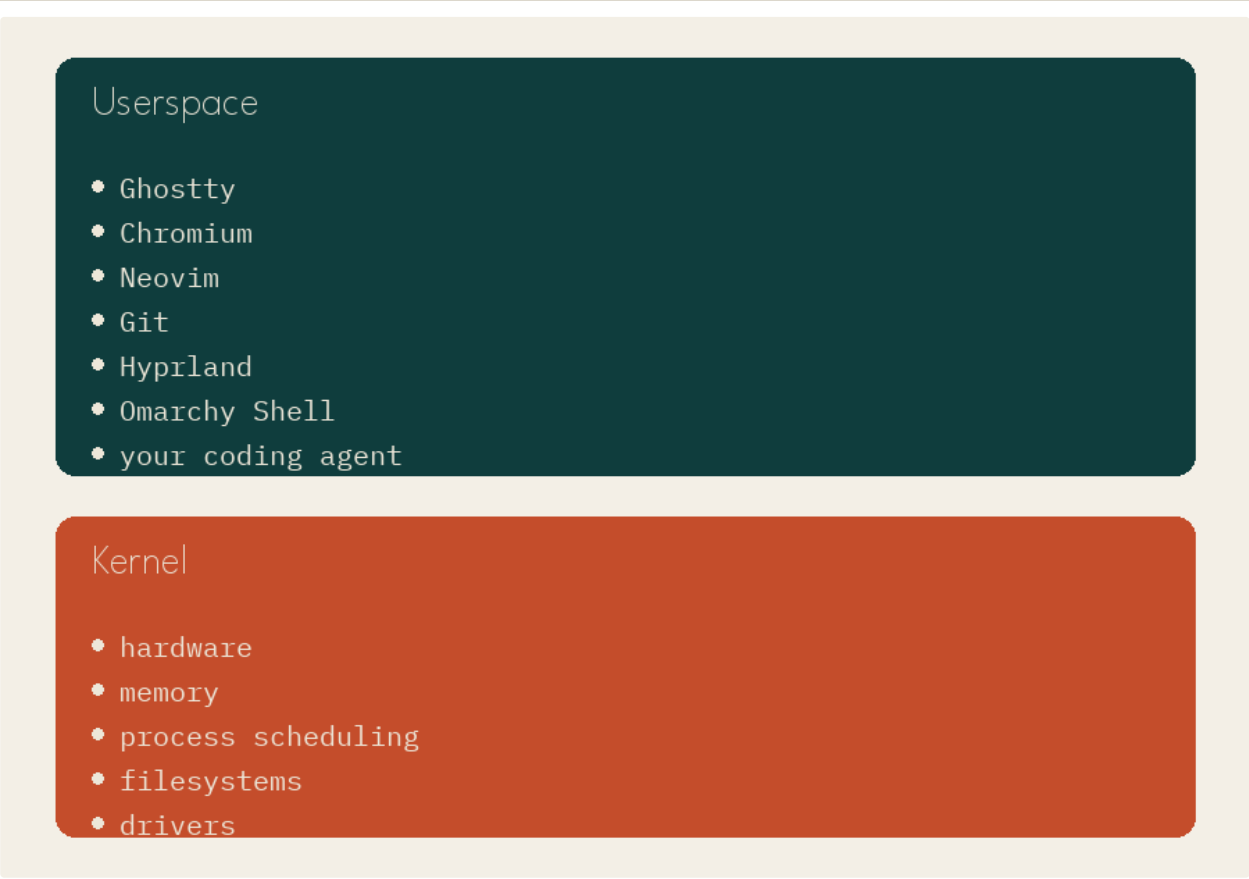
```
Userspace
_____________________
Ghostty
Chromium
Hyprland
Git
AI agent
system services
_____________________
Kernel
_____________________
CPU
RAM
GPU
SSD
USB
Network hardware
```

This matters when troubleshooting.

A problem might be:

- hardware;
- kernel/driver;
- system service;
- desktop configuration;
- application-specific.

Those are very different categories.



Screenshot 2.2-A
diagram: kernel vs userspace

2.3 — The Linux Filesystem Starts at `/`

On Windows, you are familiar with drive letters:

```
C:\
D:\
E:\
```

Linux uses one unified filesystem tree.

It begins at:

```
/
```

This is called the **root directory**.

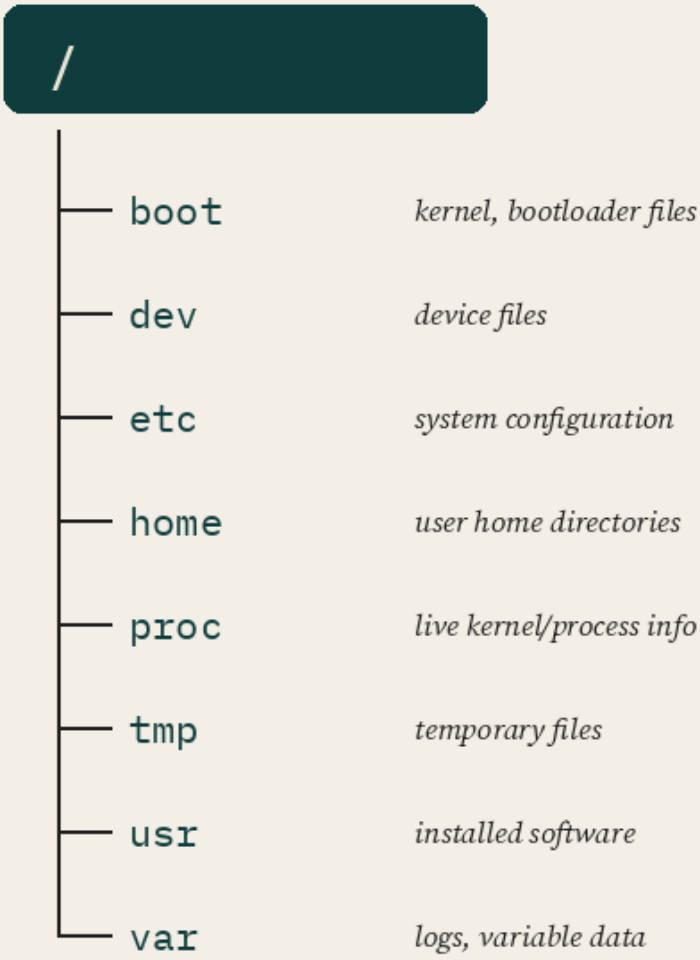
Everything exists somewhere underneath it.

Example:

```
/
├── boot
├── dev
├── etc
├── home
├── proc
├── run
├── sys
├── tmp
├── usr
└── var
```

You do not need to memorize all of these.

We will focus on the ones developers encounter constantly.



Screenshot 2.3-A
diagram: Linux filesystem hierarchy

2.4 — Your Home Directory

Each normal user has a home directory.

Usually:

```
/home/USERNAME
```

For example:

```
/home/alex
```

Linux shells provide a shortcut for your home directory:

```
~
```

So:

```
~/.config
```

means:

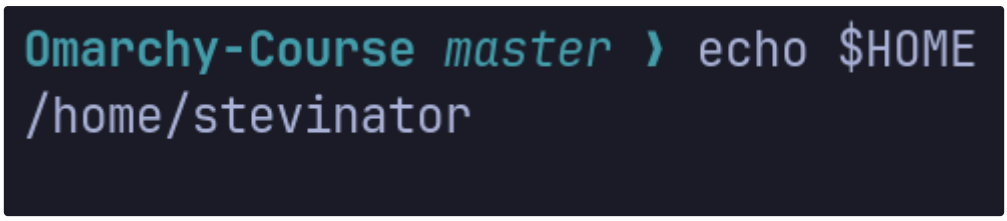
```
/home/YOUR_USERNAME/.config
```

Check yours:

```
echo "$HOME"
```

Then:

```
cd ~
pwd
```



Screenshot 2.4-A
`echo "$HOME"`

Screenshot 2.4-B
`pwd`

2.5 — Why `/home` Matters So Much

Your home directory normally contains:

- documents;

- downloads;
- projects;
- application state;
- personal configuration;
- SSH keys;
- Git configuration;
- shell configuration.

Examples:

```
~/Projects
~/.config
~/.ssh
~/.bashrc
~/.gitconfig
```

This is also why the snapshot lesson from Module 0 matters:

> *Omarchy system snapshots restore the system root state, but not your personal `/home` data.*

Your projects and personal files require a separate backup strategy.

2.6 — Important Linux Directories

You do not need every Filesystem Hierarchy Standard detail.

For this course, understand these.

`/home`

User files and personal configuration.

Example:

```
/home/alex/Projects
```

`/etc`

System-wide configuration.

Examples might include:

```
/etc/hosts
/etc/pacman.conf
/etc/systemd/
```

Files here normally affect the whole machine.

Editing them often requires elevated privileges.

`/usr`

Installed system software and shared data.

Omarchy itself places package-owned files under:

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

As covered in Module 1:

> *These are Omarchy-owned files, not your personal configuration.*

`/var`

Variable system data.

Common examples include:

- logs;
- package caches;
- databases;
- service state.

`/boot`

Boot-related files.

On your Omarchy system this may contain things such as:

- EFI data;
- Unified Kernel Images;
- bootloader-related files.

We will explore boot architecture later.

`/tmp`

Temporary files.

Applications can use this location for short-lived data.

Do not use `/tmp` as permanent storage.

2.7 — Absolute and Relative Paths

An **absolute path** begins from `/`.

Example:

```
/home/alex/Projects/my-app
```

A **relative path** begins from your current directory.

If you are currently in:

```
/home/alex
```

then:

```
Projects/my-app
```

refers to:

```
/home/alex/Projects/my-app
```

Check Your Current Directory

```
pwd
```

`pwd` means:

> *print working directory*

Move Around

```
cd ~/Projects
```

Then:

```
pwd
```

Move up one directory:

```
cd ..
```

Return home:

```
cd ~
```

or simply:

```
cd
```

2.8 — `.` and `..`

Two special directory references appear everywhere.

```
.
```

means:

> *current directory*

```
..
```

means:

> *parent directory*

Example:

```
cd ..
```

moves one level up.

Example:

```
./script.sh
```

means:

> *run `script.sh` from the current directory*

You will see `./` frequently in development tooling.

2.9 — Hidden Files

On Linux, filenames beginning with a dot are normally hidden.

Examples:

```
.config
.ssh
.bashrc
.git
```

This is why they are often called **dotfiles**.

A normal:

```
ls
```

may not show them.

Use:

```
ls -la
```

to include hidden files.

Later, Omarchy's shell aliases and tools will make directory listings more pleasant.

```
Omarchy-Course master ❱ ls -la ~ | head -12
Permissions Size User       Date Modified Name
drwxr-xr-x     - stevinator 12 Sep 17:53  .agents
drwxr-xr-x     - stevinator 12 Sep 18:08  .cache
drwxr-xr-x     - stevinator 12 Sep 17:53  .claude
drwxr-xr-x     - stevinator 12 Sep 18:15  .codex
drwxr-xr-x     - stevinator 12 Sep 17:58  .config
drwxr-xr-x     - stevinator 12 Sep 17:53  .hermes
drwxr-xr-x     - stevinator 12 Sep 17:53  .local
drwxr-xr-x     - stevinator 12 Sep 18:14  .omp
drwxr-xr-x     - stevinator 12 Sep 17:53  .pi
drwxr-xr-x     - stevinator 12 Sep 17:58  Documents
drwxr-xr-x     - stevinator 12 Sep 17:53  Downloads
```

Screenshot 2.9-A

`ls -la`

```
drwxr-xr-x   - stevinator 12 Sep 17:52 lazygit
drwxr-xr-x   - stevinator 12 Sep 17:53 microsoft-edge
drwxr-xr-x   - stevinator 12 Sep 17:53 microsoft-edge-dev
.rw-r--r-- 248 stevinator 12 Sep 17:53 mimeapps.list
drwxr-xr-x   - stevinator 12 Sep 20:20 mise
drwxr-xr-x   - stevinator 12 Sep 17:52 nvim
drwxr-xr-x   - stevinator 12 Sep 17:52 obsidian
drwxr-xr-x   - stevinator 12 Sep 21:07 omarchy
drwxr-xr-x   - stevinator 12 Sep 20:09 opencode
drwx------   - stevinator 12 Sep 17:58 pulse
.rw-r--r-- 768 stevinator  8 Sep 20:30 [T] starship.toml
drwxr-xr-x   - stevinator 12 Sep 17:52 systemd
drwxr-xr-x   - stevinator 12 Sep 17:52 tmux
.rw------- 645 stevinator 12 Sep 17:58 user-dirs.dirs
drwxr-xr-x   - stevinator 12 Sep 17:52 wireplumber
drwxr-xr-x   - stevinator 12 Sep 17:52 xournalpp
drwxr-xr-x   - stevinator 12 Sep 21:47 zed
```

Screenshot 2.9-B

example hidden dotfiles

2.10 — Files and Directories Are Case-Sensitive

Linux filesystems are usually case-sensitive.

These can all be different files:

```
README.md
readme.md
Readme.md
```

Similarly:

```
Projects
```

and:

```
projects
```

can be different directories.

This is a common source of confusion for people coming from Windows or default macOS filesystems.

2.11 — File Extensions Are Conventions

Linux generally does not require a filename extension to determine whether something is executable.

A command might simply be named:

```
omarchy
```

rather than:

```
omarchy.exe
```

File extensions are still useful conventions:

```
.md
.toml
.json
.lua
.sh
.ts
.py
```

but the system does not rely on them the same way Windows often does.

2.12 — Symbolic Links

Linux frequently uses **symbolic links**, or symlinks.

A symlink is a filesystem entry that points somewhere else.

Conceptually:

```
~/.local/bin/tool
        ↓
real executable somewhere else
```

Inspect a path with:

```
ls -l PATH
```

A symlink appears similar to:

```
tool -> /some/other/location/tool
```

You will encounter symlinks in:

- development tools;
- mise-managed executables;
- configuration management;
- dotfiles setups.

You do not need to create any yet.

2.13 — Users

Linux is a multi-user operating system.

Even if you are the only person using the workstation, the system still distinguishes:

- your normal user;
- the `root` user;
- service users.

Check your current username:

```
whoami
```

Check your numeric user and group information:

```
id
```

Screenshot 2.13-A
`whoami`

```
Omarchy-Course master ❱ id
uid=1000(stevinator) gid=1000(stevinator) groups=1000(stevinator),998
(wheel)
```

Screenshot 2.13-B
`id`

2.14 — The Root User

The `root` account has extremely broad control over the machine.

Root can:

- change system files;
- install system packages;
- modify users;
- change permissions;
- stop critical services;

- erase system data.

That is why you should not run everything as root.

Your everyday development work should happen as your normal user.

2.15 — What `sudo` Does

`sudo` allows an authorized user to execute a command with elevated privileges.

Example:

```
sudo some-command
```

This does **not** mean:

> *"make command work"*

It means:

> *"run this command with elevated privileges"*

That distinction is extremely important.

Bad Habit

A command fails:

```
Permission denied
```

and the immediate response is:

```
sudo ...
```

That can turn a harmless problem into a system-level mistake.

Instead ask:

> *Should this operation actually require root?*

For example, editing your own file:

```
~/.config/hypr/bindings.lua
```

should normally **not** require `sudo`.

If it does, ownership or some other problem may be wrong.

2.16 — Ownership

Files have an owner and a group.

Inspect:

```
ls -l ~/.bashrc
```

You may see something similar to:

```
-rw-r--r-- 1 alex alex ...
```

The relevant parts are:

```
alex alex
```

meaning:

```
owner group
```

Your own configuration should normally belong to your user.

2.17 — Permissions

You will often see permission strings like:

```
-rwxr-xr-x
```

The three primary permission types are:

```
r = read
w = write
x = execute
```

They are grouped for:

```
owner
group
others
```

Example:

```
rwx r-x r-x
```

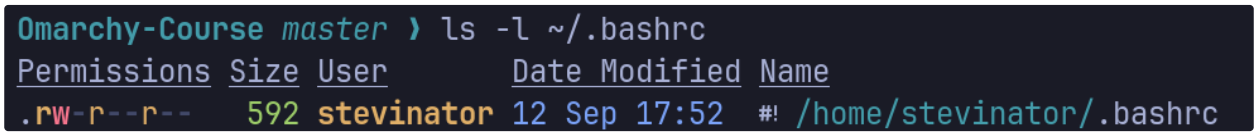
means:

```
owner  → read/write/execute
group  → read/execute
others → read/execute
```

You do not need to calculate octal permissions yet.

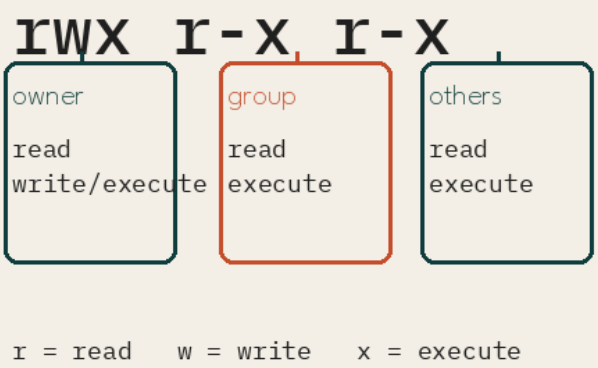
The main idea is:

> *Linux explicitly tracks who may read, write, and execute files.*



Screenshot 2.17-A

`ls -l ~/.bashrc`



Screenshot 2.17-B
permission string annotated

2.18 — Processes

A **process** is a running program.

Examples:

- Chromium process;
- Ghostty process;
- Hyprland process;
- Omarchy shell process;
- coding agent process.

Inspect running processes with tools such as:

```
ps
```

but Omarchy also provides a much nicer interactive system monitor.

Press:

```
Super + Ctrl + T
```

to open `btop`.

2.19 — Process IDs

Each process has a numeric process ID:

```
PID
```

You saw this earlier when inspecting a Hyprland window.

Example:

```
pid: 2913
```

This allows the operating system and diagnostic tools to refer to one specific running process.

2.20 — Closing a Window vs Killing a Process

Normally, close applications using the desktop.

For example:

```
Super + W
```

But sometimes a process becomes stuck.

Linux provides commands such as:

```
kill PID
```

and stronger signals when necessary.

Omarchy often provides a better component-specific restart mechanism.

Example:

```
omarchy restart app <application-name>
```

Later we will learn when to:

```
close
restart
kill
reboot
```

These are not the same thing.

2.21 — Services

Some programs are intended to run in the background.

These are commonly managed as **services**.

Examples might include:

- networking;
- Bluetooth;
- audio components;
- SSH server;
- background system tasks.

On modern Arch Linux, many services are managed using **systemd**.

2.22 — What Is systemd?

`systemd` is the system and service manager used by Arch Linux and therefore by Omarchy.

Among other things, it helps:

- start system services;
- track service state;
- restart services;
- manage dependencies;
- collect journal logs;
- manage user services.

You do not need to become a systemd administrator.

You need enough to inspect system health.

2.23 — System Services vs User Services

There are two important scopes.

System-Level Services

These affect the machine.

Inspect failed system services:

```
systemctl --failed
```

User-Level Services

These run for your user session.

Inspect failed user services:

```
systemctl --user --failed
```

This distinction is especially important on a desktop.

A broken user-session service may cause desktop problems even when:

```
systemctl --failed
```

looks perfectly clean.

2.24 — Check a Service

Generic systemd syntax:

```
systemctl status SERVICE
```

For a user service:

```
systemctl --user status SERVICE
```

Do not guess service names if you do not know them.

Use Omarchy's higher-level restart commands first when they exist.

For example:

```
omarchy restart bluetooth
```

is often a better first recovery step than manually manipulating every Bluetooth-related systemd unit.

2.25 — Why Omarchy Wrappers Matter

This pattern appears throughout the course.

Underlying Linux tools may exist:

```
systemctl
hyprctl
pacman
NetworkManager tools
PipeWire tools
```

but Omarchy may expose a higher-level command such as:

```
omarchy restart audio
```

or:

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

The Omarchy command can encode knowledge about how Omarchy expects its own system to behave.

So our preferred order is:

```
Omarchy command
        ↓
underlying Linux tool if needed
        ↓
manual low-level troubleshooting
```

2.26 — Environment Variables

An environment variable is a named value made available to commands and processes.

Example:

```
echo "$HOME"
```

`HOME` contains your home directory.

Other common examples include:

```
PATH
SHELL
EDITOR
LANG
```

Print your shell:

```
echo "$SHELL"
```

Print your language/locale:

```
echo "$LANG"
```

2.27 — Temporary Shell Variables

You can define a shell variable:

```
NAME="Stevinator"
```

Then:

```
echo "$NAME"
```

This exists in the current shell.

When that shell closes, the variable normally disappears unless it is defined through persistent shell configuration.

2.28 — Exported Environment Variables

To make a value available to child processes:

```
export EXAMPLE="hello"
```

Check:

```
echo "$EXAMPLE"
```

You will later use environment variables for things such as:

- development configuration;
- API endpoints;
- runtime behavior;
- application configuration.

Never casually place secrets into files that you intend to commit to Git.

2.29 — What Is `$PATH`?

`PATH` is one of the most important environment variables for developers.

Print it:

```
echo "$PATH"
```

It contains a list of directories separated by colons.

Conceptually:

```
/home/alex/.local/bin
/usr/local/bin
/usr/bin
...
```

When you type:

```
git
```

the shell searches those directories to find an executable named `git`.

```
Omarchy-Course master ❱ echo $PATH
/home/stevinator/.local/share/mise/installs/bun/latest/bin:/home/stev
inator/.local/share/mise/installs/codex/latest/bin:/home/stevinator/.
local/share/mise/installs/github-anomalyco-opencode/latest:/home/stev
inator/.local/share/mise/installs/github-can1357-oh-my-pi/latest:/hom
e/stevinator/.local/share/mise/installs/hunk/latest/hunkdiff-linux-x6
4:/home/stevinator/.local/share/mise/installs/node/26.8.1/bin:/home/s
tevinator/.local/share/mise/installs/python/3.13.15/bin:/home/stevina
tor/.local/share/mise/shims:/usr/share/omarchy/bin:/usr/local/sbin:/u
sr/local/bin:/usr/bin:/home/stevinator/.local/bin:/usr/bin/site_perl:
```

Screenshot 2.29-A
`$PATH` output

2.30 — How the Shell Finds a Command

Suppose you type:

```
python
```

The shell needs to decide what `python` means.

It might be:

- an alias;
- a shell function;
- a builtin;
- an executable found through `$PATH`.

This is why:

```
type python
```

is extremely useful.

Try:

```
type git
type python
type omarchy
```

Screenshot 2.30-A
`type python`

2.31 — `type` vs `which`

Many tutorials use:

```
which python
```

This can be useful, but:

```
type python
```

is generally better for understanding what the current shell will actually execute.

For example, `type` can tell you whether something is:

- an alias;

- a function;
- a builtin;
- a file.

Try:

```
type ls
```

On Omarchy, you may discover that commands you thought were raw system tools are actually aliases to more modern alternatives.

Screenshot 2.31-A
`type ls`

2.32 — Shell Configuration

Your Bash configuration lives primarily in:

```
~/.bashrc
```

Omarchy loads its shell defaults and allows your own additions to live there.

Later we will use it for:

- aliases;
- exports;
- functions.

The key principle is the same as with Hyprland:

> *Add your own configuration in your own files rather than editing Omarchy's package-owned defaults.*

2.33 — Commands Can Come From Different Places

This becomes important in a development workstation.

A command might come from:

- an Arch package;
- an Omarchy package;
- an AUR package;
- a mise-managed runtime;
- a project-local dependency;
- a shell alias;
- a shell function;
- a user script.

This is another reason `type` is valuable.

Example:

```
type node
```

might reveal a mise-managed executable rather than `/usr/bin/node`.

That matters when diagnosing version problems.

2.34 — Package Management

Arch Linux uses **pacman** for its official package repositories.

Omarchy is itself package-backed.

Current Omarchy Quattro updates are designed around:

```
omarchy update
```

rather than having users casually bypass the Omarchy update pipeline with raw:

```
sudo pacman -Syu
```

The current Omarchy update architecture explicitly treats `omarchy update` as the blessed update path because it coordinates Omarchy package updates, migrations, snapshots, hooks, and restart checks.

2.35 — System Packages vs Developer Runtimes

This distinction is extremely important.

Imagine you need Python for:

```
the operating system
```

and a completely different Python version for:

```
your project
```

Changing the system version to satisfy a project can cause problems.

Therefore:

```
system runtime
        ≠
project runtime
```

Omarchy uses **mise** for the majority of supported development environments.

The current official development tools documentation describes mise as the tool used to install and run multiple versions of programming languages on the same machine.

2.36 — Three Installation Layers

We introduced this in Module 0.

Now understand the reasoning.

Layer 1 — System Software

Omarchy/package-management layer.

Examples:

- desktop applications;
- Linux utilities;
- system dependencies.

Relevant Omarchy interface:

```
omarchy pkg
```

Layer 2 — Omarchy-Integrated Installers

Omarchy also provides curated installation flows.

Relevant interface:

```
omarchy install
```

These can include:

- editors;
- browsers;
- development environments;
- other integrated tools.

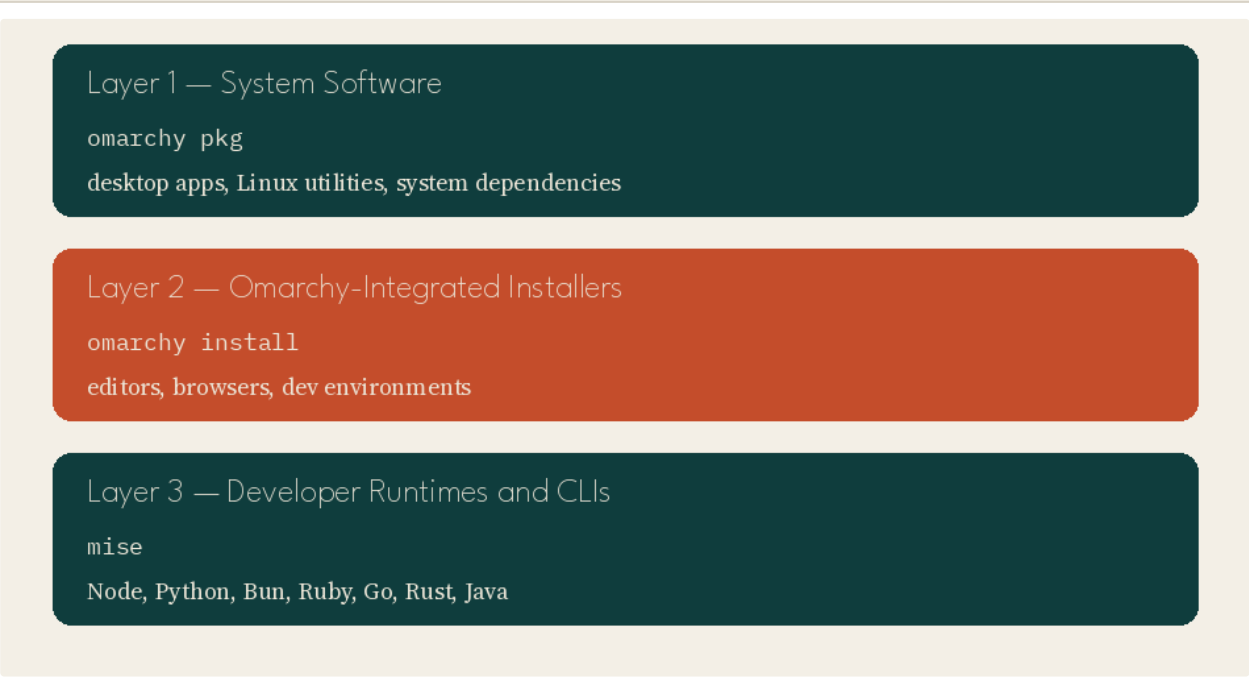
Layer 3 — Developer Runtimes and CLIs

For many developer runtimes:

```
mise
```

Examples:

- Node;
- Python;
- Bun;
- Ruby;
- Go;
- Rust;
- Java;
- other supported environments.



Screenshot 2.36-A
package vs runtime decision diagram

2.37 — Why Mise Is Useful

Suppose your machine has:

```
global Node → 26
```

but one project needs:

```
Node → 24
```

Mise can select a different version for that project.

Conceptually:

```
Machine default
Node 26

Project A
mise.toml
Node 24

Project B
mise.toml
Node 26
```

This keeps the operating system and individual projects from fighting over one global runtime.

We will configure this properly in the development-environment module.

2.38 — Services, Packages, and Commands Are Different Things

A useful mental model:

```
Package
→ installs files

Executable
→ a command you can run

Service
→ a background process managed by the system

Configuration
→ controls behavior
```

One package can install:

- several executables;
- one or more services;
- configuration files.

Do not assume package name, service name, and command name are identical.

2.39 — A Simple Boot Model

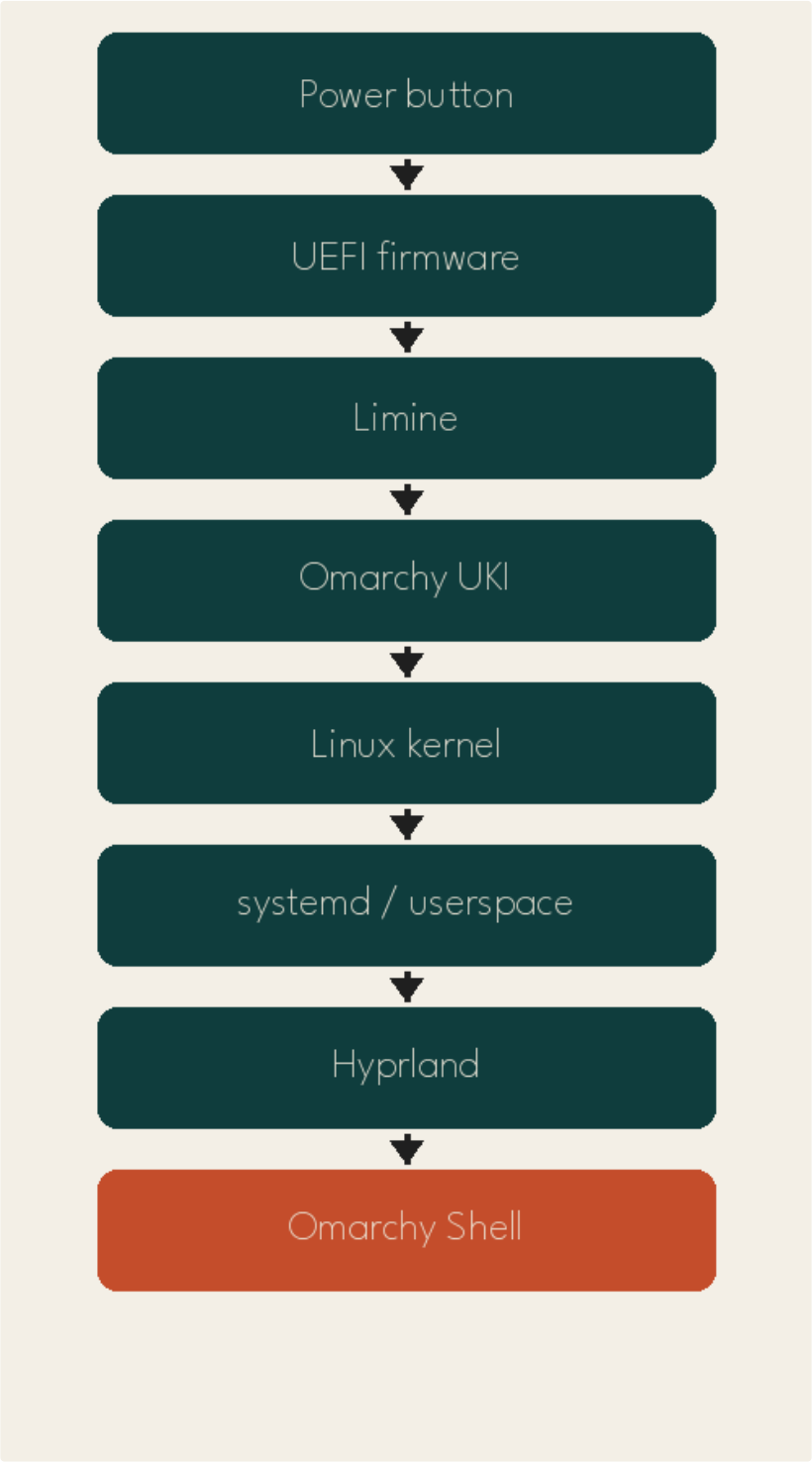
You already saw this in Module 0.

Now let's understand the pieces slightly better.

A current standard Omarchy system can conceptually boot like:

```
Power button
    ↓
UEFI firmware
    ↓
Limine
    ↓
Omarchy UKI
    ↓
Linux kernel
    ↓
systemd / userspace
    ↓
Hyprland
    ↓
Omarchy Shell
```

This model is intentionally simplified.



Screenshot 2.39-A
boot flow diagram

2.40 — UEFI

UEFI is motherboard/laptop firmware.

It:

- initializes hardware;
- maintains boot entries;
- selects boot targets;
- starts the next stage of the boot process.

UEFI exists before Linux starts.

2.41 — Limine

Limine is the default bootloader used on current Omarchy installations.

Its job is to help select and launch the bootable operating-system image.

Omarchy also integrates its snapshot recovery workflow with Limine.

If Direct Boot is configured, firmware can instead launch the Omarchy EFI image directly, bypassing the normal Limine menu.

We will cover that trade-off later.

2.42 — What Is a UKI?

UKI means:

> ***Unified Kernel Image***

A UKI packages several pieces needed for boot into one EFI executable.

You may see a path similar to:

```
/boot/EFI/Linux/omarchy_linux.efi
```

A simplified mental model is:

```
UKI
├── kernel
├── initrd
├── kernel command line
└── boot metadata
```

The exact contents and boot process are deeper than we need here.

The important point:

> *The firmware/bootloader can launch a single signed-style EFI image that contains the material needed to start Linux.*

2.43 — systemd-stub

On a current Omarchy UKI system, tools such as:

```
bootctl status
```

may report:

```
systemd-stub
```

`systemd-stub` is the EFI stub used in the UKI boot flow.

You do not need to configure it manually for normal Omarchy use.

2.44 — Userspace Startup

Once the kernel has started and prepared the machine, userspace starts.

This includes:

- systemd;
- system services;
- login/session infrastructure;
- networking;
- audio;
- desktop session;
- Hyprland;
- Omarchy Shell.

This is why a system can:

```
boot successfully
```

while still having:

```
one broken desktop service
```

The kernel booted, but a later userspace component failed.

2.45 — Wayland

Wayland is the modern display protocol used by Hyprland.

A simplified model:

```
Application
    ↓
Wayland
    ↓
Hyprland compositor
    ↓
GPU / display
```

Hyprland decides:

- where the window goes;
- how large it is;
- what workspace it belongs to;
- which monitor shows it;
- whether it floats or tiles.

2.46 — XWayland

Some older applications were designed for X11 rather than native Wayland.

XWayland provides compatibility for many of those applications.

When inspecting a Hyprland window:

```
hyprctl activewindow
```

you may see:

```
xwayland: 0
```

or:

```
xwayland: 1
```

Conceptually:

```
0 → native Wayland window
1 → XWayland compatibility window
```

This can matter later when troubleshooting application behavior.

2.47 — Hyprland Is the Compositor

On Omarchy:

```
Hyprland
```

is responsible for compositor/window-management behavior.

Examples:

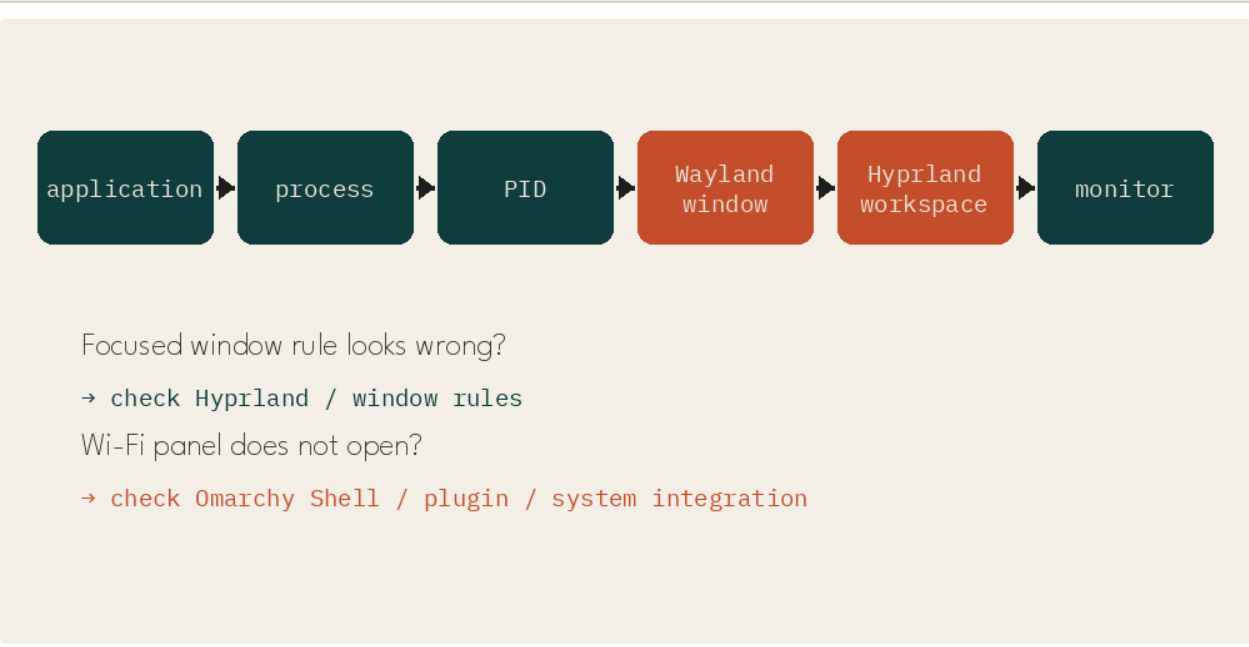
- workspaces;
- tiling;
- monitor configuration;
- input rules;
- window rules;
- focus.

The Omarchy shell is layered around this.

This distinction matters when deciding which config to inspect.

Example:

```
window always opens on wrong workspace
→ probably Hyprland/window rule

Wi-Fi panel does not open
→ probably Omarchy Shell/plugin/system integration
```



Screenshot 2.47-A
Wayland/Hyprland window-flow diagram

2.48 — Practical Exercise: Filesystem Navigation

Run:

```
cd ~
pwd
```

Then:

```
ls
```

Then:

```
ls -la
```

Move into your configuration:

```
cd ~/.config
pwd
```

Move up:

```
cd ..
pwd
```

Return home:

```
cd
```

2.49 — Practical Exercise: Inspect Users and Permissions

Run:

```
whoami
```

Then:

```
id
```

Inspect your Bash config:

```
ls -l ~/.bashrc
```

Answer:

- who owns the file?
- what group owns it?
- can your user write to it?

2.50 — Practical Exercise: Understand Commands

Run:

```
type ls
```

Then:

```
type git
```

Then:

```
type python
```

Then:

```
type omarchy
```

Compare with:

```
which git
which python
```

Do the results tell you the same story?

2.51 — Practical Exercise: Inspect Environment Variables

Run:

```
echo "$HOME"
```

Then:

```
echo "$SHELL"
```

Then:

```
echo "$PATH"
```

The `PATH` output may be long.

That is normal.

Look for locations such as:

```
~/.local/bin
/usr/bin
```

and possibly mise-managed paths.

2.52 — Practical Exercise: Check Process and Service Health

Open:

```
Super + Ctrl + T
```

and look at the process list in `btop`.

Then from a terminal:

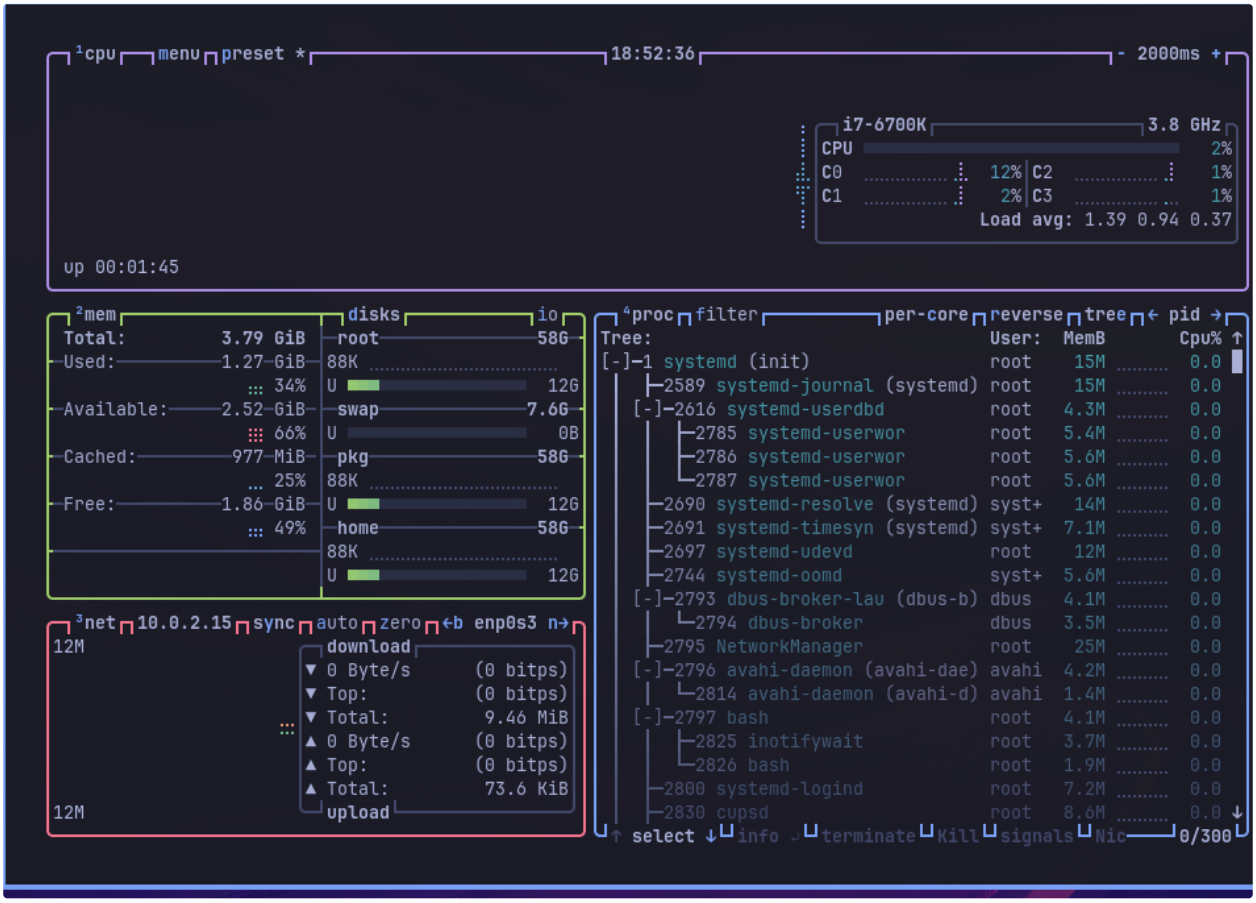
```
systemctl --failed
```

and:

```
systemctl --user --failed
```

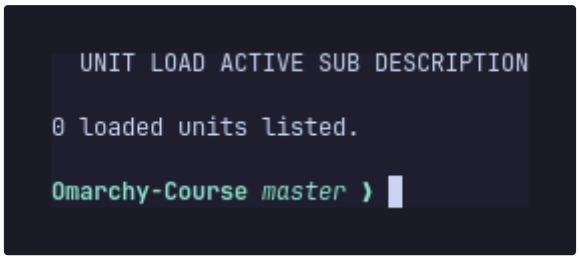
This gives you three different views:

```
btop
→ running processes and system resources

systemctl --failed
→ failed machine-level services

systemctl --user --failed
→ failed user-session services
```



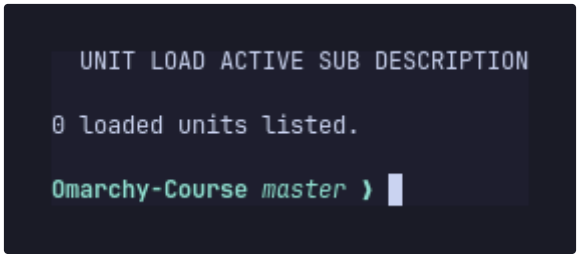
Screenshot 2.52-A

`btop`



Screenshot 2.52-B

`systemctl --failed`



Screenshot 2.52-C

`systemctl --user --failed`

2.53 — Practical Exercise: Inspect the Boot Stack

Run:

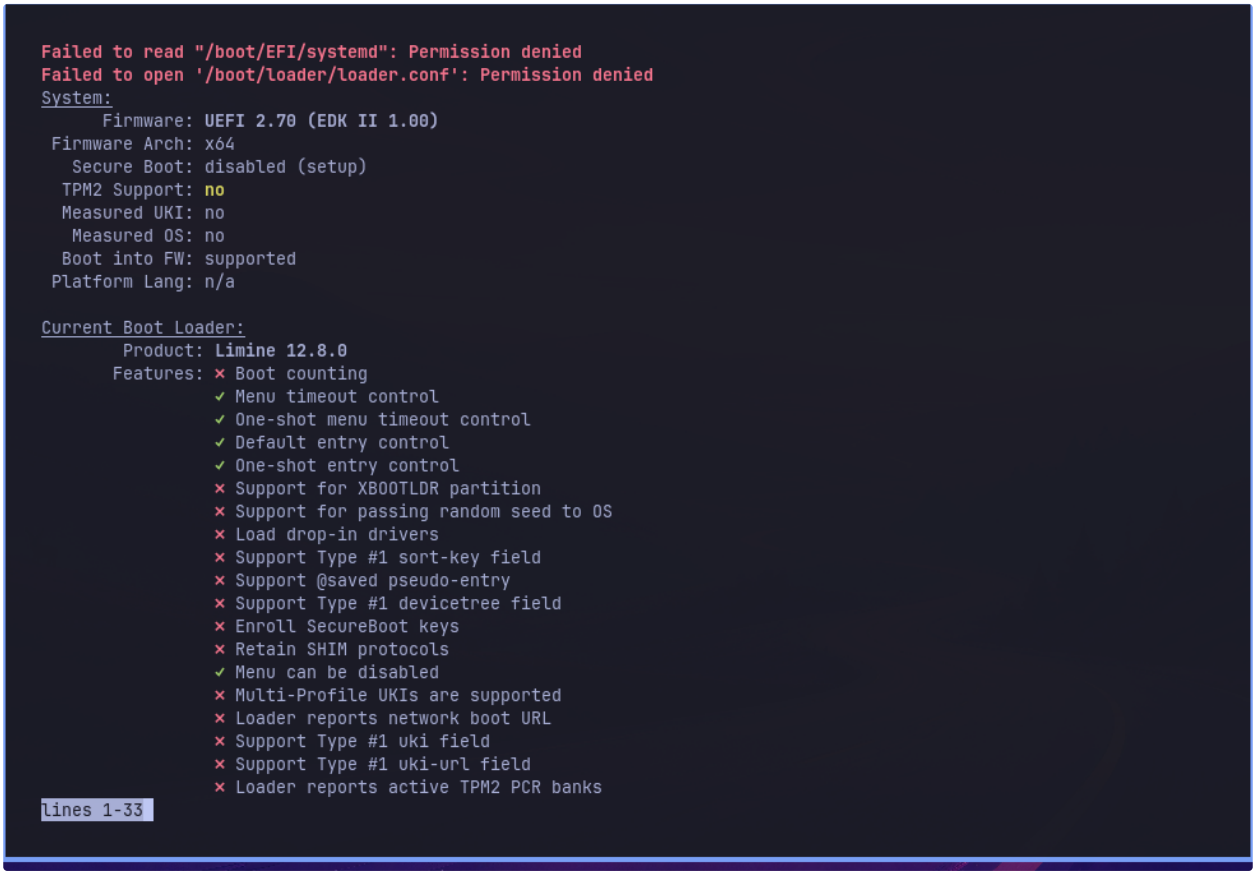
```
bootctl status
```

Do not worry about every field.

Try to identify:

- firmware type;
- Secure Boot state;
- current bootloader;
- current boot entry;
- current EFI image or UKI.

The purpose is to connect the theoretical boot diagram to your actual machine.



Screenshot 2.53-A

`bootctl status`

2.54 — Practical Exercise: Inspect the Desktop Stack

Run:

```
hyprctl activewindow
```

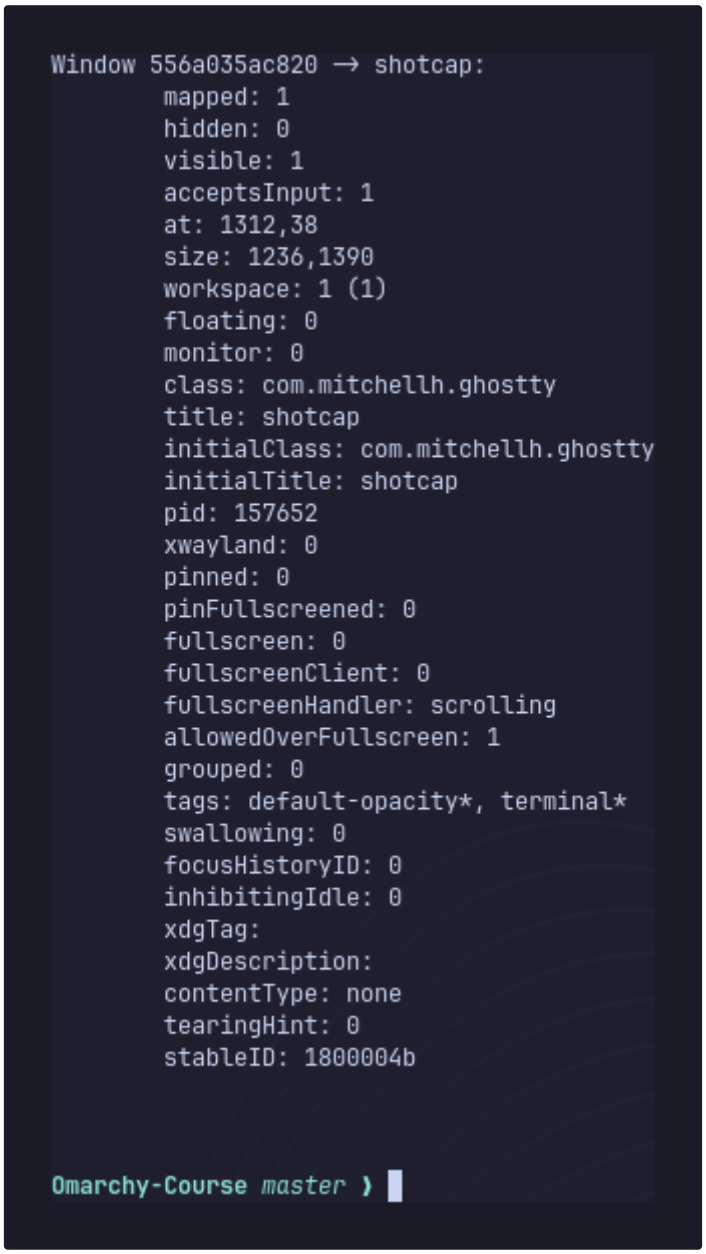
while a terminal is focused.

Find:

```
class
title
workspace
monitor
pid
xwayland
```

You have now connected several Linux concepts:

```
application
→ process
→ PID
→ Wayland window
→ Hyprland workspace
→ monitor
```



Screenshot 2.54-A

`hyprctl activewindow`

2.55 — Common Conceptual Mistakes

These will be covered in more depth in the troubleshooting module.

Mistake 1 — Thinking `/` Means Your Home Folder

It does not.

```
/
→ root of the entire filesystem

~
→ your home directory
```

Mistake 2 — Using `sudo` Whenever Something Fails

`sudo` is privilege elevation, not a repair command.

Mistake 3 — Editing User Files With `sudo`

Your personal config under:

```
~/.config
```

should normally belong to your user.

Using `sudo` carelessly can create root-owned files inside your home directory.

Mistake 4 — Treating a Service and a Process as the Same Thing

A service is a managed background component.

A process is a running instance of a program.

Mistake 5 — Thinking `command not found` Means the Package Is Broken

Possible causes include:

- tool not installed;
- PATH problem;
- wrong command name;
- runtime not active;
- shell function/alias confusion.

Use:

```
type COMMAND
```

and inspect your environment.

Mistake 6 — Installing Project Runtimes Into the System Layer

Prefer mise for supported developer runtimes.

Mistake 7 — Assuming Every Desktop Problem Is Hyprland

The issue may instead live in:

- Omarchy Shell;
- a user service;
- audio/network stack;
- application;
- configuration.

2.56 — Checkpoint

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

Filesystem

- [ ] I know that `/` is the filesystem root.
- [ ] I know that `~` means my home directory.
- [ ] I know what `/etc`, `/usr`, `/var`, `/boot`, and `/home` are broadly used for.
- [ ] I understand absolute vs relative paths.
- [ ] I understand `.` and `..`.
- [ ] I know why `.config` is called a dotfile directory.

Users and Permissions

- [ ] I know my current username.
- [ ] I know what the root user is.
- [ ] I understand that `sudo` elevates privileges.
- [ ] I know that files have owners, groups, and permissions.
- [ ] I know that my personal config normally should not require `sudo`.

Processes and Services

- [ ] I know what a process is.

- [ ] I know what a PID is.
- [ ] I understand what a service is.
- [ ] I know the difference between system and user services.
- [ ] I can check failed services.

Environment

- [ ] I understand what an environment variable is.
- [ ] I understand what `$PATH` does.
- [ ] I know how to inspect a command with `type`.
- [ ] I know why `type` may be more informative than `which`.

Development

- [ ] I understand the difference between system packages and development runtimes.
- [ ] I understand why Omarchy uses mise.
- [ ] I know that different projects can use different runtime versions.

Boot and Desktop

- [ ] I can explain the basic UEFI → Limine → UKI → kernel → userspace flow.
- [ ] I know that Hyprland is the Wayland compositor.
- [ ] I understand that the Omarchy shell and Hyprland are different layers.
- [ ] I know what XWayland is at a high level.

2.57 — What You Can Now Do

After completing Module 2, you can now:

- navigate the Linux filesystem with confidence;
- understand paths used throughout the handbook;
- inspect hidden configuration;
- understand why personal config belongs in your home directory;
- avoid unnecessary `sudo`;
- recognize file ownership and basic permissions;
- distinguish processes from services;
- check machine-level and user-level service health;
- understand environment variables;
- understand how the shell resolves commands;

- identify where a command comes from;
- understand why system packages and project runtimes should be treated differently;
- explain the basic boot chain;
- understand where Hyprland fits into the desktop;
- reason about system problems by layer instead of treating Linux as one giant black box.

Module 2 Summary

The core filesystem model:

```
/
├── boot   → boot files
├── etc    → system configuration
├── home   → user files
├── usr    → installed/shared system software
└── var    → changing system data
```

Your personal workspace:

```
~
├── .config
├── .bashrc
├── .ssh
└── Projects
```

Privilege model:

```
normal user
→ everyday work

sudo/root
→ deliberate system-level changes
```

Runtime model:

```
package
→ installs software

process
→ running program

service
→ managed background program

config
→ controls behavior
```

Developer environment model:

```
system packages
        ≠
project runtimes

Omarchy / pacman
        +
mise
```

Boot model:

```
UEFI
 ↓
Limine
 ↓
UKI
 ↓
Linux kernel
 ↓
systemd / userspace
 ↓
Hyprland
 ↓
Omarchy Shell
```

Desktop model:

```
Application
 ↓
Wayland / XWayland
 ↓
Hyprland
 ↓
Display
```

The most important lesson:

> ***Linux becomes much easier when you identify which layer you are actually interacting with.***

MODULE 03

03

Master the Omarchy Desktop

You now understand:

- what Omarchy is;
- how Arch Linux fits underneath it;
- where Hyprland and the Omarchy Shell fit;
- where your configuration lives;
- the Linux concepts needed to follow the rest of the course.

About This Module

You now understand:

- what Omarchy is;
- how Arch Linux fits underneath it;
- where Hyprland and the Omarchy Shell fit;
- where your configuration lives;
- the Linux concepts needed to follow the rest of the course.

Now we start using Omarchy the way it is designed to be used.

This module is about **daily desktop operation**.

Omarchy is heavily keyboard-driven. The official navigation documentation makes that explicit: the keyboard is the primary interface, and the Omarchy Menu is the entry point when you do not yet know the faster shortcut.

That means this module is not about memorizing dozens of keys.

It is about building a small number of strong habits:

```
Open
→ Focus
→ Move
→ Tile
→ Switch workspace
→ Use panels
→ Discover missing shortcuts
```

By the end of this module, you should feel comfortable operating Omarchy without constantly reaching for the mouse.

What You Will Learn

By the end of Module 3, you will be able to:

- open the Omarchy Menu;
- open the application launcher;
- open the system menu;
- launch core applications quickly;
- close windows;
- move focus between tiled windows;

- swap tiled windows;
- switch between tiling and floating;
- use fullscreen and full-width modes;
- navigate workspaces;
- move windows between workspaces;
- use the scratchpad;
- switch between Omarchy layouts;
- resize tiled windows;
- use the clipboard manager;
- use capture tools;
- use emoji and utility panels;
- control notifications;
- discover current keybindings instead of memorizing old ones;
- build a practical multi-workspace developer workflow.

3.1 — The Omarchy Navigation Philosophy

The official Omarchy navigation guide describes the desktop as keyboard-first.

The basic idea is:

```
Super + Space
→ discover anything

direct shortcut
→ do it faster
```

You should not try to memorize the full shortcut map on day one.

Instead, learn shortcuts in layers.

LAYER 1 — ESSENTIAL

```
Super + Space
Super + Return
Super + Shift + Return
Super + K
Super + W
```

LAYER 2 — WINDOW MANAGEMENT

```
Super + Arrows
Super + Shift + Arrows
Super + T
Super + F
Super + Alt + F
```

LAYER 3 — WORKSPACES

```
Super + 1..0
Super + Shift + 1..0
Super + Tab
Super + Shift + Tab
```

LAYER 4 — PANELS AND TOOLS

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

This gives you a manageable learning path.

3.2 — The Super Key

Throughout Omarchy documentation you will see:

```
Super
```

On a typical PC keyboard, this is the **Windows key**.

Example:

```
Super + Space
```

means:

> *Hold the Windows/Super key and press Space.*

3.3 — Open the Omarchy Menu

Press:

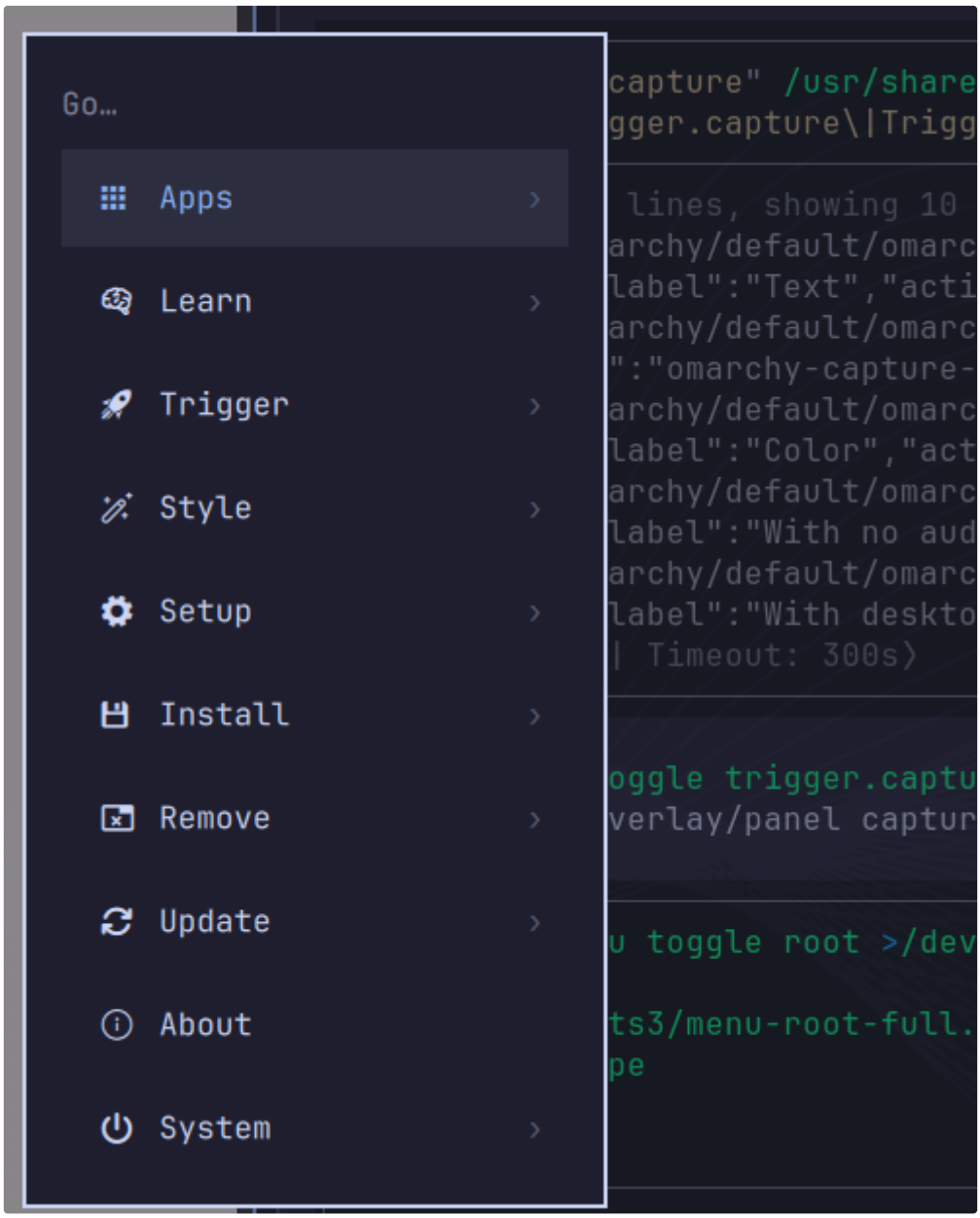
```
Super + Space
```

This opens the main Omarchy Menu.

The menu is the safest place to start when you do not remember a shortcut.

You can use it to access areas such as:

- applications;
- system controls;
- setup;
- install;
- themes;
- capture tools;
- power actions;
- configuration.



Screenshot 3.3-A
Omarchy Menu opened with `Super + Space`

3.4 — Open the Application Launcher

Press:

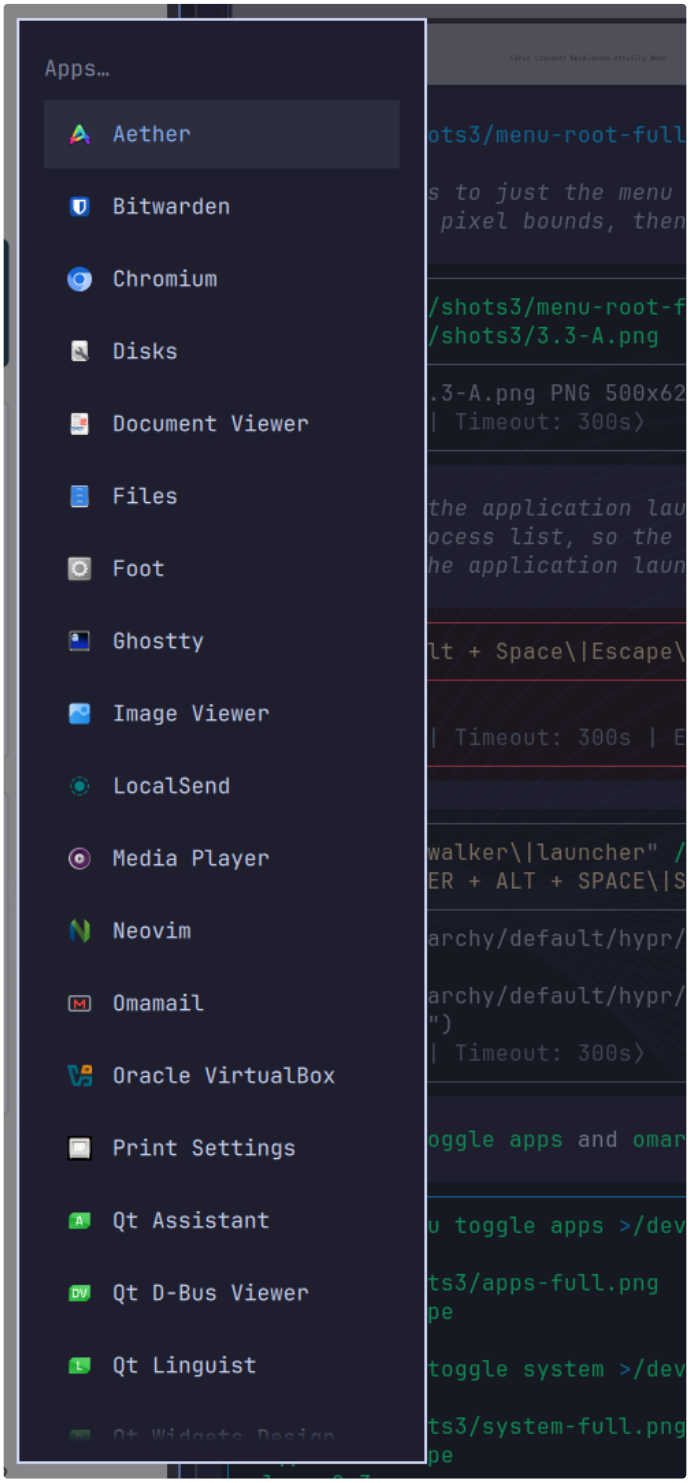
```
Super + Alt + Space
```

This opens the apps menu.

The distinction is useful:

```
Super + Space
→ Omarchy Menu

Super + Alt + Space
→ applications
```

The exact presentation may evolve between Omarchy releases, so always verify with the current keybinding list if necessary.



Screenshot 3.4-A
application launcher

3.5 — Open the System Menu

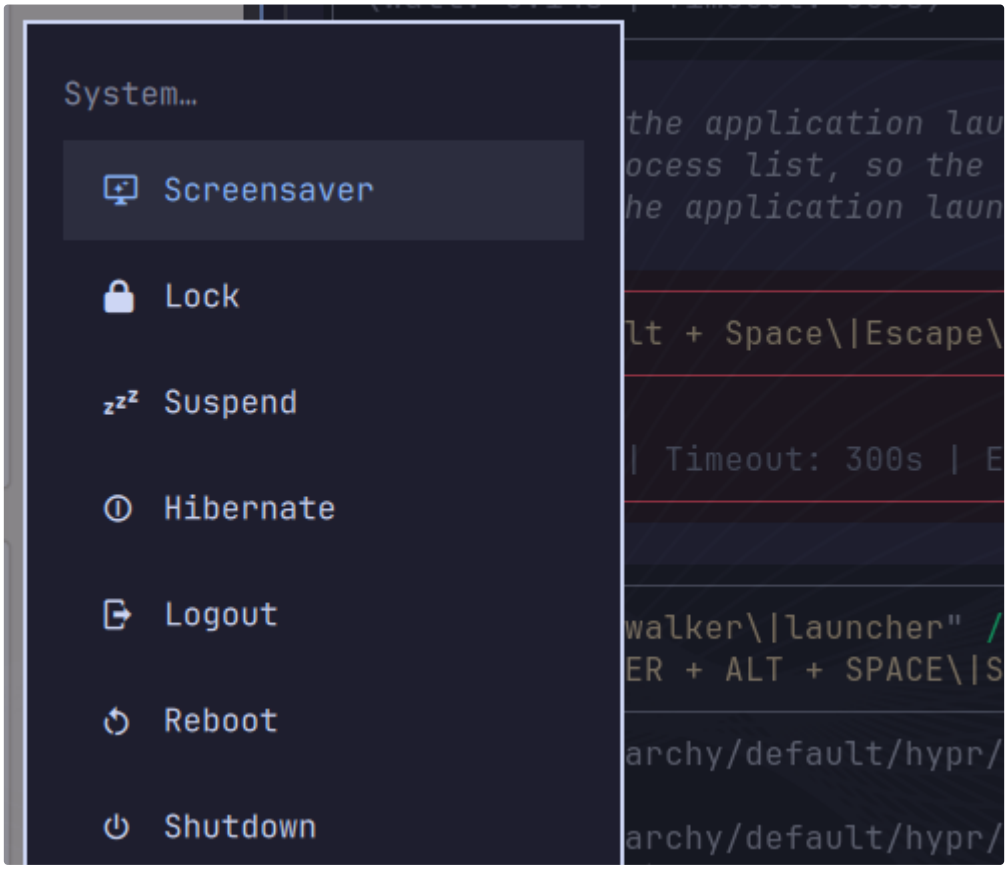
Press:

```
Super + Escape
```

This opens the system menu containing actions such as:

- lock;
- suspend;
- restart;
- power off.

Do not confuse the system menu with the main Omarchy Menu.



Screenshot 3.5-A
system menu

3.6 — Lock the Computer

Current Omarchy hotkeys include:

```
Super + Ctrl + L
```

for locking the workstation.

This is a shortcut worth learning early.

Especially on a development workstation, you should not leave an unlocked session unattended.

3.7 — Launch the Terminal

Press:

```
Super + Return
```

This launches your configured default terminal.

In our Stevinator baseline, that is:

```
Ghostty
```

but the binding follows the configured Omarchy default terminal.

This is important.

The shortcut is not conceptually:

> *"Launch Ghostty."*

It is:

> *"Launch my Omarchy default terminal."*

That makes the binding resilient if you later change terminal.

Screenshot 3.7-A
terminal opened with shortcut

3.8 — Launch the Browser

Press:

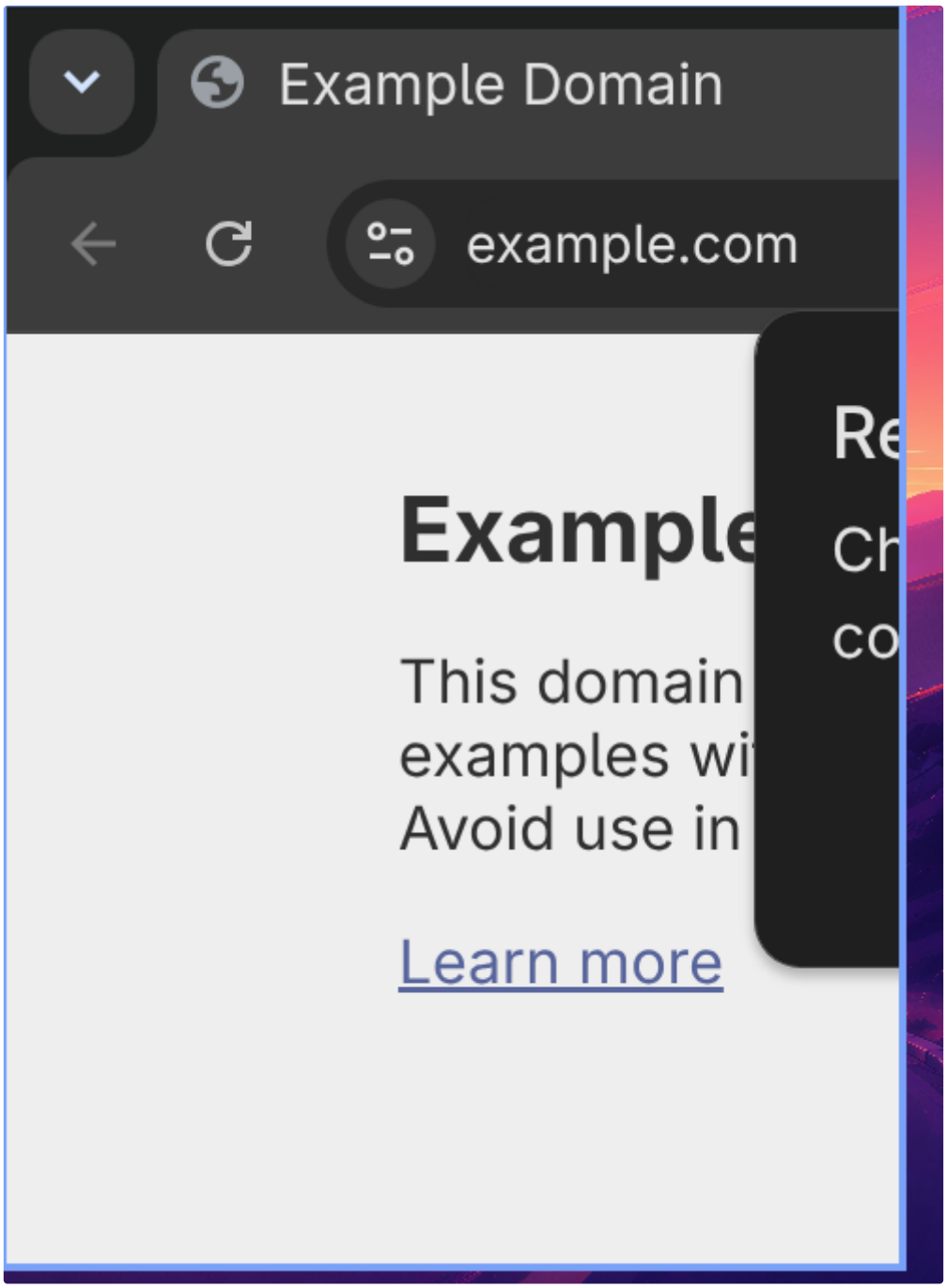
```
Super + Shift + Return
```

This launches your configured default browser.

In our baseline:

```
Chromium
```

Again, Omarchy launches the configured default rather than forcing one browser forever.



Screenshot 3.8-A
browser opened with shortcut

3.9 — See Tiling in Action

Open:

1. terminal;
2. browser.

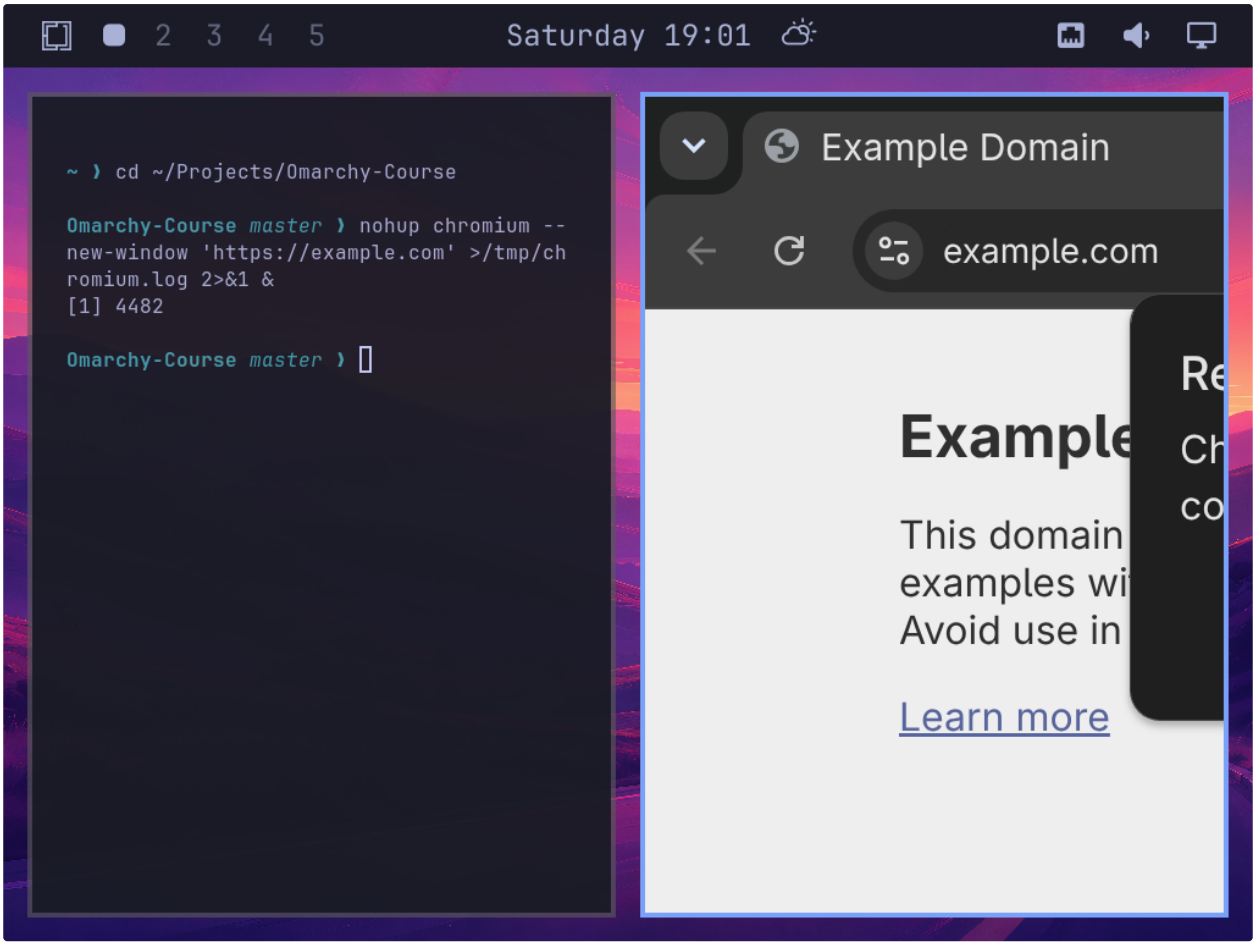
You should see Hyprland automatically arrange both windows.

Conceptually:

You did not manually resize the windows.

Hyprland tiled them.

This is one of the defining behaviors of the Omarchy desktop.



Screenshot 3.9-A
two tiled windows side by side

3.10 — Close a Window

Use:

```
Super + W
```

Current Omarchy also exposes:

```
Super + Q
```

for closing the focused window.

You only need to remember one.

STEVINATOR RECOMMENDATION

Use:

```
Super + W
```

because it is easy to associate with "window."

3.11 — Move Focus Between Windows

When multiple tiled windows are visible, move focus using:

```
Super + Left
Super + Right
Super + Up
Super + Down
```

Example:

```
Super + Right
```

moves focus to the window on the right.

This is much faster than reaching for the mouse.

3.12 — How to Know Which Window Is Focused

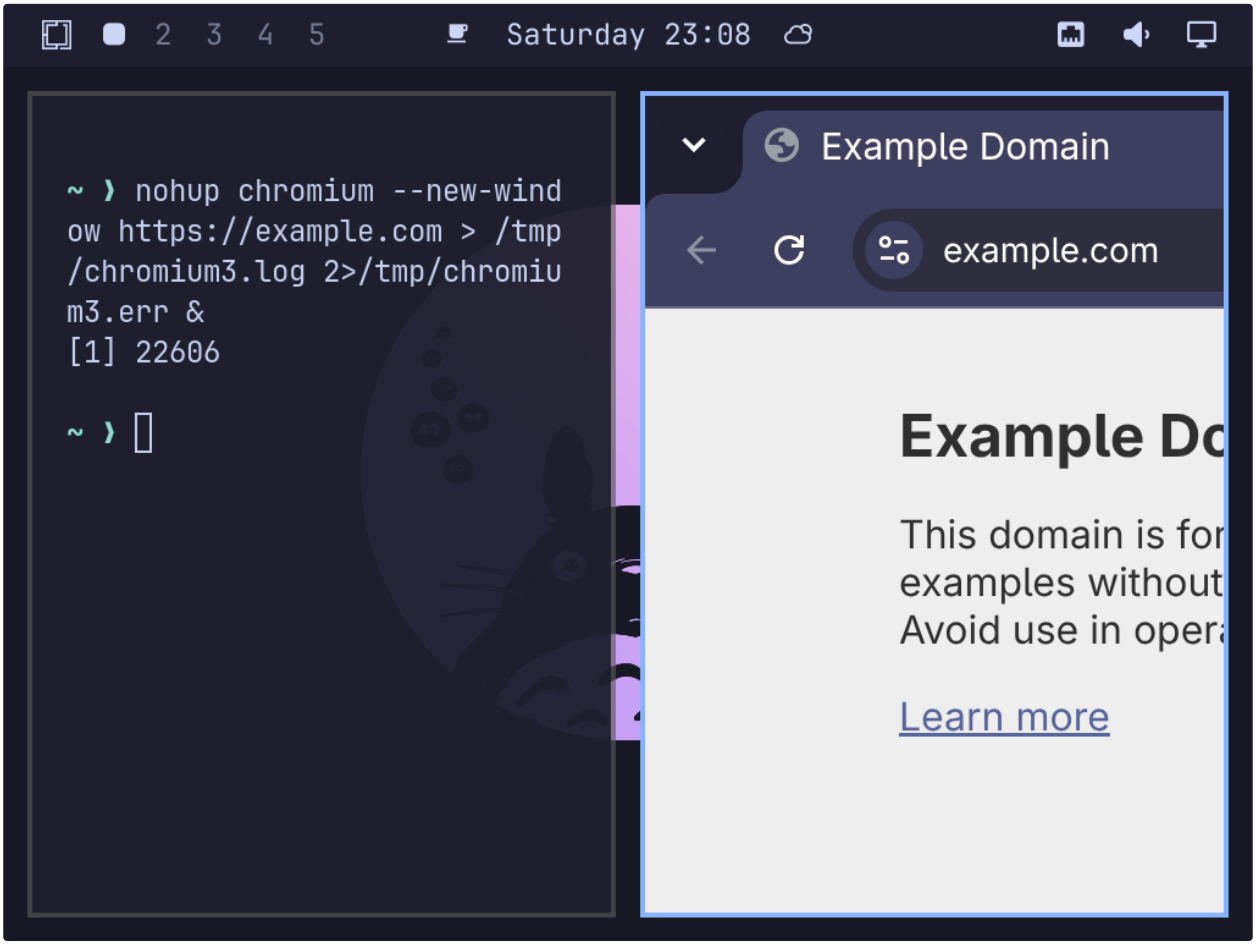
The exact visual styling depends on the current theme, but the active window generally has some visual distinction.

You can also inspect the focused window from the terminal:

```
hyprctl activewindow
```

Later we will use this command for window rules.

For daily navigation, rely on the visual focus indicator.



Screenshot 3.12-A
focused-window visual indicator

3.13 — Swap Tiled Windows

Use:

```
Super + Shift + Arrow
```

Example:

```
Super + Shift + Right
```

swaps the focused window with the window to the right.

Conceptually:

Before:

```
┌──────────────┬──────────────┐
│   Terminal   │    Browser   │
└──────────────┴──────────────┘
```

After swapping right from Terminal:

```
┌──────────────┬──────────────┐
│    Browser   │   Terminal   │
└──────────────┴──────────────┘
```

This is one of the most useful Hyprland habits.

3.14 — Toggle Tiling and Floating

Press:

```
Super + T
```

This toggles the focused window between:

```
tiled
↕
floating
```

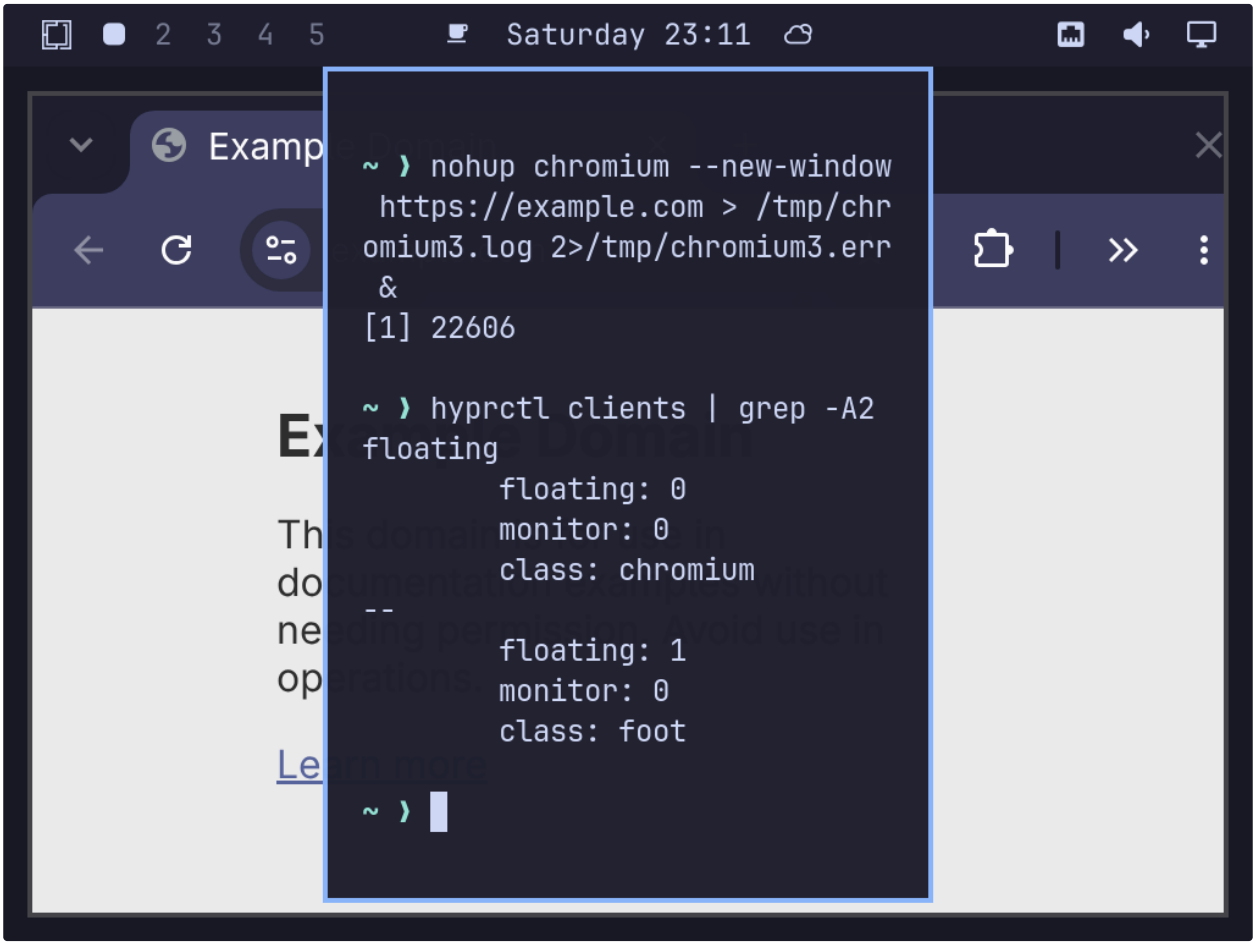
Use floating mode when a window should not occupy a permanent tile.

Examples:

- small utility windows;
- dialogs;
- media controls;
- temporary tools.

Do not float everything.

The tiling workflow is one of the main advantages of Omarchy.



Screenshot 3.14-A
floating window

3.15 — Fullscreen

Press:

```
Super + F
```

This makes the focused window fullscreen.

Use it for:

- coding;
- videos;
- focused writing;

- presentations.

Press it again to return.

Screenshot 3.15-A
fullscreen window

3.16 — Full Width

Press:

`Super + Alt + F`

This expands the focused window across the available workspace width while preserving Omarchy's window-management context.

This is useful when you want more horizontal space without necessarily using full screen.

Examples:

- large editor;

- wide terminal output;
- data table;
- browser dev tools.

Screenshot 3.16-A
full-width window

3.17 — Fullscreen Inside the Window

Current Omarchy hotkeys also include:

```
Super + Ctrl + F
```

for fullscreening content inside the focused window where supported.

This is a more specialized shortcut.

You do not need it for normal navigation.

3.18 — Change the Tiling Orientation

Current Omarchy uses:

```
Super + J
```

to toggle the arrangement direction.

For example, two windows might change from:

```
side by side
```

to:

```
stacked vertically
```

Conceptually:

```
┌──────────────┬──────────────┐
│   Terminal   │    Browser   │
└──────────────┴──────────────┘
```

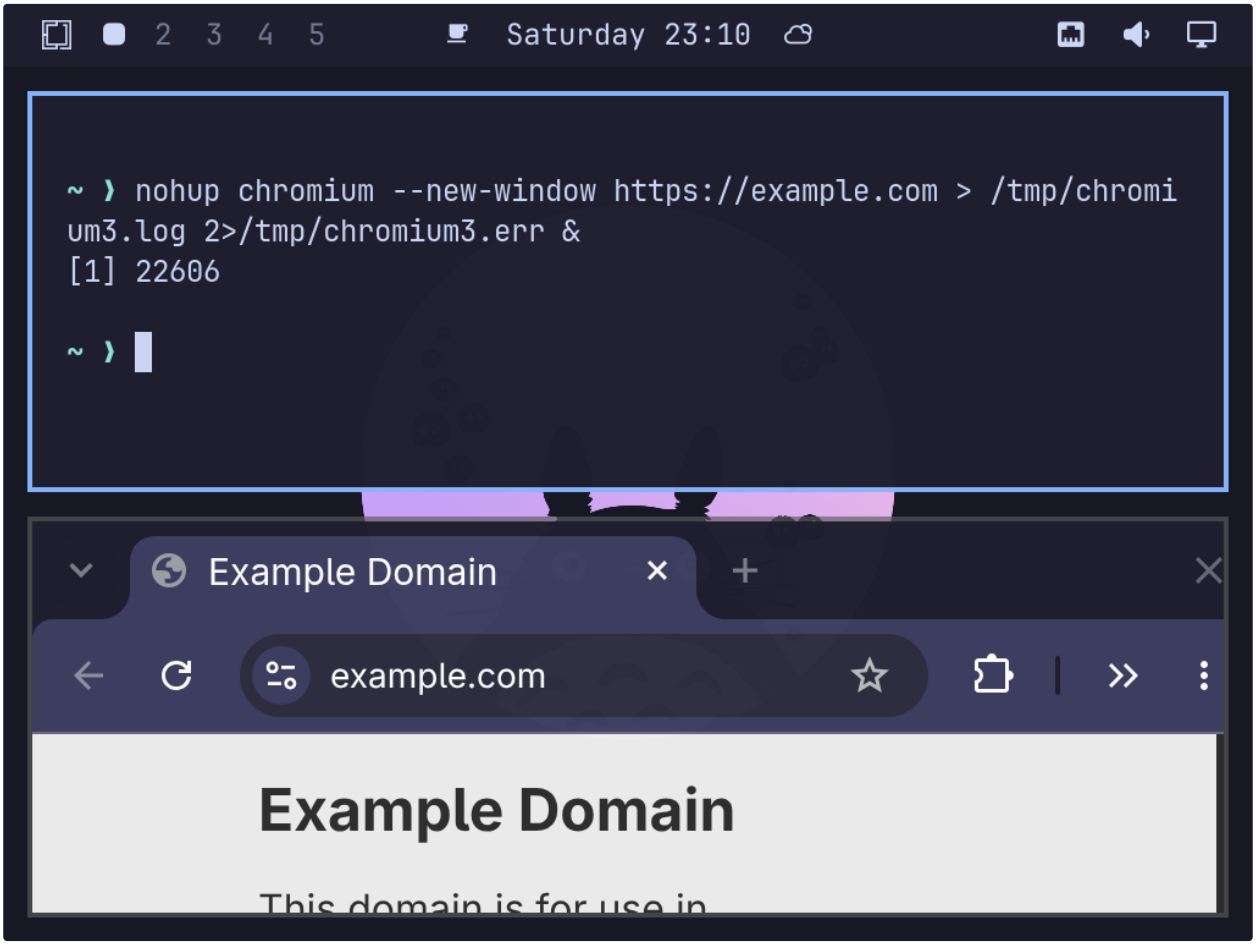
becomes:

```
┌─────────────────────────────┐
│          Terminal           │
├─────────────────────────────┤
│           Browser           │
└─────────────────────────────┘
```

Press:

```
Super + J
```

again to change the orientation back.



Screenshot 3.18-A
two tiled windows stacked

3.19 — Toggle Workspace Layout

Current Omarchy supports multiple Hyprland layout styles.

Press:

```
Super + L
```

to toggle the active workspace between:

```
dwindle
↕
scrolling
```

You do not need to decide which one is "best."

Try both.

Dwindle Layout

Dwindle recursively splits available space.

Conceptually:

This is familiar to many tiling-window-manager users.

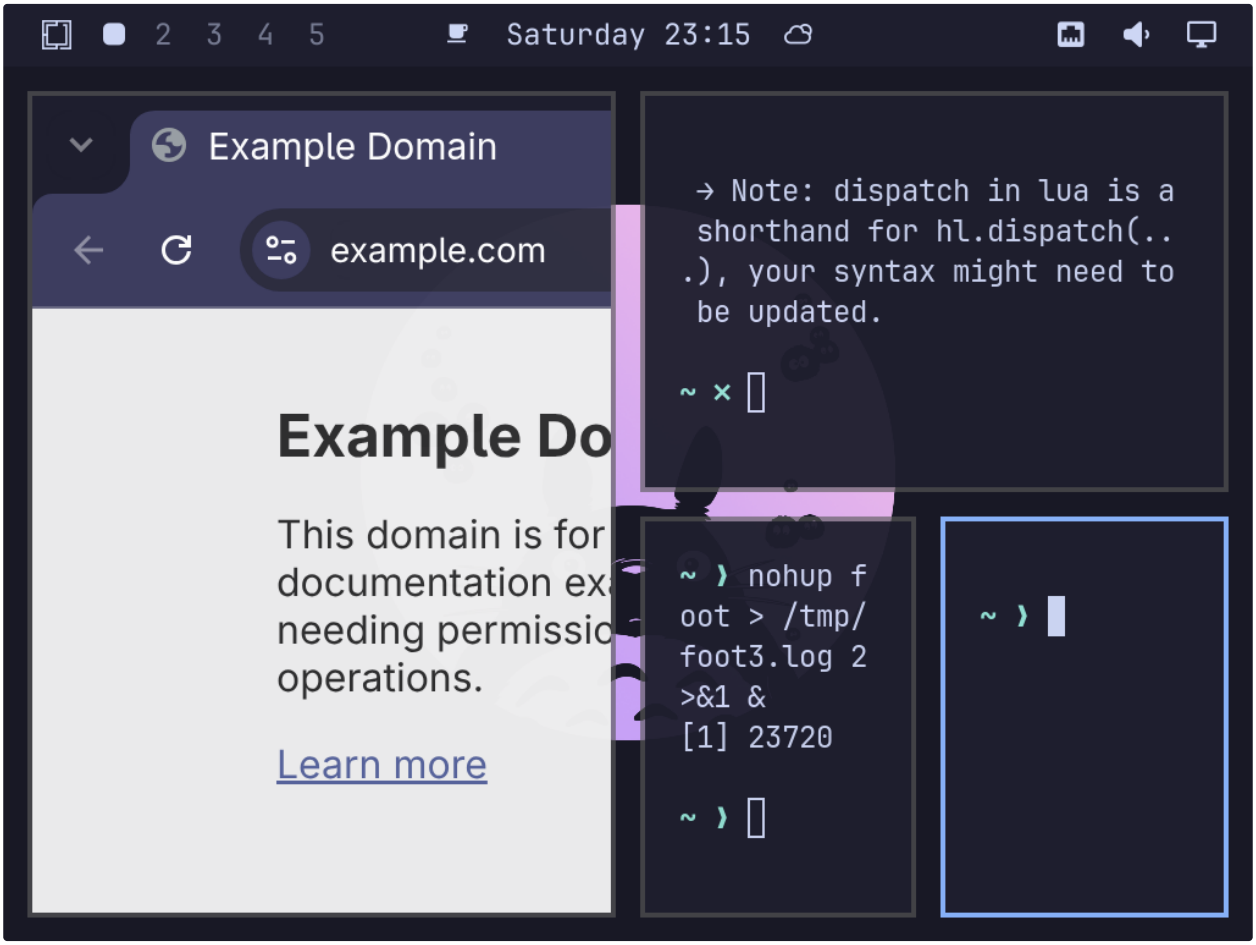
Scrolling Layout

Scrolling layout allows windows to behave more like horizontally arranged columns that you move through.

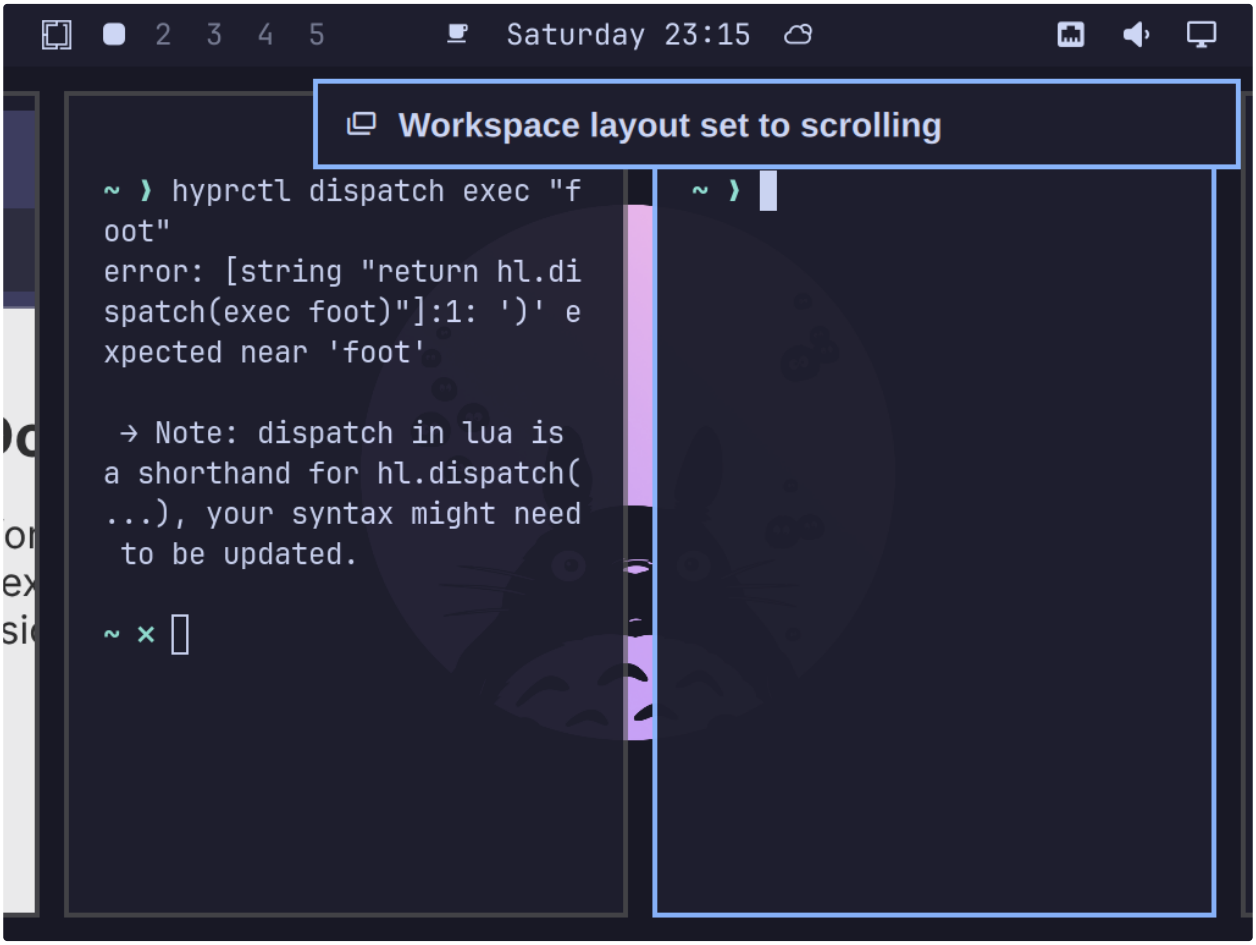
This can be particularly useful on:

- ultrawide displays;
- laptop screens;
- coding workflows with several vertical panes.

Try it rather than choosing based only on screenshots.



Screenshot 3.19-A
dwindle layout



Screenshot 3.19-B
scrolling layout

3.20 — Resize Tiled Windows

Current Omarchy hotkeys expose directional resizing.

The main shortcuts include:

```
Super + Minus
Super + Equal
Super + Shift + Minus
Super + Shift + Equal
```

The exact direction depends on the current layout/orientation.

Current hotkeys also expose:

```
Super + Alt + Minus/Equal
```

for smaller resize steps and:

```
Super + Ctrl + Minus/Equal
```

for larger resize steps.

Mouse Resize

You can also hold:

```
Super + Right Mouse
```

and drag to resize.

For learning, the mouse method can help you understand how tiling sizes behave.

Later, keyboard resizing becomes faster.

3.21 — Save and Restore Window Width

Current Omarchy exposes:

```
Super + Alt + Home
```

to save the current window width and:

```
Super + Home
```

to restore it.

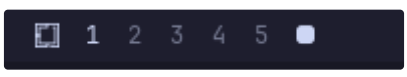
This is useful when you have a preferred working width for a terminal, editor, or browser.

You do not need to use this immediately.

Just know the feature exists.

3.22 — Workspaces

Workspaces are one of the most important parts of the Omarchy workflow.



Screenshot 3.22-A
workspace indicator

Instead of putting every application on one desktop, distribute work across multiple workspaces.

Example:

```
Workspace 1
→ browser / communication

Workspace 2
→ coding

Workspace 3
→ terminal / infrastructure

Workspace 4
→ documentation / research
```

This is much cleaner than minimizing and restoring dozens of overlapping windows.

3.23 — Switch to a Workspace

Use:

```
Super + Number
```

Examples:

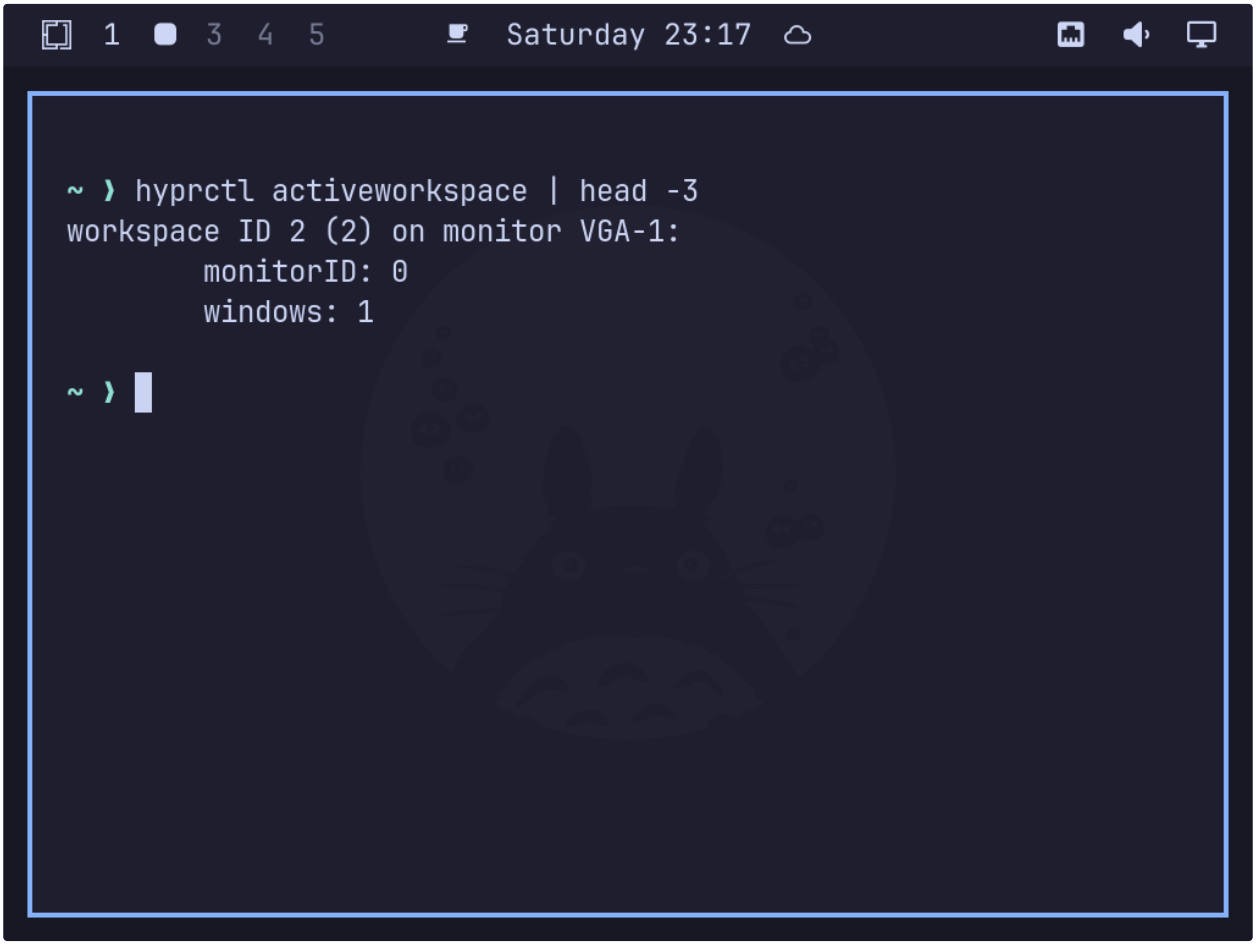
```
Super + 1
Super + 2
Super + 3
```

Current Omarchy defines numeric workspace bindings through the standard number row.

You can use workspaces 1 through 10.

3.24 — Move a Window to Another Workspace

Use:



Screenshot 3.24-A
moving window between workspaces

```
Super + Shift + Number
```

Example:

```
Super + Shift + 2
```

moves the focused window to workspace 2 and follows it.

3.25 — Move a Window Without Following

Current Omarchy also exposes:

```
Super + Shift + Alt + Number
```

Example:

```
Super + Shift + Alt + 3
```

moves the window to workspace 3 **without switching your view to workspace 3**.

This is extremely useful when organizing windows quickly.

Example:

```
browser popup appears
→ send to workspace 1 silently
→ keep coding on workspace 2
```

3.26 — Next and Previous Workspace

Use:

```
Super + Tab
```

for the next workspace.

Use:

```
Super + Shift + Tab
```

for the previous workspace.

This is useful when you are moving sequentially through several active workspaces.

3.27 — Return to the Former Workspace

Press:

```
Super + Ctrl + Tab
```

to jump back to the previous workspace you were using.

This creates a very efficient two-workspace workflow.

Example:

```
Workspace 2 → code
Workspace 1 → browser documentation
```

You can jump back and forth quickly.

3.28 — Scroll Through Workspaces

Current Omarchy also supports:

```
Super + Mouse Wheel
```

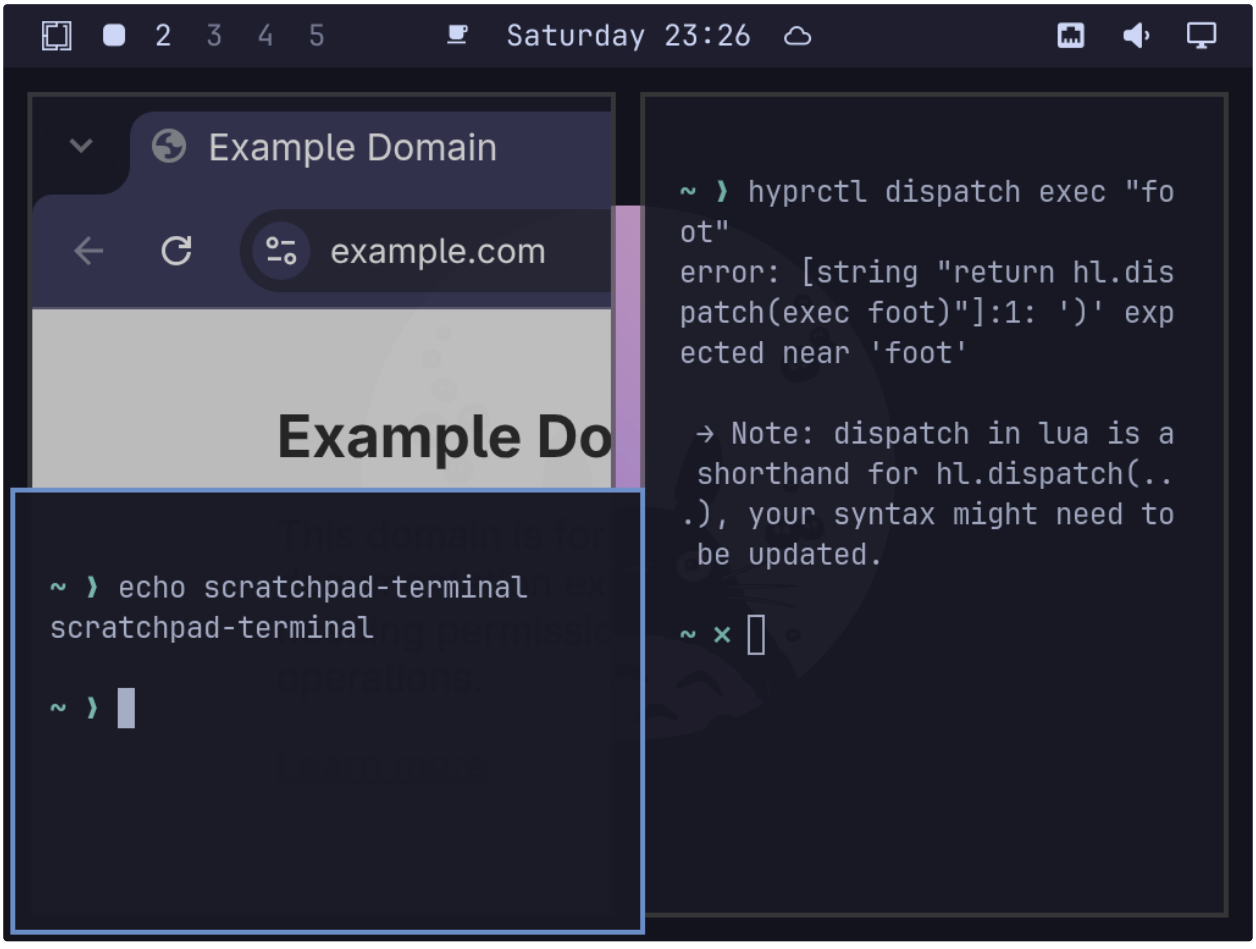
for cycling through workspaces.

This is optional.

Keyboard workspace navigation is generally faster once learned.

3.29 — The Scratchpad

Omarchy includes a special workspace called the **scratchpad**.



Screenshot 3.29-A
scratchpad open

Think of it as:

> *A temporary overlay workspace you can summon over whatever you are doing.*

Toggle it with:

```
Super + S
```

or:

```
Super + Grave
```

where `Grave` is typically the backtick key:

```
`
```

3.30 — Why the Scratchpad Is Useful

Good scratchpad candidates include:

- terminal;
- AI agent;
- music controls;
- calculator;
- monitoring tool;
- notes;
- quick reference.

Example:

```
Coding workspace
        ↓
Super + S
        ↓
temporary terminal/AI overlay
        ↓
Super + S
        ↓
back to code
```

You do not need to leave the current workspace.

3.31 — Move a Window to the Scratchpad

Current Omarchy uses:

```
Super + Alt + S
```

or:

```
Super + Shift + Grave
```

to move the focused window to the scratchpad.

Then:

```
Super + S
```

shows or hides it.

3.32 — Remove a Window from the Scratchpad

Send it to a normal workspace.

Example:

```
Super + Shift + 2
```

moves it to workspace 2.

3.33 — Pop a Window

Current Omarchy provides:

```
Super + O
```

to "pop" a tiled window into a floating, sticky-style state.

This is useful for windows that should remain visible while moving between workspaces.

Examples:

- video;
- monitoring;
- call window.

Do not confuse this with the scratchpad.

Conceptually:

```
Scratchpad
→ summon/hide a special workspace

Pop
→ make one window float/stick across workspace use
```

3.34 — Window Groups

Current Omarchy also exposes Hyprland window grouping.

Toggle grouping with:

```
Super + G
```

Grouped windows can behave like a tabbed group.

This is more advanced than normal tiling.

Inside a group, current Omarchy exposes shortcuts such as:

```
Super + Alt + Tab
```

and:

```
Super + Ctrl + Left/Right
```

for moving through grouped windows.

You do not need groups to be productive.

Treat them as an advanced desktop feature.

3.35 — The Omarchy Panels

Omarchy's top bar exposes interactive panels.

The official current top-bar documentation emphasizes that these are functional control panels, not just status indicators.

Current shortcuts include:

```
Super + Ctrl + A
→ Audio

Super + Ctrl + W
→ Network

Super + Ctrl + B
→ Bluetooth

Super + Ctrl + D
→ Display

Super + Ctrl + P
→ Power

Super + Ctrl + Alt + D
→ Calendar
```

3.36 — Audio Panel

Press:

Screenshot 3.36-A
audio panel

```
Super + Ctrl + A
```

The current Omarchy shell audio panel can expose controls such as:

- master volume;
- output device;
- per-application audio.

PRACTICAL EXERCISE

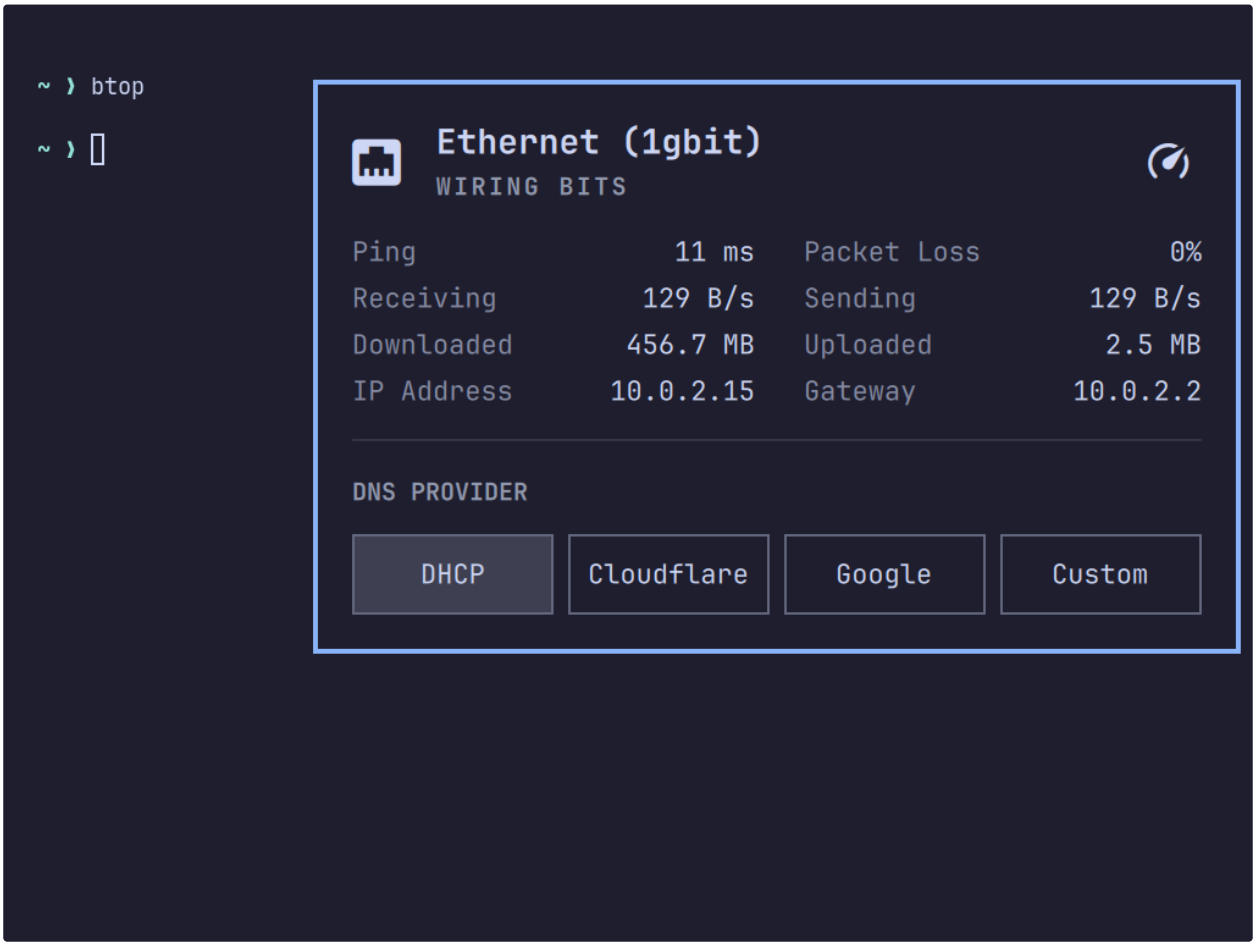
Open:

1. browser audio;
2. audio panel.

Try changing the application volume without changing the entire system volume.

3.37 — Network Panel

Press:



Screenshot 3.37-A
network panel

```
Super + Ctrl + W
```

Current functionality includes:

- Wi-Fi networks;
- signal strength;
- connecting/disconnecting;
- network settings.

Later we will also use:

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

from the terminal.

3.38 — Bluetooth Panel

Press:

Screenshot 3.38-A
Bluetooth panel

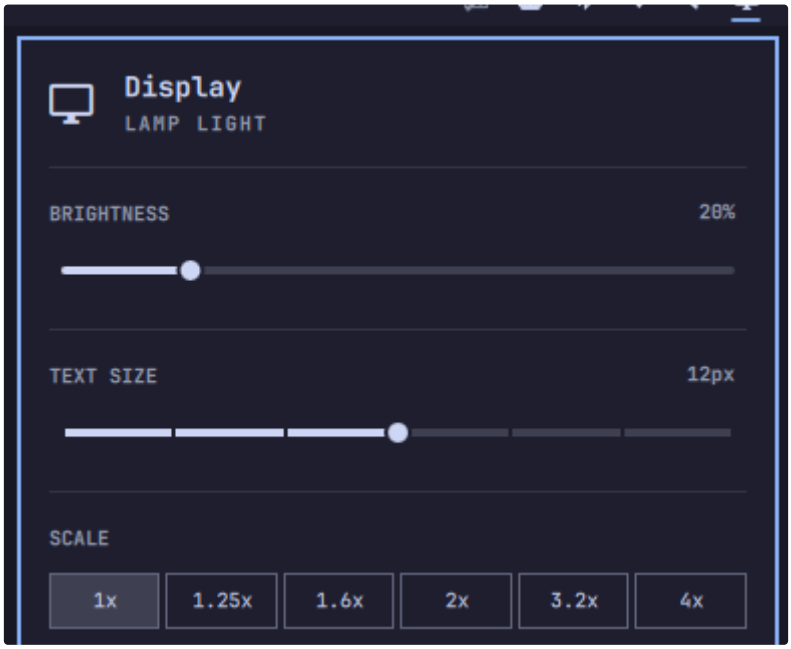
```
Super + Ctrl + B
```

Use it to:

- inspect devices;
- connect;
- disconnect;
- view device state.

3.39 — Display Panel

Press:



Screenshot 3.39-A
display panel

```
Super + Ctrl + D
```

This provides display-related controls.

For precise monitor configuration, we will later use:

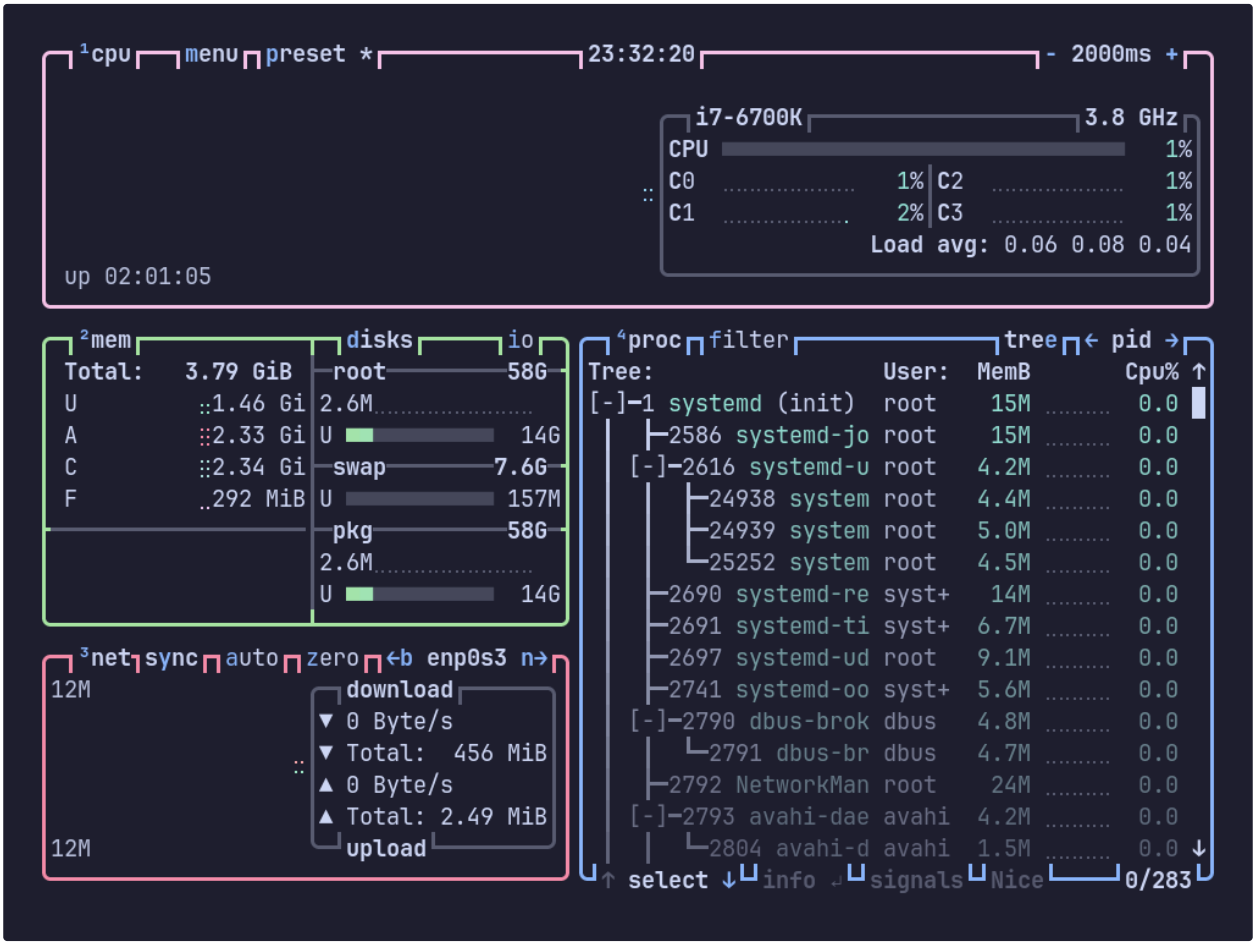
```
~/.config/hypr/monitors.lua
```

The panel is the fast daily interface.

The config is the persistent engineering interface.

3.40 — Activity Monitor

Press:



Screenshot 3.40-A

`btop`

```
Super + Ctrl + T
```

This opens:

```
btop
```

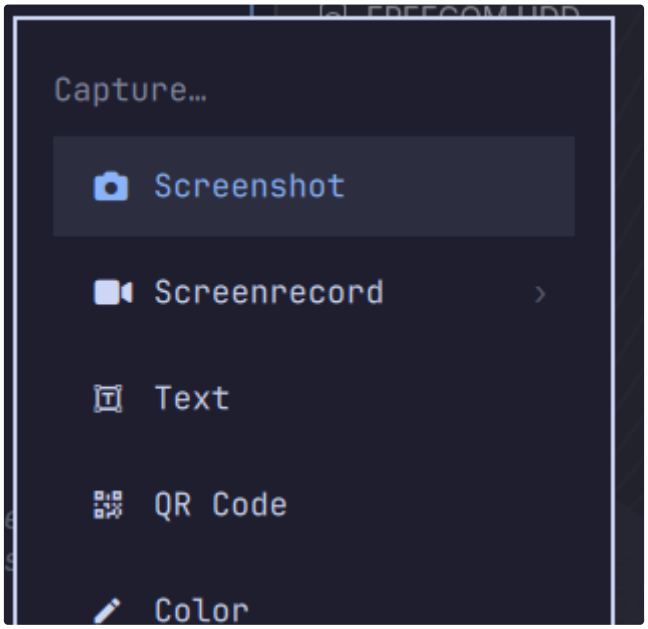
Use it to inspect:

- CPU;
- memory;
- processes;
- load;
- system activity.

This is one of the most useful quick troubleshooting shortcuts.

3.41 — Capture Menu

Press:



Screenshot 3.41-A
capture menu

```
Super + Ctrl + C
```

This opens Omarchy's capture controls.

Current Omarchy capture functionality includes:

- screenshot;
- screen recording;
- text extraction/OCR;
- color picking.

This menu is especially useful for compact keyboards without a dedicated Print Screen key.

3.42 — Print Screen Capture Shortcuts

On keyboards with a working Print Screen key, current Omarchy hotkeys include:

```
Print Screen
→ Screenshot
```

```
Alt + Print Screen
→ Screen recording
```

```
Super + Print Screen
→ Color picker
```

```
Super + Ctrl + Print Screen
→ Text extraction to clipboard
```

Because compact keyboards may map Print Screen differently, verify the actual key event if these do not work.

3.43 — Clipboard

Omarchy provides unified clipboard shortcuts.

Current hotkeys include:

```
Super + C
→ Copy

Super + X
→ Cut

Super + V
→ Paste
```

and:

```
Super + Ctrl + V
```

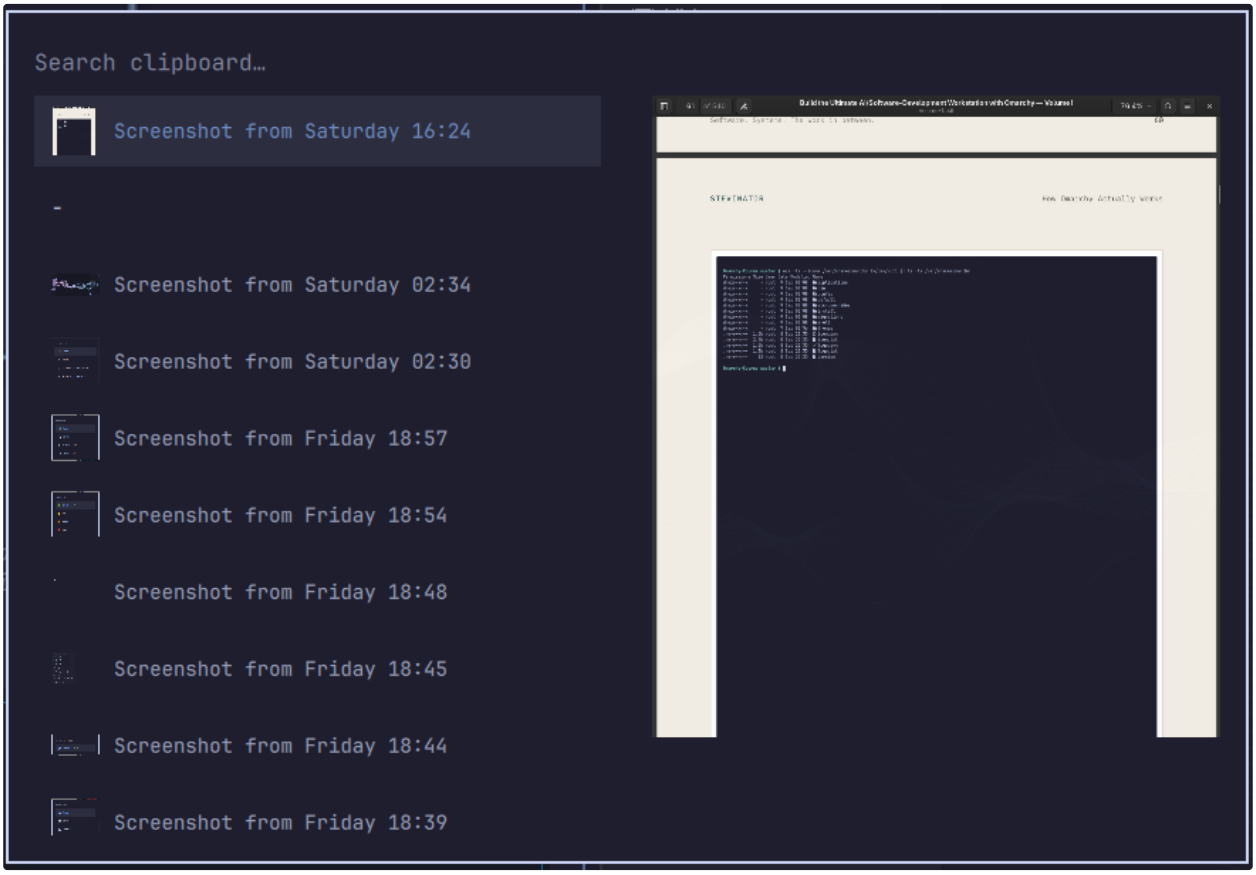
for the clipboard manager/history.

This is useful because traditional Linux terminal copy/paste shortcuts can differ from graphical applications.

Omarchy attempts to make the desktop behavior more consistent.

3.44 — Clipboard History

Press:



Screenshot 3.44-A
clipboard history

```
Super + Ctrl + V
```

This opens clipboard history.

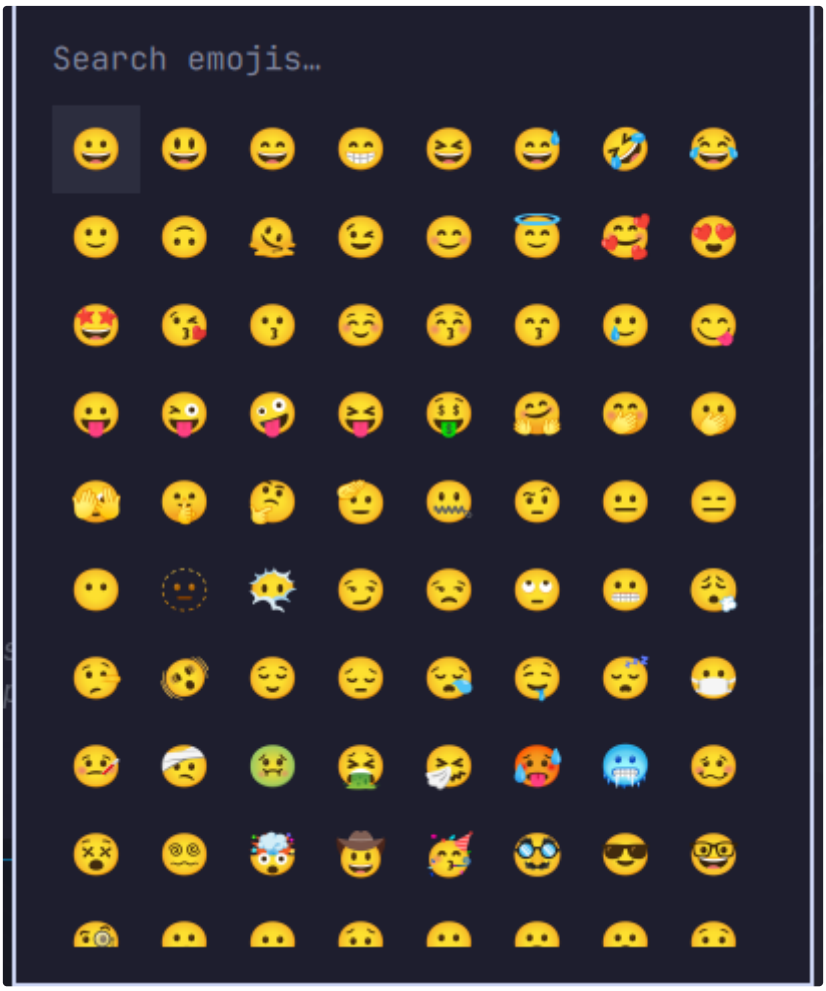
This is incredibly useful during development.

Examples:

- reuse multiple copied commands;
- recover a previous URL;
- paste an older code snippet;
- switch between copied values.

3.45 — Emoji Picker

Press:



Screenshot 3.45-A
emoji picker

```
Super + Ctrl + E
```

This opens the emoji picker.

Select an emoji and it becomes available through the clipboard workflow.

You may not use it constantly for programming, but it is useful for:

- documentation;
- communication;
- README files;
- project status;
- issue discussions.

3.46 — Calculator

Current Omarchy includes:

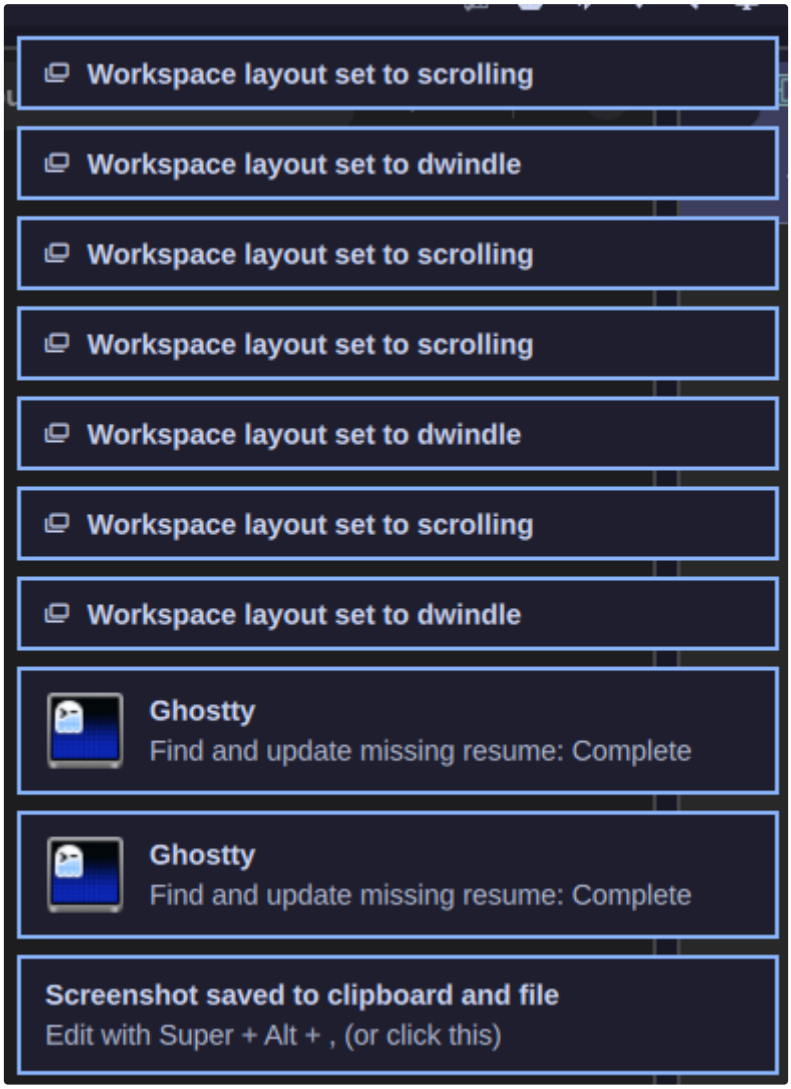
```
Super + Ctrl + Q
```

for a calculator interface.

Useful for quick calculations without opening a browser.

3.47 — Notifications

Current Omarchy provides several notification shortcuts.



Screenshot 3.47-A
notification history

Useful ones include:

```
Super + ,
→ dismiss latest notification
```

```
Super + Shift + ,
→ dismiss all notifications
```

```
Super + Ctrl + ,
→ toggle notification silencing
```

```
Super + Shift + Alt + ,
→ notification history
```

Notification silencing is effectively Omarchy's do-not-disturb mode.

This is useful during focused coding sessions.

3.48 — Style Shortcuts

Current Omarchy exposes quick visual customization.

Examples include:

```
Super + Ctrl + Shift + Space
→ choose theme
```

```
Super + Ctrl + Space
→ choose theme background
```

```
Super + Backspace
→ toggle transparency for focused window
```

We will cover themes properly later.

For now, know that visual customization is integrated into the same keyboard-first workflow.

3.49 — Scaling Shortcuts

Current Omarchy exposes monitor scaling steps through:

```
Super + /
```

and:

```
Super + Alt + /
```

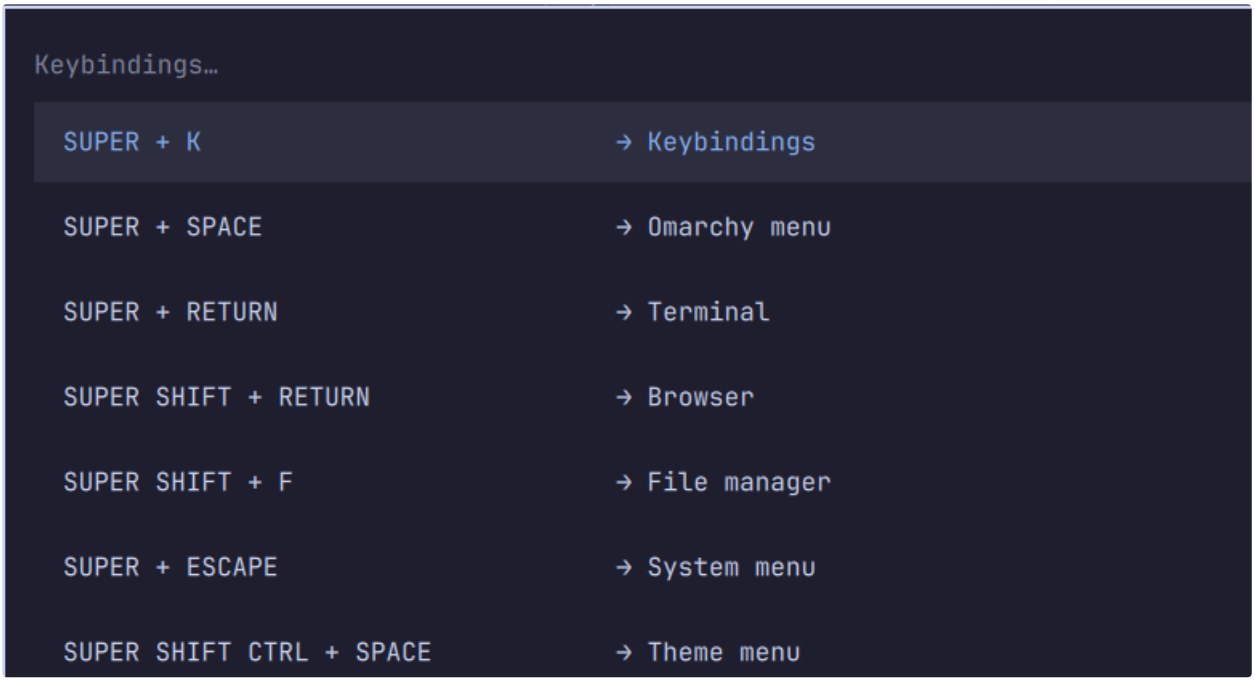
These step through monitor scaling options.

Do not randomly change scaling on a working workstation unless you understand what you are testing.

Persistent display configuration belongs in the later display/Hyprland modules.

3.50 — Discover Current Keybindings

This is one of the most important shortcuts in the entire course:



Screenshot 3.50-A
`Super + K` keybinding search

```
Super + K
```

The official Omarchy hotkey documentation recommends this as the main way to inspect current bindings.

This is better than:

- memorizing a screenshot;
- relying on a year-old YouTube video;
- assuming our handbook version exactly matches your installed version.

3.51 — CLI Keybinding Discovery

You can also use:

```
omarchy menu keybindings
```

On current versions, the keybinding menu may also support output-oriented options such as:

```
omarchy menu keybindings --print
```

if exposed by your installed version.

Verify current syntax with:

```
omarchy menu keybindings --help
```

3.52 — Why We Verify Instead of Memorize

During the real workstation setup used to create this course, several shortcut assumptions turned out to be wrong for the installed configuration.

That taught us an important rule:

> ***The installed keybinding map is the source of truth for your machine.***

So when a shortcut does not work:

```
do not immediately edit config
```

Instead:

```
1. open Super + K
2. search for the action
3. verify the current shortcut
4. only then troubleshoot further
```

3.53 — Practical Developer Workspace

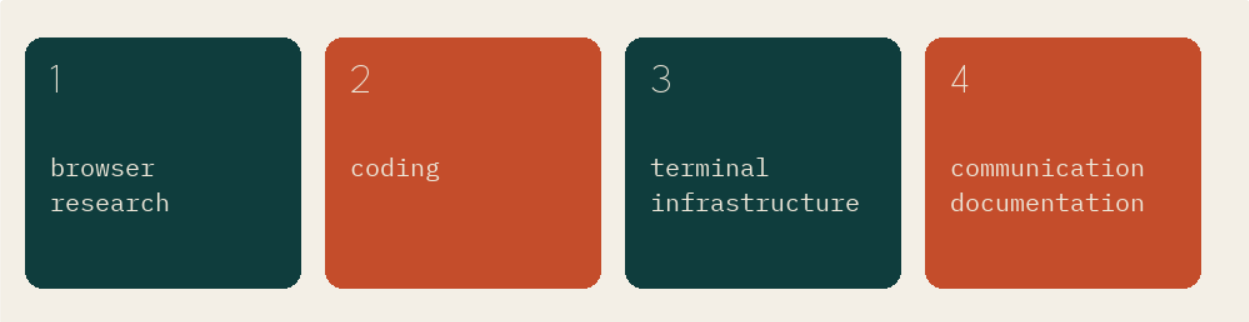
Now let's build a useful daily layout.

We will use:

```
Workspace 1
→ browser / research

Workspace 2
→ coding

Workspace 3
→ terminal / infrastructure

Workspace 4
→ communication / documentation
```

You do not have to keep this layout forever.

It is a training exercise.



Screenshot 3.53-A
four-workspace developer workflow diagram

3.54 — Exercise: Build Workspace 1

Switch:

```
Super + 1
```

Open browser:

```
Super + Shift + Return
```

This becomes your:

```
research / docs workspace
```

3.55 — Exercise: Build Workspace 2

Switch:

```
Super + 2
```

Open terminal:

```
Super + Return
```

Later, this workspace will contain:

- editor;
- coding agent;
- terminal.

For now, just establish the workspace.

3.56 — Exercise: Build Workspace 3

Switch:

```
Super + 3
```

Open another terminal.

This can become:

```
infrastructure
logs
servers
Git operations
```

3.57 — Practice Moving Windows

Focus a terminal.

Move it to workspace 4:

```
Super + Shift + 4
```

Then move it silently back to workspace 3:

```
Super + Shift + Alt + 3
```

Verify that you remain on workspace 4 when moving silently.

3.58 — Practice Former Workspace

Switch between:

```
Workspace 1
Workspace 2
```

Then press:

```
Super + Ctrl + Tab
```

several times.

This is extremely useful for:

```
code
↔
documentation
```

3.59 — Practice the Scratchpad

Open a terminal.

Move it to scratchpad:

```
Super + Alt + S
```

Toggle scratchpad:

```
Super + S
```

Hide it:

```
Super + S
```

Show it again:

```
Super + S
```

Then move the terminal back:

```
Super + Shift + 2
```

3.60 — Practice Tiling

On one workspace, open:

1. terminal;
2. browser.

Use:

```
Super + Left/Right
```

to change focus.

Use:

```
Super + Shift + Left/Right
```

to swap them.

Use:

```
Super + J
```

to change orientation.

Use:

```
Super + T
```

to float one.

Then:

```
Super + T
```

again to tile it.

3.61 — Practice Fullscreen and Full Width

With a browser or terminal focused:

```
Super + F
```

Return.

Then:

```
Super + Alt + F
```

Return.

Notice the difference.

3.62 — Practice Layout Switching

Open several windows on one workspace.

Press:

```
Super + L
```

Move between them.

Then press:

```
Super + L
```

again.

Decide which layout feels more natural for your display.

3.63 — Practical Exercise: No Mouse for Five Minutes

This exercise is intentionally simple.

For five minutes, do the following without using the mouse:

1. open a browser;
2. open two terminals;
3. move one terminal to another workspace;
4. switch between workspaces;
5. move focus between tiled windows;
6. swap two windows;
7. fullscreen one;
8. open the audio panel;
9. open the clipboard manager;
10. open the keybinding search.

The point is not speed.

The point is teaching your hands the Omarchy model.

3.64 — Common Desktop Mistakes

These will be explored more deeply in the troubleshooting module.

Mistake 1 — Trying to Operate Omarchy Like Windows

Constantly:

- minimizing;
- dragging;
- resizing;
- stacking overlapping windows;

works against the tiling workflow.

Try workspaces and focus movement first.

Mistake 2 — Memorizing Too Many Shortcuts

You do not need 50 shortcuts.

Remember:

```
Super + Space
Super + K
```

and discover the rest when needed.

Mistake 3 — Assuming a Shortcut Is Broken

First check:

```
Super + K
```

Your version or personal configuration may differ.

Mistake 4 — Confusing Clipboard Paste With Clipboard History

Current Omarchy:

```
Super + V
→ paste

Super + Ctrl + V
→ clipboard manager
```

Mistake 5 — Confusing Share With Scratchpad

Current Omarchy uses:

```
Super + S
→ scratchpad
```

while sharing functionality is under a different shortcut.

Always verify current keybindings.

Mistake 6 — Floating Everything

Floating mode is useful.

But if every application is floating, you lose much of the benefit of a tiling environment.

Mistake 7 — Putting Everything on Workspace 1

Workspaces are not an advanced feature.

They are central to the intended workflow.

Use them.

3.65 — Checkpoint

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

Core Navigation

- [ ] I can open the Omarchy Menu.
- [ ] I can open the app launcher.
- [ ] I can open the system menu.
- [ ] I can open a terminal.
- [ ] I can open a browser.
- [ ] I can close a window.
- [ ] I can open current keybinding help.

Window Management

- [ ] I can move focus with `Super + Arrow`.
- [ ] I can swap tiled windows.
- [ ] I can toggle floating.
- [ ] I can fullscreen a window.
- [ ] I understand full-width mode.
- [ ] I can change tiling orientation.
- [ ] I can switch workspace layout.

Workspaces

- [ ] I can switch to numbered workspaces.
- [ ] I can move a window to another workspace.
- [ ] I can move a window silently.
- [ ] I can go to the next and previous workspace.
- [ ] I can return to the former workspace.

Scratchpad

- [ ] I understand what the scratchpad is.
- [ ] I can toggle it.
- [ ] I can send a window to it.
- [ ] I can move a window back to a normal workspace.

System Controls

- [ ] I can open audio.
- [ ] I can open network.
- [ ] I can open Bluetooth.
- [ ] I can open display controls.
- [ ] I can open `btop`.
- [ ] I can open capture tools.

Productivity

- [ ] I know how to open clipboard history.
- [ ] I know how to open the emoji picker.
- [ ] I know how to silence notifications.
- [ ] I can discover a shortcut rather than guess it.

3.66 — What You Can Now Do

After completing Module 3, you can now:

- navigate Omarchy primarily from the keyboard;
- organize work across workspaces;
- move focus without using the mouse;
- rearrange tiled windows;
- choose between tiled and floating modes;
- use fullscreen and full-width modes;
- use the scratchpad for temporary tools;
- use Omarchy's built-in panels;
- inspect system activity quickly;
- use clipboard history;
- capture screenshots and recordings;
- control notifications;
- discover current shortcuts;
- create a practical multi-workspace developer layout.

Most importantly:

> *You have started using Omarchy as a tiling workstation rather than treating it like a traditional overlapping-window desktop.*

Module 3 Summary

Core entry points:

```
Super + Space
→ Omarchy Menu

Super + Alt + Space
→ Apps

Super + Escape
→ System Menu

Super + K
→ Keybindings
```

Core applications:

```
Super + Return
→ Terminal

Super + Shift + Return
→ Browser
```

Window management:

```
Super + Arrow
→ Focus

Super + Shift + Arrow
→ Swap

Super + T
→ Tile / float

Super + J
→ Change tiling direction

Super + F
→ Fullscreen

Super + Alt + F
→ Full width

Super + L
→ Toggle layout
```

Workspaces:

```
Super + Number
→ Switch workspace

Super + Shift + Number
→ Move window + follow

Super + Shift + Alt + Number
→ Move window silently

Super + Tab
→ Next workspace

Super + Shift + Tab
→ Previous workspace

Super + Ctrl + Tab
→ Former workspace
```

Scratchpad:

```
Super + S
→ Toggle

Super + Alt + S
→ Move window to scratchpad
```

System panels:

```
Super + Ctrl + A
→ Audio

Super + Ctrl + B
→ Bluetooth

Super + Ctrl + W
→ Network

Super + Ctrl + D
→ Display

Super + Ctrl + T
→ Activity

Super + Ctrl + C
→ Capture
```

Productivity:

```
Super + Ctrl + V
→ Clipboard history

Super + Ctrl + E
→ Emoji picker

Super + Ctrl + ,
→ Notification silencing
```

The most important lesson:

> ***Do not memorize Omarchy. Learn how to discover Omarchy.***

MODULE 04

04

Terminal Mastery for Omarchy Developers

The terminal is one of the most important parts of an Omarchy workstation.

You already used it during installation, system verification, GitHub authentication, service checks, and command discovery.

About This Module

The terminal is one of the most important parts of an Omarchy workstation.

You already used it during installation, system verification, GitHub authentication, service checks, and command discovery.

Now we turn it into a daily development tool.

This module is **not** about becoming a Bash expert.

It is about learning the terminal skills that give you the biggest return as a developer:

- navigating quickly;
- inspecting files;
- searching code;
- chaining commands;
- understanding command output;
- using shell history;
- finding commands;
- working safely with configuration;
- using the enhanced command-line tools that Omarchy already provides.

Current Omarchy ships a collection of improved shell tools, including:

- `fzf`
- `zoxide`
- `ripgrep` (`rg`)
- `eza`
- `fd`
- `bat`
- `tldr`

These are not random extras.

They are part of the intended Omarchy workflow.

By the end of this module, the terminal should stop feeling like a fallback interface and start feeling like one of the fastest ways to work.

What You Will Learn

By the end of Module 4, you will be able to:

- open and identify your default terminal;
- understand the relationship between terminal emulator and shell;
- read a command line structurally;
- navigate directories quickly;
- create, copy, move, and remove files safely;
- inspect text files;
- search filenames with `fd`;
- search file contents with `rg`;
- use `eza`-based directory listings;
- use `bat` for readable file output;
- fuzzy-find files with `fzf`;
- fuzzy-search command history with `Ctrl + R`;
- jump between frequently used directories with zoxide;
- use pipes;
- redirect output;
- combine commands;
- understand standard output and standard error;
- use quotes correctly;
- understand shell expansion;
- inspect aliases, functions, and executables with `type`;
- read command help and manual pages;
- add your own aliases and environment variables safely to `~/.bashrc`;
- avoid common shell mistakes.

4.1 — Terminal Emulator vs Shell

These two terms are often confused.

They are not the same thing.

Terminal Emulator

The terminal emulator is the graphical application displaying the terminal window.

Current Omarchy supports terminal choices including:

- Foot
- Alacritty
- Ghostty
- Kitty

Current official Omarchy documentation identifies **Foot** as the default terminal for a standard installation, while allowing supported alternatives to be installed and selected.

Our Stevinator workstation baseline uses:

```
Ghostty
```

The exact terminal does not fundamentally change the shell skills in this module.

Shell

Inside the terminal runs a shell.

Check your configured shell:

```
echo "$SHELL"
```

On the workstation used to build this course:

```
/usr/bin/bash
```

The shell is responsible for interpreting commands such as:

```
cd ~/Projects
```

or:

```
rg "TODO" src
```

Conceptually:

```
Ghostty
→ terminal window

Bash
→ reads and interprets your commands

Linux/Omarchy tools
→ perform the actual work
```

4.2 — Check Your Default Terminal

Omarchy manages the selected terminal through its default application layer.

Run:

```
omarchy default terminal
```

Current Omarchy's terminal default command integrates with `xdg-terminal-exec`.

That means shortcuts and Omarchy launch helpers can target the configured terminal instead of hard-coding one application.

To inspect the supported choices on your installed version:

```
omarchy default terminal --help
```

or:

```
omarchy commands --all | grep "default terminal"
```

4.3 — Launch a Terminal

Current Omarchy shortcut:

```
Super + Return
```

This launches the configured default terminal.

Current Omarchy also provides:

```
Super + Alt + Return
```

for launching a terminal with Tmux.

We will cover Tmux in the next module.

For now use:

```
Super + Return
```

4.4 — Anatomy of a Command

Consider:

```
rg "TODO" ~/Projects
```

Break it apart:

```
rg
```

is the command.

```
"TODO"
```

is an argument.

```
~/Projects
```

is another argument.

Now consider:

```
fd -H "config"
```

Here:

```
fd
```

is the command.

```
-H
```

is an option/flag.

```
"config"
```

is an argument.

This pattern appears everywhere:

```
COMMAND [OPTIONS] [ARGUMENTS]
```

4.5 — Command Help

When you do not remember syntax, do not guess.

Try:

```
COMMAND --help
```

Examples:

```
rg --help
```

```
fd --help
```

```
bat --help
```

For Omarchy commands:

```
omarchy network --help
```

This habit is more valuable than memorizing hundreds of options.

4.6 — Manual Pages

Many Linux tools include manual pages.

Use:

```
man COMMAND
```

Example:

```
man rg
```

or:

```
man fd
```

Navigation inside `man` commonly includes:

```
Arrow keys / Page Up / Page Down
→ move

/
→ search

q
→ quit
```

4.7 — `tldr`: Practical Command Examples

Traditional manual pages are comprehensive but can be dense.

Current Omarchy includes `tldr`, which provides concise examples.

Try:

```
tldr tar
```

or:

```
tldr grep
```

This is often useful when you know the command but cannot remember the common syntax.

A useful learning sequence is:

```
tldr
→ quick examples

--help
→ available options

man
→ full documentation
```

4.8 — Your Current Directory

Check where you are:

```
pwd
```

Example:

```
/home/alex
```

`pwd` means:

> *print working directory*

Almost every relative path you type is interpreted from this location.

4.9 — Basic Navigation

Move to your projects directory:

```
cd ~/Projects
```

Move home:

```
cd ~
```

or simply:

```
cd
```

Move one directory upward:

```
cd ..
```

Move into a directory:

```
cd path/to/directory
```

4.10 — Tab Completion

Do not type long paths manually when the shell can complete them.

Start typing:

```
cd ~/Pro
```

then press:

```
Tab
```

If the match is unambiguous, Bash completes it.

Tab completion works for many:

- paths;
- commands;
- filenames;
- options.

This reduces typos significantly.

4.11 — Listing Files

You already know:

```
ls
```

On current Omarchy, the shell environment uses `eza` to provide richer directory listings.

The official Omarchy shell-tools documentation states that `eza` is used as the enhanced replacement behind the default `ls` experience.

Try:

```
ls
```

Then inspect what `ls` actually is:

```
type ls
```

You may discover that it is an alias rather than raw `/usr/bin/ls`.

This is why we learned `type` in Module 2.

4.12 — Omarchy Directory Listing Helpers

Current official Omarchy shell tooling includes useful listing aliases.

Common Omarchy aliases include:

```
ls
```

for the enhanced directory listing.

```
lsa
```

for showing hidden entries.

```
lt
```

for a nested/tree-style view.

```
lta
```

for a nested view including hidden files.

Try all four inside:

```
cd ~/.config
```

Then:

```
ls
```

```
lsa
```

```
lt
```

```
lta
```

Do not worry if exact formatting varies with the current theme/version.

4.13 — Inspect Aliases

Run:

```
type ls
```

Then:

```
type lsa
```

Then:

```
type lt
```

The output tells you how Bash resolves each command.

You can also list shell aliases with:

```
alias
```

Do not memorize them all.

Use:

```
type NAME
```

when you encounter an unfamiliar short command.

4.14 — Create a Practice Directory

Let's create a safe sandbox.

Run:

```
mkdir -p ~/Projects/terminal-practice
```

Enter it:

```
cd ~/Projects/terminal-practice
```

Verify:

```
pwd
```

Everything in the next exercises can happen here.

4.15 — Create Directories

Create a directory:

```
mkdir notes
```

Create nested directories:

```
mkdir -p src/components
```

Check:

```
lt
```

`-p` allows `mkdir` to create missing parent directories.

4.16 — Create an Empty File

Use:

```
touch notes/todo.txt
```

Then:

```
ls notes
```

`touch` can create an empty file if it does not exist.

It can also update timestamps on existing files.

4.17 — Write Simple Text

Use shell redirection:

```
echo "Learn Omarchy terminal tools" > notes/todo.txt
```

Then inspect:

```
cat notes/todo.txt
```

Expected:

```
Learn Omarchy terminal tools
```

We will explain `>` properly shortly.

4.18 — `cat` vs `bat`

Traditional Linux tool:

```
cat notes/todo.txt
```

Omarchy also ships:

```
bat notes/todo.txt
```

Current official Omarchy documentation describes `bat` as a `cat`-style viewer with:

- syntax highlighting;
- line numbers;
- paging.

For source code and configuration, `bat` is usually more pleasant.

Use `cat` when:

- output is tiny;
- you want raw output;
- output is being piped elsewhere.

Use `bat` when:

- reading code;
- reading configuration;
- syntax highlighting helps.

4.19 — Copy Files

Create another file:

```
cp notes/todo.txt notes/todo-backup.txt
```

Inspect:

```
ls notes
```

`cp` copies.

4.20 — Move and Rename Files

Rename:

```
mv notes/todo-backup.txt notes/archive.txt
```

`mv` can both:

- move files;
- rename files.

Move into another directory:

```
mkdir archive
mv notes/archive.txt archive/
```

Check:

```
lt
```

4.21 — Removing Files

Remove a file:

```
rm archive/archive.txt
```

Remove an empty directory:

```
rmdir archive
```

Be careful with `rm`.

Unlike a graphical file manager, command-line removal may not use a trash/recycle bin.

4.22 — The Dangerous `rm -rf`

You will see this command on the internet:

```
rm -rf SOME_PATH
```

It means roughly:

```
-r
→ recursive

-f
→ force
```

This can remove directories and their contents without normal interactive protections.

Do not copy an `rm -rf` command unless you understand the exact path.

A missing variable or typo can be destructive.

COURSE RULE

> ***Read the target path before pressing Enter on recursive deletion commands.***

4.23 — Wildcards

The shell can expand wildcard patterns.

Example:

```
ls *.txt
```

means:

> *all filenames in this directory ending in* `.txt`

Another example:

```
rm *.tmp
```

would remove all `.tmp` files in the current directory.

Wildcards are powerful, which means you should inspect matches before destructive use.

Safer pattern:

```
ls *.tmp
```

first.

Then, if correct:

```
rm *.tmp
```

4.24 — Quoting

Quoting becomes important when arguments contain spaces or shell-special characters.

Create:

```
mkdir "My Notes"
```

Without quotes:

```
mkdir My Notes
```

Bash interprets that as two directory names:

```
My
Notes
```

instead of one.

4.25 — Single Quotes vs Double Quotes

Double quotes allow environment-variable expansion.

Example:

```
echo "Home: $HOME"
```

might print:

```
Home: /home/alex
```

Single quotes generally preserve the text literally:

```
echo 'Home: $HOME'
```

prints:

```
Home: $HOME
```

This distinction matters constantly in shell commands and scripts.

4.26 — Command History

Press:

```
Up Arrow
```

to move through earlier commands.

Press:

```
Down Arrow
```

to move forward again.

For a short session, this is often enough.

But Omarchy provides something much better.

4.27 — Fuzzy History Search

Current Omarchy ships `fzf`.

Press:

```
Ctrl + R
```

You should get fuzzy history search.

Start typing part of an earlier command.

Example:

```
hyprctl
```

You can find previous commands containing that text without remembering the full syntax.

For developers, this quickly becomes one of the highest-value shortcuts in the terminal.

4.28 — Why Fuzzy History Matters

Suppose three days ago you ran:

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

You vaguely remember:

```
network verbose
```

Press:

```
Ctrl + R
```

and search:

```
network
```

Instead of searching documentation again, you recover the exact command from your own history.

4.29 — `fzf` File Search

Current Omarchy provides fuzzy file finding using `fzf`.

The official shell-tools documentation exposes:

```
ff
```

as a fuzzy file finder.

Go to:

```
cd ~/Projects
```

Then:

```
ff
```

You should get an interactive finder with file previews.

Type parts of a filename to narrow results.

4.30 — Searching Filenames with `fd`

Current Omarchy includes `fd`, a friendlier alternative to traditional `find` for many everyday uses.

Return to:

```
cd ~/Projects/terminal-practice
```

Find files containing `todo` in their name:

```
fd todo
```

Find all text files:

```
fd '\.txt$'
```

4.31 — Hidden Files and `fd`

A useful gotcha:

`fd` does not necessarily include hidden files/directories in its default search.

To include hidden entries:

```
fd -H PATTERN
```

Example:

```
fd -H "config" ~
```

The current official Omarchy documentation explicitly shows `-H` when searching hidden directories.

This is important when searching:

```
~/.config
```

4.32 — Searching Inside Files with `rg`

`ripgrep`, invoked as:

```
rg
```

searches file contents.

Create a few sample files:

```
echo "TODO: learn ripgrep" > notes/search.txt
echo "DONE: learn pwd" > notes/done.txt
```

Search:

```
rg "TODO"
```

Expected result should point to:

```
notes/search.txt
```

4.33 — Search a Specific Directory

Example:

```
rg "TODO" notes/
```

Search your Hyprland config for `workspace`:

```
rg "workspace" ~/.config/hypr
```

This is a practical example from our real Omarchy learning journey.

4.34 — Case-Insensitive Search

Use:

```
rg -i "todo"
```

This can match:

```
TODO
Todo
todo
```

4.35 — Show Line Numbers

`rg` normally provides useful context automatically, but you can explicitly request line numbers:

```
rg -n "TODO"
```

This becomes especially useful when discussing results with an AI coding agent or another developer.

4.36 — Search Only Certain File Types

Example:

```
rg "function" -g '*.ts'
```

Search Lua files:

```
rg "o.bind" -g '*.lua' ~/.config/hypr
```

This is extremely useful in real codebases.

4.37 — Why `rg` Beats Manual Browsing

Imagine a project containing:

```
500 files
```

You need to find where:

```
runBacklog
```

is defined.

Do not open files one by one.

Run:

```
rg "runBacklog"
```

That is exactly the kind of codebase navigation workflow we want for this workstation.

4.38 — Pipes

A pipe sends the output of one command into another command.

The pipe character is:

```
|
```

Example:

```
omarchy commands --all | grep network
```

Conceptually:

```
omarchy commands --all
        ↓ output
grep network
        ↓ filtered output
```

4.39 — Pipe Example: Count Results

Search files:

```
fd
```

Pipe into:

```
wc -l
```

Combined:

```
fd | wc -l
```

This roughly counts the number of returned paths.

4.40 — Pipe Example: Search Command Output

You already used:

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

This is a perfect practical example.

The first command produces a large list.

The second command filters it.

4.41 — Standard Output

Commands can produce normal output called:

```
stdout
```

Example:

```
echo "hello"
```

Output:

```
hello
```

4.42 — Standard Error

Commands can also produce error output:

```
stderr
```

For example:

```
ls /path/that/does/not/exist
```

The shell displays an error.

This distinction matters when redirecting output and when debugging scripts.

4.43 — Redirect Output to a File

Use:

```
>
```

Example:

```
omarchy version > version.txt
```

Inspect:

```
cat version.txt
```

Important:

```
>
```

overwrites the target file.

4.44 — Append Output

Use:

```
>>
```

Example:

```
echo "First line" > log.txt
echo "Second line" >> log.txt
```

Inspect:

```
cat log.txt
```

Result:

```
First line
Second line
```

4.45 — Why Redirection Matters

Redirection is useful for:

- saving diagnostics;
- generating reports;
- recording command output;
- building files;
- logging scripts.

Example:

```
hyprctl monitors all > monitor-info.txt
```

Now you can attach or inspect that file later.

4.46 — Redirecting Errors

Common shell syntax:

```
COMMAND 2>/dev/null
```

This discards standard error.

You saw this pattern in some Omarchy-related inspection commands.

Example:

```
grep -R "workspace" ~/.config/hypr /usr/share/omarchy/default/hypr 2>/dev/null
```

Use this intentionally.

Do not hide errors while troubleshooting unless you know why.

4.47 — `/dev/null`

`/dev/null` is a special device that discards anything written to it.

Think:

```
output
→ /dev/null
→ gone
```

Useful for:

- intentionally suppressing noise;
- optional probes;
- scripts where failure output is irrelevant.

Not useful when:

> *you actually need to understand the error.*

4.48 — Running Commands Sequentially

You can place multiple commands on one line using:

```
;
```

Example:

```
pwd; ls
```

This runs:

```
pwd
```

then:

```
ls
```

even if the first command fails.

4.49 — Run the Next Command Only on Success

Use:

```
&&
```

Example:

```
mkdir test && cd test
```

`cd test` runs only if `mkdir test` succeeds.

This is extremely useful.

4.50 — Run the Next Command Only on Failure

Use:

```
||
```

Example:

```
test -f config.toml || echo "config.toml is missing"
```

This pattern appears frequently in shell scripts.

4.51 — Why `&&` Is Safer Than `;`

Compare:

```
cd important-directory; rm something
```

with:

```
cd important-directory && rm something
```

With `;`, the removal command still executes if `cd` fails.

With `&&`, it runs only if the directory change succeeded.

For dependent operations, prefer:

```
&&
```

4.52 — Environment Variables in Commands

Example:

```
echo "$HOME"
```

Use an environment variable inside a path:

```
ls "$HOME/.config"
```

Quotes are generally a good habit around variables that may contain spaces.

4.53 — Temporary Environment Variable for One Command

You can set a variable only for one command:

```
EXAMPLE=value some-command
```

This is common in development.

For example:

```
NODE_ENV=production node app.js
```

The variable applies to that command invocation rather than permanently changing your shell.

4.54 — Inspect a Command with `type`

We covered this earlier, but it belongs in your daily terminal toolbox.

Run:

```
type ls
```

```
type cd
```

```
type python
```

```
type node
```

```
type omarchy
```

`type` tells you what Bash would actually execute.

4.55 — Why This Matters With Mise

Suppose:

```
python --version
```

returns a version you did not expect.

Run:

```
type python
```

The result may reveal that `python` is being provided by:

- system `/usr/bin`;
- mise;
- another wrapper;
- an alias.

This is much more useful than immediately reinstalling Python.

4.56 — Check Executable Location

If `type` reports an executable, you can inspect paths with:

```
command -v python
```

or:

```
which python
```

We prefer:

```
type python
```

as the first diagnostic because it understands aliases and functions too.

4.57 — Exit Codes

Commands report whether they succeeded.

Typically:

```
0
→ success

non-zero
→ some kind of failure
```

After a command, inspect the last exit code:

```
echo $?
```

Example:

```
true
echo $?
```

Expected:

```
0
```

Then:

```
false
echo $?
```

Expected non-zero.

4.58 — Why Silent Commands Can Be Fine

Some commands intentionally print nothing when successful.

This happened during our real Omarchy learning journey.

The absence of output does not automatically mean:

```
nothing happened
```

You may need to check:

- exit code;
- state afterwards;
- corresponding status command.

Example:

```
some-command
echo $?
```

This is an important troubleshooting habit.

4.59 — `head` and `tail`

Show the beginning of output:

```
COMMAND | head
```

Example:

```
omarchy commands --all | head
```

Show the end:

```
COMMAND | tail
```

Useful for large outputs and logs.

4.60 — Follow a Log

A common pattern is:

```
tail -f FILE
```

This follows new lines as they are appended.

You do not need it frequently yet, but it becomes useful for:

- app logs;
- development servers;
- service diagnostics.

Press:

```
Ctrl + C
```

to stop following.

4.61 — `Ctrl + C`

In the terminal:

```
Ctrl + C
```

usually sends an interrupt signal to the foreground process.

Use it to stop commands such as:

- development server;
- endless output;
- `ping`;
- `tail -f`;
- long-running CLI task.

It does **not** mean "copy" in the traditional terminal context.

Omarchy provides unified desktop copy/paste shortcuts at the compositor level, but understand that `Ctrl + C` still has its classic shell meaning inside terminal applications.

4.62 — Clear the Terminal

Use:

```
clear
```

or commonly:

```
Ctrl + L
```

This clears/redraws the terminal view without deleting shell history.

4.63 — Autocomplete and History Are More Important Than Typing Speed

A developer should rarely type repetitive full commands manually.

Use:

```
Tab
→ completion

Up Arrow
→ recent command

Ctrl + R
→ fuzzy history

zoxide
→ remembered directories

ff
→ fuzzy files
```

The goal is not to become a faster typist.

The goal is to type less.

4.64 — Zoxide

Current Omarchy ships zoxide as a smarter directory-jumping tool.

Zoxide learns directories you visit.

Suppose you visit:

```
cd ~/.config/omarchy
```

After zoxide has learned the path, you can often jump back using only a fragment.

Current official Omarchy documentation integrates zoxide so that remembered directory navigation becomes dramatically shorter.

Try visiting several directories normally first:

```
cd ~/.config/hypr
cd ~/Projects
cd ~/.config/omarchy
```

Then experiment with remembered fragments according to the current shell integration.

4.65 — Why Zoxide Matters

Without directory memory:

```
cd ~/Projects/client/backend/services/auth
```

every time.

With a learned directory:

```
jump using a small unique fragment
```

This adds up quickly during a development day.

4.66 — `fd` + `rg` + `fzf` : The Core Search Trio

A useful mental model:

```
fd
→ Where is the file?

rg
→ Where is the text?

fzf / ff
→ Let me interactively choose
```

Examples:

```
fd "config"
```

Find files.

```
rg "databaseUrl"
```

Find content.

```
ff
```

Interactively find and preview a file.

These three tools cover a huge percentage of everyday codebase navigation.

4.67 — `bat` + `rg` : Inspect Search Results

Search:

```
rg -n "workspace" ~/.config/hypr
```

Once you identify the relevant file:

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

This pattern is excellent:

```
search
→ identify file
→ inspect file
→ edit only if necessary
```

4.68 — Searching Command Output with `rg`

`rg` is primarily a file-content search tool, while standard tools such as `grep` are common for streams.

For large command output you will frequently see:

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

Use whichever tool fits the input naturally.

The important concept is the pipe.

4.69 — Avoid `sudo` in Your Home Directory

You learned this in Module 2.

It deserves repetition because the terminal makes dangerous commands easy to type.

Do not edit:

```
~/.config/hypr/bindings.lua
```

with:

```
sudo nvim ...
```

under normal circumstances.

Your personal files should remain owned by your user.

Use `sudo` only when the operation genuinely requires system-level privileges.

4.70 — Your `.bashrc`

Your personal Bash customizations belong in:

```
~/.bashrc
```

The current official Omarchy dotfiles documentation specifically recommends putting your own:

- aliases;
- functions;
- exports;

in this file.

It is intended to survive Omarchy updates.

4.71 — Inspect `.bashrc`

Run:

```
bat ~/.bashrc
```

Your file may be intentionally small because Omarchy loads its defaults separately.

That is good.

Your file should represent your personal customizations, not a copy of all Omarchy shell internals.

4.72 — Add a Personal Alias

For example:

```
alias cproj='cd ~/Projects'
```

You could add this to:

```
~/.bashrc
```

Then reload:

```
source ~/.bashrc
```

Now:

```
cproj
```

would jump to your project directory.

IMPORTANT

Do not add aliases merely because you can.

A good alias should reduce a repeated friction point.

4.73 — Add an Environment Variable

Example:

```
export EDITOR=nvim
```

could be placed in `.bashrc` if that is intentionally how you want tools to resolve the editor.

However, Omarchy already has its own default-editor integration.

Before adding environment overrides, check whether Omarchy already manages the setting you want.

This is the same principle throughout the course:

> ***Use the Omarchy layer when it already solves the problem cleanly.***

4.74 — Functions

A shell function can wrap several commands.

Example:

```
mkcd() {
  mkdir -p "$1" && cd "$1"
}
```

Then:

```
mkcd experiment
```

creates and enters the directory.

Functions become useful later for personal workflows and automation.

For now, understand the concept.

4.75 — Reloading Shell Configuration

After editing `.bashrc`, either open a new terminal or run:

```
source ~/.bashrc
```

`source` evaluates the file in the current shell.

You may also see shorthand:

```
. ~/.bashrc
```

Both forms are common.

4.76 — Do Not Edit Omarchy's Internal Shell Files

You can inspect Omarchy's shell defaults under:

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

but your personal changes should go in:

```
~/.bashrc
```

The official Omarchy dotfiles guidance is clear that package-owned files under `/usr/share/omarchy` can be overwritten by updates.

This is the same upgrade-safe configuration strategy you learned with Hyprland.

4.77 — Environment Debugging Pattern

If a command behaves unexpectedly:

```
1. type COMMAND
2. command -v COMMAND
3. COMMAND --version
4. echo "$PATH"
```

Example:

```
type node
command -v node
node --version
```

This often reveals the problem without reinstalling anything.

4.78 — Working With Large Output

Do not let huge command output flood the terminal if you only need a small part.

Use:

```
COMMAND | head
```

or:

```
COMMAND | tail
```

or:

```
COMMAND | grep PATTERN
```

or save it:

```
COMMAND > output.txt
```

Then inspect:

```
bat output.txt
```

4.79 — Copying Commands Safely

When copying a command from documentation:

1. read it;
2. identify the command;
3. identify the paths;
4. identify flags;
5. look for `sudo`;
6. look for destructive commands such as `rm`;
7. only then execute.

Especially inspect:

```
rm
mv
chmod
chown
dd
mkfs
pacman
sudo
```

The terminal is powerful because it assumes you know what you asked it to do.

4.80 — Commands From AI Agents

AI coding agents can generate shell commands.

Treat them the same way as commands from a human tutorial.

Before execution:

```
What does it change?
Does it need sudo?
Which files does it touch?
Can it delete data?
Can I inspect first?
```

One of the reasons we are learning shell fundamentals is so you can judge AI-generated commands rather than blindly approving them.

4.81 — Practical Exercise: Navigate Without the File Manager

Starting from home:

```
cd
```

Do the following:

1. enter `~/Projects`;
2. create `terminal-practice-2`;
3. enter it;
4. create `src`, `docs`, and `tests`;
5. create `README.md`;
6. display the directory tree;
7. return home.

Possible commands:

```
cd ~/Projects
mkdir terminal-practice-2
cd terminal-practice-2
mkdir src docs tests
touch README.md
lt
cd
```

4.82 — Practical Exercise: Search a Project

Inside a project repository, perform:

```
fd
```

Then:

```
rg "TODO"
```

Then:

```
rg -n "TODO"
```

Then:

```
ff
```

Your goal is to understand the different search experiences.

4.83 — Practical Exercise: Search Omarchy Configuration

Search your own Hyprland config:

```
rg "monitor" ~/.config/hypr
```

Then:

```
rg "bind" ~/.config/hypr
```

Search Omarchy's defaults without editing them:

```
rg "Toggle scratchpad" /usr/share/omarchy
```

This demonstrates a powerful troubleshooting workflow.

4.84 — Practical Exercise: Build a Diagnostic File

Run:

```
{
  echo "=== Omarchy ==="
  omarchy version
  echo
  echo "=== Network ==="
  omarchy network status --verbose
  echo
  echo "=== Failed system services ==="
  systemctl --failed
} > diagnostic.txt
```

Inspect:

```
bat diagnostic.txt
```

You just combined:

- grouping;
- commands;
- redirection;
- diagnostics.

This is a useful real-world pattern.

4.85 — Practical Exercise: Fuzzy History

Run several commands:

```
omarchy version
hyprctl monitors all
omarchy network status --verbose
git --version
mise --version
```

Now press:

```
Ctrl + R
```

Search for:

```
network
```

Select the earlier network command.

Then repeat with:

```
monitor
```

This exercise is more important than it looks.

4.86 — Practical Exercise: Inspect Command Resolution

Run:

```
type ls
type cat
type bat
type fd
type rg
type python
type node
type omarchy
```

Write down which are:

- aliases;
- builtins;
- executable paths;
- wrappers.

Your exact results can vary by current Omarchy configuration.

That variation is part of the lesson.

4.87 — Practical Exercise: One Pipeline

Run:

```
omarchy commands --all | grep -i "restart"
```

Then:

```
omarchy commands --all | grep -i "network"
```

Then:

```
omarchy commands --all | grep -i "hyprland" | head -n 20
```

You are now chaining tools instead of manually scrolling through hundreds of lines.

4.88 — Five-Minute Terminal Challenge

Complete the following without a graphical file manager:

1. go to `~/Projects`;
2. find a project directory using `fd`;
3. enter the project;
4. find every file containing `config` in its name;
5. search all files for the word `TODO`;
6. inspect one matching file with `bat`;
7. check Git status;
8. return to a previous command using `Ctrl + R`;
9. return home.

The objective is not speed.

It is fluency.

4.89 — Common Terminal Mistakes

Mistake 1 — Typing Everything Manually

Use:

- Tab completion;
- history;
- `Ctrl + R`;
- zoxide;
- `ff`.

The best command is often the command you do not have to retype.

Mistake 2 — Reaching for `sudo` Too Quickly

Permission errors are not automatically requests for root.

Mistake 3 — Using `rm -rf` Without Inspecting the Path

Recursive deletion can be irreversible.

Mistake 4 — Forgetting Quotes Around Paths With Spaces

Use:

```
cd "My Project"
```

not:

```
cd My Project
```

unless you actually mean two arguments.

Mistake 5 — Confusing `>` and `>>`

```
>
→ overwrite

>>
→ append
```

Mistake 6 — Thinking `fd` Searches Hidden Files Automatically

When hidden paths matter:

```
fd -H ...
```

Mistake 7 — Searching Files One by One

Use:

```
rg
```

for content and:

```
fd
```

for filenames.

Mistake 8 — Assuming `ls` Is Always Raw `ls`

On Omarchy, enhanced shell aliases may change what familiar commands resolve to.

Check:

```
type ls
```

Mistake 9 — Assuming Silent Output Means Failure

Check:

```
echo $?
```

or inspect the resulting state.

Mistake 10 — Editing Omarchy Internals for a Personal Shell Change

Use:

```
~/.bashrc
```

for your own aliases, exports, and functions.

Do not put normal personal changes inside package-owned `/usr/share/omarchy` files.

4.90 — Checkpoint

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

Terminal Basics

- [ ] I understand the difference between terminal emulator and shell.
- [ ] I can identify my shell.
- [ ] I can check the configured Omarchy terminal.
- [ ] I know how to get command help.

Navigation

- [ ] I can use `pwd`.

- [ ] I can use `cd`.
- [ ] I understand `~`, `.`, and `..`.
- [ ] I use Tab completion.

Files

- [ ] I can create directories.
- [ ] I can create files.
- [ ] I can copy files.
- [ ] I can move/rename files.
- [ ] I understand the danger of `rm -rf`.

Omarchy Shell Tools

- [ ] I understand what `eza` provides.
- [ ] I can use `lsa`, `lt`, and `lta`.
- [ ] I can use `bat`.
- [ ] I can use `fd`.
- [ ] I know when `fd -H` matters.
- [ ] I can use `rg`.
- [ ] I can use `ff`/fzf.
- [ ] I can fuzzy-search history with `Ctrl + R`.
- [ ] I understand what zoxide does.
- [ ] I can use `tldr`.

Shell Composition

- [ ] I understand pipes.
- [ ] I understand `>`.
- [ ] I understand `>>`.
- [ ] I understand `&&`.
- [ ] I understand `||`.
- [ ] I know what `/dev/null` does at a high level.

Environment

- [ ] I understand `$PATH`.
- [ ] I can inspect commands with `type`.
- [ ] I understand why command resolution matters with mise.

- [ ] I know where my personal Bash configuration belongs.

Safety

- [ ] I inspect destructive commands before running them.
- [ ] I do not use `sudo` automatically.
- [ ] I can recognize when an AI-generated command deserves extra inspection.

4.91 — What You Can Now Do

After completing Module 4, you can:

- operate comfortably inside the terminal;
- navigate without a graphical file manager;
- search large codebases quickly;
- inspect code and configuration cleanly;
- fuzzy-search files and history;
- jump around frequently used directories;
- chain tools using pipes;
- save command output;
- inspect command resolution;
- understand why runtime paths matter;
- customize Bash safely;
- read terminal commands rather than treating them as magic;
- evaluate commands suggested by documentation or AI agents more safely.

This is the foundation for nearly everything that follows:

- Tmux;
- Git;
- mise;
- coding agents;
- development servers;
- logs;
- automation;
- remote systems;
- recovery.

Module 4 Summary

Terminal stack:

```
Terminal emulator
→ Ghostty / Foot / Alacritty / Kitty

Shell
→ Bash

Commands
→ Linux + Omarchy + developer tools
```

Navigation:

```
pwd
cd
Tab
```

Directory inspection:

```
ls
lsa
lt
lta
```

Search:

```
fd
→ filenames

rg
→ file contents

ff / fzf
→ interactive fuzzy selection

Ctrl + R
→ fuzzy history
```

File inspection:

```
cat
→ raw/simple output

bat
→ developer-friendly viewing
```

Command composition:

```
|
→ pipe

>
→ overwrite output file

>>
→ append output

&&
→ next command on success

||
→ next command on failure
```

Command diagnosis:

```
type COMMAND
command -v COMMAND
COMMAND --version
echo "$PATH"
```

Personal shell configuration:

```
~/.bashrc
```

And the most important lesson:

> ***The terminal becomes fast when you stop retyping, stop browsing manually, and start composing small tools.***

MODULE 05

05

Tmux for Omarchy Developers

You now know how to use the terminal efficiently.

The next step is learning how to keep multiple terminal tasks organized without opening a large number of separate terminal windows.

About This Module

You now know how to use the terminal efficiently.

The next step is learning how to keep multiple terminal tasks organized without opening a large number of separate terminal windows.

That is where **Tmux** comes in.

Tmux is a terminal multiplexer.

It gives you:

- persistent terminal sessions;
- multiple windows inside one session;
- multiple panes inside one window;
- keyboard navigation between terminal areas;
- layouts for development;
- resumable work after closing the terminal;
- the same terminal workflow on local and remote machines.

Current Omarchy includes a customized Tmux configuration and several developer-focused layout helpers.

This makes Tmux especially useful for the workstation we are building.

A typical AI development layout may contain:

```
Editor
+
AI coding agent
+
Terminal
```

inside one Tmux session.

The goal of this module is **not** to teach every Tmux command.

It is to make you productive with the Omarchy Tmux workflow.

What You Will Learn

By the end of Module 5, you will be able to:

- understand what Tmux is;

- understand sessions, windows, and panes;
- launch Tmux through Omarchy;
- use the Omarchy Tmux prefix;
- create and close panes;
- move between panes;
- resize panes;
- create and switch windows;
- detach from a session;
- resume an existing session;
- inspect active sessions;
- use copy mode;
- reload Tmux configuration;
- understand Tmux working-directory behavior;
- use Omarchy's `tdl`, `tds`, `tdlm`, and `tsl` development layouts;
- build an editor + AI + terminal workspace;
- understand when Tmux is useful and when it is unnecessary.

5.1 — What Is Tmux?

Tmux stands for:

> ***terminal multiplexer***

A normal terminal window usually contains one shell.

Conceptually:

```
Terminal
└── Bash
```

Tmux adds another layer:

```
Terminal
└── Tmux
    ├── Window 1
    │   ├── Pane 1
    │   └── Pane 2
    │
    ├── Window 2
    │   └── Pane 1
    │
    └── Window 3
        ├── Pane 1
        ├── Pane 2
        └── Pane 3
```

This gives you a programmable terminal workspace.

5.2 — Why Tmux Matters on Omarchy

The current official Omarchy terminal documentation recommends Tmux as a way to gain:

- panes;
- windows/tabs;
- resumable sessions;

regardless of which terminal emulator you use.

This matters because Omarchy supports several terminals:

- Foot;
- Ghostty;
- Alacritty;
- Kitty.

Even if the terminal itself has tabs or splits, Tmux gives you the same workflow everywhere.

That includes remote machines over SSH.

5.3 — Tmux Persists Separately from the Terminal Window

This is one of the most important concepts.

Without Tmux:

```
close terminal
→ shell process disappears
→ running work may stop
```

With Tmux:

```
close terminal
→ Tmux session can continue running
→ reconnect later
```

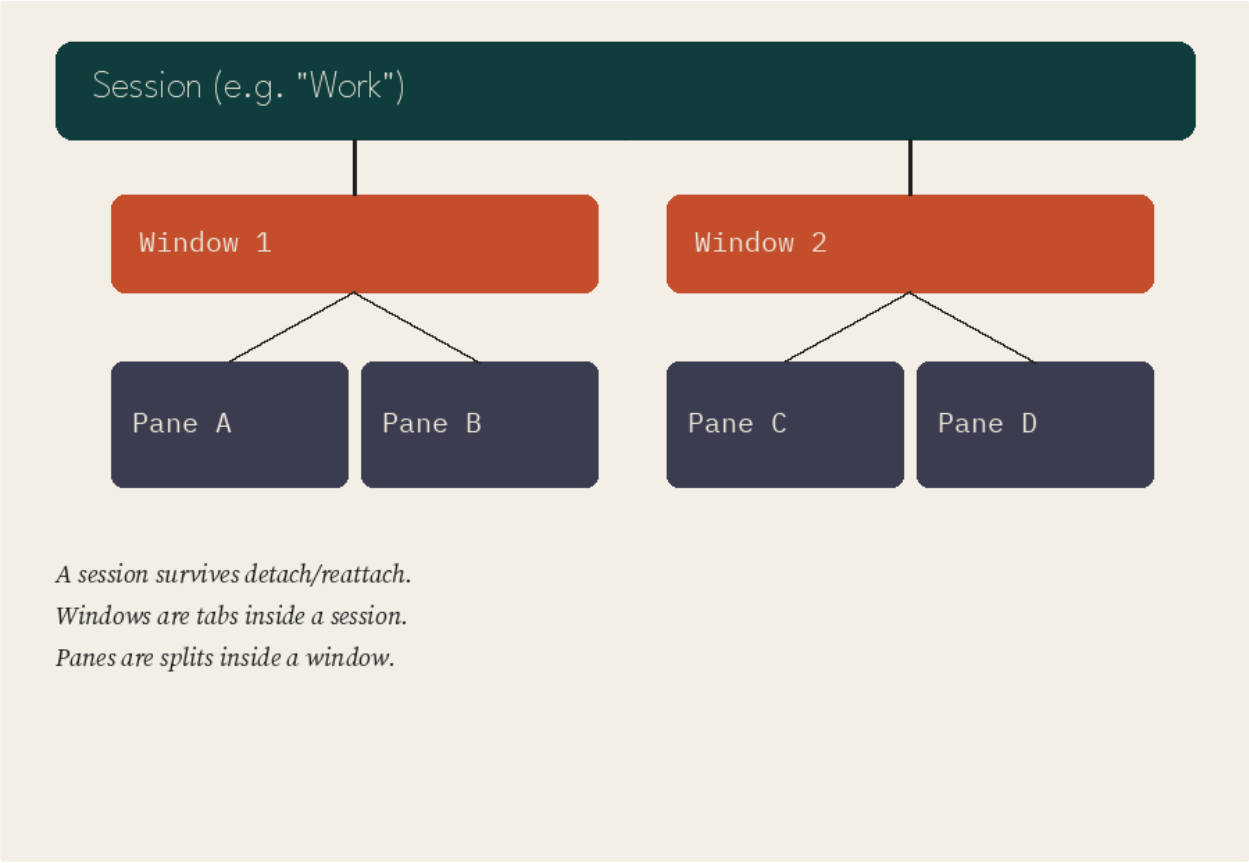
Conceptually:

```
Ghostty window
      ↓
Tmux client
      ↓
Tmux server/session
      ↓
shells and programs
```

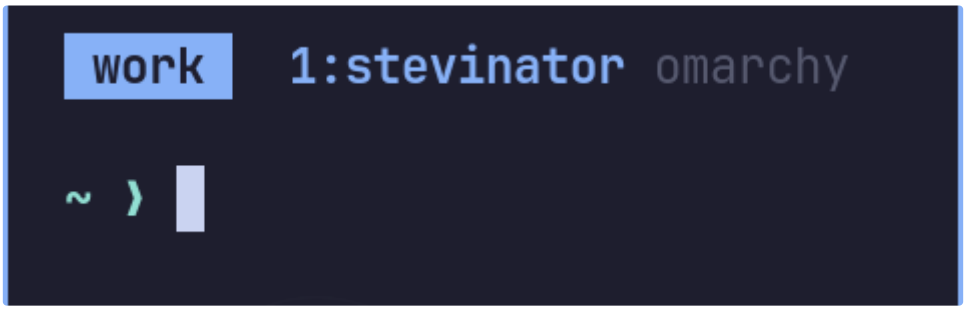
The terminal window is only a view into the Tmux session.

5.4 — Sessions, Windows, and Panes

Tmux has three core levels.



Screenshot 5.4-A
session/window/pane hierarchy diagram



Screenshot 5.4-B
Tmux status bar

Session

A session is the entire persistent workspace.

Example:

```
Session: Work
```

Window

A window is similar to a tab inside a Tmux session.

Example:

```
Work
├── Window 1: editor
├── Window 2: server
└── Window 3: logs
```

Pane

A pane is a split inside a Tmux window.

Example:

```
Window 1

┌──────────────────────┬──────────────────────┐
│                      │                      │
│       Editor         │      AI Agent        │
│                      │                      │
├──────────────────────┴──────────────────────┤
│                  Terminal                   │
└─────────────────────────────────────────────┘
```

This distinction is fundamental:

```
Session
→ collection of windows

Window
→ collection of panes

Pane
→ one terminal area
```

5.5 — Launch Tmux Through Omarchy

Current official Omarchy shortcut:

```
Super + Alt + Return
```

This launches a fresh terminal with Tmux.

The official current hotkey documentation identifies this as the **Tmux terminal** launcher.

Screenshot 5.5-A
Fresh Omarchy Tmux terminal

5.6 — Launch Tmux from an Existing Terminal

You can also run Tmux manually.

Check the installed version:

```
tmux -V
```

Then start a session:

```
tmux
```

However, Omarchy provides shell helpers that make the normal workflow easier.

5.7 — The Omarchy `t` Helper

Current Omarchy shell configuration includes Tmux helpers.

On the workstation used to build this course, the shorthand:

```
t
```

attaches to or creates the default working Tmux session.

Before assuming this helper exists on a future version, verify it:

```
type t
```

If available, it should reveal the actual shell alias or function.

This is a recurring course principle:

> ***Inspect the helper before assuming what it does.***

5.8 — The Tmux Prefix Key

Tmux uses a **prefix key** for many commands.

Current Omarchy documentation uses:

```
Ctrl + Space
```

as the primary Tmux prefix.

The current hotkey documentation also states that:

```
Ctrl + B
```

continues to work.

So:

```
Prefix + s
```

means:

1. press `Ctrl + Space`;
2. release it;
3. press `s`.

It does **not** mean pressing all keys simultaneously.

5.9 — Why a Prefix Exists

Tmux runs inside your terminal.

If every Tmux command used ordinary keys directly, it would interfere with applications such as:

- Bash;
- Neovim;
- coding agents;
- TUIs.

The prefix tells Tmux:

> *"The next key is for you."*

Example:

```
Ctrl + Space
then
c
```

creates a Tmux window.

Without the prefix, typing:

```
c
```

should simply type `c` into your shell.

5.10 — Show the Tmux Keybinding Help

Current Omarchy provides:

```
Super + Alt + K
```

for Tmux keybinding help.

Inside Tmux, current bindings also include:

```
Prefix + ?
```

for the Tmux keybinding cheatsheet.

This is important.

Just as:

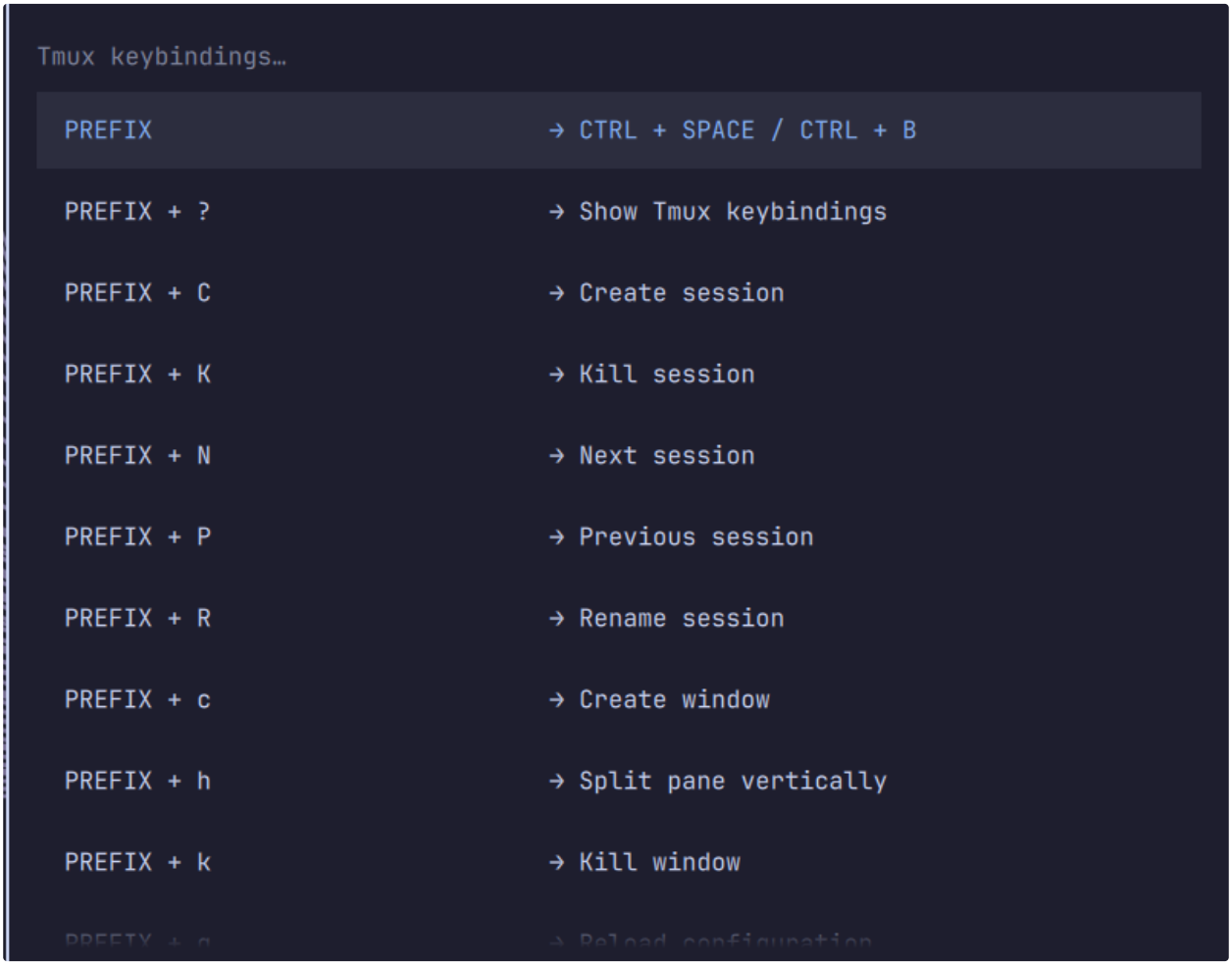
```
Super + K
```

helps you discover Omarchy shortcuts,

```
Super + Alt + K
```

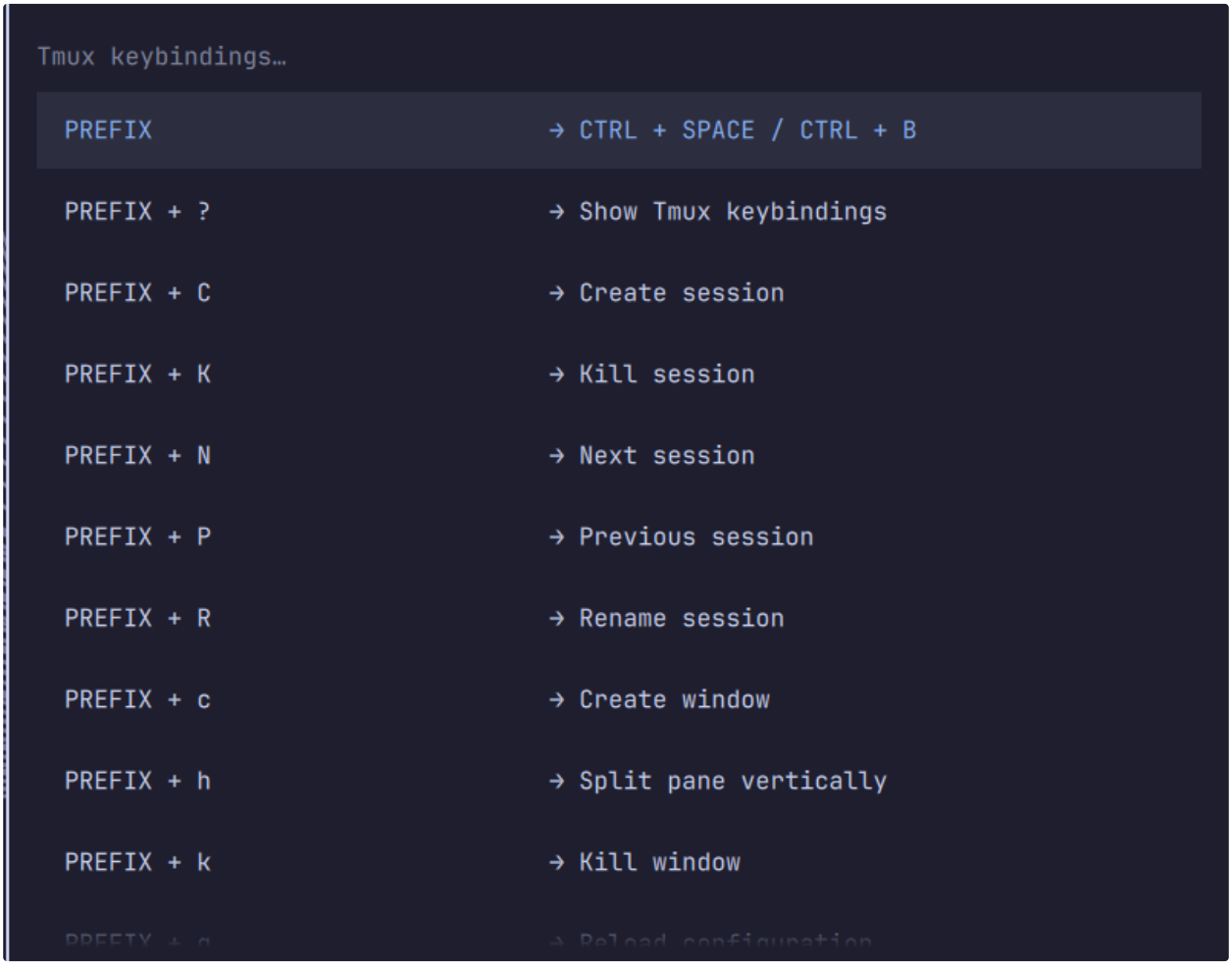
helps you discover Tmux shortcuts.

Do not try to memorize the full Tmux map.



Screenshot 5.10-A

`Super + Alt + K` Tmux help



Screenshot 5.64-A
`Prefix + ?` cheatsheet

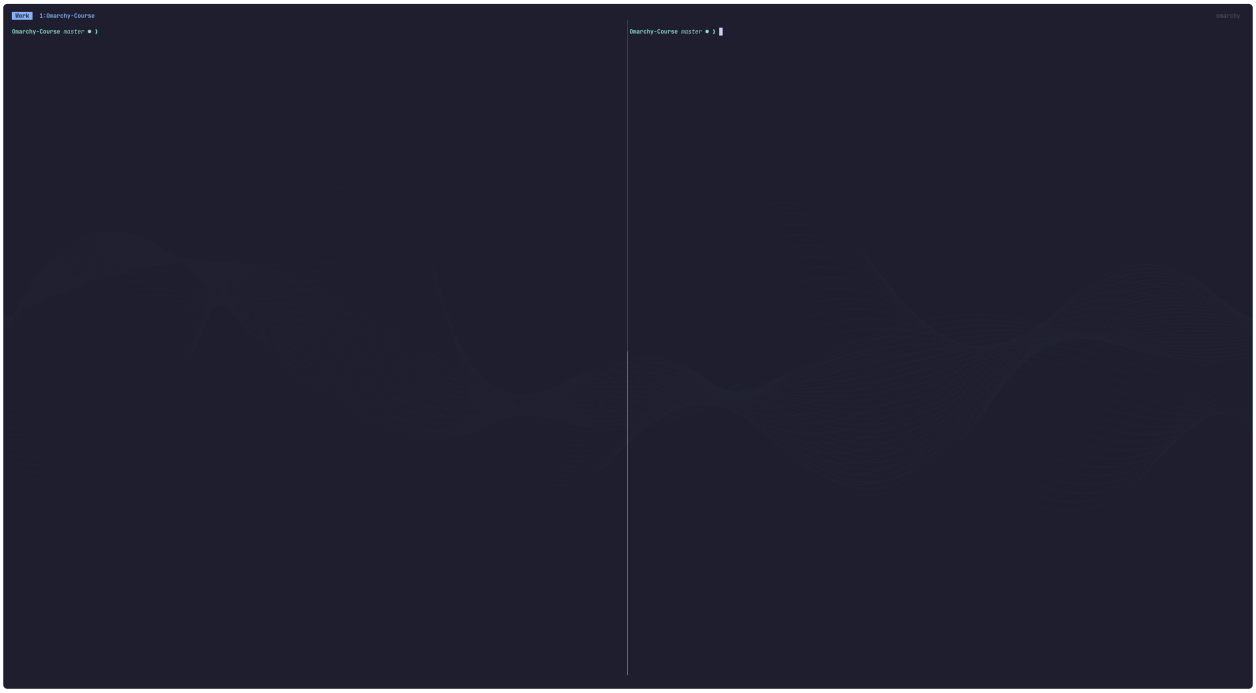
5.11 — Create a Pane Beside the Current Pane

Current Omarchy Tmux binding:

```
Prefix + v
```

This creates a vertical divider, placing another pane beside the current one.

Conceptually:



Screenshot 5.11-A
vertical pane split

Before

```
┌──────────────────────────────┐
│                              │
│           Terminal           │
│                              │
└──────────────────────────────┘
```

After:

```
┌──────────────┬──────────────┐
│              │              │
│    Pane 1    │    Pane 2    │
│              │              │
└──────────────┴──────────────┘
```

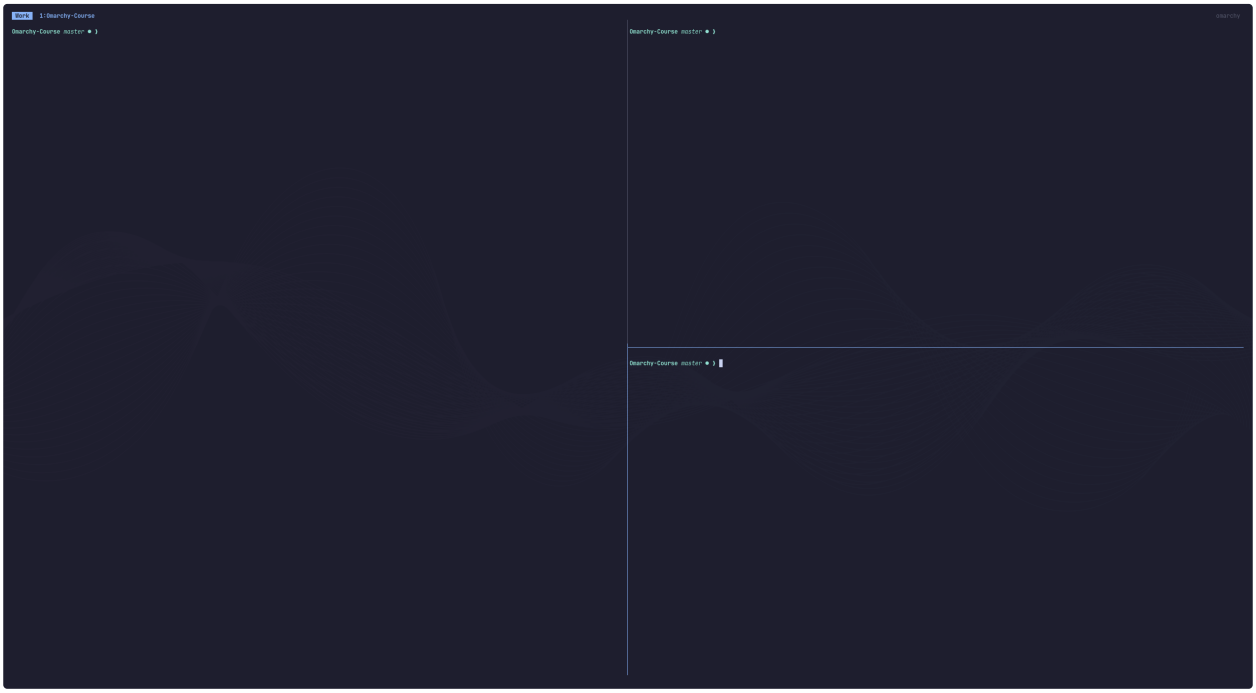
5.12 — Create a Pane Below

Current Omarchy binding:

```
Prefix + h
```

This splits the current pane horizontally.

Conceptually:



Screenshot 5.12-A
horizontal pane split

```
┌──────────────────┐
│      Pane 1      │
├──────────────────┤
│      Pane 2      │
└──────────────────┘
```

5.13 — Split Without Using the Prefix

Current Omarchy Tmux configuration also provides direct shortcuts.

Split below:

```
Alt + Return
```

Split beside:

```
Alt + Shift + Return
```

These are convenient because they avoid the two-step prefix sequence.

5.14 — Move Between Panes

Current Omarchy bindings:

```
Ctrl + Alt + Left
Ctrl + Alt + Right
Ctrl + Alt + Up
Ctrl + Alt + Down
```

This lets you move focus between panes directly.

Screenshot 5.14-A
pane focus movement

Example:

```
Ctrl + Alt + Right
```

moves into the pane to the right.

5.15 — Resize Panes

Current Omarchy provides:

```
Ctrl + Alt + Shift + Arrow
```

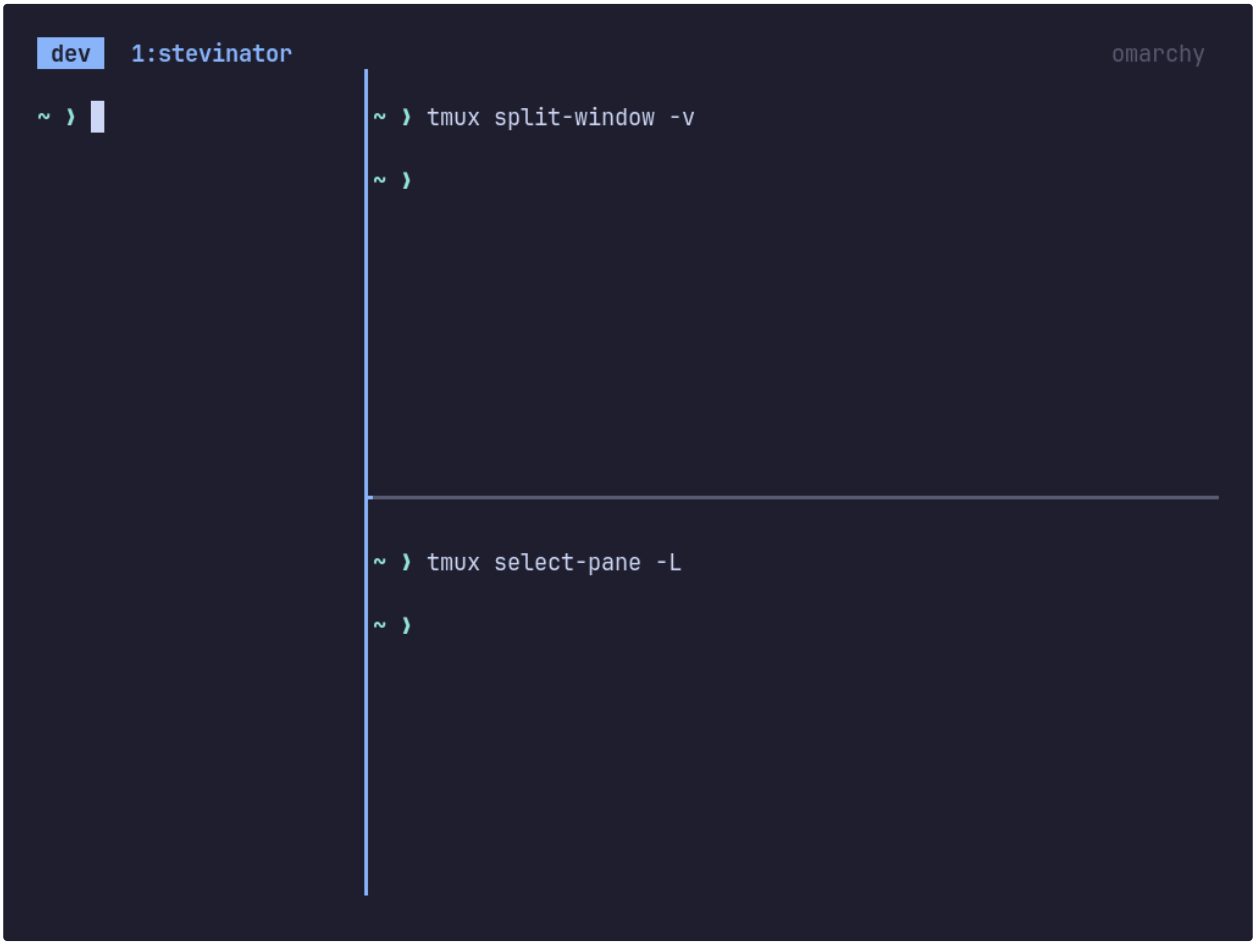
for resizing panes.

Examples:

```
Ctrl + Alt + Shift + Left
Ctrl + Alt + Shift + Right
```

Use these to allocate more room to:

- editor;
- AI agent;
- logs;
- terminal.



Screenshot 5.15-A
pane resize

5.16 — Resize Using the Tmux Command Prompt

Tmux also provides a command prompt.

Open it with:

```
Prefix + :
```

Then you can use commands such as:

```
resize-pane -L 10
```

or:

```
resize-pane -R 10
```

This is useful when you need a precise adjustment.

You do not need it for normal use if the direct resize shortcuts are enough.

5.17 — Zoom a Pane

Current Omarchy binding:

```
Prefix + z
```

This temporarily expands the current pane to fill the whole Tmux window.

Example:

```
Editor + AI + terminal
        ↓
focus terminal
        ↓
Prefix + z
        ↓
terminal fills entire Tmux window
```

Press it again to restore the previous layout.

This is extremely useful.

Screenshot 5.17-A
zoomed pane

5.18 — Close a Pane

Current Omarchy binding:

```
Prefix + x
```

You can also use the direct shortcut:

```
Alt + Escape
```

Another natural method is simply to exit the shell running in that pane:

```
exit
```

When the pane's shell ends, the pane closes.

5.19 — Create a New Tmux Window

Current Omarchy binding:

```
Prefix + c
```

This creates another Tmux window.

Think:

```
Tmux window ≈ terminal tab
```

but managed by Tmux rather than the terminal emulator.

Screenshot 5.19-A
multiple Tmux windows

5.20 — Switch Between Windows

Current Omarchy provides:

```
Alt + 1
Alt + 2
Alt + 3
...
Alt + 9
```

to jump directly to a Tmux window.

It also provides:

```
Alt + Left
Alt + Right
```

to move between windows.

This is very fast once you use Tmux regularly.

5.21 — Move Tmux Windows

Current Omarchy shortcut:

```
Alt + Shift + Left
Alt + Shift + Right
```

This moves the current Tmux window left or right in the window order.

Useful when organizing larger sessions.

5.22 — Rename a Window

Current Omarchy binding:

```
Prefix + r
```

Give windows meaningful names such as:

```
code
server
logs
database
deploy
```

This becomes useful when a session contains several windows.

Screenshot 5.22-A
renamed Tmux window

5.23 — Close a Window

Current Omarchy binding:

```
Prefix + k
```

This kills the current Tmux window.

Be careful if it contains running processes you still need.

5.24 — List Sessions

Inside Tmux:

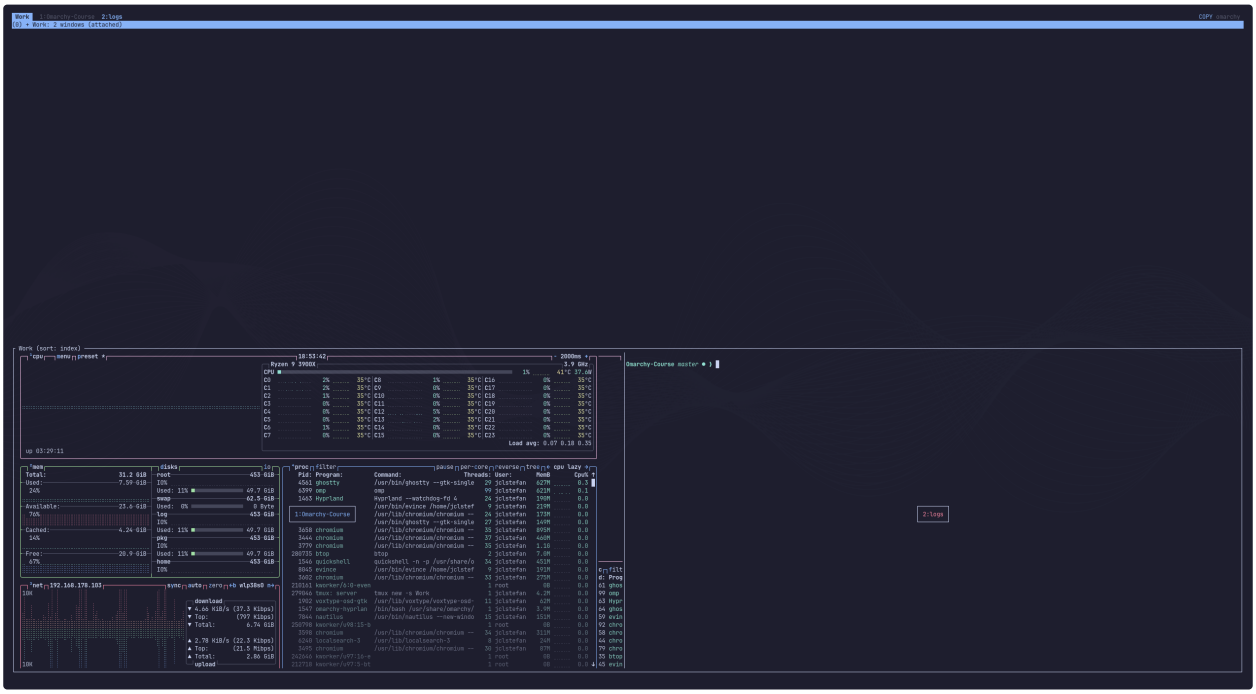
```
Prefix + s
```

The official Omarchy terminal documentation specifically recommends this to see active sessions.

Screenshot 5.26-A

`tmux ls`

This gives you a session chooser.



Screenshot 5.24-A

session chooser

5.25 — Detach from a Session

Current Tmux binding:

```
Prefix + d
```

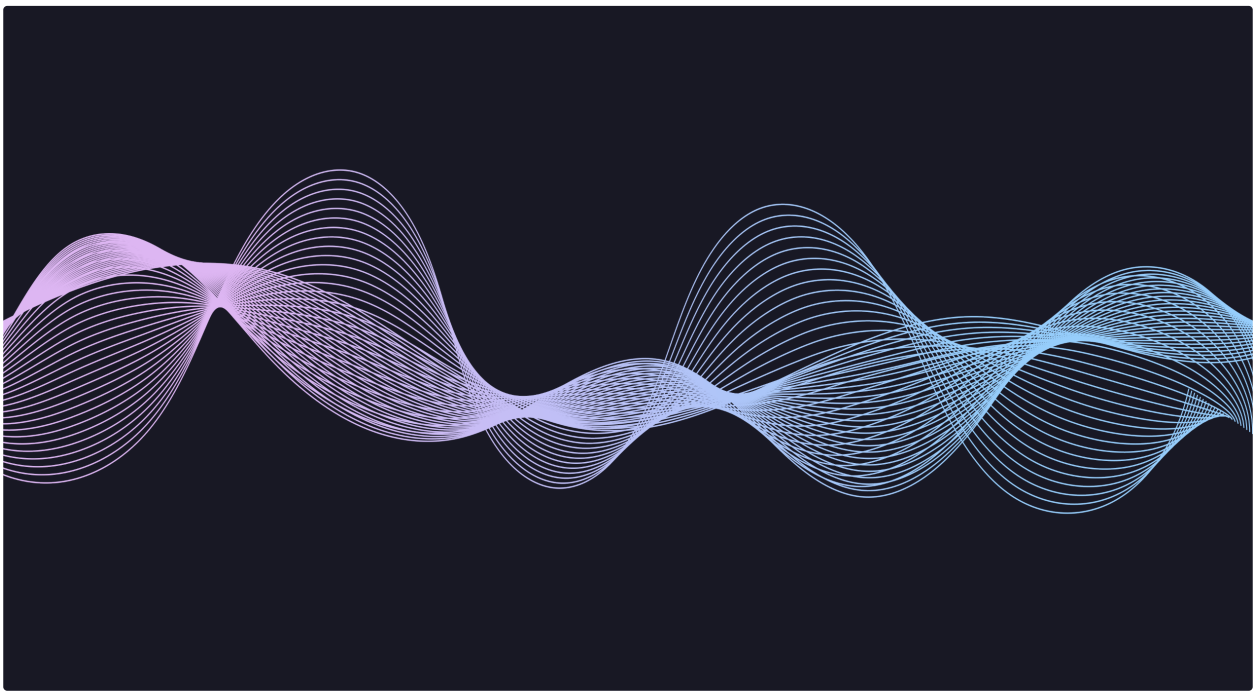
Detach means:

```
leave the Tmux session
without stopping it
```

Conceptually:

```
Tmux session
├── editor
├── server
└── logs

Prefix + d
        ↓

terminal returns to normal shell

Tmux session still exists
```

This is one of Tmux's most important features.



Screenshot 5.25-A
detach/reattach demonstration (detach)

5.26 — Check Sessions from the Shell

Outside Tmux:

```
tmux ls
```

Example:

```
Work: 3 windows ...
```

If there are no sessions, Tmux may report that no server is running.

That is normal.

5.27 — Reattach to a Session

Generic Tmux syntax:

```
tmux attach
```

or:

```
tmux attach -t SESSION_NAME
```

Example:

```
tmux attach -t Work
```

If your Omarchy shell provides the `t` helper, it may handle the common attach/create workflow for you.

Verify with:

```
type t
```

Screenshot 5.27-A
detach/reattach demonstration (reattach)

5.28 — Why Persistence Matters

Imagine you have:

```
Window 1
→ Neovim

Window 2
→ development server

Window 3
→ logs
```

You accidentally close Ghostty.

Without Tmux:

```
everything may disappear
```

With Tmux:

```
open a new terminal
→ reattach
→ continue
```

This becomes even more useful on remote servers.

5.29 — Tmux over SSH

Tmux runs on whichever machine you start it on.

For example:

```
Your Omarchy laptop
        ↓ SSH
Remote Linux server
        ↓
Tmux
        ↓
server processes
```

If your network connection drops:

```
SSH disconnects
```

but the remote Tmux session can continue.

Reconnect and attach again.

This is one of Tmux's classic strengths.

5.30 — Sessions

Current Omarchy Tmux bindings include:

```
Prefix + C
→ new session

Prefix + K
→ kill session

Prefix + R
→ rename session

Prefix + N
→ next session

Prefix + P
→ previous session
```

And direct movement:

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

for moving between sessions.

Most developers do not need many sessions at first.

Start with one.

5.31 — A Simple Session Strategy

For a beginner:

```
Session: Work
```

is enough.

Inside:

```
Window 1
→ current project

Window 2
→ infrastructure

Window 3
→ misc
```

As your workflow develops, you may prefer:

```
one Tmux session per project
```

or:

```
one long-lived Work session
```

Both are valid.

Choose based on actual workflow rather than theoretical purity.

5.32 — Copy Mode

Tmux has its own scrollback/copy mode.

Enter it with:

```
Prefix + [
```

Current Omarchy config uses vi-style copy-mode bindings.

Inside copy mode:

```
v
```

starts selection.

Then:

```
y
```

copies the selection.

This is useful for terminal output that has scrolled away.

Screenshot 5.32-A
copy mode

5.33 — Tmux Copy Mode vs Omarchy Clipboard History

These solve different problems.

```
Tmux copy mode
→ navigate and select historical terminal output
```

```
Super + Ctrl + V
→ browse clipboard history
```

You can use both.

5.34 — Reload Tmux Configuration

Current Omarchy binding:

```
Prefix + q
```

reloads Tmux configuration.

Omarchy also exposes:

```
omarchy restart tmux
```

which reloads/restarts Tmux integration using current configuration when appropriate.

Later, if you customize:

```
~/.config/tmux/tmux.conf
```

you will know how to apply changes.

5.35 — Tmux Configuration Location

Current official Omarchy hotkey documentation identifies the user Tmux configuration at:

```
~/.config/tmux/tmux.conf
```

This is where you would place personal Tmux configuration.

As always:

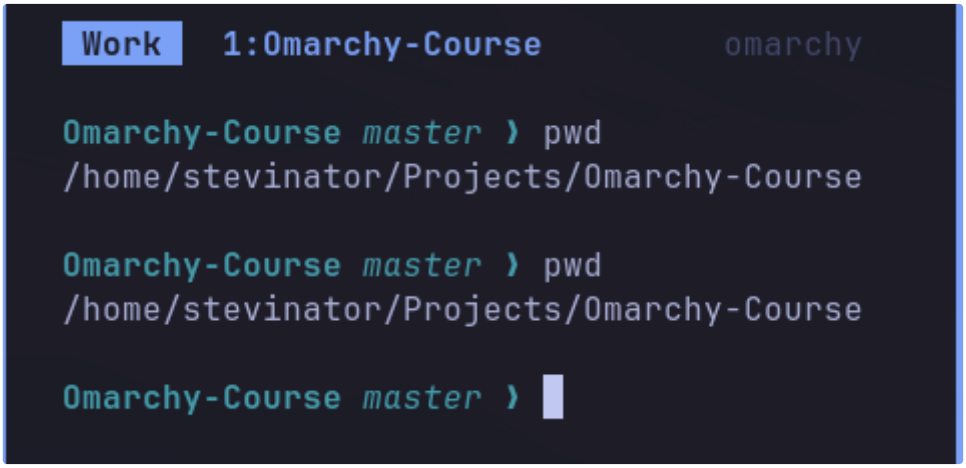
> *Prefer user-owned configuration over editing package-owned Omarchy defaults.*

You do not need to customize Tmux yet.

The current Omarchy configuration is already designed for its workflow.

5.36 — Working Directory Behavior

This is an important real-world gotcha.



Screenshot 5.36-A
working-directory persistence demonstration

Suppose you already have a Tmux session named:

```
Work
```

and one of its panes is currently in:

```
~/Projects/project-a
```

Now you leave Tmux and run:

```
cd ~/Projects/project-b
```

Then you attach to the **existing** `Work` session.

The old pane can still be in:

```
~/Projects/project-a
```

Why?

Because the shell inside that pane never changed directories.

You merely reattached to the same running session.

5.37 — Important Mental Model

Attaching to a Tmux session does **not** mean:

> *"Recreate this session using my current terminal directory."*

It means:

> *"Show me the already-running session exactly where it was."*

This distinction matters when starting project layouts.

5.38 — Starting Fresh in the Correct Project

If you want a new Tmux workspace for a project:

1. go to the project directory;
2. create a new session or ensure you are not attaching to an old unrelated one;
3. then start the layout.

Example:

```
cd ~/Projects/my-project
```

Then start a fresh Tmux environment.

This prevents confusion about why tools opened in the wrong repository.

5.39 — Omarchy Developer Layout Helpers

Current Omarchy ships special Tmux layout functions for development.

The official current terminal documentation describes:

```
tdl
tds
tdlm
tsl
```

These are one of the strongest reasons for developers to learn Tmux on Omarchy.

5.40 — `tdl`: Development Layout

Current official syntax:

```
tdl [agent]
```

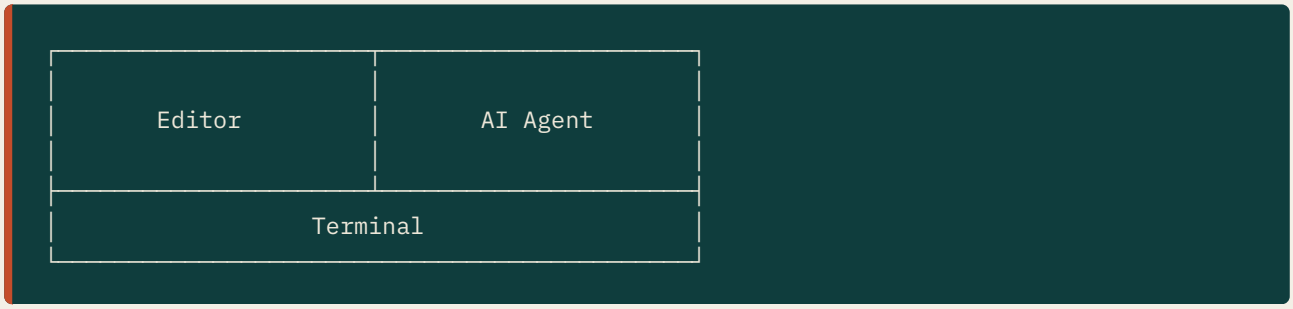
or with two AI agents:

```
tdl [agent] [second_agent]
```

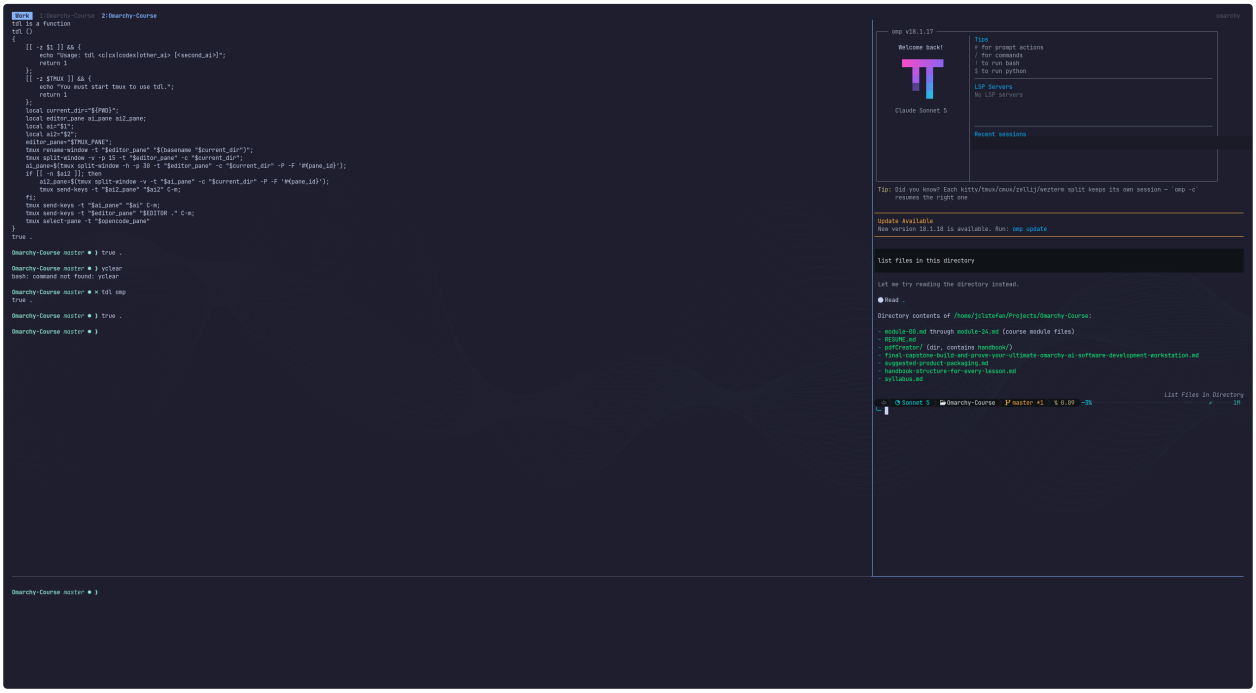
The official current Omarchy documentation describes `tdl` as a three-way IDE-like layout containing:

- editor on the left;
- AI coding agent on the right;
- terminal at the bottom.

Conceptually:



This is an excellent layout for the workstation we are building.



Screenshot 5.40-A

`tdl` layout

5.41 — `tdl` with Your AI Agent

The exact agent command depends on which coding agent you have configured and installed.

Examples in the current official Omarchy documentation use short agent commands for tools such as:

- OpenCode;
- Claude;
- Codex.

On our Stevinator workstation, we also tested layouts with OMP.

Do **not** assume a specific agent shorthand exists.

Verify:

```
type AGENT_COMMAND
```

and:

```
type tdl
```

before building a custom workflow around it.

5.42 — Start `tdl` from the Project Directory

This is important.

First:

```
cd ~/Projects/YOUR_PROJECT
```

Then enter your Tmux session.

Then:

```
tdl YOUR_AGENT_COMMAND
```

This keeps:

- editor;
- AI;
- terminal;

anchored to the intended project.

5.43 — `tdl` with Two Agents

Current official Omarchy documentation supports:

```
tdl AGENT_1 AGENT_2
```

This can give you a development layout with two agent processes.

That may be useful for:

- implementation + review;
- primary agent + second opinion;
- parallel research.

But do not use two agents merely because the feature exists.

More agents create:

- more context;
- more cost;
- more coordination;

- more opportunities for conflicting edits.

Start with one.

5.44 — `tds`: Square Development Layout

Current official Omarchy documentation also provides:

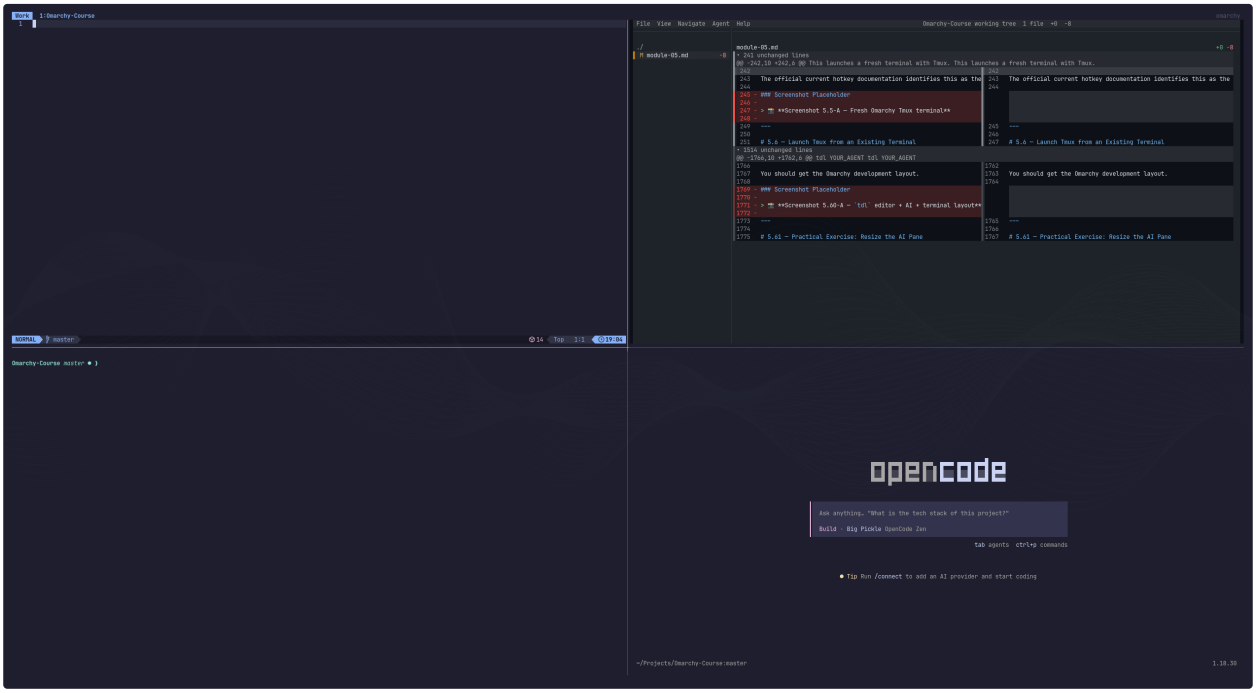
```
tds
```

The current documented layout includes:

- editor;
- live diff watcher;
- terminal;
- OpenCode.

Conceptually:

This layout is particularly useful when reviewing continuous AI-generated changes.



Screenshot 5.44-A

`tds` layout

5.45 — Dependencies of `tds`

A helper layout may depend on other tools.

If `tds` opens a pane and reports:

```
command not found
```

do not assume Tmux is broken.

Inspect the missing command.

Use:

```
type COMMAND
```

Then determine whether that tool belongs to:

- Omarchy package;
- system package;
- mise-managed CLI.

This exact situation occurred while building the workstation used for this course.

That makes it an excellent troubleshooting example.

5.46 — `tdlm` : One Layout Per Subdirectory

Current official Omarchy documentation provides:

```
tdlm [agent]
```

This creates a development layout for subdirectories in the current directory.

The official current workflow allows navigation using:

```
Alt + 1
Alt + 2
Alt + 3
...
```

This can be useful for repositories containing multiple related applications or services.

Example:

```
project/
├── frontend/
├── backend/
├── worker/
└── infrastructure/
```

`tdlm` can help organize those as separate Tmux windows.

Screenshot 5.46-A
`tdlm` example

5.47 — When `tdlm` Makes Sense

Useful:

```
monorepo
multi-service project
directory containing several independent repos
```

Not useful:

```
small single-folder project
```

Use the simplest layout that solves the problem.

5.48 — `tsl` : Swarm Layout

Current official Omarchy syntax:

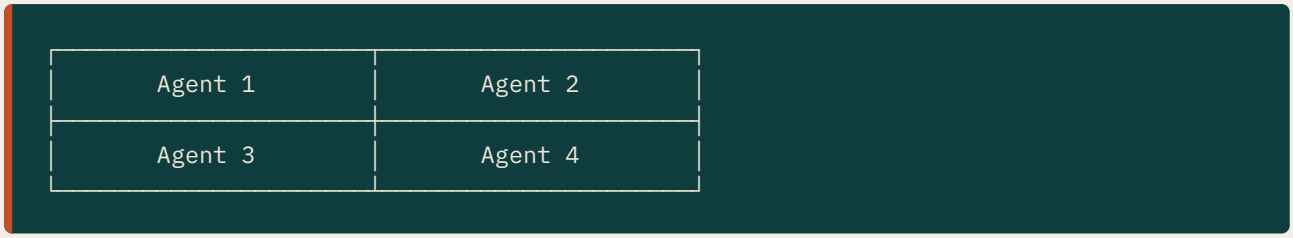
```
tsl <count> <command>
```

Example from current documentation:

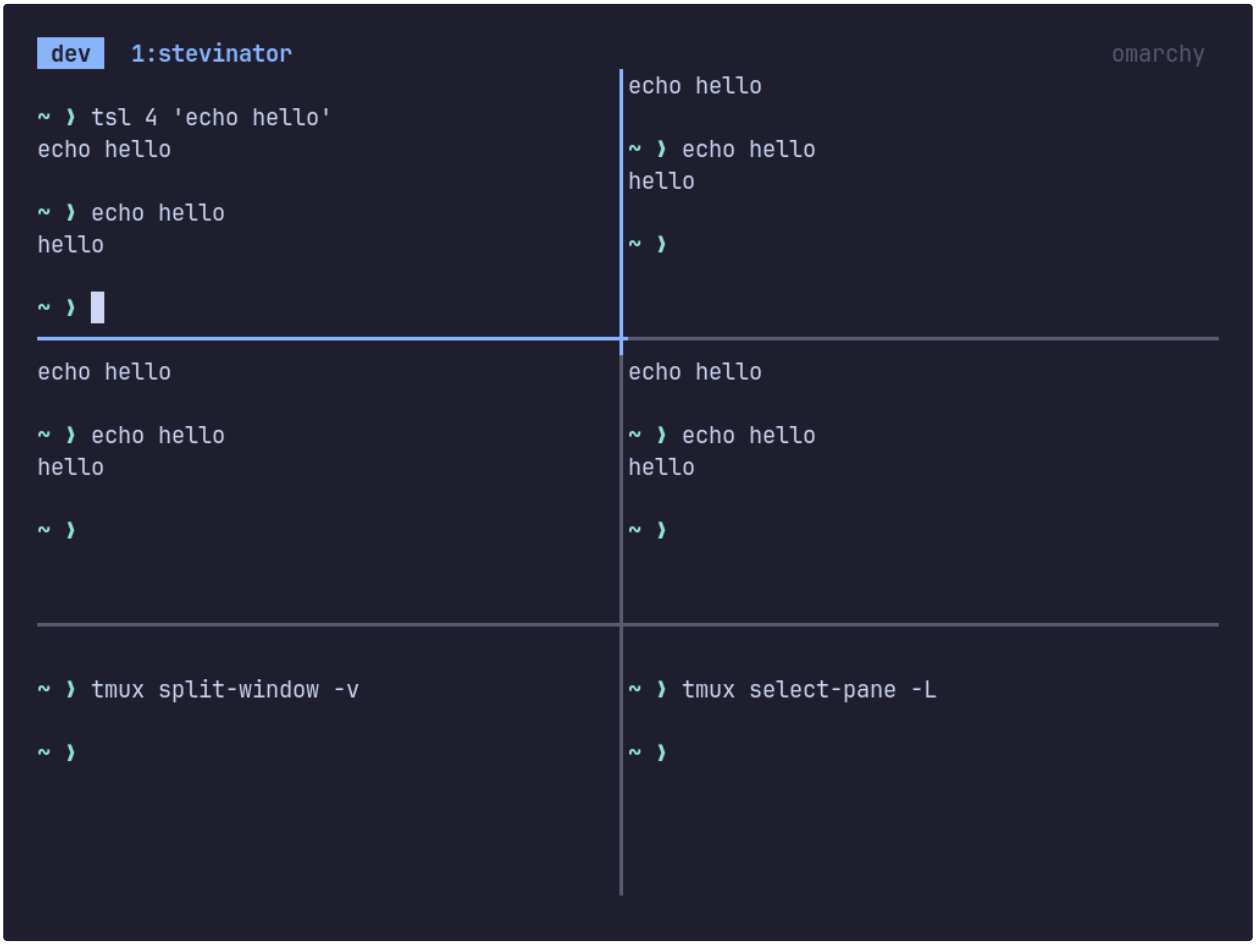
```
tsl 4 c
```

creates four panes running the selected command.

Conceptually:



This is intended for parallel/swarm workflows.



Screenshot 5.48-A
`tsl` grid

5.49 — Do Not Start With Swarms

A multi-agent swarm looks impressive.

That does not mean it should be your default development workflow.

Potential issues include:

- competing file changes;
- duplicated work;
- higher model cost;
- difficult coordination;
- confusing Git state.

Use swarms only when tasks can genuinely be separated.

We will discuss parallel AI work later.

5.50 — Tmux and Git Worktrees

Tmux becomes especially powerful when combined with Git worktrees.

Conceptually:

```
Tmux Window 1
→ Worktree A
→ Agent A

Tmux Window 2
→ Worktree B
→ Agent B

Tmux Window 3
→ Worktree C
→ Agent C
```

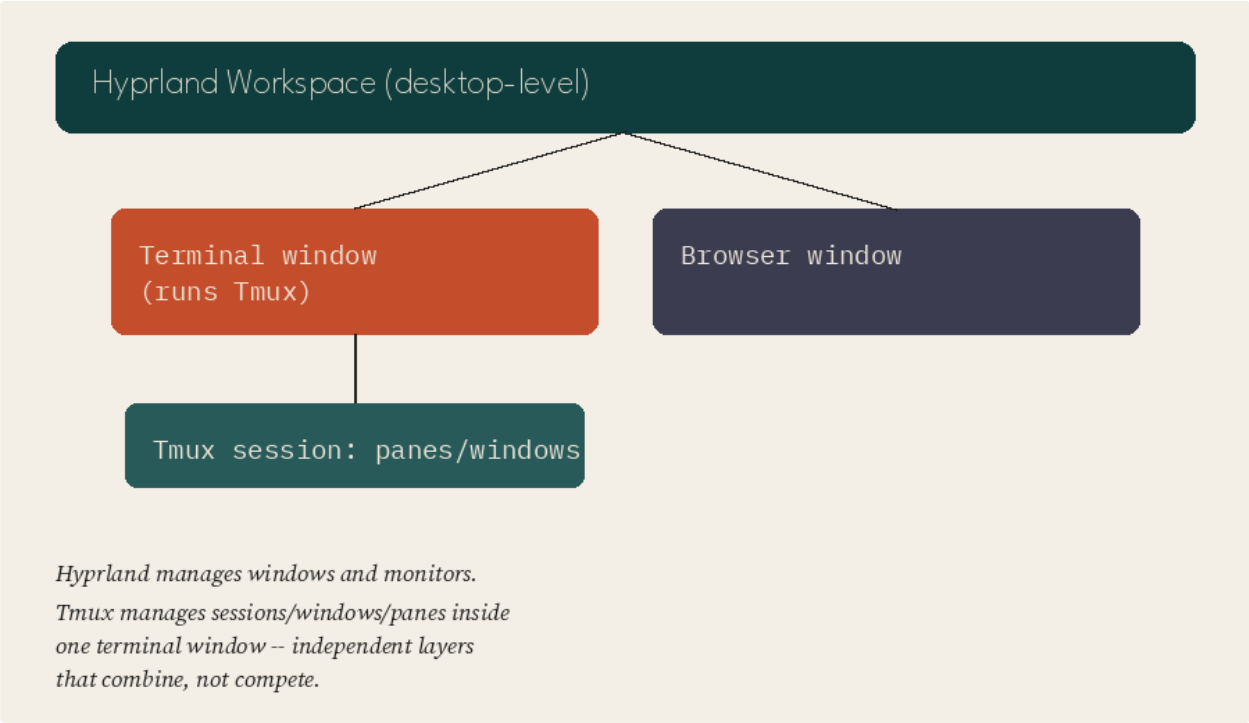
Each agent can work in a separate filesystem checkout.

We will cover Git worktrees conceptually in the Git module and revisit the AI use case later.

Do not set this up yet.

5.51 — Tmux vs Hyprland Workspaces

These are not competitors.



Screenshot 5.51-A
Hyprland workspace + Tmux hierarchy diagram

They operate at different layers.

Hyprland Workspace

Organizes graphical applications.

Example:

```
Workspace 1
→ browser

Workspace 2
→ terminal/Tmux

Workspace 3
→ communication
```

Tmux

Organizes terminal tasks inside the terminal.

Example:

```
Hyprland Workspace 2
└── Ghostty
    └── Tmux
        ├── editor
        ├── AI agent
        └── terminal
```

This layered organization is extremely powerful.

5.52 — Tmux vs Terminal Tabs

If your terminal emulator supports tabs, you may wonder why Tmux is necessary.

Terminal tabs are fine for simple local work.

Tmux adds:

- persistence;
- remote portability;
- programmable layouts;
- pane/window/session hierarchy;
- Omarchy developer layout helpers.

You do not have to use Tmux for every terminal.

A useful rule:

```
quick command
→ normal terminal

long development session
→ Tmux
```

5.53 — Tmux vs Multiple Terminal Windows

Multiple windows can be perfectly valid.

Tmux becomes more valuable when you want:

- one project workspace;
- reproducible layouts;
- long-running processes;

- remote sessions;
- editor + AI + terminal together.

Use the tool because it solves a problem, not because terminal multiplexers are fashionable.

5.54 — A Recommended Omarchy Development Layout

A very practical workstation structure is:

```
Hyprland Workspace 1
→ Browser / documentation

Hyprland Workspace 2
→ Ghostty + Tmux

Hyprland Workspace 3
→ infrastructure / monitoring

Hyprland Workspace 4
→ communication
```

Inside Workspace 2:

```
Tmux Work session

┌──────────────────────┬──────────────────────┐
│                      │                      │
│       Editor         │       AI Agent       │
│                      │                      │
├──────────────────────┴──────────────────────┤
│                  Terminal                   │
└─────────────────────────────────────────────┘
```

This gives you two levels of organization:

```
Hyprland
→ graphical workflow

Tmux
→ development workflow
```

5.55 — Practical Exercise: Create Your First Split

Launch Tmux:

```
Super + Alt + Return
```

Create a pane beside:

```
Prefix + v
```

Move between panes:

```
Ctrl + Alt + Left/Right
```

Close one pane:

```
exit
```

5.56 — Practical Exercise: Horizontal Split

Create:

```
Prefix + h
```

Move:

```
Ctrl + Alt + Up/Down
```

Zoom:

```
Prefix + z
```

Restore:

```
Prefix + z
```

5.57 — Practical Exercise: Windows

Create a new Tmux window:

```
Prefix + c
```

Switch:

```
Alt + 1
Alt + 2
```

Rename the second window:

```
Prefix + r
```

Name it:

```
server
```

Return to Window 1.

5.58 — Practical Exercise: Detach and Reattach

Inside Tmux:

```
Prefix + d
```

Now check sessions:

```
tmux ls
```

Reattach:

```
tmux attach
```

You should return to the same Tmux state.

This is the exercise that proves you understand Tmux persistence.

5.59 — Practical Exercise: Persistence Test

Inside a Tmux pane:

```
cd ~/Projects
```

Then run:

```
pwd
```

Detach:

```
Prefix + d
```

Close the terminal window.

Open a new terminal.

Reattach:

```
tmux attach
```

Run:

```
pwd
```

You should still be in the same directory inside that pane.

This demonstrates why existing Tmux sessions retain their own working directories.

5.60 — Practical Exercise: Use `tdl`

Choose an existing Git project.

Example:

```
cd ~/Projects/YOUR_PROJECT
```

Enter Tmux.

Verify:

```
type tdl
```

Then launch the layout using an installed agent command:

```
tdl YOUR_AGENT
```

You should get the Omarchy development layout.

Screenshot 5.60-A
`tdl` editor + AI + terminal layout

5.61 — Practical Exercise: Resize the AI Pane

Inside the `tdl` layout, focus the AI pane.

Try the current Omarchy resize shortcuts:

```
Ctrl + Alt + Shift + Arrow
```

If you want a precise resize:

```
Prefix + :
```

then:

```
resize-pane -L 10
```

or the direction appropriate to your layout.

The goal is to understand that Tmux layouts are adjustable, not fixed.

5.62 — Practical Exercise: Check the Working Directory

With `tdl` open:

Run:

```
pwd
```

inside the terminal pane.

Confirm that it matches the intended project.

If it does not, ask:

> *Did I attach to an existing Tmux session whose pane already had a different working directory?*

Do not assume the helper is broken.

5.63 — Practical Exercise: Try `tds`

From a Git repository:

```
type tds
```

Then:

```
tds
```

Observe the layout.

If a pane reports a missing command:

1. record the command name;
2. do not randomly install packages;
3. inspect with:

```
type COMMAND
```

1. determine the appropriate installation layer.

This is a troubleshooting exercise as much as a Tmux exercise.

5.64 — Practical Exercise: Tmux Cheatsheet

Press:

```
Super + Alt + K
```

Then inside Tmux:

```
Prefix + ?
```

Compare the two help experiences.

You should leave this module knowing:

> ***I do not have to remember everything. I know where to look.***

5.65 — Common Tmux Mistakes

Mistake 1 — Treating Prefix as a Simultaneous Key Combination

Wrong mental model:

```
Ctrl + Space + c
```

Correct:

```
Ctrl + Space
release
c
```

Mistake 2 — Forgetting You Are Inside Tmux

When shortcuts behave unexpectedly, check whether you are:

```
normal shell
or
Tmux pane
```

The status line normally makes this visually clear.

Mistake 3 — Attaching to an Existing Session and Expecting a New Working Directory

Existing panes retain their running shell state.

Your current directory outside Tmux does not magically replace their directory.

Mistake 4 — Opening Endless Tmux Sessions

Start simple.

One useful session is better than ten forgotten sessions.

Check:

```
tmux ls
```

Mistake 5 — Killing a Session Instead of Detaching

If you want to return later:

```
Prefix + d
```

Do not kill the session.

Mistake 6 — Using Tmux for Every Five-Second Command

A normal terminal is perfectly fine for:

```
git status
```

or:

```
omarchy version
```

Use Tmux when persistence/layout provides value.

Mistake 7 — Assuming a Missing `tds` Tool Means Tmux Is Broken

Layout helpers can depend on additional CLIs.

Identify the missing dependency first.

Mistake 8 — Starting an Agent Swarm Before Understanding Single-Agent Workflows

More panes do not automatically mean more productivity.

Mistake 9 — Hard-Coding Old Agent Aliases

Agent commands can change.

Inspect:

```
type AGENT
```

and current Omarchy/agent configuration.

Mistake 10 — Memorizing Old Tmux Bindings

Current Omarchy provides:

```
Super + Alt + K
```

and:

```
Prefix + ?
```

Use them.

5.66 — Checkpoint

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

Concepts

- [ ] I understand what Tmux is.
- [ ] I understand the difference between session, window, and pane.
- [ ] I understand that Tmux persists independently of the terminal window.
- [ ] I understand how Hyprland workspaces and Tmux complement each other.

Launching

- [ ] I can launch a Tmux terminal with `Super + Alt + Return`.
- [ ] I know the current Omarchy prefix is `Ctrl + Space`.
- [ ] I know `Ctrl + B` also works on the current documented config.
- [ ] I know how to open Tmux keybinding help.

Panes

- [ ] I can split beside.

- [ ] I can split below.
- [ ] I can move between panes.
- [ ] I can resize panes.
- [ ] I can zoom a pane.
- [ ] I can close a pane.

Windows

- [ ] I can create a Tmux window.
- [ ] I can switch between windows.
- [ ] I can rename a window.
- [ ] I understand that Tmux windows are similar to terminal tabs.

Sessions

- [ ] I can list sessions.
- [ ] I can detach.
- [ ] I can reattach.
- [ ] I understand why working directories persist inside existing sessions.

Developer Layouts

- [ ] I understand what `tdl` does.
- [ ] I understand what `tds` does.
- [ ] I understand what `tdlm` is for.
- [ ] I understand what `tsl` is for.
- [ ] I know not to use multi-agent swarms by default.
- [ ] I can verify helper commands using `type`.

5.67 — What You Can Now Do

After completing Module 5, you can now:

- organize a terminal development environment;
- keep terminal work running independently of the terminal window;
- resume work after closing or reconnecting;
- create pane-based development layouts;
- organize tasks into Tmux windows;
- use the same terminal structure locally and over SSH;

- create an editor + AI + terminal workspace;
- understand the Omarchy developer-layout helpers;
- diagnose working-directory confusion;
- inspect Tmux helpers rather than guessing;
- choose intelligently when Tmux is useful.

Most importantly:

> *You now have a persistent terminal workspace capable of hosting your editor, coding agent, development server, logs, and other tools in one organized environment.*

Module 5 Summary

The hierarchy:

```
Tmux Session
├── Window
│   ├── Pane
│   └── Pane
└── Window
    └── Pane
```

Launch:

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

Prefix:

```
Ctrl + Space
```

with current documented compatibility for:

```
Ctrl + B
```

Panes:

```
Prefix + v
→ split beside

Prefix + h
→ split below

Ctrl + Alt + Arrow
→ move focus

Ctrl + Alt + Shift + Arrow
→ resize

Prefix + z
→ zoom

Prefix + x
→ close pane
```

Windows:

```
Prefix + c
→ new window

Alt + 1..9
→ switch window

Alt + Left/Right
→ previous/next window

Prefix + r
→ rename
```

Sessions:

```
Prefix + s
→ list sessions

Prefix + d
→ detach

tmux ls
→ list from shell

tmux attach
→ reattach
```

Help:

```
Super + Alt + K
→ Omarchy Tmux keybindings

Prefix + ?
→ Tmux cheatsheet
```

Developer layouts:

```
tdl
→ editor + AI + terminal

tds
→ editor + diff + terminal + AI

tdlm
→ layout per subdirectory

tsl
→ multi-pane command/swarm layout
```

The most important mental model:

> ***Closing a terminal window and ending a Tmux session are not the same thing.***

And the most important workflow lesson:

> ***Start the project in the correct directory before creating a fresh development layout. Reattaching to an existing Tmux session restores that session's existing state, including its working directories.***

MODULE 06

06

Development Environment Management with Omarchy and Mise

You now have:

- a working Omarchy desktop;
- strong terminal fundamentals;
- a Tmux-based development workspace.

About This Module

You now have:

- a working Omarchy desktop;
- strong terminal fundamentals;
- a Tmux-based development workspace.

The next step is making your development environment **clean, predictable, and reproducible**.

This is one of the most important modules in the entire course.

Many developers eventually end up with a workstation where:

- Node comes from one package manager;
- Python comes from the system;
- another Python came from somewhere else;
- a CLI was installed globally with npm;
- another tool came from the AUR;
- project requirements conflict;
- one update breaks another project;
- nobody remembers which version is active.

We want the opposite.

The goal is:

```
system packages
→ managed as system packages

Omarchy-integrated software
→ installed through Omarchy

developer runtimes and CLIs
→ managed through mise

project requirements
→ declared inside the project
```

This gives us a workstation that is easier to:

- understand;
- update;
- troubleshoot;

- reproduce;
- share with teammates;
- automate with AI agents.

Current official Omarchy documentation explicitly uses **mise** for the majority of supported development environments.

What You Will Learn

By the end of Module 6, you will understand:

- the three software-installation layers on our Omarchy workstation;
- when to use `omarchy pkg`;
- when to use `omarchy install`;
- when to use `mise`;
- what a development runtime is;
- why system Python and project Python should be treated separately;
- how mise manages multiple tool versions;
- the difference between global and project-specific tool versions;
- where mise configuration lives;
- how `mise.toml` works;
- why `.mise.toml` is not the default current filename;
- how project configuration overrides global configuration;
- how to install declared tools after cloning a repository;
- how to inspect active mise configuration;
- how to verify which executable the shell actually uses;
- how to pin or broadly request versions;
- how to use `mise.local.toml` for non-shared configuration;
- how mise trust affects project configuration;
- how to build a reproducible developer environment.

6.1 — The Core Rule: Do Not Install Everything the Same Way

On our Omarchy workstation, software falls into different categories.

A useful mental model is:

This is not an absolute rule for every package in existence.

It is our preferred decision process.

6.2 — Layer 1: System Packages

Omarchy is based on Arch Linux, so system software ultimately lives in the Arch package ecosystem.

Current Omarchy exposes package-management helpers through:

```
omarchy pkg
```

Inspect the current package command surface:

```
omarchy pkg --help
```

Depending on the current version, you may see functionality for:

- installing packages;

- removing packages;
- checking package presence;
- AUR packages;
- package discovery.

When to Think "System Package"

Examples include:

- Linux utilities;
- desktop applications;
- system libraries;
- hardware-related software;
- tools that should exist as part of the operating system.

The key question is:

> *Does this tool belong to the machine, or does it belong to a development environment?*

6.3 — Layer 2: Omarchy-Integrated Installation

Omarchy also exposes curated installation flows through:

```
omarchy install
```

Inspect:

```
omarchy install --help
```

Current Omarchy provides installation flows for areas such as:

- editors;
- browsers;
- development environments;
- AI tooling;
- services;
- other integrated applications.

The exact list evolves.

Do not maintain a static mental list.

Inspect your installed command surface.

6.4 — Layer 3: Mise

For the majority of development environments, current official Omarchy documentation uses:

```
mise
```

Mise is a development-tool version manager.

It can install and select versions of tools such as:

- Node.js;
- Python;
- Ruby;
- Go;
- Rust;
- Java;
- Bun;
- many development CLIs.

The current official Omarchy manual describes mise as the system that allows multiple versions of programming languages to coexist on the same machine.

6.5 — Why This Separation Matters

Consider this bad workstation state:

```
/usr/bin/python
→ manually replaced

npm -g
→ dozens of global tools

AUR
→ another Node version

curl | bash
→ random CLI

project
→ expects something else
```

This is difficult to reason about.

Instead:

```
Operating system
→ keeps its own system dependencies

mise global defaults
→ tools you prefer across projects

mise project config
→ exact project requirements
```

Much easier.

6.6 — The System Runtime Is Not Your Project Runtime

This is particularly important with Python.

Linux distributions may depend on their own Python version.

Suppose:

```
System
→ Python 3.14
```

but your project needs:

```
Project
→ Python 3.13
```

Do not "fix" the project by replacing the system Python.

Instead, let mise provide the project version.

Conceptually:

```
/usr/bin/python
→ operating-system Python

mise Python
→ development/project Python
```

Your shell can select the mise-managed version for development without destroying the system copy.

6.7 — Check Mise

Run:

```
mise --version
```

You should receive the installed mise version.

Then:

```
mise --help
```

You do not need to memorize the command list.

6.8 — Inspect Current Mise Configuration

Current mise provides:

```
mise config ls
```

to show active configuration files.

Run it from your home directory:

```
cd ~
mise config ls
```

Then run it inside one of your projects:

```
cd ~/Projects/YOUR_PROJECT
mise config ls
```

You may see additional project configuration appear.

This demonstrates mise's layered configuration system.

6.9 — Global Mise Configuration

Current mise uses:

```
~/.config/mise/config.toml
```

as the default global configuration file.

Inspect it:

```
bat ~/.config/mise/config.toml
```

If the file exists, it may contain something like:

```
[tools]
node = "26"
python = "3.13"
bun = "1"
```

These are personal defaults.

6.10 — What "Global" Means in Mise

Global does **not** mean:

> *Install into `/usr/bin` for the whole operating system.*

It means:

> *Use this as my personal default when no more specific project configuration overrides it.*

That is an important distinction.

Conceptually:

```
Global mise config
~/.config/mise/config.toml

        ↓ default

Project mise.toml

        ↓ overrides global
```

6.11 — Set a Global Tool Version

Current official mise syntax:

```
mise use --global node@24
```

Short form:

```
mise use -g node@24
```

This:

1. installs the tool version if necessary;
2. records it in your global configuration.

For example:

```
mise use --global python@3.13
```

Do not blindly copy these exact versions into a future workstation.

Choose versions appropriate to the current toolchain and projects.

6.12 — Check the Global Config After `mise use`

After:

```
mise use --global node@24
```

inspect:

```
bat ~/.config/mise/config.toml
```

You should see a `[tools]` entry.

Example:

```
[tools]
node = "24"
```

This is far better than a mysterious global installation with no declared configuration.

6.13 — Project Configuration

Current mise's default project filename is:

```
mise.toml
```

This point matters because older examples on the internet may use:

```
.mise.toml
```

The current default filename is:

```
mise.toml
```

unless the user deliberately changes mise's filename settings.

6.14 — Create a Practice Project

Create:

```
mkdir -p ~/Projects/mise-practice
cd ~/Projects/mise-practice
```

Initialize Git:

```
git init
```

Check:

```
pwd
```

6.15 — Add a Project-Specific Node Version

Inside the practice project:

```
mise use node@24
```

Current mise behavior:

- installs Node 24 if needed;
- creates or updates the project's `mise.toml`.

Inspect:

```
bat mise.toml
```

You should see something similar to:

```
[tools]
node = "24"
```

6.16 — Verify the Active Version

Run:

```
node --version
```

Then:

```
mise ls --current
```

Current mise documentation recommends:

```
mise ls --current
```

for seeing which tools are selected by the active configuration.

6.17 — Compare Project and Global Versions

Suppose your global default is:

```
Node 26
```

and your project requests:

```
Node 24
```

Inside the project:

```
node --version
```

should resolve to Node 24.

Now:

```
cd ~
node --version
```

should return your global default.

This demonstrates the core principle:

```
project config
→ more specific
→ overrides global config
```

6.18 — Real-World Example

On the workstation used to create this course, the global Node version was:

```
26.8.1
```

A project was configured with:

```
[tools]
node = "24"
```

Inside that repository:

```
Node 24.x
```

became active.

Outside it:

```
Node 26.8.1
```

remained the global default.

The system did not need two competing global Node installations.

This is exactly the behavior we want.

6.19 — `mise use` vs `mise install`

These commands have different purposes.

`mise use`

Example:

```
mise use node@24
```

This:

- installs the tool if needed;
- records the version requirement in configuration.

`mise install`

Example:

```
mise install
```

This installs tools already declared by the active configuration.

It does **not** need to create a new tool declaration.

6.20 — Typical Workflow After Cloning a Project

Imagine a repository contains:

```
mise.toml
```

with:

```
[tools]
node = "24"
python = "3.13"
```

After cloning:

```
git clone ...
cd project
```

run:

```
mise install
```

Mise installs the declared tool versions.

This is one of the major reproducibility advantages.

6.21 — The Official Omarchy Shortcut: `mise i`

Current official Omarchy development documentation also references:

```
mise i
```

as the short form for:

```
mise install
```

For clarity in the handbook, we will normally use:

```
mise install
```

until the learner is comfortable.

6.22 — Check Which Configs Are Active

Inside a project:

```
mise config ls
```

This is an extremely useful diagnostic.

If mise selects an unexpected version, ask:

> *Which config file is telling it to do that?*

`mise config ls` helps answer that.

6.23 — Check Which Tools Are Active

Run:

```
mise ls --current
```

This shows selected tools for the current location.

Use it when:

- Node version looks wrong;

- Python version looks wrong;
- one project behaves differently;
- an AI agent is using an unexpected runtime.

6.24 — Check Where the Executable Comes From

Suppose:

```
python --version
```

looks unexpected.

Run:

```
type python
```

Then:

```
command -v python
```

Then:

```
mise ls --current
```

This is a strong troubleshooting sequence.

Do not immediately reinstall Python.

6.25 — Tool Selection Through `$PATH`

Mise works by making the selected tool versions available through your command environment.

Conceptually:

```
project mise config
        ↓
mise selects tool version
        ↓
shell PATH/environment
        ↓
python / node / bun command
```

This is why the `$PATH` and command-resolution concepts from Module 4 matter.

6.26 — Run a Tool Through Mise Explicitly

Current mise supports:

```
mise exec -- node --version
```

This executes the command using the active mise environment.

You can also temporarily request a specific version without saving it.

Example:

```
mise exec node@24 -- node --version
```

This is useful for testing.

6.27 — Temporary Version Testing

Suppose you want to know whether your project works with a new Node version.

Instead of immediately editing `mise.toml`, try:

```
mise exec node@25 -- node --version
```

or run a relevant test through that environment.

This helps separate:

```
temporary experiment
```

from:

```
declared project requirement
```

6.28 — Version Requests vs Exact Pins

A project can request a version family.

Example:

```
[tools]
node = "24"
```

This means:

> *use an appropriate version in the Node 24 series.*

Current mise also supports exact pinning.

Example:

```
mise use --pin node@24.20.0
```

This records a concrete version.

6.29 — Which Should You Use?

Use a broad series when:

- patch updates are desirable;
- the project supports any version in that major/minor range.

Use exact pins when:

- reproducibility requires exact behavior;
- a dependency is sensitive;
- production matches one exact runtime;
- CI expects one precise version.

For many projects:

```
major/minor request
+
lockfile
```

is a good balance.

6.30 — Mise Lockfiles

Current mise supports:

```
mise.lock
```

for recording resolved versions.

The official mise documentation recommends committing normal project lockfiles.

Conceptually:

```
mise.toml
→ declares what the project wants

mise.lock
→ records what was resolved
```

This can improve reproducibility across machines.

We will not require lockfiles for every beginner project.

Know that they exist.

6.31 — Project Config Should Usually Go Into Git

A project-level:

`mise.toml`

usually belongs in version control.

Why?

Because it communicates:

> *"This repository expects these tools."*

Example:

```
[tools]
node = "24"
python = "3.13"
```

Then:

```
git add mise.toml
git commit -m "chore: add mise project config"
```

Other developers and AI environments can reproduce the same toolchain.

6.32 — Real Course Example

During development of the project used in this course, we created:

```
mise.toml
```

and committed it to Git.

This transformed the runtime version from:

```
something known only to one developer's machine
```

into:

```
part of the project's declared environment
```

That is the philosophy we want.

6.33 — `mise.local.toml`

Sometimes you need project configuration that should **not** be shared.

Current mise provides:

```
mise.local.toml
```

for personal project-specific configuration.

Examples:

- personal tool override;
- local development preference;
- machine-specific environment setting.

This file should generally be added to:

```
.gitignore
```

6.34 — Shared vs Local Configuration

A useful rule:

```
Does every developer need this?
        ↓
yes
→ mise.toml

no, only me / this machine
→ mise.local.toml
```

Do not put secrets in normal committed configuration.

6.35 — Configuration Layering

Current mise can combine configuration from multiple levels.

Conceptually:

```
~/.config/mise/config.toml
        ↓
~/Projects/mise.toml
        ↓
~/Projects/my-app/mise.toml
        ↓
~/Projects/my-app/mise.local.toml
```

More specific configuration can override broader defaults.

This allows sophisticated setups without duplicating everything.

6.36 — Example: Work-Wide Defaults

Suppose every repository under:

```
~/Projects/company
```

uses:

```
Node 24
```

You could have:

```
~/Projects/company/mise.toml
```

with:

```
[tools]
node = "24"
```

Then an individual project could override it.

This is powerful, but not necessary for a beginner.

Start with:

```
global
+
project
```

and add deeper layering only when needed.

6.37 — Mise Can Manage More Than Languages

Mise is not limited to:

- Node;
- Python;
- Ruby.

Current mise supports many CLI tools through multiple backends.

This means some development CLIs are better managed through mise than through the system package manager.

Examples can include:

- AI coding CLIs;
- formatting tools;
- build tools;
- release utilities.

Always check the current registry and installation source.

6.38 — Omarchy's Mise-Managed Lazy Tools

Current Omarchy uses mise-backed wrappers for a number of development tools.

This is especially visible in current AI-agent integration.

The current official Omarchy AI documentation states that major coding-agent CLIs are exposed through lazy-loaded launcher stubs under:

```
~/.local/bin
```

and installed through mise when first used.

This reinforces the architecture:

```
Omarchy integration
+
mise-managed CLI
```

rather than forcing every developer CLI into the base operating system.

6.39 — Inspect a Mise-Managed Command

For example:

```
type omp
```

or another installed tool.

Then:

```
command -v omp
```

You may see a wrapper under:

```
~/.local/bin
```

This is a good example of why command resolution matters.

6.40 — Omarchy Development Environment Installers

Current official Omarchy documentation provides a broad set of development environments through:

```
Super + Space
→ Install
→ Development
```

Current documented environments include categories such as:

- Node;
- Bun;
- Deno;
- Python;
- Ruby;
- Go;
- Rust;
- Java;
- PHP frameworks;
- Elixir;
- .NET;
- Zig;
- OCaml;
- Clojure;
- Scala.

The exact list can change.

Use the current menu or:

```
omarchy install --help
```

to discover what your installed version supports.

6.41 — Why Use Omarchy's Development Installer?

The Omarchy installer can provide a curated setup around the runtime.

That may include:

- dependencies;
- environment setup;
- integrations;
- expected conventions.

For a supported development environment, it is often a good starting point.

For precise runtime versioning and project overrides, mise remains the core tool.

6.42 — A Practical Decision Tree

When you want to install something, ask:

```
Is it a desktop/system application?
        ↓
yes → check Omarchy install/pkg

Is it an Omarchy-supported development environment?
        ↓
yes → consider Omarchy development installer

Is it a runtime or development CLI whose version matters by project?
        ↓
yes → mise
```

Examples:

```
Chromium
→ Omarchy browser/install layer

system USB utility
→ package layer

Node.js
→ mise / Omarchy development setup

Python project runtime
→ mise

AI coding CLI
→ Omarchy lazy wrapper / mise-backed tooling
```

6.43 — Do Not Default to `sudo pacman -S`

Arch documentation frequently uses:

```
sudo pacman -S PACKAGE
```

That is a valid underlying Arch command.

But our workstation policy is:

> ***First decide whether the tool belongs in the system layer.***

If it is really a project runtime, pacman may be the wrong layer.

This is especially important for:

- Node;
- Python;
- Ruby;
- Go;
- Java;
- development CLIs.

6.44 — Do Not Default to `npm install -g`

Another common pattern:

```
npm install -g TOOL
```

Global npm installation can be appropriate in some contexts.

But if a CLI can be managed through mise or belongs to the project, that may provide a cleaner lifecycle.

Ask:

```
Does the tool need a specific version?
Should teammates reproduce it?
Does it belong to one project?
Can mise manage it?
```

before installing globally.

6.45 — The Same Principle Applies to Python `pip`

Avoid casually filling your global environment with:

```
pip install ...
```

For project Python packages, use an appropriate project environment/package manager.

Mise manages the **Python runtime version**, not the entire Python dependency strategy.

Later project tooling may use:

- `uv`;

- virtual environments;
- Poetry;
- pip;
- project-specific tooling.

Those are a different layer from runtime selection.

6.46 — Runtime Manager vs Package Manager

This distinction matters.

Example:

```
mise
→ selects Python 3.13

uv / pip
→ installs Python packages for the project
```

Similarly:

```
mise
→ selects Node 24

npm / pnpm / bun
→ installs project JavaScript dependencies
```

Mise does not replace every ecosystem package manager.

6.47 — Verify Node

Run:

```
type node
```

Then:

```
node --version
```

Then:

```
mise ls --current
```

You want these three outputs to tell a consistent story.

6.48 — Verify Python

Run:

```
type python
```

Then:

```
python --version
```

Then:

```
mise ls --current
```

If you are inside a project with a Python override, expect the project version.

6.49 — Verify Bun

Run:

```
type bun
```

Then:

```
bun --version
```

Again:

```
mise ls --current
```

6.50 — Compare with System Python

If you want to inspect the underlying Arch Python explicitly:

```
/usr/bin/python --version
```

Then compare:

```
python --version
```

If mise controls your active development Python, these may differ.

That is fine.

In fact, that separation is intentional.

6.51 — Do Not Remove System Python Because You Use Mise

This deserves a direct warning.

Even if:

```
python
```

resolves to a mise-managed version,

do not assume:

```
/usr/bin/python
```

is redundant.

System packages may depend on it.

Leave operating-system dependencies alone unless you understand the package dependency graph.

6.52 — Trust and Project Configuration

Current mise documentation warns that project configuration can include:

- tasks;
- hooks;
- environment behavior;

that may execute code.

Therefore:

> *Treat a `mise.toml` from an unknown repository as executable development configuration, not harmless text.*

Review unfamiliar project configuration before trusting/running it.

6.53 — `mise trust`

Current mise provides:

```
mise trust
```

for explicitly trusting configuration you have reviewed.

Exact trust behavior can depend on current mise mode/settings.

Use current official mise documentation when configuring stricter trust policies.

6.54 — Why This Matters for AI Development

AI agents frequently:

- clone repositories;
- enter project directories;
- run setup commands;
- execute tests.

If the repository contains development configuration capable of executing hooks/tasks, trust boundaries matter.

A strong AI workstation does not mean:

> *auto-approve every repository setup script.*

It means:

> *make environment setup reproducible **and** inspect what will execute.*

6.55 — Environment Variables in `mise.toml`

Current mise project configuration can also declare environment variables.

Example:

```
[env]
NODE_ENV = "development"
```

This can make environment expectations explicit.

However:

> *Do not commit secrets directly into `mise.toml`.*

Bad:

```
[env]
API_KEY = "super-secret-real-key"
```

Use a proper secrets strategy.

6.56 — Mise Tasks

Current mise can also define project tasks.

Example:

```
[tasks.test]
run = "npm test"
```

Then:

```
mise run test
```

This can become useful for standardized project commands.

However, we will not turn Module 6 into a complete mise course.

For our workstation, the primary role is:

```
tool version management
+
reproducible project environment
```

6.57 — Upgrade Tools Carefully

Current mise supports commands such as:

```
mise upgrade
```

for moving within configured version requests.

Do not automatically upgrade every project runtime before checking compatibility.

For example:

```
Project says Node 24
```

does not mean:

```
switch project to Node 26 because newer exists
```

Project requirements come first.

6.58 — Omarchy Updates and Mise Tools

Current Omarchy's update pipeline includes mise-managed tooling.

The current official Omarchy update architecture runs mise updates as part of the supported update process.

This is another reason to prefer:

```
omarchy update
```

for normal workstation maintenance.

It keeps Omarchy's package-backed and mise-managed layers aligned with its expected update flow.

6.59 — Global Defaults Should Be Conservative

Do not install every language "just in case."

A good global configuration contains tools you genuinely use frequently.

Example:

```
[tools]
node = "26"
python = "3.13"
bun = "1"
```

If you never use:

- Rust;
- Go;

- Ruby;
- Java;

there is no reason to populate the machine with all of them.

Install environments when needed.

6.60 — Project Configuration Should Express Project Reality

Suppose a project genuinely requires:

```
Node 24
```

Declare:

```
[tools]
node = "24"
```

Do not change it to Node 26 merely because your machine's global default is newer.

The project configuration exists specifically to prevent this kind of machine-dependent assumption.

6.61 — Reproducibility Workflow

A healthy project workflow looks like:

```
git clone repository
        ↓
cd repository
        ↓
inspect mise.toml
        ↓
mise install
        ↓
mise ls --current
        ↓
install project dependencies
        ↓
run tests
```

This gives both humans and AI agents a predictable setup path.

6.62 — New Project Workflow

For a new project:

```
mkdir -p ~/Projects/new-project
cd ~/Projects/new-project
git init
```

Choose runtime:

```
mise use node@24
```

Check:

```
bat mise.toml
```

Verify:

```
node --version
mise ls --current
```

Commit:

```
git add mise.toml
git commit -m "chore: add mise project config"
```

Now the runtime requirement is part of the repository.

6.63 — Practical Exercise: Global vs Project Node

First inspect your global version:

```
cd ~
node --version
mise ls --current
```

Now create:

```
mkdir -p ~/Projects/mise-node-test
cd ~/Projects/mise-node-test
```

Set a different major version:

```
mise use node@24
```

Check:

```
node --version
mise ls --current
```

Return home:

```
cd ~
node --version
```

You should see the project override disappear.

6.64 — Practical Exercise: Inspect Configuration Resolution

From home:

```
mise config ls
```

Then from your test project:

```
cd ~/Projects/mise-node-test
mise config ls
```

Identify:

- global configuration;
- project configuration.

This is one of the most useful mise debugging exercises.

6.65 — Practical Exercise: Clone-Style Setup

Inside your practice project:

```
cat mise.toml
```

Then simulate a new setup by asking:

> *If another developer cloned this repository, what command would install the declared runtime?*

Answer:

```
mise install
```

That simple contract is the foundation of reproducibility.

6.66 — Practical Exercise: Command Resolution

Inside the project:

```
type node
command -v node
node --version
mise ls --current
```

Then return home and repeat.

Observe what changes.

6.67 — Practical Exercise: Add Python Too

Inside the project:

```
mise use python@3.13
```

Inspect:

```
bat mise.toml
```

You should now see multiple tools.

Example:

```
[tools]
node = "24"
python = "3.13"
```

Verify:

```
node --version
python --version
mise ls --current
```

6.68 — Practical Exercise: Commit the Environment

Run:

```
git status
```

Then:

```
git add mise.toml
git commit -m "chore: define project runtime versions"
```

You have now made the development environment part of the project history.

6.69 — Practical Exercise: Local-Only Override

Create:

```
mise.local.toml
```

with a harmless local tool/config example if you want to experiment.

Then ensure:

```
mise.local.toml
```

is listed in:

```
.gitignore
```

The lesson is:

```
shared config
→ version control

personal local config
→ not version control
```

6.70 — Common Development Environment Mistakes

Mistake 1 — Installing Every Runtime with Pacman

If runtime version matters by project, use mise.

Mistake 2 — Replacing System Python

Do not overwrite or remove the operating system's Python just because your project needs a different version.

Mistake 3 — Using the Wrong Mise Filename

Current default:

```
mise.toml
```

not:

```
.mise.toml
```

unless you have explicitly configured otherwise.

Mistake 4 — Setting Everything Globally

If a project requires a particular version, declare it in the project.

Mistake 5 — Forgetting to Commit `mise.toml`

A local-only runtime choice is not reproducible.

Mistake 6 — Committing Secrets

Do not place real credentials in shared mise configuration.

Mistake 7 — Running `mise install` on an Unknown Repository Without Review

Project configuration can include executable behavior.

Review unfamiliar configuration first.

Mistake 8 — Assuming `python` Comes from `/usr/bin`

Run:

```
type python
```

before making assumptions.

Mistake 9 — Installing a CLI Globally with npm/pip Before Checking Mise

Determine the correct software layer first.

Mistake 10 — Upgrading Runtime Versions Because "Newer Is Better"

Project compatibility is more important than novelty.

6.71 — Checkpoint

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

Architecture

- [ ] I understand the difference between system packages, Omarchy-integrated installers, and mise.
- [ ] I know why not every development tool belongs in pacman.
- [ ] I understand why system Python should remain separate from project Python.

Mise

- [ ] I can check the mise version.
- [ ] I know the global config path.
- [ ] I know the default project config filename is `mise.toml`.
- [ ] I can use `mise use`.
- [ ] I can use `mise use --global`.
- [ ] I can use `mise install`.
- [ ] I can use `mise config ls`.
- [ ] I can use `mise ls --current`.

Project Environments

- [ ] I can create a project-specific runtime override.
- [ ] I understand that project config overrides global config.

- [ ] I know why `mise.toml` should usually be committed.
- [ ] I know what `mise.local.toml` is for.
- [ ] I understand the purpose of a mise lockfile at a high level.

Debugging

- [ ] I can use `type node`.
- [ ] I can use `type python`.
- [ ] I can use `command -v`.
- [ ] I know how to inspect which mise configs are active.
- [ ] I know not to reinstall a runtime before checking command resolution.

Security

- [ ] I understand that project development config should be reviewed.
- [ ] I know not to commit secrets into `mise.toml`.
- [ ] I understand why AI agents should not blindly execute unknown project setup.

6.72 — What You Can Now Do

After completing Module 6, you can now:

- choose the correct installation layer for development tools;
- protect the operating system from project runtime changes;
- manage multiple Node/Python/etc. versions;
- create global developer defaults;
- override those defaults per project;
- reproduce a project's runtime environment after cloning;
- commit runtime requirements to Git;
- inspect active environment configuration;
- diagnose incorrect tool versions;
- distinguish runtime management from dependency management;
- use mise safely in AI-assisted development workflows.

Most importantly:

> *Your workstation's development environment is becoming declarative instead of accidental.*

Module 6 Summary

The three layers:

```
System software
→ Omarchy package layer

Omarchy-integrated apps/environments
→ omarchy install

Developer runtimes / versioned CLIs
→ mise
```

Global config:

```
~/.config/mise/config.toml
```

Project config:

```
mise.toml
```

Local-only project config:

```
mise.local.toml
```

Core mise commands:

```
mise use node@24
```

```
mise use --global node@24
```

```
mise install
```

```
mise config ls
```

```
mise ls --current
```

```
mise exec -- node --version
```

Debugging:

```
type node
command -v node
node --version
mise config ls
mise ls --current
```

The core hierarchy:

```
global defaults
        ↓
project configuration
        ↓
local project overrides
```

And the most important rule:

> ***The operating system owns its system runtimes. Your projects own their development runtime requirements. Mise lets both coexist cleanly.***

MODULE 07

07

Git and GitHub on Omarchy

You now have:

- a working Omarchy workstation;
- strong terminal fundamentals;
- Tmux for persistent development sessions;
- mise for reproducible runtimes.

About This Module

You now have:

- a working Omarchy workstation;
- strong terminal fundamentals;
- Tmux for persistent development sessions;
- mise for reproducible runtimes.

Now we add the version-control layer.

Git is the foundation of modern software development.

GitHub adds:

- remote hosting;
- collaboration;
- pull requests;
- issues;
- code review;
- CI/CD integration;
- repository discovery;
- backup of source history.

Omarchy already provides a strong Git-centered developer workflow.

Current official Omarchy documentation includes:

- GitHub CLI integration;
- Lazygit;
- Git worktree helper functions;
- terminal-based development layouts that pair naturally with Git and AI coding agents.

This module teaches the Git workflow you actually need for daily development.

It is **not** a complete Git internals course.

The goal is:

```
create
→ inspect
→ stage
→ commit
→ push
→ branch when needed
→ use worktrees when parallel work becomes useful
```

By the end, you should be able to create a local project, turn it into a Git repository, create a private GitHub repository, connect the two, push your work, and understand the concepts needed for AI-assisted development.

What You Will Learn

By the end of Module 7, you will be able to:

- understand what Git tracks;
- understand repository, working tree, staging area, and commit history;
- initialize a repository;
- inspect repository state;
- understand tracked vs untracked files;
- stage files;
- create commits;
- inspect history;
- understand remotes;
- authenticate with GitHub CLI;
- create a private GitHub repository from an existing folder;
- push local commits;
- clone repositories;
- pull remote changes;
- understand branches;
- create and switch branches;
- understand merge conflicts conceptually;
- understand `.gitignore`;
- understand when `git restore` and `git reset` are appropriate;
- use Lazygit as a visual terminal interface;
- understand Git worktrees;

- use Omarchy's `ga` and `gd` worktree helpers at a high level;
- understand why worktrees are useful for parallel AI agents;
- avoid common Git mistakes.

7.1 — What Is Git?

Git is a distributed version-control system.

At a high level, Git records snapshots of your project's history.

Without Git:

```
project/
├── app-final.js
├── app-final-2.js
├── app-really-final.js
└── app-final-working.js
```

With Git:

```
project/
└── app.js

Git history:
A → B → C → D
```

Each commit records a meaningful point in the project's history.

This allows you to:

- see what changed;
- understand why it changed;
- return to older versions;
- work on multiple lines of development;
- collaborate safely.

7.2 — Git vs GitHub

These are different things.

Git

Git is the version-control system.

It works locally.

You can use Git without GitHub.

GitHub

GitHub is an online hosting and collaboration platform for Git repositories.

Conceptually:

```
Your computer
└── Git repository
       ↓ push

GitHub
└── remote Git repository
```

Do not confuse:

`git`

with:

`GitHub`

7.3 — Check Git

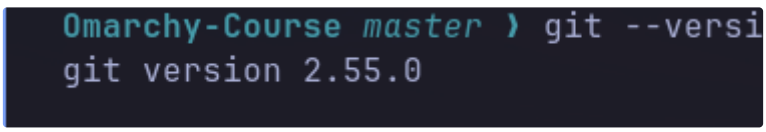
Run:

```
git --version
```

Current Git documentation recommends this as the basic version check.

On the workstation used while creating this course, Git was already installed through the Omarchy environment.

If Git is missing on a future Omarchy release, use the current Omarchy package/install workflow instead of assuming an old package command.



Screenshot 7.3-A
`git --version` output

7.4 — Check GitHub CLI

Run:

```
gh --version
```

The GitHub CLI command is:

```
gh
```

Current Omarchy development documentation includes GitHub CLI integration as part of its developer tooling.

The official GitHub CLI documentation is available at:

```
https://cli.github.com/manual/
```

Screenshot 7.4-A
`gh --version` output

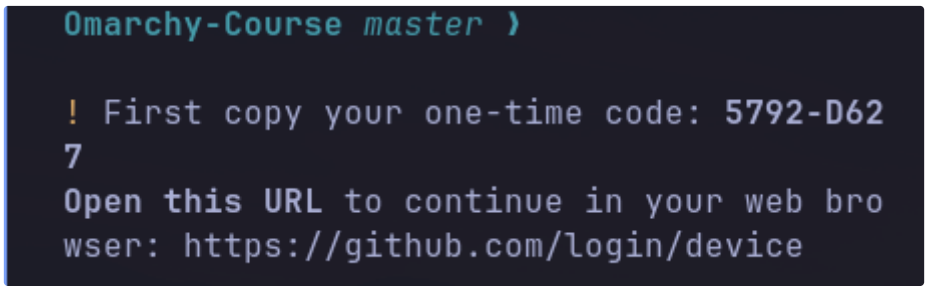
7.5 — Authenticate with GitHub

Run:

```
gh auth login
```

Current GitHub CLI behavior uses a browser-based authentication flow by default for `github.com`.

Follow the prompts.

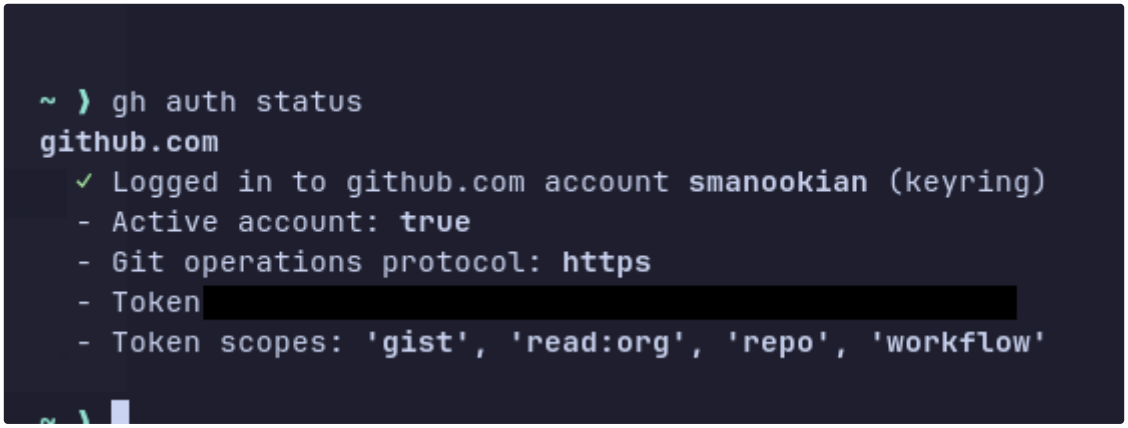


Screenshot 7.5-B
`gh auth login` device-code prompt

After authentication:

```
gh auth status
```

The official GitHub CLI documentation states that `gh auth status` reports the active account and authentication state for known GitHub hosts.



Screenshot 7.5-A
Successful `gh auth status`

7.6 — Verify Which Account Is Active

Run:

```
gh auth status
```

Read the output carefully.

If you have multiple GitHub accounts, verify the active one before creating or pushing repositories.

The repository will otherwise be created under the wrong account or organization.

7.7 — What Is a Repository?

A Git repository is a project directory whose history is tracked by Git.

A normal directory:

```
my-project/
├── src/
└── README.md
```

After Git initialization:

```
my-project/
├── .git/
├── src/
└── README.md
```

The hidden:

```
.git
```

directory contains Git's repository metadata.

Do not manually edit files inside `.git` unless you deeply understand what you are doing.

7.8 — Create a Practice Project

Run:

```
mkdir -p ~/Projects/git-practice
cd ~/Projects/git-practice
```

Create a README:

```
echo "# Git Practice" > README.md
```

Check:

```
ls -la
```

At this point, it is just a normal directory.

Screenshot 7.9-A
directory before `git init`

7.9 — Initialize Git

Run:

```
git init
```

Check again:

```
ls -la
```

You should now see:

```
.git
```

The directory is now a Git repository.

Screenshot 7.9-B
`.git` directory after initialization

7.10 — The Four Important Git States

For the beginner workflow, think in four layers:

```
Working tree
    ↓ git add

Staging area
    ↓ git commit

Local commit history
    ↓ git push

Remote repository
```

This simple model explains most everyday Git commands.

7.11 — Working Tree

The working tree is the actual files you are editing.

Example:

```
README.md
src/app.ts
package.json
```

When you modify a file, the working tree changes.

Git notices the difference.

7.12 — The Staging Area

Before committing, you normally select which changes belong in the next commit.

This selection is called the **staging area** or **index**.

Conceptually:

```
Working tree

README.md changed
app.ts changed
debug.log changed

        ↓ git add README.md app.ts

Staging area

README.md
app.ts

debug.log remains unstaged
```

This lets you create focused commits.

7.13 — Commits

A commit is a recorded snapshot in the repository history.

Example:

```
A
↓
B
↓
C
```

Each commit includes information such as:

- changes;
- author;
- timestamp;
- commit message;

- parent commit.

A good commit should represent one understandable unit of work.

7.14 — Remote Repository

A remote repository is another copy of the Git repository.

For this course, that is normally:

```
GitHub
```

The conventional remote name is:

```
origin
```

Conceptually:

```
Local repository
      ↓ push
origin
      ↓
GitHub repository
```

7.15 — `git status`: Your Most Important Git Command

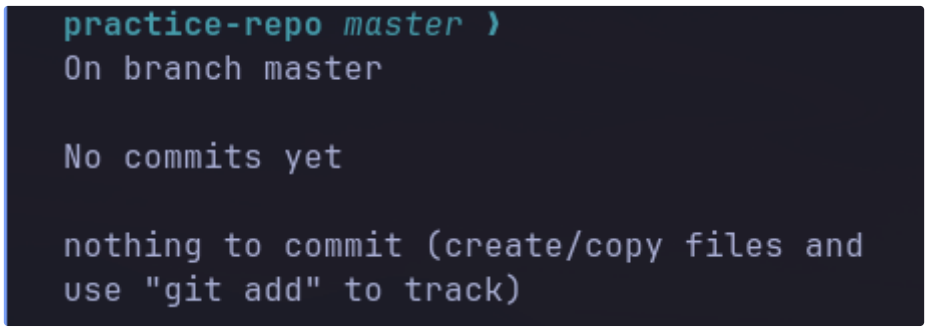
Run:

```
git status
```

The official Git documentation describes `git status` as showing:

- changes staged for commit;
- changes not yet staged;
- untracked files.

You will use it constantly.



Screenshot 7.15-A
`git status`

COURSE RULE

> ***When you are unsure what state your repository is in, run `git status`.***

7.16 — Short Status

Git also provides:

```
git status --short
```

or:

```
git status -s
```

Example:

```
?? README.md
```

means:

```
untracked file
```

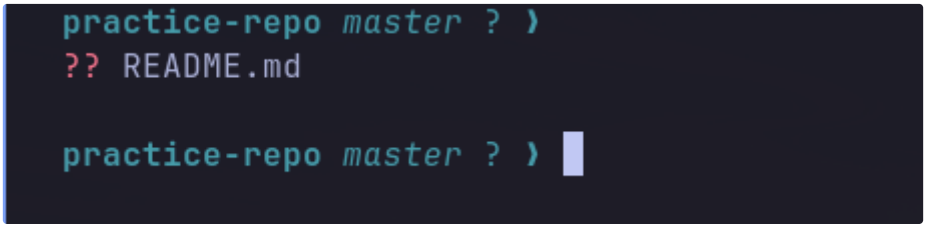
After staging:

```
A  README.md
```

means:

```
added to staging area
```

Short status becomes very useful once you are comfortable with the symbols.



Screenshot 7.16-A
`git status --short`

7.17 — Untracked Files

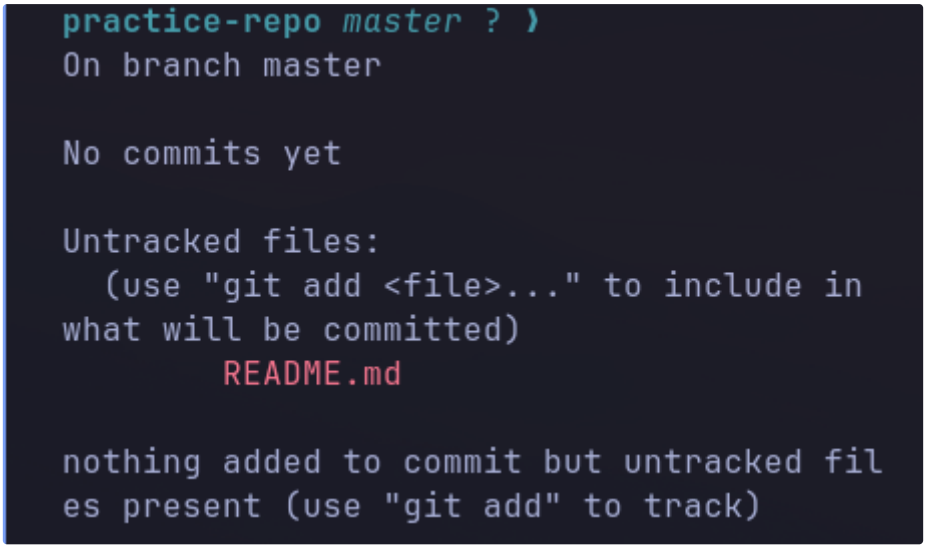
Run:

```
git status
```

Your new `README.md` should initially be untracked.

That means:

> *Git sees the file, but it has not yet been added to version control.*



Screenshot 7.17-A
untracked files in `git status`

7.18 — Stage a File

Run:

```
git add README.md
```

Then:

```
git status
```

The file should now appear under changes to be committed.

7.19 — Stage Everything

You will often see:

```
git add .
```

This stages changes under the current directory.

This is convenient, but do not use it blindly.

Before:

```
git add .
```

run:

```
git status
```

and make sure you actually want everything staged.

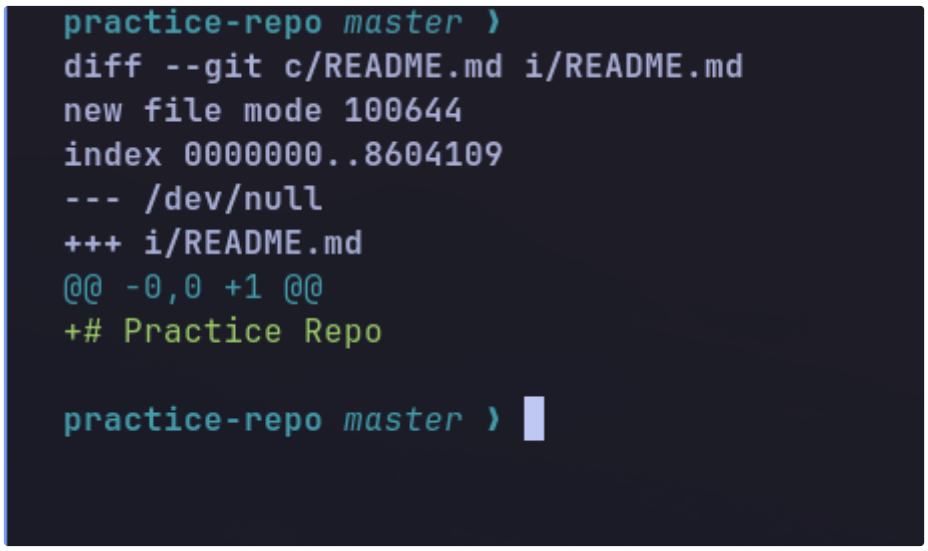
7.20 — Inspect Staged Changes

Before committing:

```
git diff --staged
```

This shows the changes currently staged for the next commit.

This is a very good habit.



Screenshot 7.20-A

`git diff --staged`

Workflow:

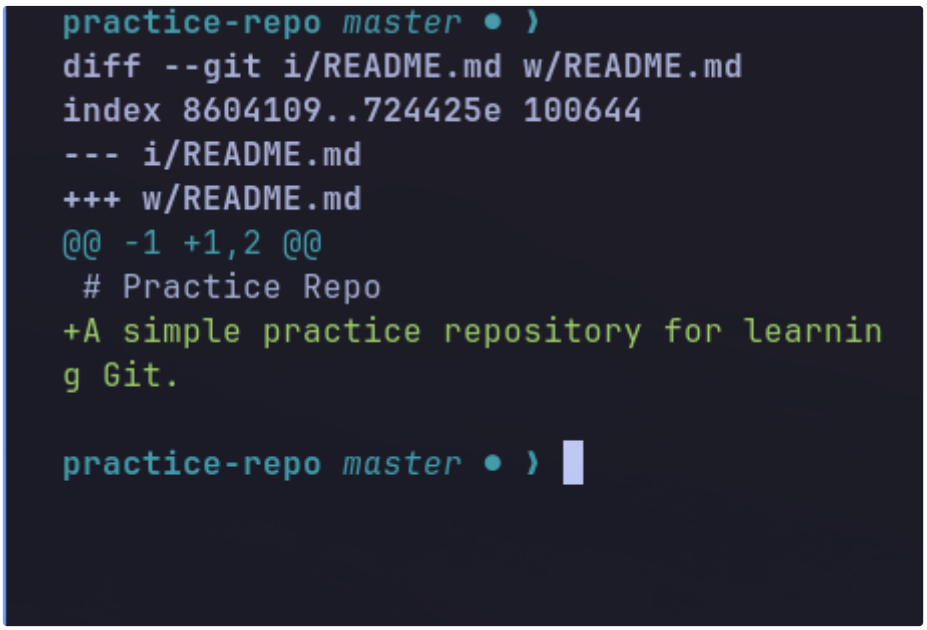
```
git status
↓
git diff
↓
git add
↓
git diff --staged
↓
git commit
```

7.21 — Inspect Unstaged Changes

Use:

```
git diff
```

This shows tracked changes in the working tree that have not yet been staged.



Screenshot 7.21-A
`git diff`

7.22 — Create Your First Commit

Run:

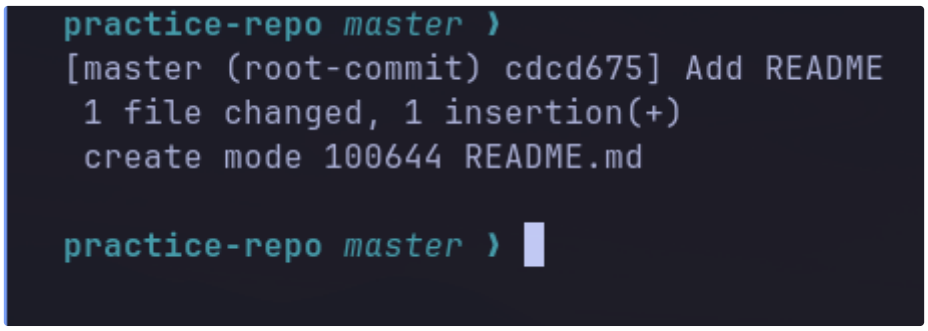
```
git commit -m "docs: add initial README"
```

The `-m` option supplies the commit message.

Check:

```
git status
```

You should now have a clean working tree.



Screenshot 7.22-A
first commit

7.23 — Good Commit Messages

A commit message should explain the change.

Bad:

```
stuff
```

Bad:

```
changes
```

Better:

```
add project runtime configuration
```

Better:

```
fix monitor refresh-rate configuration
```

Many development teams use a conventional style such as:

```
feat:
fix:
docs:
chore:
refactor:
test:
```

Example:

```
chore: add mise project config
```

This course will often use that style because it is readable and automation-friendly.

7.24 — Check Commit History

Run:

```
git log
```

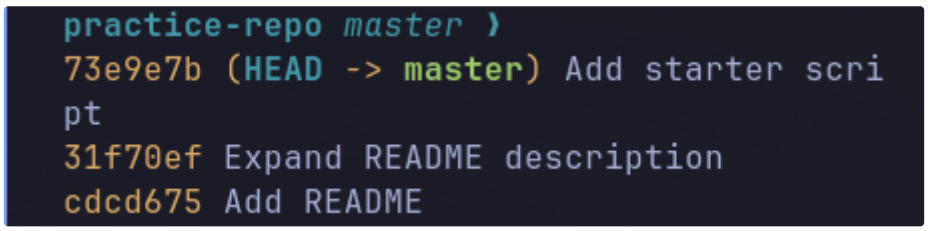
For a compact view:

```
git log --oneline
```

Example:

```
4c8a3f4 chore: add mise project config
```

This is an excellent everyday history view.



Screenshot 7.24-A
`git log --oneline`

7.25 — Check the Current Branch

Run:

```
git branch --show-current
```

You may see:

```
main
```

or another configured default branch.

7.26 — Create a Private GitHub Repository from an Existing Folder

This is one of the most useful GitHub CLI workflows.

Suppose you already have:

```
~/Projects/my-project
```

with Git initialized and at least one commit.

Inside the project:

```
gh repo create my-project --private --source=. --remote=origin --push
```

Current official GitHub CLI documentation states that:

- `--private` creates a private repository;
- `--source=.` uses the current local repository as the source;
- `--remote=origin` creates the remote with that name;
- `--push` pushes local commits.

This is the exact workflow we want for quickly putting an existing local project on GitHub.

7.27 — What That Command Does

Break it down:

```
gh repo create my-project
```

creates a GitHub repository.

```
--private
```

sets the repository visibility to private.

```
--source=.
```

uses the current directory as the local source repository.

```
--remote=origin
```

adds the GitHub repository as:

```
origin --push
```

pushes local commits.

One command replaces several manual steps.

7.28 — Verify the Remote

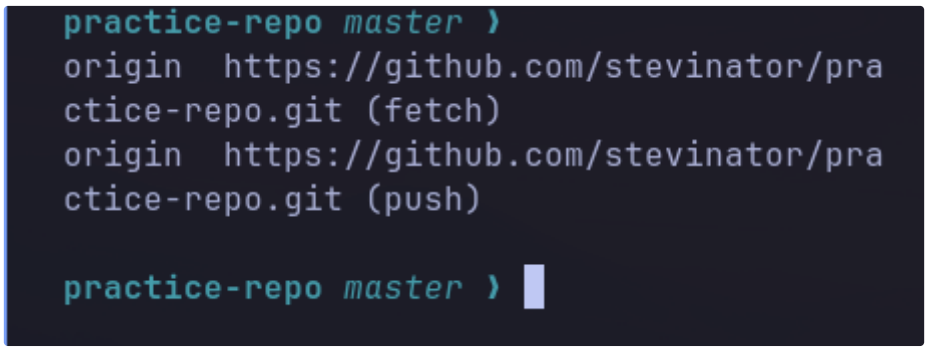
Run:

```
git remote -v
```

You should see something similar to:

```
origin  https://github.com/USERNAME/my-project.git (fetch)
origin  https://github.com/USERNAME/my-project.git (push)
```

Exact transport may differ depending on your Git/GitHub configuration.



Screenshot 7.28-A
`git remote -v`

7.29 — Open the Repository on GitHub

Current GitHub CLI supports:

```
gh repo view --web
```

This opens the current repository in the browser.

Very useful after repository creation.

7.30 — Push Later Changes

Make a change:

```
echo "Git and GitHub practice repository." >> README.md
```

Inspect:

```
git status
```

Then:

```
git diff
```

Stage:

```
git add README.md
```

Commit:

```
git commit -m "docs: expand README"
```

Push:

```
git push
```

This is the basic everyday loop.

7.31 — The Core Daily Git Workflow

Memorize this flow, not dozens of commands:

```
git status
```

```
git diff
```

```
git add ...
```

```
git diff --staged
```

```
git commit -m "..."
```

```
git push
```

That is enough for a large percentage of daily development.

7.32 — Clone an Existing Repository

To download a repository from GitHub:

```
gh repo clone OWNER/REPOSITORY
```

or standard Git:

```
git clone REPOSITORY_URL
```

With GitHub CLI, example:

```
gh repo clone octocat/Hello-World
```

For your own repository:

```
gh repo clone YOUR_USERNAME/YOUR_REPO
```

7.33 — Pull Remote Changes

If the remote repository changed:

```
git pull
```

Conceptually:

```
GitHub
  ↓ fetch changes
  ↓ integrate
local repository
```

For simple beginner workflows, `git pull` is convenient.

Later, advanced teams may prefer more explicit fetch/rebase strategies.

Do not overcomplicate this yet.

7.34 — Fetch Without Integrating

Use:

`git fetch`

This retrieves remote information without automatically changing your working branch.

This is useful when you want to inspect what changed before integrating it.

Conceptually:

```
git fetch
→ update knowledge of remote state

git pull
→ fetch + integrate
```

7.35 — `.gitignore`

Some files should not be committed.

Examples:

```
node_modules/
.env
build output
temporary files
logs
IDE caches
mise.local.toml
```

Git uses:

`.gitignore`

to define paths that should normally remain untracked.

7.36 — Create a `.gitignore`

Example:

```
cat > .gitignore <<'EOF'
node_modules/
.env
mise.local.toml
*.log
EOF
```

Inspect:

```
bat .gitignore
```

Then:

```
git status
```

Ignored files should no longer appear as ordinary untracked files.

Screenshot 7.36-A
`.gitignore` content

7.37 — Do Not Commit Secrets

Never commit:

```
API keys
passwords
private SSH keys
database credentials
access tokens
real .env secrets
```

A repository being private does **not** make committing secrets a good practice.

Git preserves history.

Deleting a secret later does not automatically erase it from every historical commit.

7.38 — A Good Project Start

A clean project might contain:

```
my-project/
├── .git/
├── .gitignore
├── AGENTS.md
├── mise.toml
├── README.md
├── package.json
└── src/
```

Each file has a different role:

```
.git
→ Git history

.gitignore
→ files Git should ignore

AGENTS.md
→ AI-agent project instructions

mise.toml
→ development runtime requirements

README.md
→ human project documentation
```

This is a strong foundation for AI-assisted software development.

7.39 — Restore an Unstaged File

Suppose you modify a tracked file and want to discard those unstaged changes.

Modern Git provides:

```
git restore FILE
```

Example:

```
git restore README.md
```

WARNING

This discards the working-tree changes to that file.

Inspect first:

```
git diff README.md
```

Then restore only if you are sure.

7.40 — Unstage a File

Suppose you ran:

```
git add README.md
```

but do not want it in the next commit.

Use:

```
git restore --staged README.md
```

The file remains changed in the working tree.

It simply leaves the staging area.

This distinction is important.

7.41 — Do Not Reach for `git reset --hard` Casually

You will see:

```
git reset --hard
```

in many tutorials.

It can discard working-tree changes.

Do not use it as a generic "make Git clean" button.

Prefer targeted operations when possible:

```
git restore FILE
```

or:

```
git restore --staged FILE
```

Understand the state you are changing.

7.42 — Branches

A branch is a named line of development.

Suppose:

```
main
```

contains stable project history.

You want to develop:

```
new-login
```

without immediately changing `main`.

Conceptually:

```
A ── B ── C
          │
          ├── main
          │
          └── feature/login
```

You create a branch and work there.

Screenshot 7.42-A
branch graph (`git log --oneline --graph --all`)

7.43 — Create and Switch to a Branch

Modern Git provides:

```
git switch -c feature/login
```

The official Git documentation defines:

```
git switch -c <new-branch>
```

as creating and switching to a new branch.

Check:

```
git branch --show-current
```

Expected:

```
feature/login
```

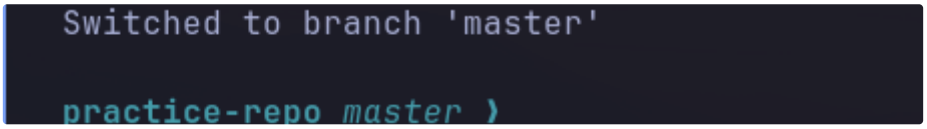
Screenshot 7.43-A
`git switch -c`

7.44 — Switch Back

Run:

```
git switch main
```

The working tree changes to represent the selected branch.



Screenshot 7.44-A
switch back to previous branch

7.45 — List Branches

Run:

```
git branch
```

The current branch is marked with:

```
*
```

Screenshot 7.45-A
`git branch` listing

Example:

```
* main
  feature/login
```

7.46 — Why Branches Matter

Branches allow:

- isolated feature development;
- bug fixes;
- experimentation;
- code review;
- pull requests.

They also become important when AI agents work on changes.

But a normal Git branch still uses the same working directory.

Only one branch is checked out there at a time.

That is where worktrees become interesting.

7.47 — Merging Conceptually

Suppose:

```
main
A — B ————————————
     \
      C — D
      feature/login
```

After the feature is accepted, its changes can be merged back.

Conceptually:

```
A — B ——————— E
     \       /
      C — D
```

You do not need deep merge strategy knowledge yet.

The key idea is:

> *A branch isolates a line of development; merge integrates it.*

7.48 — Merge Conflicts

A conflict occurs when Git cannot automatically decide how to combine changes.

Example:

`main` changes:

```
line 10
```

and your branch independently changes the same area.

Git may ask you to resolve the conflict manually.

Conflict does **not** mean:

> *Git is broken.*

It means:

> *Git needs a human or agent to decide the intended final content.*

We will cover conflict resolution in the troubleshooting material rather than overloading this module.

7.49 — Branches and AI Agents

An AI coding agent should generally not make unrelated experimental changes directly on your important stable branch.

A common workflow:

```
main
↓
feature branch
↓
AI makes changes
↓
tests
↓
review
↓
merge
```

This gives you a clean rollback/review boundary.

7.50 — What Is a Worktree?

Git worktrees allow the same repository to have multiple working directories checked out at the same time.

The official Git documentation describes a repository as having:

- one main worktree;
- zero or more linked worktrees.

Conceptually:

```
Repository history
      │
      ├── ~/Projects/app
      │       main
      │
      ├── ~/Projects/app-feature-auth
      │       feature/auth
      │
      └── ~/Projects/app-fix-api
              fix/api
```

All share the same repository history.

But each has its own working directory.

7.51 — Why Worktrees Are Powerful

Without worktrees:

```
one repository folder
→ switch branches
→ files change underneath you
```

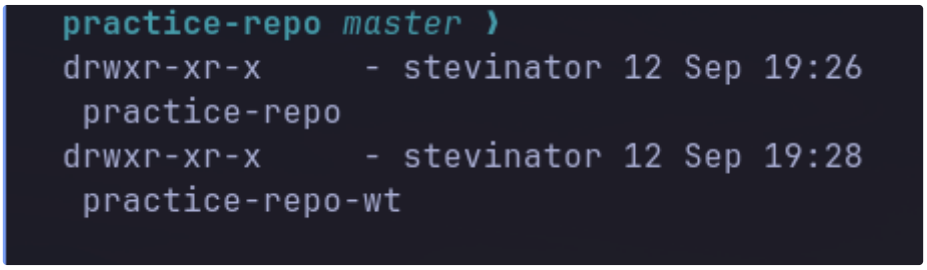
With worktrees:

```
folder A
→ branch A

folder B
→ branch B
```

You can keep both open simultaneously.

This is excellent for:

- parallel development;
- code review;
- comparing branches;
- AI agents working independently.



Screenshot 7.51-A
two worktrees side by side in the filesystem

7.52 — Standard Git Worktree Commands

List worktrees:

```
git worktree list
```

Create a worktree with a new branch:

```
git worktree add -b feature/example ../project-feature-example
```

Remove one:

```
git worktree remove ../project-feature-example
```

The official Git documentation recommends using `git worktree remove` rather than simply deleting a linked worktree directory manually.

Screenshot 7.52-A
`git worktree list`

7.53 — Omarchy Worktree Helpers

Current official Omarchy shell-function documentation provides:

```
ga [branch]
```

and:

```
gd
```

Current documentation describes:

```
ga [branch]
→ create a new branch/worktree next to the current repository and enter it

gd
→ remove the current worktree and its branch after confirmation
```

These helpers are designed to make parallel development fast.

7.54 — Inspect the Helpers Before Using Them

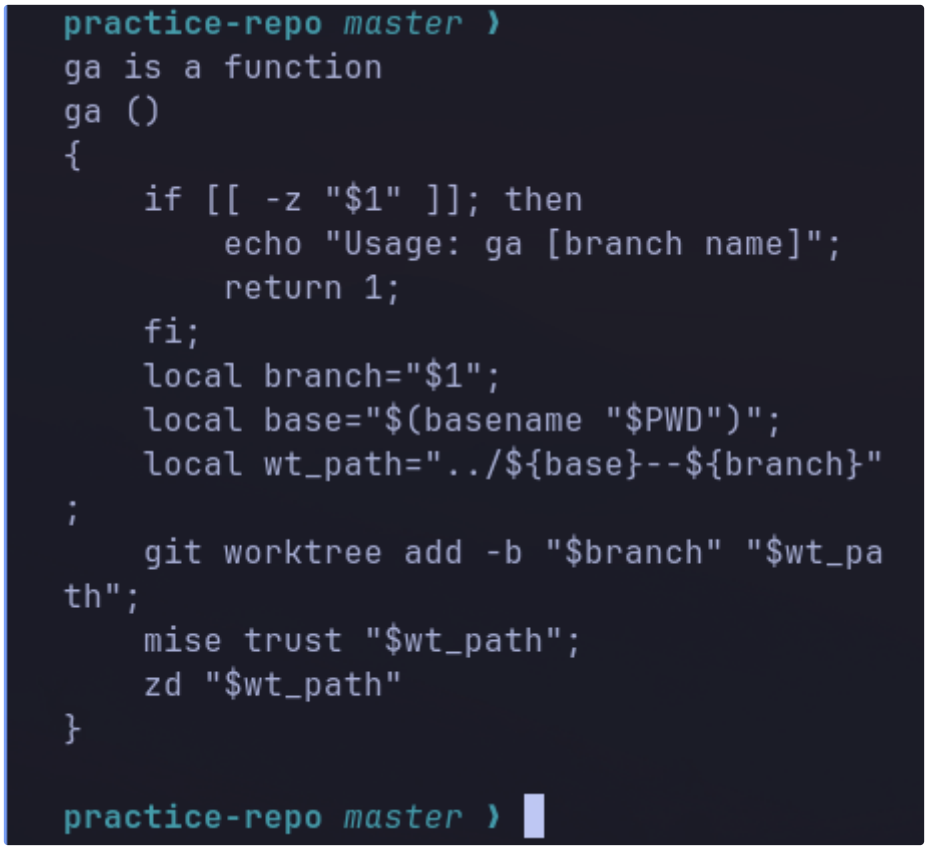
Run:

```
type ga
```

and:

```
type gd
```

Your installed Omarchy version is the final source of truth for the actual helper implementation.



Screenshot 7.54-A

`type ga`

Because Omarchy evolves quickly, do not rely solely on an old blog post describing helper behavior.

7.55 — Why Worktrees Matter for AI

Imagine three independent tasks:

```
Task A
→ build authentication feature

Task B
→ fix API bug

Task C
→ improve tests
```

A parallel AI workflow can use:

```
main worktree
→ your stable baseline

worktree A
→ agent A

worktree B
→ agent B

worktree C
→ agent C
```

This avoids multiple agents changing the exact same working directory.

That is much safer than simply opening three coding-agent panes in one repository and telling all of them to edit files.

7.56 — Do Not Use Worktrees Just Because They Exist

For one small feature:

```
normal branch
```

may be enough.

For one simple solo project:

```
main + occasional feature branch
```

may be enough.

Worktrees become valuable when:

- several branches need to remain open;
- you are reviewing another branch while coding;
- parallel AI agents need filesystem isolation.

Use complexity only when it solves a real problem.

7.57 — Lazygit

Current official Omarchy documentation includes **Lazygit**, a terminal user interface for Git.

Run inside a Git repository:

```
lazygit
```

The current Omarchy manual describes:

```
Tab
```

for moving between panes,

```
Space
```

for staging files from the Files pane,

and:

```
c
```

for creating a commit.

Use:

```
?
```

inside Lazygit to see current commands.

Screenshot 7.57-A
Lazygit main screen

7.58 — Why Lazygit Is Useful

Git commands are essential because they teach the underlying model.

Lazygit is useful once you understand that model.

It provides a visual view of:

- files;

- staged changes;
- commits;
- branches;
- remotes;
- diffs.

COURSE RECOMMENDATION

Learn the core Git CLI first.

Then use Lazygit when its visual interface makes the task faster.

Do not use a TUI to avoid understanding what staging and commits mean.

7.59 — Lazygit and Neovim

Current Omarchy documentation also provides Lazygit integration inside Neovim.

The current TUI documentation identifies:

```
Space G G
```

inside Neovim as the Lazygit launcher.

We will keep editor-specific details in the editor module.

For now:

```
lazygit
```

is enough.

7.60 — GitHub Repository Visibility

GitHub repositories may be:

- public;
- private;
- internal for eligible organizations.

For personal proprietary work or private workstation configuration, use:

```
private
```

when appropriate.

Current GitHub CLI syntax:

```
gh repo create NAME --private
```

7.61 — Private Does Not Mean "Safe for Secrets"

A private repository is access-controlled.

But credentials should still not be committed.

Reasons include:

- accidental sharing;
- collaborator access;
- compromised account;
- repository visibility changes;
- secrets preserved in commit history.

Use proper secret management.

7.62 — Creating a GitHub Repository Interactively

You can also run:

```
gh repo create
```

with no repository name.

Current GitHub CLI documentation provides an interactive creation flow.

This is useful for beginners.

Once you know exactly what you want, the non-interactive command is faster.

7.63 — Existing Folder → Private GitHub Repo: Recommended Workflow

This is the workflow you recently used.

Enter the folder:

```
cd ~/Projects/YOUR_PROJECT
```

Initialize Git if necessary:

```
git init
```

Inspect:

```
git status
```

Create a `.gitignore` if appropriate.

Stage:

```
git add .
```

Inspect staged changes:

```
git diff --staged
```

Commit:

```
git commit -m "chore: initial commit"
```

Create the GitHub repo and push:

```
gh repo create YOUR_REPO_NAME --private --source=. --remote=origin --push
```

Verify:

```
git remote -v
```

Then:

```
gh repo view --web
```

This is one of the most useful repeatable workflows in the handbook.

7.64 — Practical Exercise: Build a Repository from Scratch

Create:

```
mkdir -p ~/Projects/git-module-exercise
cd ~/Projects/git-module-exercise
```

Initialize:

```
git init
```

Create files:

```
echo "# Git Module Exercise" > README.md
echo "console.log('hello');" > app.js
```

Inspect:

```
git status
```

Stage only README:

```
git add README.md
```

Inspect:

```
git status --short
```

You should see one staged file and one untracked file.

7.65 — Practical Exercise: Two Focused Commits

Commit the README:

```
git commit -m "docs: add README"
```

Now stage:

```
git add app.js
```

Commit:

```
git commit -m "feat: add starter application"
```

Inspect:

```
git log --oneline
```

You should now see two understandable commits instead of one vague "everything" commit.

7.66 — Practical Exercise: Create a Private Remote

Ensure GitHub authentication:

```
gh auth status
```

Then:

```
gh repo create git-module-exercise --private --source=. --remote=origin --push
```

Verify:

```
git remote -v
```

Open:

```
gh repo view --web
```

7.67 — Practical Exercise: Daily Workflow

Modify:

```
echo "Second line" >> README.md
```

Inspect:

```
git status
```

Then:

```
git diff
```

Stage:

```
git add README.md
```

Inspect:

```
git diff --staged
```

Commit:

```
git commit -m "docs: expand README"
```

Push:

```
git push
```

7.68 — Practical Exercise: Create a Branch

Run:

```
git switch -c feature/example
```

Verify:

```
git branch --show-current
```

Modify:

```
echo "Feature branch" >> app.js
```

Commit:

```
git add app.js
git commit -m "feat: add example branch change"
```

Switch back:

```
git switch main
```

Inspect:

```
git log --oneline --all --decorate --graph
```

This provides a visual terminal representation of the branch history.

7.69 — Practical Exercise: Create a Worktree

From the main repository:

```
git worktree add -b feature/worktree-example ../git-module-exercise-worktree
```

List:

```
git worktree list
```

Enter:

```
cd ../git-module-exercise-worktree
```

Check:

```
git branch --show-current
```

You now have another branch checked out in another folder at the same time.

7.70 — Practical Exercise: Remove the Worktree

Return to the main worktree:

```
cd ../git-module-exercise
```

Remove:

```
git worktree remove ../git-module-exercise-worktree
```

Delete the branch if no longer needed:

```
git branch -d feature/worktree-example
```

Only delete a branch when you are sure you no longer need it.

7.71 — Practical Exercise: Inspect Omarchy Worktree Helpers

Run:

```
type ga
```

Then:

```
type gd
```

If they exist, inspect what your current Omarchy version provides.

You do not need to use them yet.

The goal is simply to understand that Omarchy includes worktree ergonomics for developer workflows.

7.72 — Practical Exercise: Lazygit

Inside the exercise repository:

```
lazygit
```

Explore:

- Files;
- Local branches;
- Commits;
- Remotes.

Press:

```
?
```

for help.

Exit when finished.

The purpose is to connect the Git concepts you learned to a visual terminal interface.

7.73 — Common Git Mistakes

Mistake 1 — Running `git add .` Without Checking Status

Always inspect:

```
git status
```

first.

You may accidentally stage:

- secrets;
- logs;
- generated files;
- huge directories.

Mistake 2 — Committing `.env`

Put sensitive local environment files in:

```
.gitignore
```

before the first commit.

Mistake 3 — Using One Giant Commit

Prefer understandable units of work.

A commit should tell a story.

Mistake 4 — Using `git reset --hard` as a Universal Fix

Understand which Git state you want to change first.

Mistake 5 — Assuming "Private Repo" Means Secrets Are Fine

Private repositories are not secret stores.

Mistake 6 — Pulling with Uncommitted Changes and No Idea What Will Happen

Run:

```
git status
```

before integrating remote changes.

Mistake 7 — Letting an AI Agent Commit Unrelated Files

Always inspect:

```
git status
git diff
git diff --staged
```

before accepting a commit.

Mistake 8 — Using Multiple AI Agents in One Working Tree

For genuinely parallel edits, prefer isolated branches/worktrees.

Mistake 9 — Deleting a Worktree Folder Manually

Prefer:

```
git worktree remove PATH
```

so Git cleans up its linked-worktree metadata properly.

Mistake 10 — Treating GitHub as the Source of Truth Instead of Git

GitHub hosts the repository.

Git is still the version-control system.

Understand local state first.

7.74 — AI-Assisted Git Workflow

A strong coding-agent workflow looks like:

```
clean Git status
        ↓
create branch/worktree if appropriate
        ↓
agent inspects project
        ↓
agent makes focused changes
        ↓
run tests
        ↓
git diff
        ↓
human/agent review
        ↓
stage
        ↓
git diff --staged
        ↓
commit
        ↓
push
```

Git provides the safety boundary around AI-generated changes.

7.75 — Why Git Is Essential for AI Development

AI agents can change files quickly.

That is powerful.

It also increases the value of:

- clean history;
- small commits;
- branches;
- diffs;
- worktrees;
- review.

Without Git:

```
agent changed 14 files
→ what exactly happened?
```

With Git:

```
git diff
```

you can inspect the precise change set.

Git is therefore not just source control in an AI workstation.

It is part of the control and review system.

7.76 — Recommended Commit Boundaries for AI Work

Try to keep one commit focused on one objective.

Examples:

```
feat: add login screen
```

```
fix: handle expired auth tokens
```

```
test: add router regression coverage
```

Avoid:

```
feat: auth + refactor database + update docs + formatting everywhere
```

Smaller commits are easier to:

- review;
- revert;
- cherry-pick;
- debug;
- compare.

7.77 — Checkpoint

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

Git Fundamentals

- [ ] I understand what a Git repository is.
- [ ] I understand working tree, staging area, commit history, and remote.
- [ ] I know the difference between Git and GitHub.
- [ ] I can initialize a repository.
- [ ] I can use `git status`.
- [ ] I can use `git diff`.
- [ ] I can stage files.
- [ ] I can create a commit.
- [ ] I can inspect history.

GitHub

- [ ] I can authenticate with `gh auth login`.
- [ ] I can verify authentication with `gh auth status`.
- [ ] I can create a private GitHub repository from an existing local folder.
- [ ] I can inspect remotes.
- [ ] I can push commits.
- [ ] I can open the repository in the browser.

Repository Hygiene

- [ ] I understand `.gitignore`.

- [ ] I know not to commit secrets.
- [ ] I inspect staged changes before committing.
- [ ] I understand why focused commits are useful.

Branches

- [ ] I understand what a branch represents.
- [ ] I can create and switch branches.
- [ ] I understand merge conflicts at a high level.

Worktrees

- [ ] I understand what a Git worktree is.
- [ ] I understand why worktrees are useful for parallel development.
- [ ] I can list worktrees.
- [ ] I understand the Omarchy `ga` and `gd` helpers conceptually.
- [ ] I understand why worktrees are especially useful for parallel AI agents.

Tools

- [ ] I can open Lazygit.
- [ ] I understand that Lazygit is an interface over Git, not a replacement for understanding Git.

7.78 — What You Can Now Do

After completing Module 7, you can now:

- create version-controlled software projects;
- inspect project state confidently;
- create meaningful commits;
- push projects to GitHub;
- create private repositories from the terminal;
- clone and update repositories;
- protect secrets with `.gitignore` and sane practices;
- create feature branches;
- understand worktrees;
- use Omarchy's Git ergonomics;
- use Lazygit for visual Git workflows;
- use Git as a safety and review layer around AI-generated code.

Most importantly:

> *Your code is no longer just a collection of files. It has history, review boundaries, recovery points, and a reproducible remote home.*

Module 7 Summary

Core Git model:

```
Working tree
    ↓ git add

Staging area
    ↓ git commit

Local history
    ↓ git push

GitHub remote
```

Daily workflow:

```
git status
git diff
git add ...
git diff --staged
git commit -m "..."
git push
```

Create a repository:

```
git init
```

Create a private GitHub repository from the current local repository:

```
gh repo create YOUR_REPO_NAME --private --source=. --remote=origin --push
```

Verify:

```
git remote -v
gh repo view --web
```

Branch:

```
git switch -c feature/name
```

Switch:

```
git switch main
```

Worktrees:

```
git worktree list
git worktree add -b feature/name ../project-feature-name
git worktree remove ../project-feature-name
```

Omarchy helpers:

```
ga [branch]
→ create branch + worktree and enter it

gd
→ remove current worktree + branch after confirmation
```

Visual Git:

```
lazygit
```

And the most important AI-development lesson:

> ***Never give an AI agent a blank check over your working tree. Use Git status, diffs, commits, branches, and worktrees to keep changes inspectable and reversible.***

MODULE 08

08

Editor Integration on Omarchy

You now have:

- a working Omarchy workstation;
- terminal fluency;
- Tmux;
- reproducible runtimes with mise;
- Git and GitHub.

About This Module

You now have:

- a working Omarchy workstation;
- terminal fluency;
- Tmux;
- reproducible runtimes with mise;
- Git and GitHub.

Now we add the editor layer.

Current Omarchy ships with **Neovim** as its default editor and provides a complete Neovim setup through the `omarchy-nvim` package. The current official Omarchy documentation states that this setup is built on **LazyVim**, so you get a powerful development environment without building a Neovim configuration from scratch.

At the same time, Omarchy supports alternative editors.

Current official Omarchy development documentation lists:

- VS Code;
- Cursor;
- Zed;
- Sublime Text;
- Helix;
- Vim;
- Emacs;

through the Omarchy editor-install workflow.

The goal of this module is not to turn you into a Neovim expert.

The goal is:

```
open project
→ find file
→ search code
→ edit safely
→ save
→ recover from common mistakes
→ return to Git / AI workflow
```

That is enough to make the editor a reliable part of the workstation.

What You Will Learn

By the end of Module 8, you will be able to:

- understand how Omarchy selects the default editor;
- launch the configured editor;
- understand why Neovim is Omarchy's default;
- understand the difference between Neovim and LazyVim;
- open files and directories;
- understand Normal, Insert, and Command mode;
- save and quit;
- fuzzy-find files;
- search project contents;
- use the file sidebar;
- move between editor windows;
- use Git controls from the editor;
- understand safe configuration editing;
- understand Neovim swap-file recovery;
- choose a different editor if Neovim is not for you;
- keep editor configuration upgrade-safe.

8.1 — Omarchy's Default Editor

Current official Omarchy documentation states that Neovim is the default editor.

You can change the system-wide default editor through:

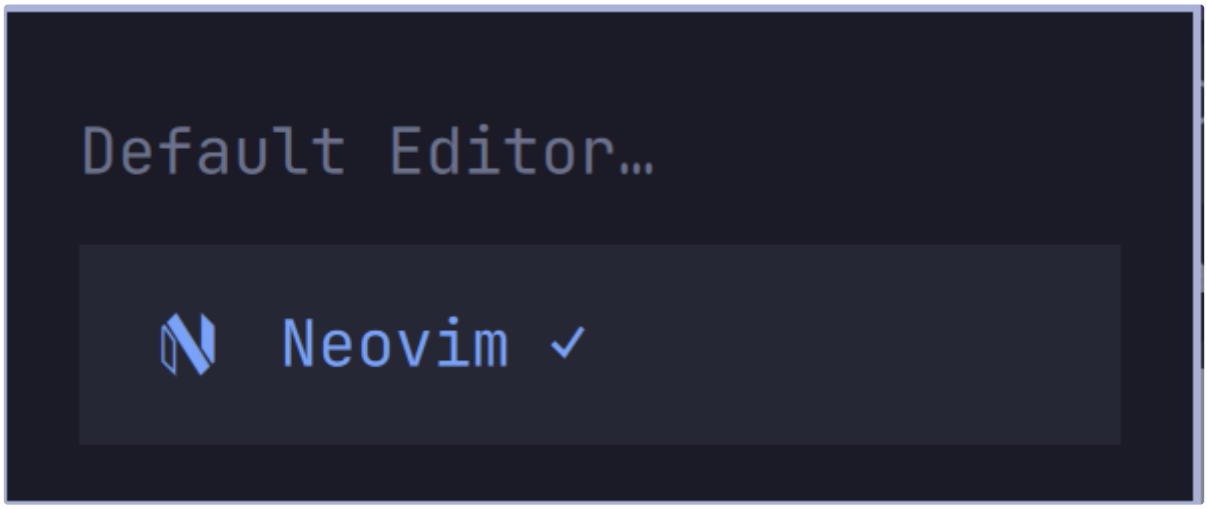
```
Super + Space
→ Setup
→ Defaults
→ Editor
```

The currently documented alternative editors include:

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

The available list can evolve.

Use the current Omarchy menu or current CLI help instead of assuming a static list forever.



Screenshot 8.1-A
Default editor menu

8.2 — Why the Omarchy Default Matters

The default editor is used by more than one shortcut.

For example, Omarchy configuration actions may open a config file using the selected default editor.

That means:

```
Setup > Monitors
Setup > Keybindings
Setup > Input
Setup > Config
```

can all respect the editor you selected.

This is better than manually hard-coding an editor in every workflow.

8.3 — Launch the Editor

Current official Omarchy hotkeys include:

```
Super + Shift + N
```

for launching the editor.

On the default setup, this opens Neovim.

Screenshot 8.3-A
Neovim launched from `Super + Shift + N`

8.4 — Launch Neovim from the Terminal

Open a file:

```
nvim README.md
```

Open a project directory:

```
cd ~/Projects/YOUR_PROJECT
nvim .
```

Opening the directory is usually the more useful development workflow.

8.5 — Neovim vs LazyVim

This distinction matters.

Neovim

Neovim is the editor itself.

LazyVim

LazyVim is a curated configuration/plugin distribution built on Neovim.

Current Omarchy documentation states that the default `omarchy-nvim` setup is built on LazyVim.

Conceptually:

```
Neovim
    ↓
LazyVim configuration/plugins
    ↓
Omarchy-tuned setup
```

So when you press:

```
Space
```

and see a command menu, that behavior is largely part of the configured LazyVim experience rather than bare Neovim alone.

Screenshot 8.5-A
Neovim/LazyVim home screen with a project open

8.6 — Modal Editing

Neovim is a modal editor.

The most important modes for this course are:

```
Normal mode
Insert mode
Command-line mode
```

You do not need to learn every Vim mode.

8.7 — Normal Mode

Normal mode is the main navigation/command mode.

When Neovim starts, you are normally in Normal mode.

In this mode:

```
keys perform editor actions
```

rather than directly entering text.

This is the biggest conceptual difference from mainstream editors.

8.8 — Insert Mode

Press:

```
i
```

to enter Insert mode.

Now typing inserts text normally.

Return to Normal mode with:

```
Esc
```

A core survival loop is:

```
i
→ type

Esc
→ stop typing / return to commands
```

8.9 — Save

From Normal mode:

```
:w
```

then:

```
Enter
```

This means:

```
write the file
```

8.10 — Quit

From Normal mode:

```
:q
```

then:

```
Enter
```

8.11 — Save and Quit

Use:

```
:wq
```

This is especially useful because Omarchy's own dotfile documentation explicitly references `:wq` when editing configuration through the Omarchy menu.

8.12 — Quit Without Saving

Use:

```
:q!
```

This discards unsaved changes.

Use it intentionally.

8.13 — Undo

In Normal mode:

```
u
```

undoes the previous change.

Redo:

```
Ctrl + R
```

8.14 — Basic Movement

In Normal mode, traditional Vim movement keys are:

```
h
→ left

j
→ down

k
→ up

l
→ right
```

You can also use arrow keys.

For this course, using arrow keys initially is completely acceptable.

Learn Vim movement gradually.

8.15 — The Leader Key

Current Omarchy/LazyVim documentation uses:

```
Space
```

as the leader key.

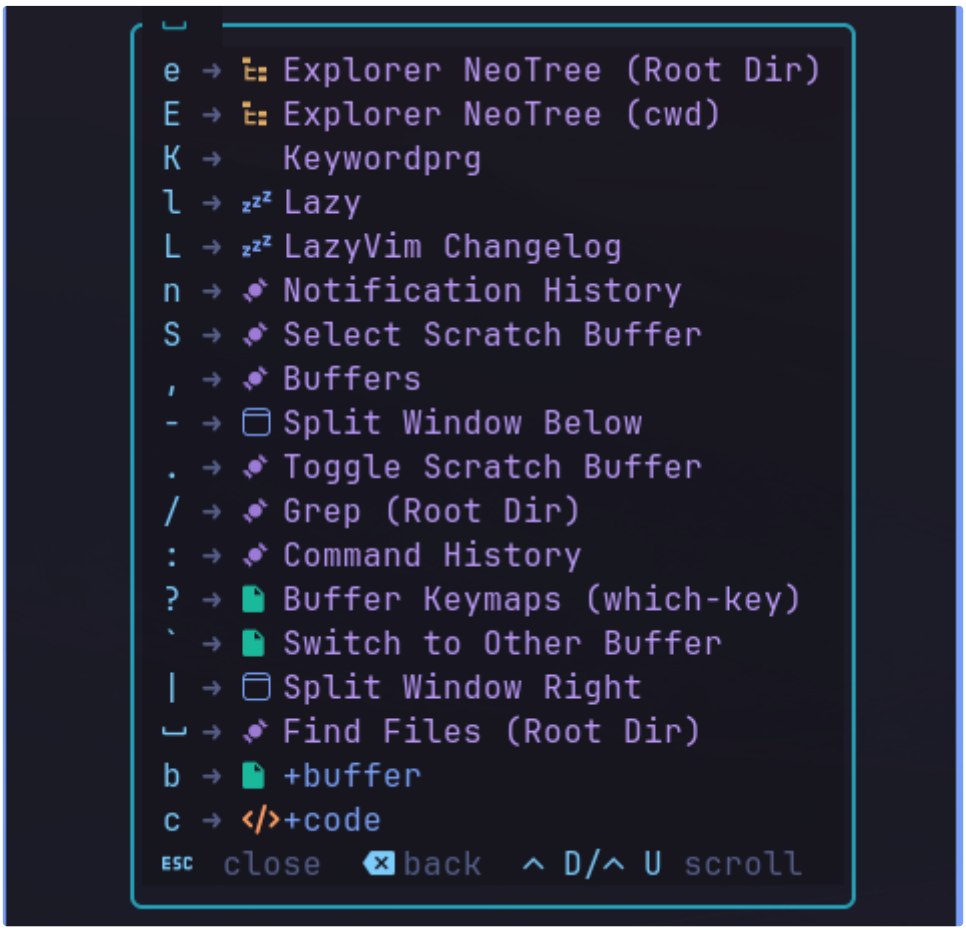
Press:

```
Space
```

and wait briefly.

You should see available command groups.

This is one of the most useful discovery features in the editor.



Screenshot 8.15-A
`Space` leader menu

COURSE RULE

> ***Do not memorize the whole LazyVim command map. Press `Space` and discover.***

8.16 — Fuzzy Find Files

Current official Omarchy Neovim documentation identifies:

```
Space Space
```

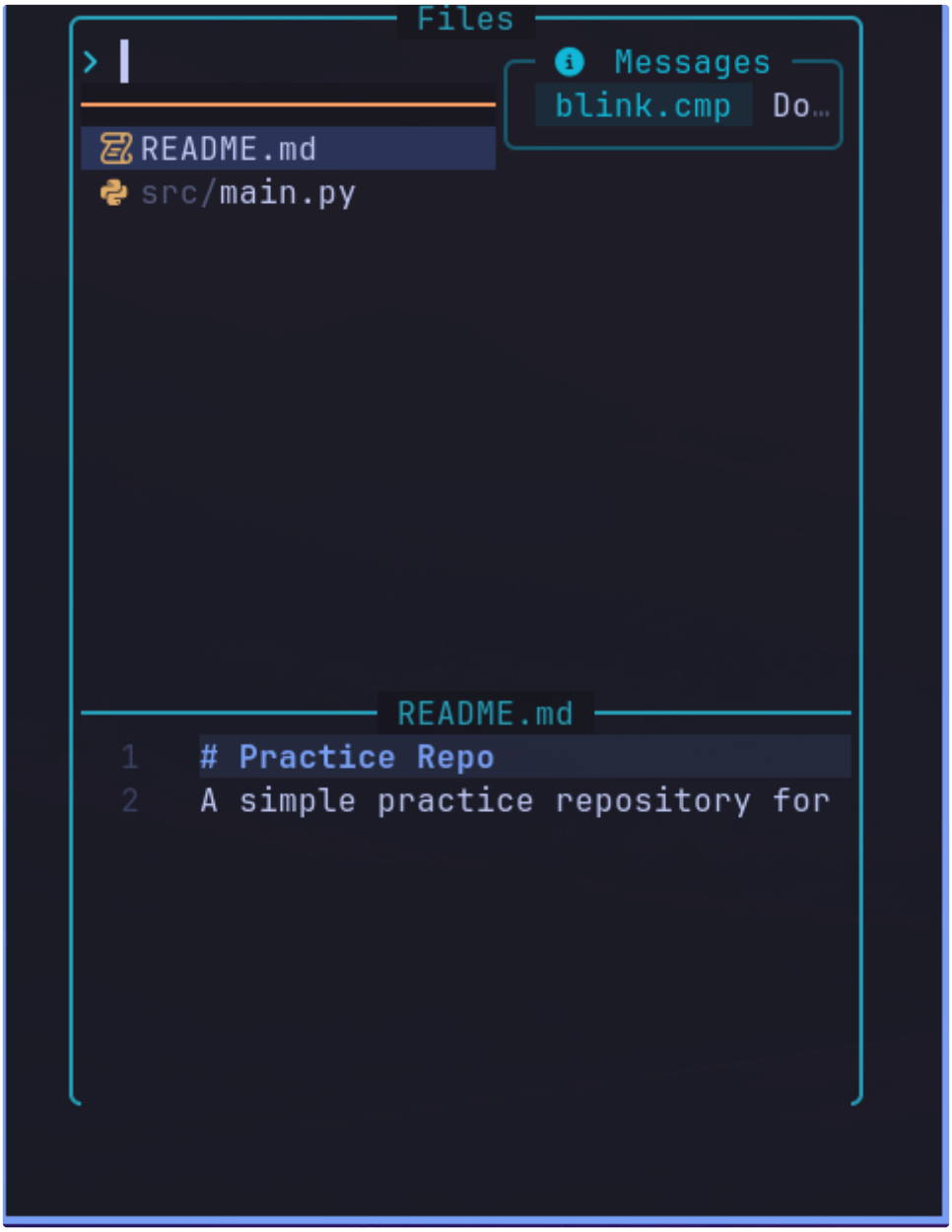
as the shortcut to fuzzy-find a file in the current project.

This is one of the most valuable editor shortcuts.

Workflow:

```
Space Space
→ type part of filename
→ select
→ Enter
```

You no longer need to browse directory trees manually.



Screenshot 8.16-A
`Space Space` fuzzy file finder

8.17 — Search Project Contents

Current official Omarchy documentation identifies:

```
Space S G
```

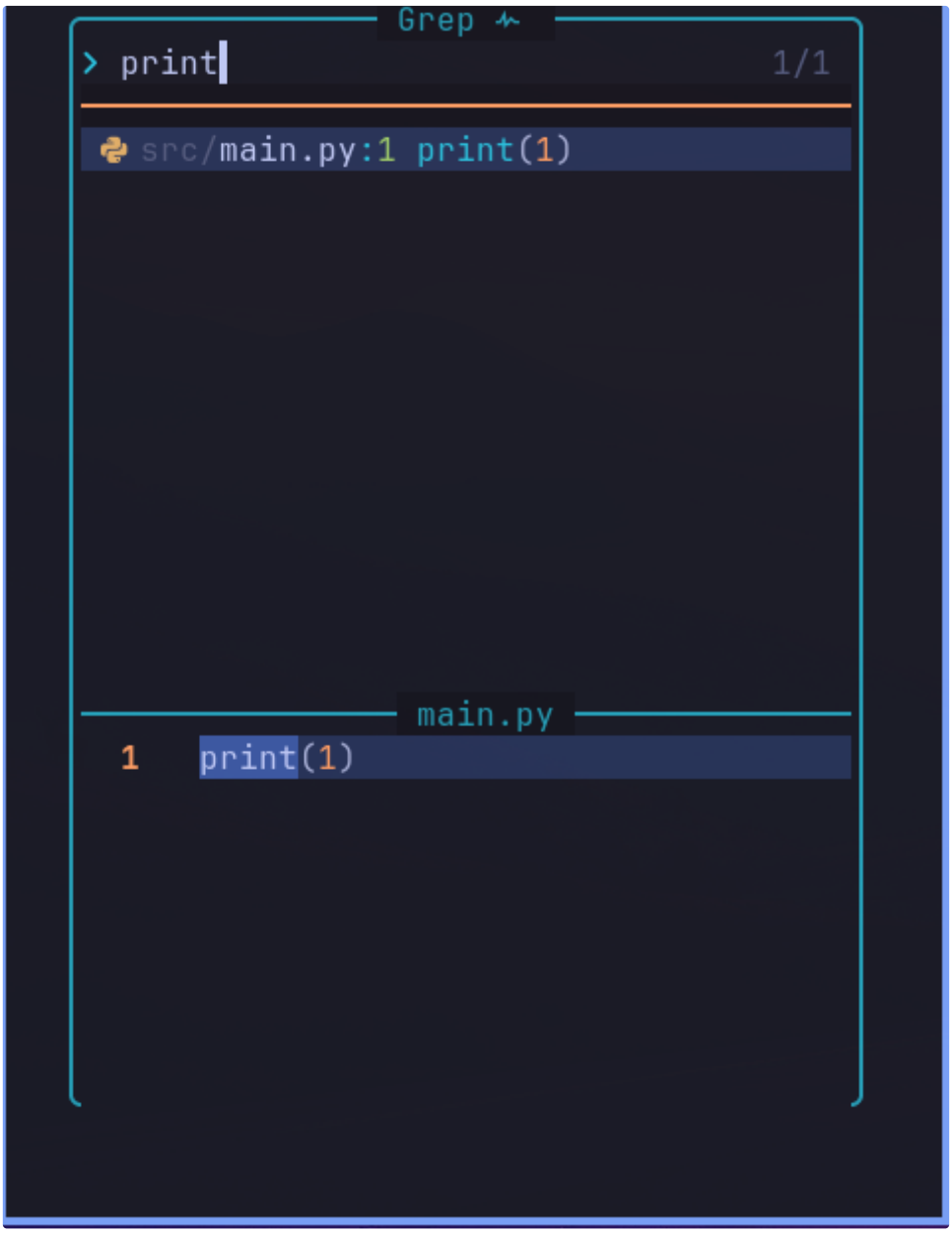
for searching file contents across the project.

Conceptually:

```
ripgrep-style project search
+
interactive preview
```

This is ideal for:

- finding functions;
- finding TODOs;
- finding config references;
- tracing API names.



Screenshot 8.17-A

`Space S G` project grep

8.18 — File Sidebar

Current official hotkey documentation identifies:

```
Space E
```

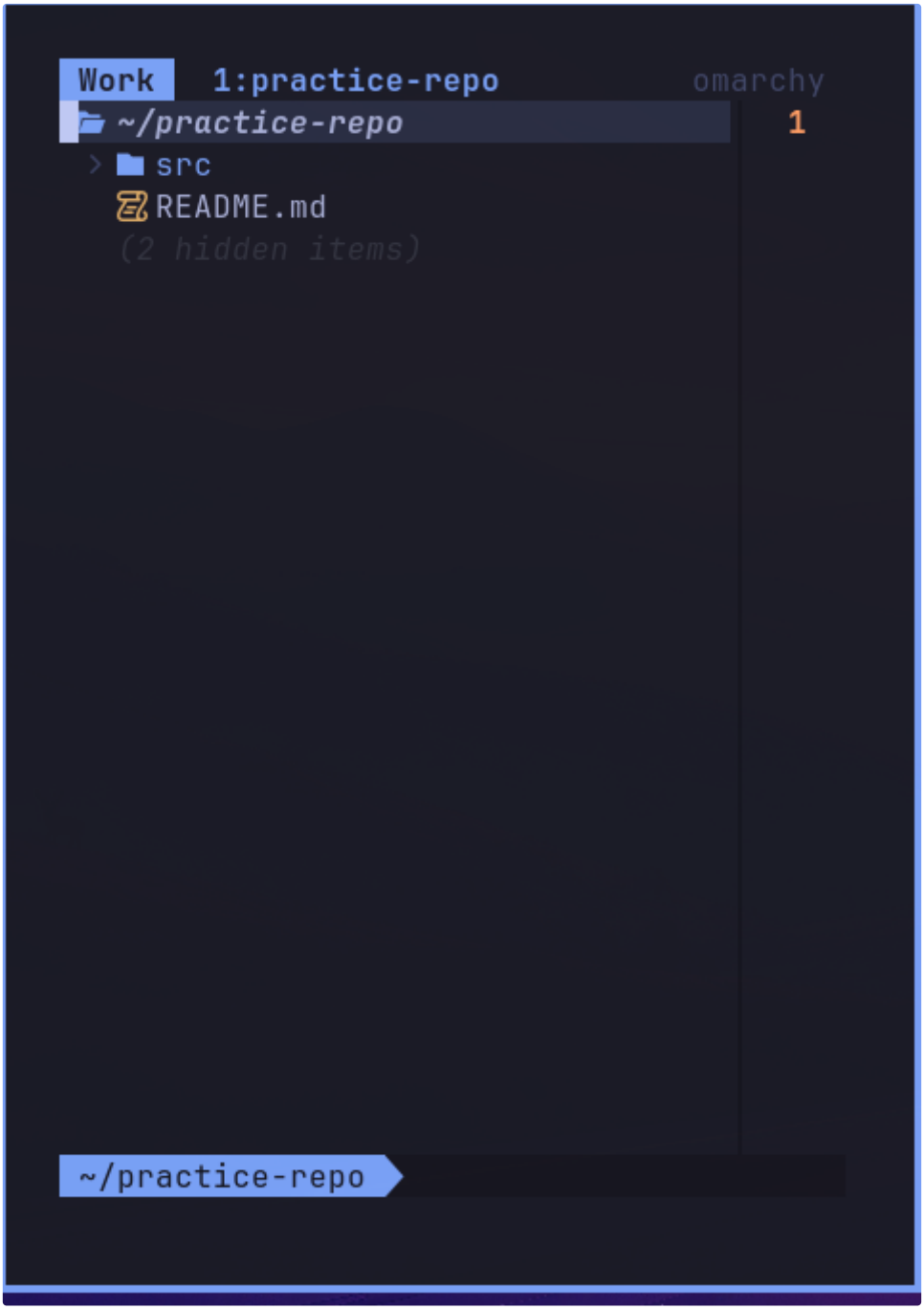
for toggling the file sidebar.

Use this when you want a visual project tree.

Do not feel obligated to keep it open permanently.

A strong workflow is:

```
Space Space
→ fast file navigation

Space E
→ structural overview
```



Screenshot 8.18-A

Space E sidebar

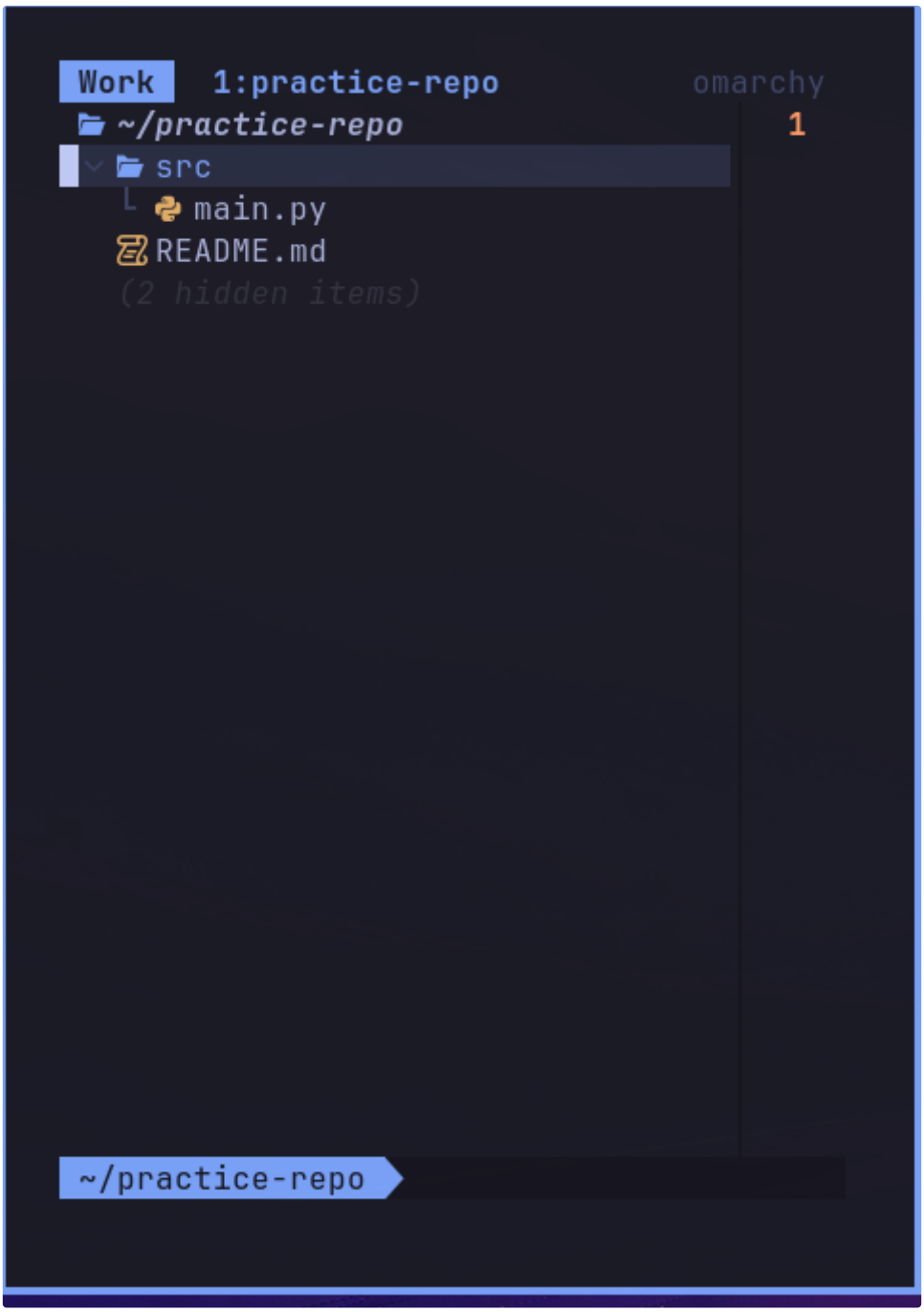
8.19 — Move Between Sidebar and Editor

Current official Omarchy hotkey documentation identifies:

```
Ctrl + W W
```

for jumping between sidebar and editor windows.

This is useful when using the file tree.



Screenshot 8.19-A
sidebar/editor navigation

8.20 — Resize the Sidebar

Current official Omarchy hotkey documentation identifies:

```
Ctrl + Left
Ctrl + Right
```

for changing the sidebar width.

Exact behavior is tied to the current configured Neovim/LazyVim setup, so use the current in-editor command hints if a future release differs.

8.21 — Git Controls

Current official Omarchy hotkey documentation identifies:

```
Space G G
```

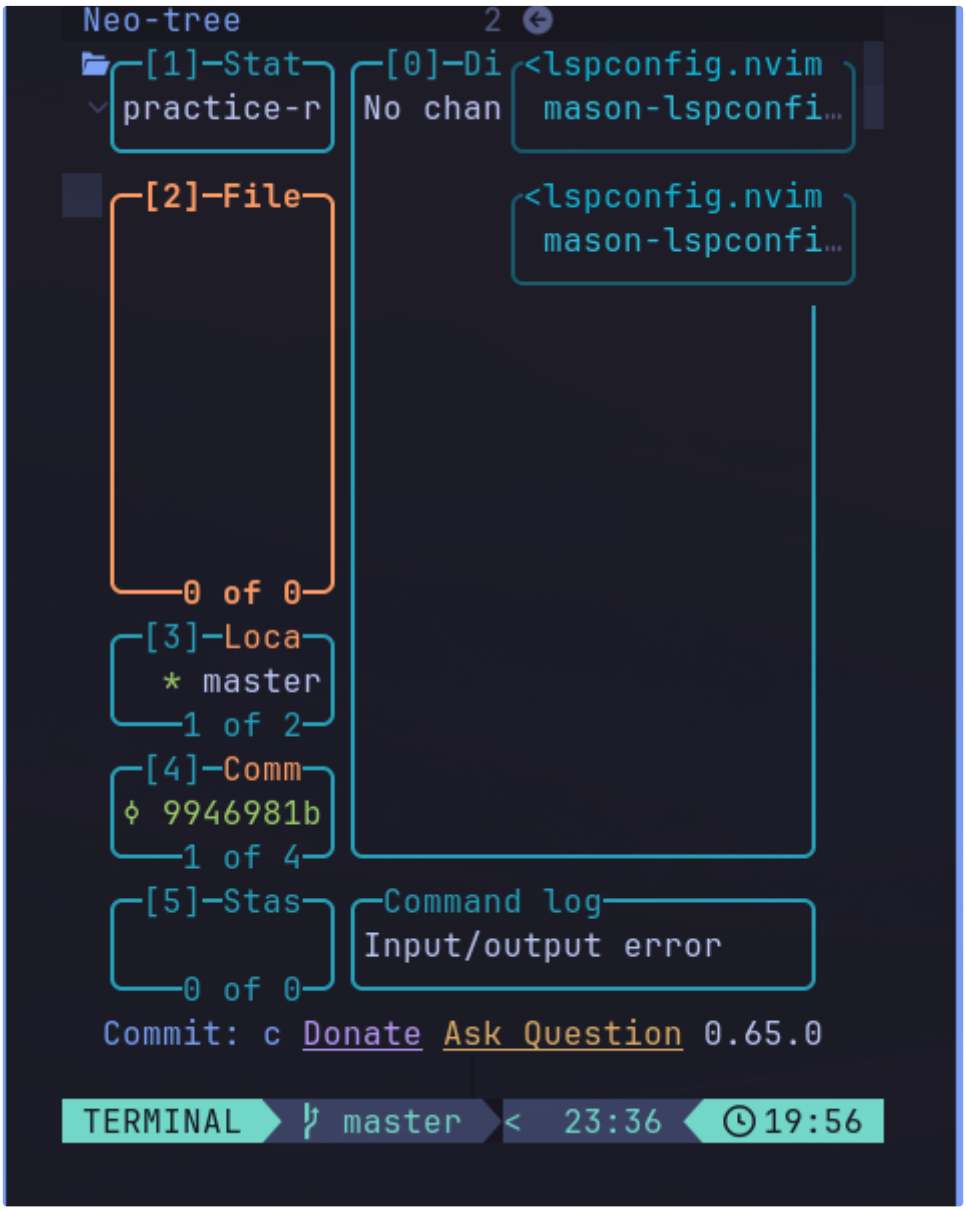
for Git controls.

This opens Lazygit integration from inside Neovim.

You already learned Lazygit in Module 7.

This creates a useful workflow:

```
edit
↓
Space G G
↓
inspect diff
↓
stage
↓
commit
↓
return to editor
```



Screenshot 8.21-A

`Space G G` Lazygit integration

8.22 — Editor + Git + AI Workflow

A practical development loop looks like:

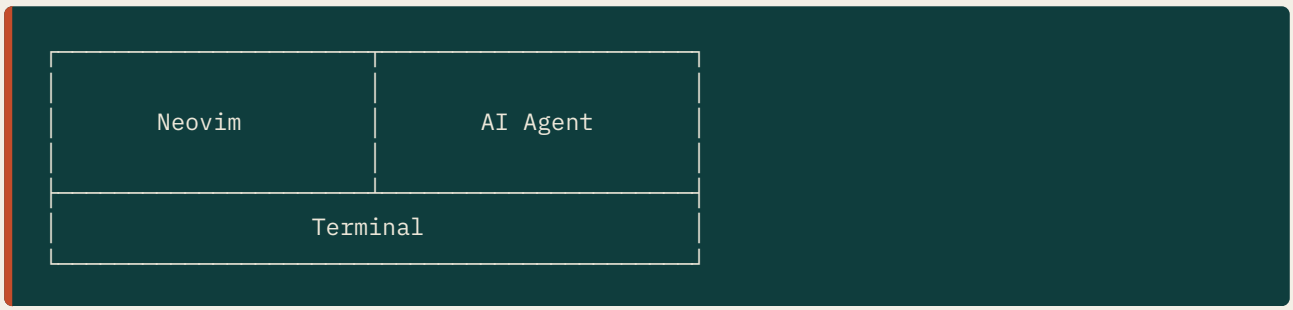
```
Editor
→ inspect/change code

AI agent
→ analyze or implement

Git
→ review exact diff

Tests
→ verify behavior

Editor
→ refine
```

The editor is one part of the system, not the entire system.

This matters because our workstation is deliberately multi-tool.

8.23 — Use the Editor Inside Tmux

A common development layout:



This is exactly the kind of layout Omarchy's current `tdl` helper is designed to support.

The editor, agent, and shell remain visible at the same time.

Screenshot 8.23-A
Neovim inside a Tmux `tdl` layout (editor + agent + terminal)

8.24 — Open the Correct Project

Before launching the editor:

```
cd ~/Projects/YOUR_PROJECT
```

Then:

```
nvim .
```

This matters because:

- fuzzy file search;
- content search;
- Git integration;
- AI context;

all depend heavily on the current working directory/project root.

8.25 — Safe Configuration Editing

Omarchy is heavily configured through files under:

```
~/.config
```

Examples:

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

Current official Omarchy dotfile documentation recommends editing user-owned configuration under `~/.config`, not package-owned files under `/usr/share/omarchy`.

This same rule applies whether the editor is Neovim, Helix, VS Code, or another editor.

8.26 — Open an Omarchy Config

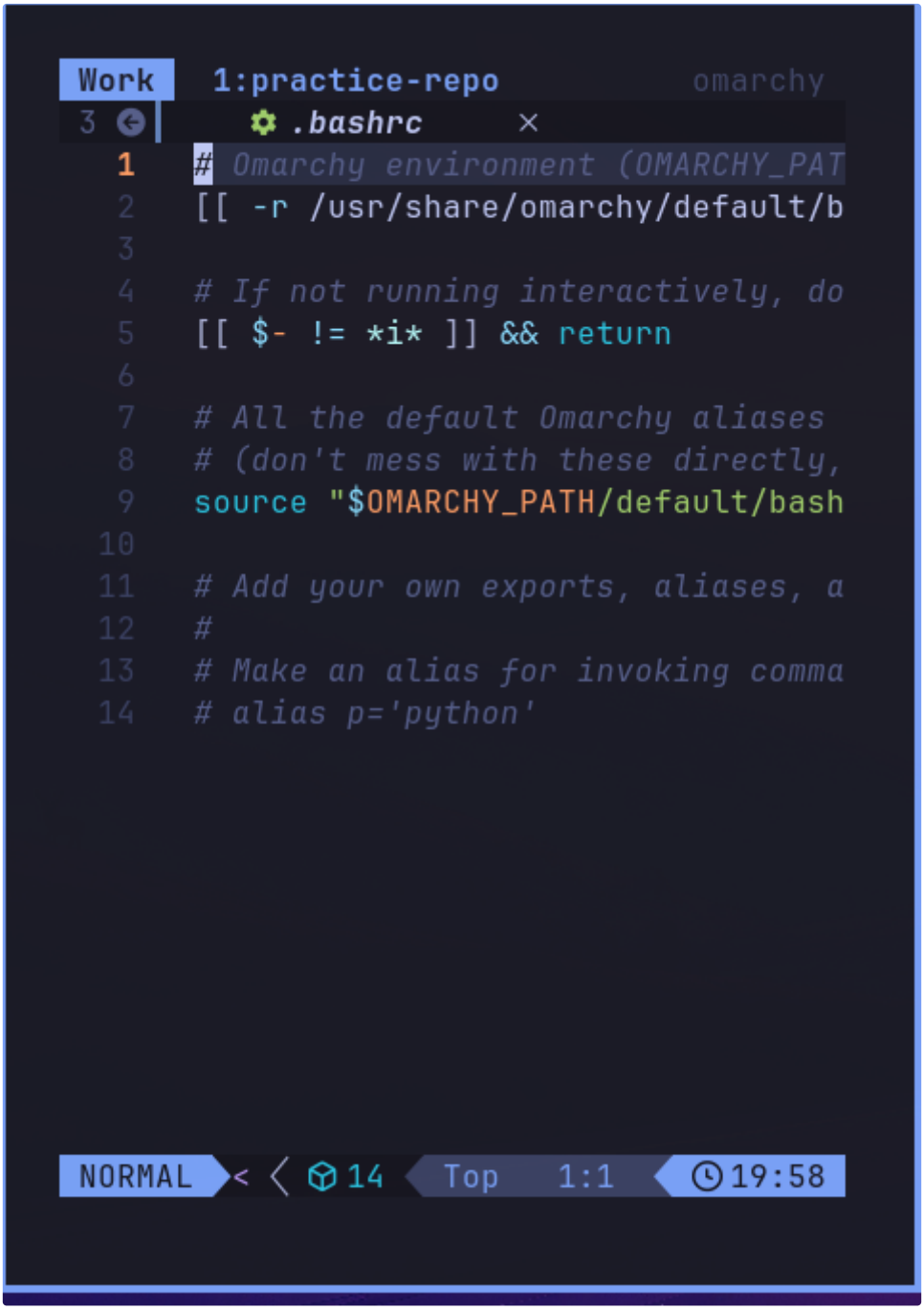
Example:

```
nvim ~/.config/hypr/bindings.lua
```

Before changing anything:

1. read the file;
2. understand the existing structure;
3. change one thing;
4. save;
5. verify behavior.

Do not rewrite entire config files because a tutorial showed a full replacement.



Screenshot 8.26-A
`~/.config` file opened safely (e.g. `~/.bashrc`)

8.27 — Prefer Small Changes

Good:

```
add one binding
```

Bad:

```
replace entire bindings.lua
with random configuration from GitHub
```

Small changes are easier to:

- understand;
- reverse;
- commit;
- troubleshoot;
- carry forward across Omarchy updates.

8.28 — Omarchy Config Editing Through the Menu

Current official dotfile documentation states that key configs can be opened directly through the Omarchy Menu.

Examples include:

```
Super + Space
→ Setup
→ Monitors
```

or:

```
Setup
→ Keybindings
```

or:

```
Setup
→ Input
```

When opened through Omarchy's own configuration workflow, processes that require a restart/reload after editing can be handled automatically once the editor exits.

This can be safer than manually guessing which component needs restarting.

8.29 — Do Not Use `sudo` for Personal Config

Do not normally run:

```
sudo nvim ~/.config/hypr/bindings.lua
```

Your personal config belongs to your normal user.

If you need `sudo` to edit your own files, investigate ownership before forcing the issue.

8.30 — Neovim Swap Files

Neovim may create a swap file while a file is open.

This helps recover unsaved work after:

- crash;
- killed terminal;
- machine restart;
- editor failure.

Sometimes, when opening a file, Neovim may warn that a swap file already exists.

This does **not** automatically mean the file is corrupted.

Screenshot 8.30-A
swap-file warning example

8.31 — Why Swap Warnings Appear

Common reasons:

```
another Neovim instance still has the file open
```

or:

```
a previous Neovim session crashed
```

The warning exists to stop you from accidentally overwriting unsaved changes.

8.32 — Safe Swap Recovery Workflow

If Neovim reports an existing swap file:

1. read the warning;
2. check whether the file is already open elsewhere;
3. if another editor instance is active, return to it;
4. if the previous session crashed, use Neovim's recovery option;
5. compare recovered content;
6. save only when you understand which version should win;
7. remove stale swap state only after recovery is complete.

Do not automatically delete swap files because the warning is annoying.

8.33 — Recover From a Swap File

A standard Neovim recovery command can be:

```
nvim -r FILE
```

Example:

```
nvim -r ~/.config/hypr/bindings.lua
```

The exact recovery prompt may also offer actions directly when opening the affected file.

Use the current Neovim prompt as the primary guide.

8.34 — If You Do Not Want Neovim

That is completely valid.

Current Omarchy documentation explicitly supports alternative editors.

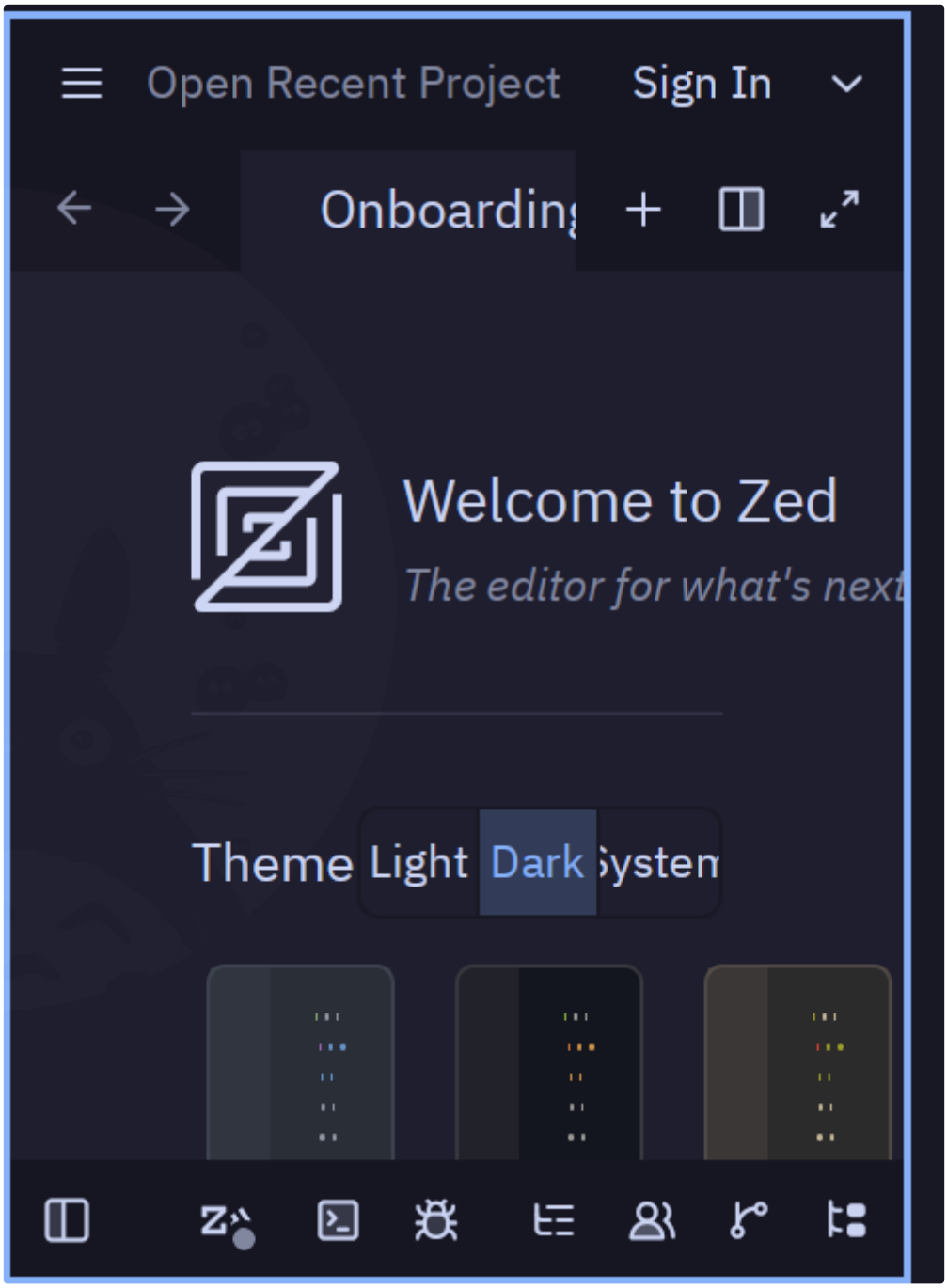
Use:

```
Super + Space
→ Install
→ Editor
```

Then set the default under:

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

The best editor is the one that lets you work effectively.



Screenshot 8.34-A
an alternative editor (Zed) installed and running

8.35 — Recommended Editor Strategy

For this course:

```
Learn enough Neovim to survive
```

because:

- it is the Omarchy default;
- it is integrated deeply;
- it works entirely in the terminal;
- it fits Tmux perfectly;
- it is always useful for quick config editing.

But:

> ***You do not need to become a Vim specialist to use Omarchy successfully.***

If your productive editor is Cursor, Zed, VS Code, Helix, or another supported option, use it.

8.36 — Theme Integration

Current Omarchy documentation states that theme matching is offered for editors including:

- VS Code;
- Cursor;
- VSCodium;
- Helix.

Omarchy's own theme system also generates styling for Neovim.

This allows the editor to visually match the workstation.

You will explore this properly in the themes module.

8.37 — Do Not Build a Neovim Config From Scratch Yet

Current Omarchy already ships a complete `omarchy-nvim` configuration built on LazyVim.

For a beginner, immediately replacing it with a custom Neovim configuration creates unnecessary complexity.

Use the provided setup first.

Only customize when you can clearly answer:

> *What specific problem am I solving?*

8.38 — Editor Configuration vs Omarchy Configuration

Keep these concepts separate.

```
Neovim config
→ controls editor behavior

Omarchy config
→ controls workstation behavior
```

You may edit Omarchy config *using* Neovim, but that does not make the file part of Neovim configuration.

This distinction prevents a lot of confusion.

8.39 — Find Before Opening

In a large project, prefer:

```
Space Space
```

over manually drilling through nested directories.

If you know text rather than filename:

```
Space S G
```

This mirrors what you learned in the terminal:

```
fd
→ filenames

rg
→ contents
```

and now gives you an editor-integrated version of the same workflow.

8.40 — Search First, Edit Second

A strong codebase-navigation pattern is:

```
search symbol
↓
inspect results
↓
open relevant file
↓
understand context
↓
edit
```

This is much safer than guessing where functionality lives.

The same principle applies to AI agents.

8.41 — Practical Exercise: Open a Project

Run:

```
cd ~/Projects/YOUR_PROJECT
nvim .
```

Then:

```
Space Space
```

Search for:

```
README
```

Open it.

8.42 — Practical Exercise: Edit and Save

Open a harmless practice file.

Press:

```
i
```

Add one line.

Press:

```
Esc
```

Then:

```
:w
```

Verify the file from another terminal:

```
git diff
```

This connects editing to Git immediately.

8.43 — Practical Exercise: Search the Project

Inside Neovim:

```
Space S G
```

Search for:

```
TODO
```

or another known project symbol.

Open one result.

Notice how much faster this is than manually searching files.

8.44 — Practical Exercise: File Sidebar

Toggle:

```
Space E
```

Move between sidebar and editor:

```
Ctrl + W W
```

Resize:

```
Ctrl + Left/Right
```

Then close/toggle the sidebar again.

8.45 — Practical Exercise: Git Controls

Make one harmless edit.

Then:

```
Space G G
```

Inspect the change in Lazygit.

Do not commit if this is only a test.

Return to Neovim.

8.46 — Practical Exercise: Open Omarchy Configuration Safely

Open:

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

Do **not** change anything.

Identify:

- file structure;
- comments;
- existing monitor rule.

Quit:

```
:q
```

The goal is simply to become comfortable inspecting configuration.

8.47 — Practical Exercise: Default Editor Discovery

Open:

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

Inspect the currently available choices.

Do not change the editor unless you want to.

The lesson is knowing where the system-wide choice lives.

8.48 — Practical Exercise: No-Mouse Editing Survival

Complete the following:

1. open a project in Neovim;
2. fuzzy-find a file;
3. enter Insert mode;
4. add a harmless line;
5. return to Normal mode;
6. save;
7. search project contents;
8. open Git controls;
9. inspect the diff;
10. quit Neovim.

If you can do this, you have enough Neovim knowledge for the rest of the course.

8.49 — Common Editor Mistakes

Mistake 1 — Typing and Nothing Happens as Expected

You are probably in Normal mode.

Press:

```
i
```

to enter Insert mode.

Mistake 2 — Cannot Quit Neovim

Return to Normal mode:

```
Esc
```

Then:

```
:q
```

or:

```
:wq
```

Mistake 3 — `:q` Says There Are Unsaved Changes

Either save:

```
:wq
```

or intentionally discard:

```
:q!
```

Do not use `:q!` unless you want to lose those unsaved changes.

Mistake 4 — Deleting a Swap File Immediately

The swap may contain recoverable work.

Investigate first.

Mistake 5 — Editing Omarchy's Package-Owned Files

Normal personal configuration belongs under:

```
~/.config
```

not:

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

Mistake 6 — Editing Personal Config with `sudo`

That can create ownership problems.

Mistake 7 — Replacing the Entire Omarchy Neovim Setup on Day One

Use the provided setup first.

Customize intentionally later.

Mistake 8 — Thinking You Must Use Neovim

Omarchy supports alternatives.

Use the editor that keeps you productive.

Mistake 9 — Navigating Huge Projects Only Through the Sidebar

Use:

```
Space Space
```

and:

```
Space S G
```

for fast navigation.

Mistake 10 — Editing Before Understanding the Change

Search and inspect first.

Then edit.

This is especially important for system configuration and AI-generated changes.

8.50 — Checkpoint

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

Editor Integration

- [ ] I know how Omarchy selects the default editor.
- [ ] I know the current Omarchy default is Neovim.
- [ ] I know how to install/select another editor.
- [ ] I can launch the editor from Omarchy.

- [ ] I can launch Neovim from the terminal.

Neovim Survival

- [ ] I understand Normal mode.
- [ ] I understand Insert mode.
- [ ] I can return to Normal mode with `Esc`.
- [ ] I can save.
- [ ] I can quit.
- [ ] I can save and quit.
- [ ] I can undo.

Navigation

- [ ] I know the leader key is `Space` in the current LazyVim setup.
- [ ] I can fuzzy-find files with `Space Space`.
- [ ] I can search project contents with `Space S G`.
- [ ] I can toggle the sidebar with `Space E`.
- [ ] I can open Git controls with `Space G G`.

Safety

- [ ] I know that personal Omarchy config belongs under `~/.config`.
- [ ] I do not edit personal config with `sudo`.
- [ ] I understand what a Neovim swap warning means.
- [ ] I know to recover before deleting stale swap state.

8.51 — What You Can Now Do

After completing Module 8, you can now:

- use Omarchy's default editor productively;
- survive the important Neovim modes;
- open projects correctly;
- find files without browsing manually;
- search code across the project;
- inspect Git changes from inside the editor;
- edit Omarchy configuration safely;
- recover from common Neovim swap situations;

- choose a different editor without breaking the Omarchy workflow;
- integrate the editor into your Tmux + Git + AI environment.

Most importantly:

> *The editor is now a tool inside your development system, not a separate world you need to master before you can be productive.*

Module 8 Summary

Default editor:

```
Current Omarchy default
→ Neovim

Configuration
→ omarchy-nvim
→ LazyVim-based setup
```

Launch:

```
Super + Shift + N
```

or:

```
nvim .
```

Core modes:

```
Normal mode
→ commands/navigation

i
→ Insert mode

Esc
→ Normal mode
```

Survival commands:

```
:w
→ save

:q
→ quit

:wq
→ save + quit

:q!
→ quit and discard
```

Current Omarchy/LazyVim navigation:

```
Space
→ command discovery

Space Space
→ fuzzy-find file

Space E
→ file sidebar

Space S G
→ search project contents

Space G G
→ Git controls
```

Safe config editing:

```
~/.config
→ your configuration

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

Swap recovery:

```
nvim -r FILE
```

And the most important lesson:

> *Learn enough Neovim to be safe and productive. Go deeper only if the editor itself becomes something you want to master.*

MODULE 09

09

AI Coding Agents in Omarchy

Omarchy treats AI coding agents as first-class development tools.

Current Omarchy 4 / Quattro integrates a growing list of coding-agent CLIs directly into the workstation through:

About This Module

Omarchy treats AI coding agents as first-class development tools.

Current Omarchy 4 / Quattro integrates a growing list of coding-agent CLIs directly into the workstation through:

- lazy-loaded launchers;
- mise-backed installation;
- a configurable default agent;
- keyboard shortcuts;
- terminal aliases;
- Omarchy's own agent launcher;
- usage tracking in the Omarchy Shell;
- crash-diagnosis integration;
- theme synchronization for supported agents.

This module is about **using AI coding agents as part of Omarchy**.

It is deliberately **not** a complete course on:

- OMP;
- Claude Code;
- Codex;
- OpenCode;
- Pi;
- or any other individual agent.

Those products evolve quickly and each can justify its own course.

Our goal is to understand the stable Omarchy layer around them.

The workflow we want is:

```
Project
    ↓
Git
    ↓
Declared runtime environment
    ↓
AI coding agent
    ↓
Inspect
    ↓
Change
    ↓
Test
    ↓
Review diff
    ↓
Commit
```

The AI agent is not a replacement for the rest of the workstation.

It becomes much more useful because the workstation around it is structured.

What You Will Learn

By the end of Module 9, you will understand:

- how Omarchy integrates coding agents;
- how lazy-loaded agent launchers work;
- how to inspect currently supported agents;
- how to choose the default agent;
- how to launch the default agent;
- how to launch an agent in the current terminal;
- how to prompt an agent directly through Omarchy;
- why working directory matters;
- why Omarchy avoids launching agents directly from your home directory;
- how agent CLIs fit into mise;
- how to verify where an agent command comes from;
- how project instruction files such as `AGENTS.md` fit into AI workflows;
- how to use read-only analysis safely;
- how to use Git as the safety boundary around AI-generated changes;
- when a second agent or worktree makes sense;

- how Omarchy's AI usage panel works at a high level;
- how Omarchy can hand crashes to your default agent for diagnosis;
- how to avoid dangerous auto-approval habits.

9.1 — Omarchy's AI Philosophy

The current official Omarchy AI documentation says that Omarchy treats AI coding agents as **first-class citizens** while not choosing one favorite agent for you.

That is an important design choice.

Omarchy provides the integration layer.

You choose the agent.

Conceptually:

```
Omarchy
├── agent launcher
├── default agent selection
├── mise integration
├── keyboard shortcuts
├── usage panel
├── theme integration
└── crash diagnosis

        ↓

Your chosen coding agent
```

This means your workstation workflow can remain relatively stable even if you change coding agents later.

9.2 — Current Supported Agent Launchers

As of the current Omarchy 4 / Quattro documentation in September 2026, Omarchy provides lazy-loaded launchers for major coding-agent CLIs.

The current official AI manual lists commands including:

```
claude
codex
opencode
agy
copilot
crush
grok
pi
omp
ori
```

The current Quattro source also includes additional current options such as:

```
hermes
muse
cursor-agent
openclaw
```

The exact list changes as Omarchy evolves.

Therefore:

> ***Do not treat this handbook's list as permanent.***

Inspect the current default-agent command on your installed system.

9.3 — Discover the Current Agent Choices

Run:

```
omarchy default agent --help
```

You can also inspect the current command surface:

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

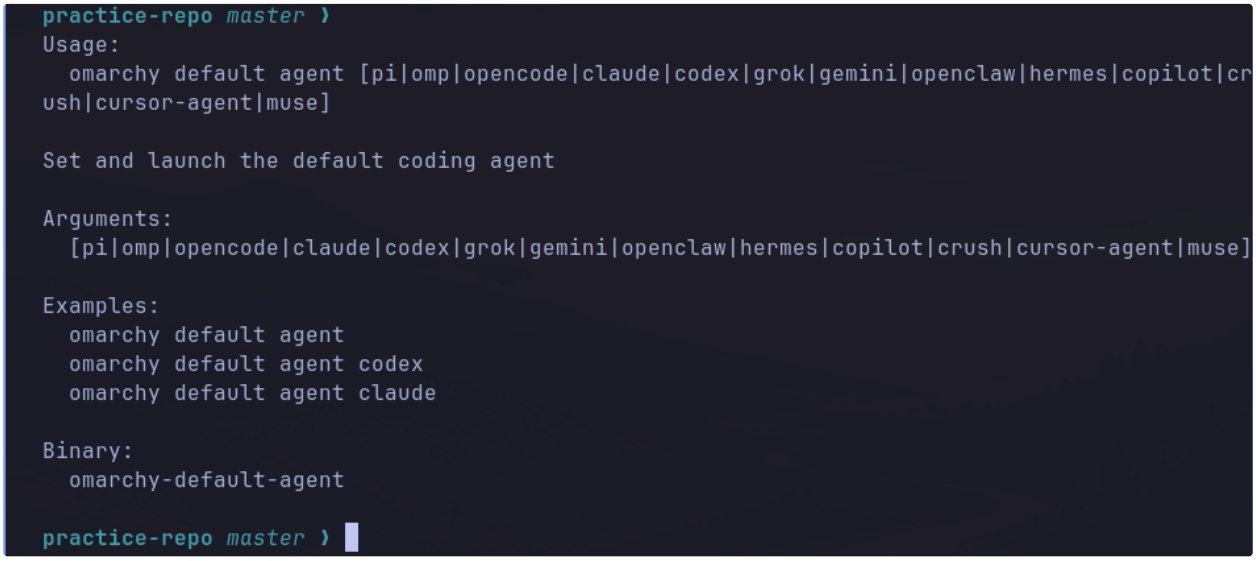
And inspect the current selected default:

```
omarchy default agent
```

If no agent has been selected yet, the command may return no value.

That is intentional.

Current Omarchy does not force one default agent on you.



Screenshot 9.3-A

`omarchy default agent --help`

9.4 — Lazy-Loaded Agent Launchers

Current official Omarchy documentation explains that coding-agent commands are exposed as tiny launchers under:

```
~/.local/bin/
```

The actual tool is installed only when it is first needed.

The installation is managed through mise.

Conceptually:

```
type agent command
        ↓
Omarchy launcher stub
        ↓
mise installs agent if needed
        ↓
agent starts
```

This keeps a fresh Omarchy installation from downloading every coding agent immediately.

9.5 — Why Lazy Loading Is Useful

Imagine Omarchy supports ten coding agents.

Without lazy loading:

```
fresh install
→ install all ten agents
→ large unnecessary setup
→ constant updates
```

With lazy loading:

```
fresh install
→ tiny launcher stubs

first time you use one agent
→ install only that agent
```

This is cleaner and fits the same environment-management philosophy we learned in Module 6.

9.6 — Inspect an Agent Command

Suppose you use:

```
omp
```

Check:

```
type omp
```

Then:

```
command -v omp
```

Do the same for another supported agent:

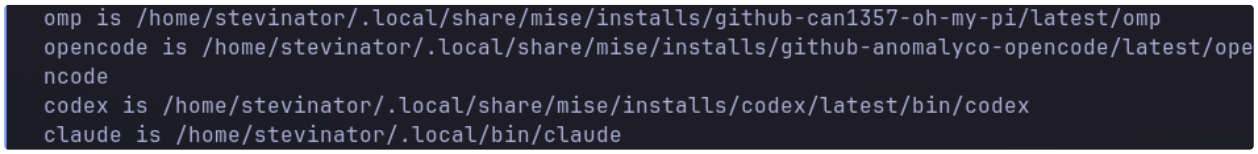
```
type pi
```

This tells you what your shell will actually execute.

You learned this pattern earlier for:

- Python;
- Node;
- Omarchy helpers.

It is equally useful for coding agents.



Screenshot 9.6-A

`type omp` (or equivalent agent)

9.7 — Choose the Default Agent

Current official syntax:

```
omarchy default agent NAME
```

Example:

```
omarchy default agent omp
```

or:

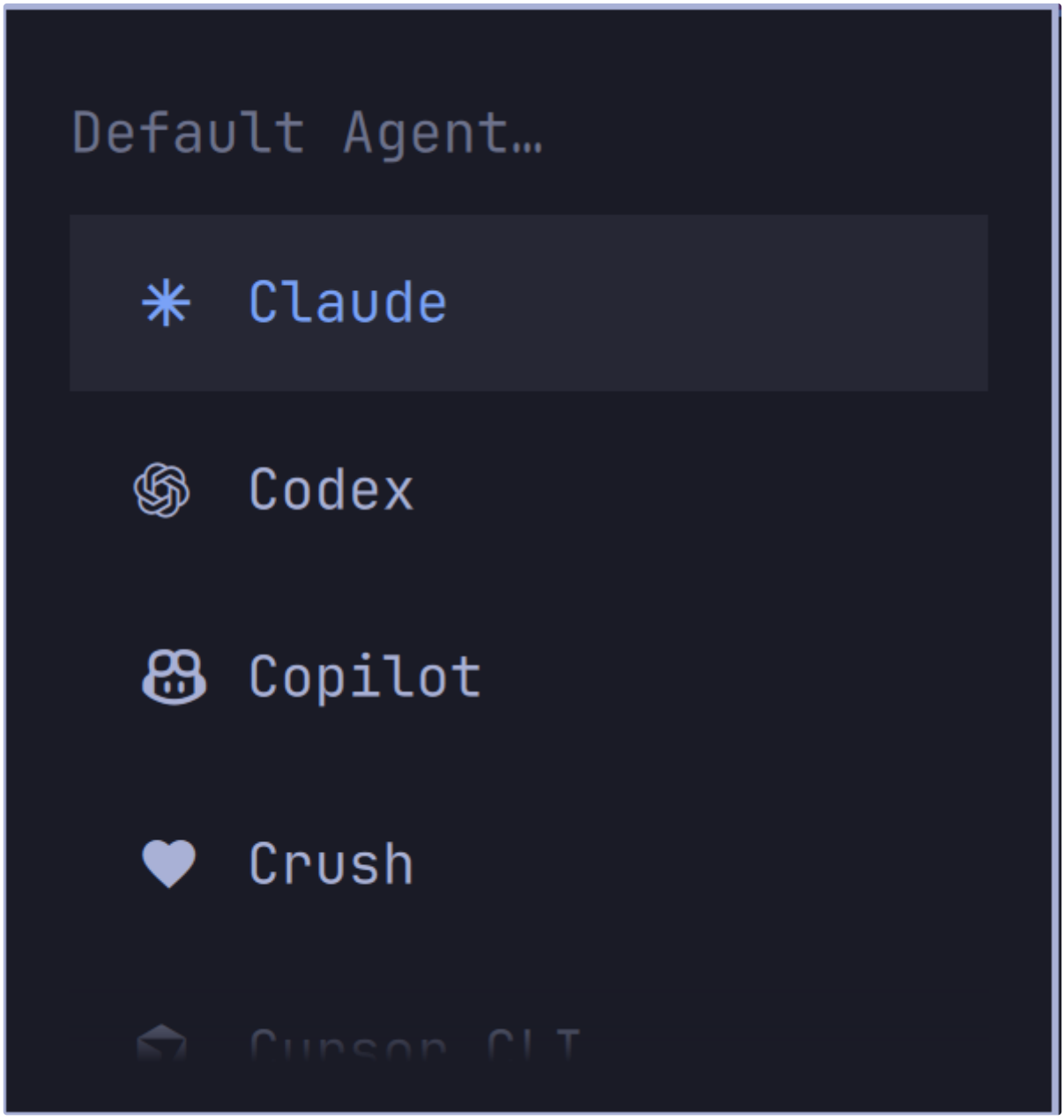
```
omarchy default agent codex
```

or another currently supported name.

You can also use the Omarchy Menu:

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

Current Omarchy installs the selected agent if necessary.



Screenshot 9.7-A
Omarchy default-agent menu

9.8 — Our Stevinator Baseline

The workstation used while developing this course currently uses:

`OMP`

as the default agent.

That is a workstation choice.

It is **not** a course requirement.

You may prefer:

- Pi;
- Codex;
- Claude Code;
- OpenCode;
- Cursor CLI;
- another currently supported agent.

The principles in this module are intentionally agent-independent.

9.9 — Verify the Selected Default

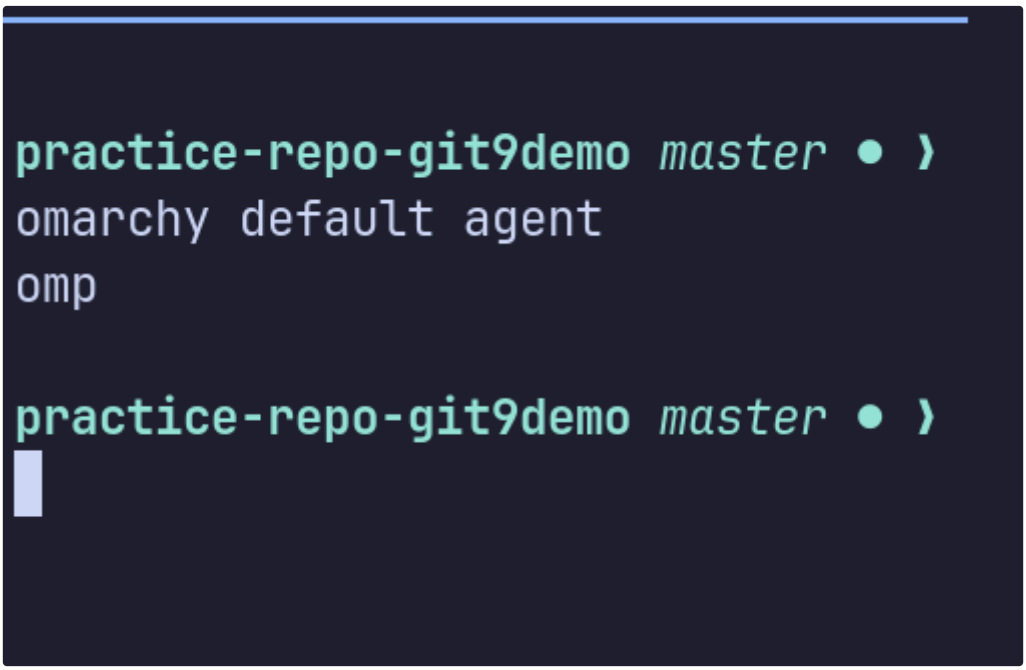
Run:

```
omarchy default agent
```

Expected example:

```
omp
```

Your result may differ.



Screenshot 9.9-A
`omarchy default agent`

9.10 — Launch the Default Agent

Current official Omarchy shortcut:

```
Super + Shift + Ctrl + A
```

This launches the configured default coding agent in a dedicated terminal window.

If no default agent has been selected, current Omarchy can present the agent picker instead.

Screenshot 9.10-A
Default coding agent launched from Omarchy

9.11 — Launch the Default Agent Inline

Current official Omarchy documentation also provides the terminal shortcut:

```
a
```

for launching the default agent inline in the current terminal.

Verify first:

```
type a
```

This is useful inside:

- a normal terminal;
- a Tmux pane;
- an existing project directory.

9.12 — Dedicated Window vs Inline Agent

Both approaches are useful.

Dedicated Agent Window

```
Super + Shift + Ctrl + A
```

Good when:

- you want the agent as a separate Omarchy application;
- you want it independent from your current terminal layout.

Inline Agent

```
a
```

Good when:

- you are already inside Tmux;
- you want the agent in a specific pane;
- you are already inside the correct project directory.

9.13 — Launch an Agent with a Direct Prompt

Current official Omarchy supports:

```
omarchy agent prompt "Review this project"
```

This allows you to launch the configured agent directly into a task.

Example:

```
omarchy agent prompt "Map this repository and explain the main components"
```

IMPORTANT WARNING

The current official Omarchy AI documentation states that agents launched through this prompt workflow run in their respective unattended / don't-stop-to-ask modes.

That means they may be allowed to actually perform actions rather than asking for approval at every step.

Treat this as a powerful automation interface.

Do not use it casually with destructive or vague prompts.

9.14 — The Working Directory Is Part of the Prompt

Consider:

```
cd ~/Projects/project-a
a
```

The agent starts with:

```
project-a
```

as its working context.

Now consider:

```
cd ~/Projects/project-b
a
```

The same agent command now starts in a different project.

This is one reason terminal fluency matters so much for AI development.

Screenshot 9.14-A
`pwd` + `git status` before launching an agent

9.15 — Verify the Directory Before Launching

Before starting an agent:

```
pwd
```

Then:

```
git status
```

Then launch:

```
a
```

This tiny habit prevents a huge category of mistakes.

COURSE RULE

> ***Before an AI agent edits anything, know exactly which repository and branch it is standing in.***

9.16 — Why Omarchy Avoids `$HOME` for Agent Launches

Current official Omarchy documentation explains that agents launched from:

```
$HOME
```

are instead started in:

```
~/Work
```

because coding agents generally should not permanently trust or treat the entire home directory as one development workspace.

This is a sensible safety boundary.

Your home directory can contain:

- SSH keys;
- personal config;
- credentials;
- browser state;
- unrelated projects.

An agent usually does not need project-style access to everything under `$HOME`.

9.17 — Projects vs `~/Work`

In this course, we have used:

```
~/Projects
```

as our preferred development folder.

Current Omarchy's automatic home-directory fallback uses:

```
~/Work
```

for agent launching.

These do not need to be the same directory.

Our recommendation remains:

> *Launch the agent from the actual project directory whenever possible.*

9.18 — AI Agents Are Just Processes

Even though an AI agent may feel like a special system component, at the operating-system level it is still a running process.

That means all of the Linux concepts you learned still apply:

- working directory;
- environment variables;
- PATH;
- permissions;
- files;
- processes;
- network access.

This is useful when troubleshooting.

9.19 — The Agent Uses Your Development Environment

Suppose the project contains:

```
mise.toml
```

with:

```
[tools]
node = "24"
```

You enter the project:

```
cd ~/Projects/my-app
```

Mise activates the project's environment.

Then you launch:

```
a
```

The agent inherits that project environment.

This gives the agent access to the same runtime context you use.

9.20 — Why Reproducible Environments Improve AI Agents

Without a declared environment:

```
Agent:
"Which Node version should I use?"
```

With:

```
mise.toml
```

the project answers that question.

Likewise:

```
package.json
pyproject.toml
Cargo.toml
AGENTS.md
README.md
```

can all provide structure.

The more explicit your project is, the less the agent must guess.

9.21 — Project Instructions

Many modern coding-agent workflows support repository-level instruction files.

A common convention is:

```
AGENTS.md
```

Omarchy itself uses an `AGENTS.md` file in its official repository to give coding agents instructions about the codebase.

That makes it a useful pattern for our own projects.

9.22 — What Belongs in `AGENTS.md`?

A useful project instruction file may contain:

```
Project purpose
Architecture overview
Important directories
Build command
Test command
Lint command
Formatting rules
Files agents should not modify
Expected Git workflow
Safety constraints
```

Example:

```
## AGENTS.md

### Project

This is a Node.js API.

### Development

Use the runtime versions declared in `mise.toml`.

Install dependencies with:

npm install

Run tests with:

npm test

### Rules

- Do not modify `.env`.
- Do not commit generated files.
- Run tests before proposing a commit.
- Keep changes focused on the requested task.
```

9.23 — Agent Support for Instruction Files Varies

Do not assume every coding agent interprets:

```
AGENTS.md
```

in exactly the same way.

Some agents have their own instruction mechanisms or naming conventions.

Therefore:

1. follow the current documentation for your chosen agent;
2. keep `AGENTS.md` useful to humans even if one agent ignores it;
3. avoid putting sensitive information into instruction files.

9.24 — Our Recommended AI Project Baseline

A strong repository may look like:

```
project/
├── .git/
├── .gitignore
├── AGENTS.md
├── mise.toml
├── README.md
├── package.json
└── src/
```

The files communicate different things:

```
Git
→ project history

.gitignore
→ what not to track

mise.toml
→ runtime requirements

AGENTS.md
→ AI/development instructions

README.md
→ project documentation
```

This is a much better AI workspace than an undocumented folder full of files.

Screenshot 9.24-A
project structure with `AGENTS.md` + `mise.toml`

9.25 — Start with Read-Only Analysis

A good first agent task is not:

```
Rewrite this application.
```

A better first task is:

```
Inspect this repository without modifying files.
Explain the architecture, important directories, runtime, test commands, and likely
entry points.
```

Why?

Because you first want to verify whether the agent understands the project.

9.26 — Example Read-Only Prompt

A useful prompt:

```
Analyze this repository without changing any files.

Tell me:
1. what the application does;
2. the main entry points;
3. the important directories;
4. the runtime/toolchain;
5. how tests are run;
6. any obvious architectural risks or unclear areas.

Do not edit anything.
```

This is one of the best ways to begin work with a new codebase.

9.27 — Verify That Read-Only Really Means Read-Only

Different agent harnesses have different permission systems.

A natural-language instruction such as:

```
Do not edit anything.
```

is useful, but it is not a cryptographic sandbox.

If the agent offers a genuine read-only or restricted permission mode, use it when appropriate.

Always refer to the current documentation for the specific agent.

9.28 — Approval Modes Matter

Coding agents vary widely in how they handle:

- file writes;
- command execution;
- network access;
- destructive operations.

Some agents:

```
ask before writes
```

Others can:

```
write automatically
```

Others can run in:

```
fully unattended mode
```

Do not generalize permission semantics from one agent to another.

9.29 — The Omarchy Prompt Launcher Is Powerful

Remember:

```
omarchy agent prompt "..."
```

currently launches agents in unattended modes appropriate to the selected harness.

This makes it excellent for:

- deliberate automation;
- repeatable agent tasks;
- scripted workstation workflows.

It also means a vague prompt can have real consequences.

9.30 — Bad Automated Prompt

Avoid:

```
omarchy agent prompt "Fix everything"
```

Why?

The scope is undefined.

The agent may:

- edit many files;
- run unnecessary commands;
- change configuration;
- create difficult-to-review diffs.

9.31 — Better Automated Prompt

Prefer:

```
Inspect the failing test in tests/auth.test.ts.

Identify the root cause.

Change only the files required to fix that failure.

Run the relevant test suite.

Do not commit or push.

At the end, summarize the files changed and remaining risks.
```

Boundaries make agents more useful.

9.32 — Git Before Agent

Before meaningful AI changes:

```
git status
```

Ideally:

```
working tree clean
```

Why?

Because after the agent works:

```
git diff
```

can show exactly what it changed.

If your repository already contains unrelated modifications, review becomes much harder.

9.33 — Recommended Agent Editing Loop

Use:

```
git status
      ↓
launch agent
      ↓
agent edits
      ↓
run tests
      ↓
git status
      ↓
git diff
      ↓
human review
      ↓
git add
      ↓
git diff --staged
      ↓
commit
```

This is our default AI-development discipline.

Screenshot 9.33-A
`git diff` after an agent-style change

9.34 — Never Let "Agent Finished" Mean "Task Verified"

An agent saying:

```
Done.
```

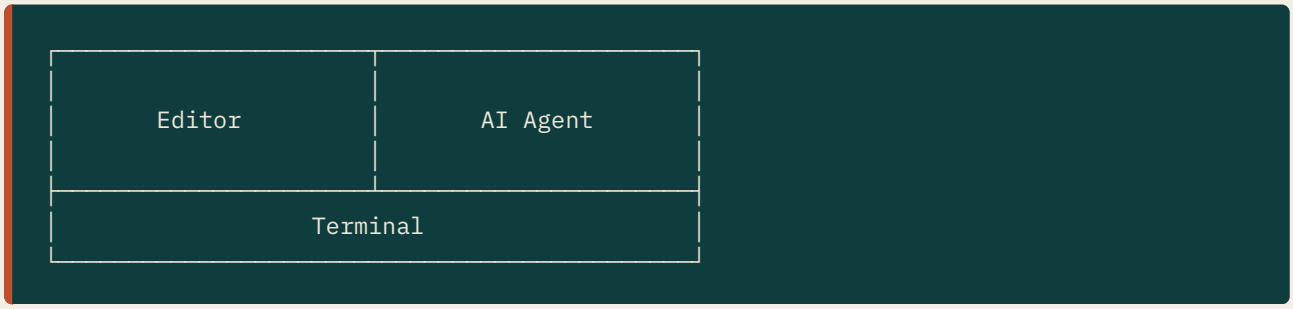
does not prove:

- the application works;
- tests pass;
- no unrelated files changed;
- security is preserved;
- the intended behavior is correct.

Verification still matters.

9.35 — Agent + Tmux

A strong setup:



You already learned this in the Tmux module.

Current Omarchy's:

```
tdl
```

helper is built specifically around this type of workflow.

Screenshot 9.35-A
editor + agent + terminal Tmux layout (`tdl`)

9.36 — Why Tmux Helps AI Development

One pane:

```
editor
```

Second pane:

```
agent
```

Third pane:

```
tests / Git / server
```

This keeps the control loop visible.

You are less likely to blindly trust the agent because you can immediately inspect and test its work.

9.37 — Agent + Git Worktree

For one agent:

```
normal branch
```

is often enough.

For genuinely parallel agents:

```
separate Git worktrees
```

are much safer.

Conceptually:

```
Worktree A
→ Agent A

Worktree B
→ Agent B
```

Each agent has its own filesystem checkout.

9.38 — Do Not Parallelize Too Early

Using multiple agents at once creates coordination overhead.

Potential problems:

- duplicate work;
- conflicting design decisions;
- higher cost;
- unclear ownership;
- difficult merges.

Start with:

```
one agent
+
one focused task
```

Parallelize only when the tasks are truly separable.

9.39 — Model Roles Are Agent-Specific

Some coding-agent tools allow multiple model roles such as:

```
fast model
planning model
deep reasoning model
review model
```

This can be useful.

But model configuration belongs to the specific agent.

Omarchy's job is mainly to:

- install;
- select;
- launch;
- integrate.

We will not turn this course into a model-routing course.

9.40 — Use the Right Model for the Job

At a high level:

```
fast/cheap model
→ simple search, mechanical tasks

strong reasoning model
→ architecture, difficult debugging

specialized reviewer
→ second-pass review
```

Exact model names, pricing, and capabilities change rapidly.

Follow the current documentation for your agent/provider.

9.41 — Agent Sessions

Most coding agents support some concept of conversation/session history.

Useful session behavior includes:

- resuming a long investigation;
- preserving project discussion;
- returning to a plan later.

However:

> *A saved agent session is not a substitute for project documentation.*

Important decisions should still live in durable places such as:

- Git commits;
- issues;
- README;
- architecture docs;
- `AGENTS.md`.

9.42 — Omarchy's Agents Panel

Current Omarchy 4 documentation includes an AI usage panel in the top bar.

The agents icon appears once Omarchy detects AI coding usage on the machine.

The current official documentation states that the panel can track supported subscription/token usage such as:

- plan information;
- session/weekly usage;
- prepaid balance where relevant;
- tokens by day;
- tokens by model.

Current built-in coverage includes selected providers such as Claude Code, Codex, and Fireworks.

The exact providers and metrics can evolve.

9.43 — Agent Usage Updates

Current official Omarchy documentation states that usage records are regenerated periodically using:

```
omarchy agent usage-update
```

You generally do not need to run this manually.

The shell integration updates the panel automatically.

9.44 — Why Usage Visibility Matters

AI coding can become expensive without obvious feedback.

A visible usage panel helps answer:

```
Which provider am I using?
How much of my plan is consumed?
Which model is generating most tokens?
```

This supports better model/cost decisions.

We will revisit cost-aware workflows in Module 10.

9.45 — Theme Synchronization

Current Omarchy AI integration can synchronize the active Omarchy theme with supported coding agents.

The current official documentation explicitly mentions theme synchronization for agents including:

- Claude Code;
- Pi;
- OpenCode;

and current Quattro source also references Hermes where supported.

This is a quality-of-life feature rather than a core engineering capability.

But it reinforces the idea that agents are integrated into the workstation rather than bolted on as unrelated tools.

9.46 — Crash Diagnosis with the Default Agent

Current Omarchy documentation includes a particularly interesting AI integration.

Omarchy watches:

```
systemd-coredump
```

for process crashes.

When a process crashes, current Omarchy can display a:

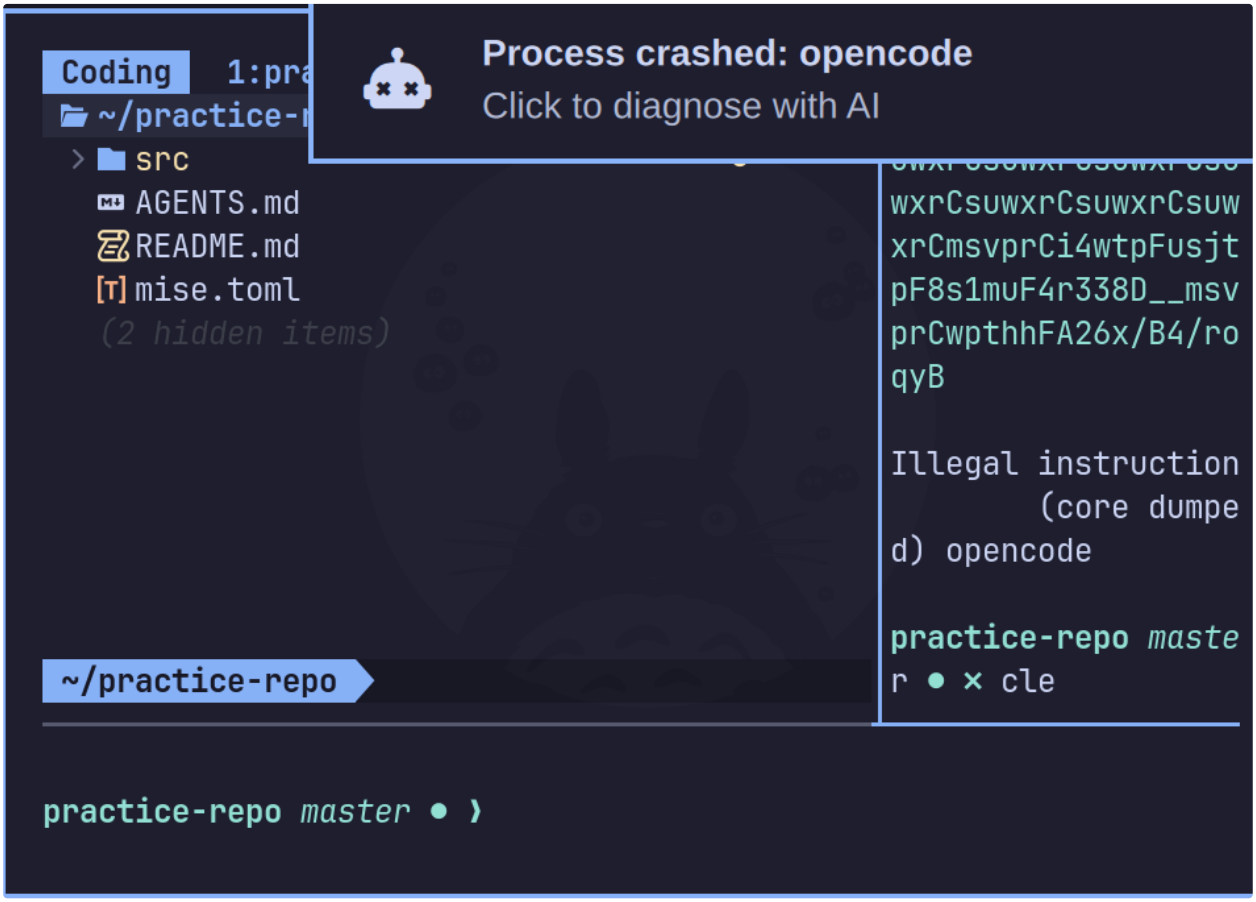
```
Process crashed
```

notification.

Clicking it can hand the crash information to your default coding agent together with Omarchy's crash-diagnosis guidance.

The agent can then inspect the available crash data and help determine whether the issue:

- is understandable locally;
- needs more investigation;
- may be worth reporting upstream.



Screenshot 9.46-A
real "Process crashed" notification with crash-diagnosis prompt

9.47 — Why Crash Diagnosis Is a Good AI Use Case

Crash investigation often requires:

- reading logs;
- inspecting stack traces;
- identifying binaries;
- understanding package versions;
- correlating symptoms.

Those are tasks AI agents can assist with well.

But the agent still needs evidence.

The correct workflow is:

```
crash
↓
collect real system information
↓
agent analyzes evidence
```

not:

```
crash
↓
agent guesses
```

9.48 — AI-Assisted Workstation Troubleshooting

Your coding agent can also help inspect the workstation itself.

Good read-only tasks include:

```
Inspect failed user services and explain which ones matter.
```

```
Read my Hyprland monitor configuration and compare it with `hyprctl monitors all`.
```

```
Inspect this application's logs and propose likely causes without changing anything.
```

This can be extremely useful because the agent can combine:

- shell commands;
- config reading;
- log analysis.

9.49 — Let the Agent Inspect Before It Repairs

Good troubleshooting workflow:

```
symptom
↓
ask agent to inspect
↓
collect facts
↓
explain likely cause
↓
propose minimal change
↓
you approve
↓
apply
↓
verify
```

Avoid:

```
Something is broken.
Fix my system.
```

especially in unattended mode.

9.50 — Good System-Troubleshooting Prompt

Example:

```
My Bluetooth panel is not showing my device.

Do not change anything yet.

Inspect the relevant Omarchy commands, Bluetooth state, failed services, and recent
relevant logs.

Tell me what you find and propose the smallest next step.
```

This encourages evidence-first debugging.

9.51 — Agent Permissions and `sudo`

Be very cautious about giving unattended agents broad `sudo` capability.

With root privileges, an agent could potentially:

- modify system configuration;

- install/remove packages;
- alter services;
- change permissions;
- damage the boot system.

The same rule from the terminal module applies:

> `sudo` *is privilege elevation, not a magic problem-solving command.*

9.52 — Agent Permissions and Secrets

Your development machine may contain:

```
~/.ssh
API credentials
GitHub tokens
cloud credentials
.env files
browser session data
```

Do not assume every agent needs access to all of them.

Use:

- project-scoped working directories;
- separate secret-management mechanisms;
- minimal permissions;
- explicit approval for sensitive operations.

9.53 — Agent Permissions and Network Access

Some tasks require network access:

- package installation;
- documentation lookup;
- Git push;
- API testing.

Others do not.

When the agent offers network permission controls, grant only what the task needs.

The exact controls depend on the chosen agent.

9.54 — Verify Agent Authentication Separately

Each coding agent can have its own authentication mechanism.

Examples may include:

- subscription login;
- API key;
- OAuth;
- provider account.

Omarchy installs and launches the tool.

It does not make all third-party service accounts interchangeable.

Follow the current official authentication instructions for your selected agent.

9.55 — Changing the Default Agent

One advantage of Omarchy's abstraction is that switching agents is straightforward.

Example:

```
omarchy default agent pi
```

Later:

```
omarchy default agent codex
```

Then:

```
Super + Shift + Ctrl + A
```

launches the newly selected default.

Your surrounding Omarchy workflow remains similar.

9.56 — Agent-Specific Aliases

Current official Omarchy documentation also exposes direct terminal shortcuts for certain agents.

Examples currently documented include:

```
c
cx
cy
```

for selected agent CLIs.

Exact mappings can change.

Inspect them:

```
type c
type cx
type cy
```

Do not assume old aliases forever.

9.57 — Custom Agent Wrappers

Current official Omarchy AI documentation provides:

```
omarchy-mise-install <package> [command-name]
```

for wrapping additional CLIs through the same mise-backed pattern.

This is an advanced capability.

You do not need it for normal use.

The important concept is:

> *Omarchy's agent system is extensible beyond the built-in launcher list.*

9.58 — Do Not Install Duplicate Copies of the Same Agent

Before manually running:

```
npm install -g ...
curl | bash ...
```

check:

```
type AGENT
```

and:

```
omarchy default agent --help
```

Omarchy may already provide a mise-backed installation path.

Duplicate installations can create PATH confusion.

9.59 — Practical Exercise: Inspect Agent Integration

Run:

```
omarchy default agent
```

Then:

```
omarchy default agent --help
```

Then:

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

Write down:

- your current default;
- the available agent-related Omarchy commands.

9.60 — Practical Exercise: Inspect the Agent Command

If your default is OMP:

```
type omp
command -v omp
```

If your default is another agent, substitute its command.

The purpose is to connect:

```
Omarchy
→ shell command
→ mise-managed installation
```

9.61 — Practical Exercise: Launch From the Correct Project

Run:

```
cd ~/Projects/YOUR_PROJECT
```

Verify:

```
pwd
git status
```

Launch inline:

```
a
```

Exit when finished.

This is the basic AI-agent entry workflow.

9.62 — Practical Exercise: Read-Only Repository Map

Ask your agent:

```
Analyze this repository without modifying files.

Explain:
- what the project does;
- the major directories;
- the main entry points;
- the runtime/tooling;
- how tests are run;
- any important project instructions.

Do not make changes.
```

Compare the response with the real repository.

Does it understand the project correctly?

9.63 — Practical Exercise: Create `AGENTS.md`

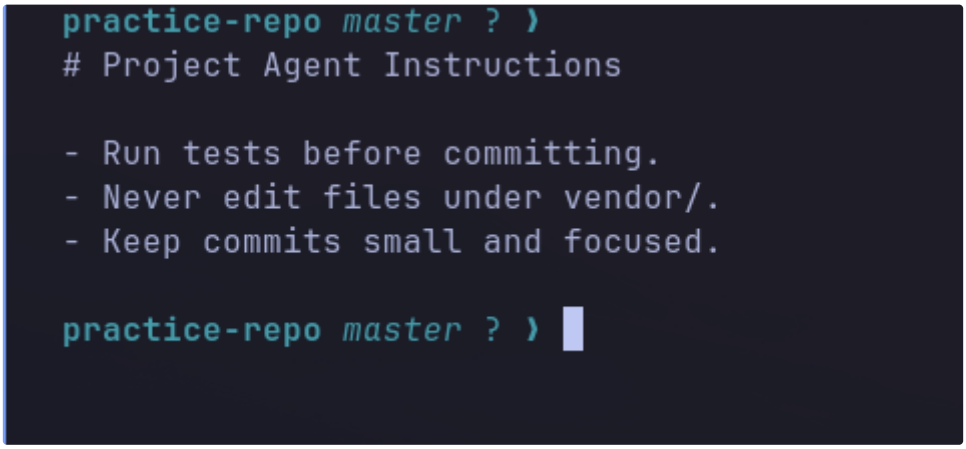
If your project does not already have one, create:

`AGENTS.md`

Keep it small.

Example:

```
## AGENTS.md

### Project

Briefly describe the project.

### Environment

Use the runtime versions declared in `mise.toml`.

### Validation

Run the relevant tests before considering a change complete.

### Rules

- Keep changes focused.
- Do not modify secrets.
- Do not commit or push without explicit approval.
```



Screenshot 9.63-A
sample `AGENTS.md`

Add project-specific details.

Commit it if it is intended to be shared:

```
git add AGENTS.md
git commit -m "docs: add agent development instructions"
```

9.64 — Practical Exercise: AI Change with Git Review

Start clean:

```
git status
```

Ask the agent for one small code/documentation change.

After it finishes:

```
git status
```

Then:

```
git diff
```

Check:

- expected files only;
- no secrets;
- no unrelated formatting;
- no generated junk.

Run the relevant validation.

Only then stage/commit.

9.65 — Practical Exercise: Tmux Agent Pane

Enter your project:

```
cd ~/Projects/YOUR_PROJECT
```

Open your Tmux development layout.

Place:

`editor`

in one pane,

`agent`

in another,

and:

`tests / Git`

in a third.

Practice moving between them without leaving the project workspace.

9.66 — Practical Exercise: System Inspection Without Changes

Ask your default agent:

```
Inspect this Omarchy machine without changing anything.

Check:
- Omarchy version;
- failed system services;
- failed user services;
- active network state.

Explain anything that appears abnormal.

Do not modify configuration or restart services.
```

Then manually verify the commands/results.

This demonstrates AI-assisted system inspection while keeping the human in control.

9.67 — Common AI-Agent Mistakes

Mistake 1 — Launching From the Wrong Directory

Always check:

```
pwd
git status
```

before editing tasks.

Mistake 2 — Giving a Huge Vague Task

Bad:

```
Improve the whole project.
```

Better:

```
Fix this failing endpoint and run these tests.
```

Mistake 3 — Assuming "Read Only" from Natural Language Is a Security Boundary

Use actual agent permission controls where available.

Mistake 4 — Letting an Unattended Agent Use `sudo` Casually

System-level changes deserve stronger scrutiny.

Mistake 5 — Starting With a Dirty Git Working Tree

You lose the ability to distinguish your changes from the agent's changes.

Mistake 6 — Accepting the Agent's Summary Instead of Inspecting `git diff`

The diff is the evidence.

Mistake 7 — Letting Multiple Agents Edit the Same Working Tree

Use separate worktrees when true parallel editing is required.

Mistake 8 — Treating `AGENTS.md` as Universal Agent Configuration

Different agents may interpret repository instructions differently.

Follow the current documentation for the selected harness.

Mistake 9 — Installing Another Copy of an Agent Manually

Check the existing Omarchy/mise integration first.

Mistake 10 — Turning the Omarchy Course Into an Agent-Specific Course

The point of this module is the workstation integration.

Agent-specific advanced topics belong in separate training.

9.68 — Checkpoint

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

Omarchy Integration

- [ ] I understand that Omarchy treats coding agents as first-class development tools.
- [ ] I know how to inspect current supported agent choices.
- [ ] I know how to choose the default agent.
- [ ] I know how to check the current default.
- [ ] I can launch the default agent through Omarchy.
- [ ] I can launch it inline in a terminal.

Environment

- [ ] I understand that agent launchers are mise-backed.
- [ ] I can inspect an agent command with `type`.
- [ ] I understand why working directory matters.
- [ ] I understand why launching from `$HOME` is handled specially.

Project Context

- [ ] I understand what `AGENTS.md` is used for conceptually.
- [ ] I know not to assume every agent interprets it identically.
- [ ] I understand how `mise.toml`, Git, README, and agent instructions work together.

Safety

- [ ] I check `git status` before an editing task.
- [ ] I inspect `git diff` after the task.
- [ ] I understand that unattended Omarchy agent prompts can perform real actions.
- [ ] I do not casually grant agents `sudo`.
- [ ] I understand that read-only prompts are not necessarily hard security boundaries.

Workflow

- [ ] I can ask an agent to map a repository without modifying it.
- [ ] I can use an agent inside Tmux.
- [ ] I understand when a worktree is useful for a second agent.
- [ ] I understand that agent output still requires validation.

9.69 — What You Can Now Do

After completing Module 9, you can now:

- choose and switch coding agents through Omarchy;
- understand Omarchy's mise-backed lazy agent installation;
- launch an agent from the correct project;
- give an agent structured project context;
- use project-level AI instructions;
- start with read-only repository analysis;
- use Git as a control boundary around AI changes;
- integrate AI into Tmux-based development;
- use separate worktrees for parallel tasks;
- inspect AI usage from the Omarchy Shell;
- use AI as a system-diagnostics assistant;
- understand the risk of unattended prompt execution.

Most importantly:

> *You are not merely "using an AI chatbot while coding." You now have an AI coding agent integrated into a structured, version-controlled, reproducible development workstation.*

Module 9 Summary

Omarchy integration:

```
Agent command
        ↓
lazy Omarchy launcher
        ↓
mise-managed installation
        ↓
coding agent
```

Choose default:

```
omarchy default agent NAME
```

Check:

```
omarchy default agent
```

Launch:

```
Super + Shift + Ctrl + A
```

Inline:

```
a
```

Prompt directly:

```
omarchy agent prompt "TASK"
```

with the important current warning:

```
direct prompt launcher
→ unattended agent mode
→ real actions may occur
```

Project AI baseline:

```
project/
├── .git/
├── .gitignore
├── AGENTS.md
├── mise.toml
├── README.md
└── source
```

Recommended AI loop:

```
pwd
↓
git status
↓
agent
↓
test
↓
git status
↓
git diff
↓
review
↓
commit
```

Parallel work:

```
one task
→ one agent / branch

independent parallel tasks
→ separate Git worktrees
```

And the most important rule:

> ***AI agents are fastest when the environment is explicit, and safest when every meaningful change is reviewable through Git.***

MODULE 10

10

Building an AI Development Workflow on Omarchy

You now have all of the core building blocks:

- Omarchy desktop workflows;
- terminal fluency;
- Tmux;
- reproducible development environments with mise;
- Git and GitHub;
- an editor;
- coding-agent integration.

About This Module

You now have all of the core building blocks:

- Omarchy desktop workflows;
- terminal fluency;
- Tmux;
- reproducible development environments with mise;
- Git and GitHub;
- an editor;
- coding-agent integration.

Now we combine them into a repeatable development workflow.

This module is where the workstation starts to feel like one system rather than a collection of tools.

The goal is not:

```
Use AI everywhere.
```

The goal is:

```
Use AI where it helps,
keep the environment explicit,
keep changes reviewable,
and keep the developer in control.
```

A strong AI-native workflow should make it easy to move from:

```
idea
→ project
→ plan
→ implementation
→ test
→ review
→ commit
```

without losing track of:

- which repository you are in;
- which branch you are on;
- which runtime is active;

- what the agent changed;
- whether the code actually works;
- what should be committed.

This module turns the components from Modules 3–9 into one daily operating model.

What You Will Learn

By the end of Module 10, you will be able to:

- design a repeatable multi-workspace Omarchy development layout;
- bootstrap a new project cleanly;
- prepare a cloned project for AI-assisted work;
- structure project context for coding agents;
- decide when to use editor, terminal, AI agent, or browser;
- use Tmux as the project control center;
- keep Git as the review and rollback boundary;
- separate analysis, implementation, testing, and review;
- use a second agent or worktree only when it provides real value;
- understand the current Omarchy Tmux development helpers;
- use live diff watching where appropriate;
- reason about local vs remote development;
- think about model strength, latency, and cost without tying the workflow to one provider;
- create a practical daily start and shutdown routine;
- define a repeatable AI-native development loop.

10.1 — The Workstation as a System

The workstation we built can now be viewed as one layered development system.

```
Omarchy / Hyprland
→ graphical workspace organization

Ghostty / terminal
→ command-line interface

Tmux
→ persistent project workspace

Editor
→ code editing

Mise
→ runtime/tool environment

Git
→ history, review, rollback

GitHub
→ remote collaboration

AI agent
→ analysis and implementation assistance
```

Each layer has a specific job.

That separation is useful.

10.2 — Do Not Make the AI Agent the Center of Everything

It is tempting to structure the entire workstation around one coding agent.

That is fragile.

Agents change.

Pricing changes.

Model quality changes.

Tools disappear.

A better architecture is:

```
Project
├── Git
├── runtime configuration
├── documentation
├── tests
└── code

Developer tools
├── editor
├── terminal
└── AI agent
```

The project remains usable even if you switch from one agent to another.

That is the correct abstraction boundary.

10.3 — Recommended Omarchy Workspace Layout

A strong baseline is:

```
Workspace 1
→ Browser / documentation / research

Workspace 2
→ Main development workspace

Workspace 3
→ Infrastructure / logs / databases / servers

Workspace 4
→ Communication / notes / project management
```

You can customize this later.

For training, this gives each activity a predictable location.

10.4 — Workspace 1: Research

Use:

```
Super + 1
```

for:

- documentation;

- issue trackers;
- API references;
- browser-based tools;
- external research.

This keeps browser research from constantly covering your code.

10.5 — Workspace 2: Development

Use:

```
Super + 2
```

for the primary project workspace.

A typical layout:

```
Ghostty
└── Tmux
    ├── Editor
    ├── AI agent
    └── terminal/test shell
```

This becomes the core of the AI development workstation.

10.6 — Workspace 3: Infrastructure

Use:

```
Super + 3
```

for:

- dev servers;
- Docker/LazyDocker if needed;
- database shells;
- logs;
- SSH;
- system monitoring;
- deployment tools.

Separating infrastructure from the coding workspace reduces noise.

10.7 — Workspace 4: Communication

Use:

```
Super + 4
```

for:

- chat;
- issue discussions;
- notes;
- project planning;
- email.

This helps create a clean focus boundary.

10.8 — Fast Code ↔ Documentation Switching

A useful pattern:

```
Workspace 2
→ code

Workspace 1
→ docs
```

Then use:

```
Super + Ctrl + Tab
```

to jump back to the former workspace.

This creates a fast:

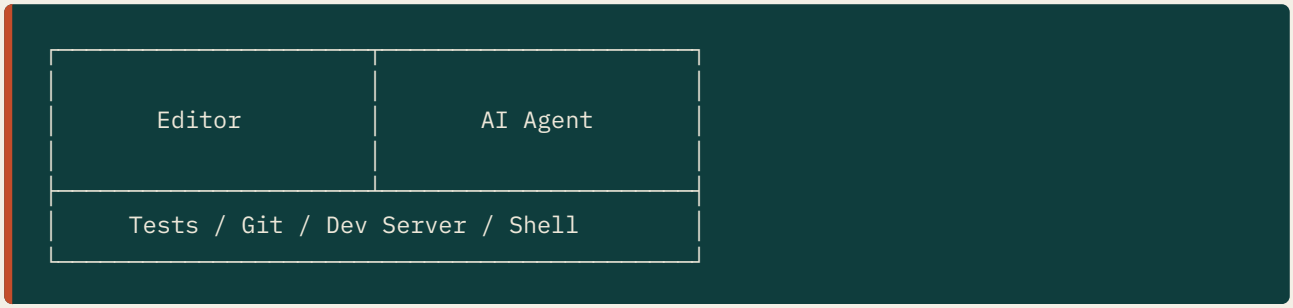
```
code
⇔
documentation
```

loop.

10.9 — Build the Project Control Center in Tmux

Inside the development workspace, Tmux becomes the control center.

A strong baseline:



Current Omarchy provides a helper specifically for this style:

```
tdl AGENT
```

The official current Omarchy terminal and shell-function documentation describes `tdl` as an editor + AI agent + terminal development layout.

10.10 — Start From the Project Directory

Before launching the layout:

```
cd ~/Projects/YOUR_PROJECT
```

Verify:

```
pwd
```

Then:

```
git status
```

Then:

```
mise ls --current
```

Only then create/attach your development session.

This gives you three critical facts:

```
Where am I?
What Git state am I in?
What runtime environment is active?
```

10.11 — A Good Daily Start Sequence

A practical daily startup can be:

```
cd ~/Projects/YOUR_PROJECT
git status
git branch --show-current
mise ls --current
```

Then:

```
t
```

or your preferred Tmux session workflow.

If starting a fresh layout:

```
tdl YOUR_AGENT
```

The exact shell helper behavior should always be verified with:

```
type t
type tdl
```

on the learner's current Omarchy version.

10.12 — Why `git status` Comes Before AI

Suppose your working tree already contains:

```
3 files modified by you
```

Then an agent modifies:

```
5 more files
```

Now:

```
git diff
```

contains mixed authorship and mixed intent.

That makes review harder.

Prefer:

```
clean working tree
        ↓
agent task
        ↓
agent diff
```

whenever practical.

10.13 — New Project Bootstrap

For a new project:

```
mkdir -p ~/Projects/my-project
cd ~/Projects/my-project
git init
```

Add a runtime:

```
mise use node@CURRENT_REQUIRED_VERSION
```

or another required runtime.

Create:

```
README.md
AGENTS.md
.gitignore
```

Then make an initial commit.

Conceptually:

```
project/
├── .git/
├── .gitignore
├── AGENTS.md
├── mise.toml
├── README.md
└── source/
```

This is a strong AI-native project foundation.

10.14 — Cloned Project Bootstrap

For an existing repository:

```
gh repo clone OWNER/REPOSITORY
cd REPOSITORY
```

Then inspect before executing setup.

Recommended order:

```
git status
```

```
ls
```

```
bat README.md
```

if present.

Inspect:

```
bat AGENTS.md
```

if present.

Inspect:

```
bat mise.toml
```

if present.

Then:

```
mise install
```

after you have reviewed the configuration.

10.15 — Do Not Let the Agent Discover Basic Project Facts by Guessing

Before a coding session, the project should make important facts explicit.

Examples:

```
runtime versions
→ mise.toml

dependency manager
→ package metadata / lockfile

project instructions
→ AGENTS.md

purpose
→ README.md

validation
→ test/lint scripts
```

The goal is to reduce ambiguity.

10.16 — A Useful `AGENTS.md` Structure

Keep it practical.

Example:

```
## AGENTS.md

### Project

Describe the application and its purpose.

### Architecture

- `src/api` — API routes
- `src/services` — business logic
- `tests` — test suite

### Environment

Use the versions declared in `mise.toml`.

### Commands

Install dependencies:

npm install

Run tests:

npm test

Run lint:

npm run lint

### Rules

- Keep changes focused.
- Do not modify secrets.
- Do not commit or push unless explicitly requested.
- Run relevant tests after code changes.
- Explain any architectural change before making it.
```

This is useful to both humans and compatible agents.

10.17 — The Four AI Work Phases

A strong AI coding task can be divided into:

```
1. Analyze
2. Plan
3. Implement
4. Verify
```

Do not collapse all four phases automatically into one giant prompt.

Separating them gives you checkpoints.

10.18 — Phase 1: Analyze

Example:

```
Inspect the authentication flow.

Do not edit files.

Explain:
- entry points;
- important modules;
- current behavior;
- likely cause of the bug;
- files that would probably need to change.
```

Goal:

```
understand before editing
```

10.19 — Phase 2: Plan

After reviewing the analysis:

```
Propose the smallest implementation plan.

For each step, list:
- file;
- change;
- risk;
- validation method.

Do not edit yet.
```

This is particularly useful for:

- architectural work;
- risky refactors;
- unfamiliar repositories.

For tiny tasks, a separate planning phase may be unnecessary.

10.20 — Phase 3: Implement

After you agree with the direction:

```
Implement the approved plan.

Keep the change focused.

Do not commit or push.

Run the relevant tests.
```

The agent should now have a constrained task.

10.21 — Phase 4: Verify

Verification belongs to both the agent and the developer.

Run:

```
git status
git diff
```

Then run:

- tests;
- lint;
- type checking;
- build;
- manual validation;

as relevant.

Do not treat an agent's statement:

```
Tests pass.
```

as proof without checking output when the task matters.

10.22 — The Core AI Development Loop

A reliable daily loop is:

```
understand task
        ↓
git status
        ↓
agent analysis
        ↓
implementation
        ↓
tests
        ↓
git diff
        ↓
review
        ↓
refine
        ↓
git add
        ↓
git diff --staged
        ↓
commit
        ↓
push
```

This is the central workflow of the workstation.

10.23 — Editor and Agent Should Have Different Jobs

A common mistake is duplicating the same work in both panes.

Use the editor for:

- precise inspection;
- manual edits;
- code navigation;
- human judgment.

Use the agent for:

- codebase mapping;
- repetitive implementation;
- multi-file changes;
- debugging hypotheses;
- test generation;
- review.

Use the terminal for:

- Git;
- tests;
- development server;
- diagnostics.

Clear roles reduce cognitive overhead.

10.24 — Ask the Agent to Explain Before Large Changes

For a one-line fix, this may be unnecessary.

For a large refactor:

```
Explain the proposed change first.
Do not edit until the plan is clear.
```

This gives you a chance to catch:

- wrong architecture assumptions;
- unnecessary scope;
- poor dependency choices;
- security risks.

10.25 — Keep Agent Tasks Small Enough to Review

Good task:

```
Add validation for expired refresh tokens.
```

Risky task:

```
Redesign the authentication system, database schema, API, frontend, and tests.
```

The larger the task, the harder:

```
git diff
```

becomes to review.

Small reviewable tasks are usually more reliable.

10.26 — Use the Terminal Pane for Independent Verification

Do not make the agent pane the only source of command execution.

Keep a separate terminal pane for your own:

```
git status
```

```
git diff
```

```
npm test
```

or equivalent.

This keeps independent verification easy.

10.27 — Run the Smallest Relevant Test First

If the task changes one component:

```
run component test
```

before:

```
run full test suite
```

Typical pattern:

```
targeted test
↓
broader test
↓
build/lint if appropriate
```

This gives faster feedback.

10.28 — Let Project Configuration Define Validation

Avoid hard-coding universal commands such as:

```
npm test
```

for every project.

The correct command may be:

```
pnpm test
bun test
pytest
cargo test
go test ./...
mise run test
```

The project should tell you.

Sources include:

- README;
- AGENTS.md;
- package metadata;
- `mise.toml`;
- CI configuration.

10.29 — Mise Tasks Can Standardize Workflows

Current mise supports project tasks.

A project can define:

```
[tasks.test]
run = "npm test"

[tasks.lint]
run = "npm run lint"
```

Then everyone can use:

```
mise run test
```

and:

```
mise run lint
```

This can make AI-agent instructions more portable.

Instead of:

```
Run npm test unless we're in some other package manager...
```

you can say:

```
Run `mise run test`.
```

10.30 — Git Diff Is the Ground Truth of File Changes

After an agent works:

```
git diff
```

tells you what actually changed.

The agent's summary is useful.

The diff is authoritative.

Review for:

- unexpected files;
- deleted code;
- large formatting churn;
- security changes;
- dependency changes;
- config changes;
- generated files.

10.31 — Diff Watching

Current Omarchy provides:

```
tds
```

as a square Tmux development layout containing:

- editor;
- live diff watcher via `hunk diff --watch`;
- terminal;
- AI agent.

This is especially useful when you want to observe changes as the agent edits.

Current official Omarchy shell-function documentation describes this layout explicitly.

10.32 — When `tds` Is Useful

Good use cases:

```
AI makes multi-file changes
you want live visual feedback
you are reviewing iterative edits
```

Less useful:

```
tiny one-file change
```

Do not use a four-pane setup just because it looks advanced.

10.33 — Verify Helper Dependencies

If:

```
tds
```

opens a pane with:

```
command not found
```

do not blame Tmux immediately.

The current documented layout depends on:

```
hunk diff --watch
```

Check:

```
type hunk
```

Then determine whether the missing tool should be installed through:

- Omarchy;
- mise;
- system package layer.

Use the installation-layer decision tree from Module 6.

10.34 — Branches for Normal AI Tasks

A reasonable feature workflow:

```
git switch -c feature/auth-refresh
```

Then launch the agent.

This isolates the work from:

```
main
```

After implementation:

```
git diff
```

review and test.

Later merge or open a pull request as appropriate.

10.35 — Worktrees for Parallel AI Tasks

Current Omarchy provides worktree helpers:

```
ga [branch]
```

and:

```
gd
```

The current official shell-function documentation says:

```
ga
→ create a new branch/worktree next to the current repo and enter it

gd
→ remove the current worktree and branch after confirmation
```

Parallel structure:

```
project/
→ main worktree

project-feature-a/
→ Agent A

project-feature-b/
→ Agent B
```

This is far safer than two agents editing one checkout.

10.36 — Parallelize Independent Tasks Only

Good parallel tasks:

```
Agent A
→ backend endpoint

Agent B
→ unrelated documentation

Agent C
→ independent test suite
```

Bad parallel tasks:

```
Agent A
→ refactor auth module

Agent B
→ fix bugs in same auth module
```

Shared files create merge and reasoning conflicts.

10.37 — Omarchy's Swarm Layout

Current Omarchy provides:

```
tsl <count> <command>
```

to create several Tmux panes running the same command.

The official documentation describes it as useful for AI-agent swarms.

Example:

```
tsl 4 c
```

on the current documented OpenCode alias setup.

This is an advanced tool.

10.38 — A Swarm Is Not a Workflow by Itself

Four agent panes do not solve coordination.

You still need:

- task decomposition;
- separate worktrees;
- clear ownership;
- integration strategy;
- review.

Without those, a swarm can simply create four times as much confusion.

10.39 — The Best Default Is One Agent

For most development tasks:

```
one developer
+
one agent
+
one branch
+
one reviewable diff
```

is an excellent default.

Use multiple agents only when there is a measurable reason.

10.40 — A Second Agent as Reviewer

One useful pattern does not require simultaneous editing.

Example:

```
Agent A
→ implement

Agent B
→ review the final diff
```

The reviewer does not need permission to edit.

Prompt:

```
Review the current Git diff.

Do not modify files.

Look for:
- correctness;
- regressions;
- security issues;
- missing tests;
- unnecessary complexity.

Prioritize concrete findings.
```

This can provide a useful second perspective.

10.41 — Do Not Use AI Review as a Substitute for Tests

An AI reviewer can miss bugs.

Tests can miss bugs too.

Use multiple signals:

```
tests
+
diff review
+
static tools
+
AI review
+
manual validation
```

depending on project risk.

10.42 — Model Strength vs Task Complexity

Not every task needs the strongest available model.

A useful conceptual strategy:

```
simple repetitive task
→ fast/cheap model

codebase navigation
→ capable general coding model

hard architecture/debugging
→ stronger reasoning model

final review
→ independent second pass
```

Exact model names change quickly.

Do not hard-code the course around one provider.

10.43 — Cost Awareness

Current Omarchy 4 includes an agents usage panel for supported providers.

Current official documentation says it can show information such as:

- subscription plan;
- session/weekly usage;
- prepaid balance where relevant;
- token usage by day;

- token usage by model.

Use this as feedback.

If a simple task uses an expensive model unnecessarily, adjust your workflow.

10.44 — Latency Matters Too

A slower, stronger model may be appropriate for architecture.

For:

```
rename this function in five places
```

waiting a long time for maximal reasoning may be unnecessary.

Developer productivity is a balance of:

```
quality
latency
cost
reviewability
```

not just raw model strength.

10.45 — Browser Research vs Agent Research

Use the browser when:

- you want to personally inspect official docs;
- you need current product information;
- you want visual context;
- you need authoritative sources.

Use the coding agent when:

- it needs to relate external information to the codebase;
- it needs to search local files;
- it needs to propose code changes.

Do not let local coding convenience replace source verification when documentation accuracy matters.

10.46 — Prefer Official Documentation for Tool Behavior

This course itself follows:

```
current official Omarchy docs
→ current official Omarchy source
→ upstream official tool docs
```

Use the same approach while developing.

Example:

If a Hyprland option behaves differently than expected:

```
official current Hyprland docs
```

are a better source than an old Reddit comment.

The agent can help interpret docs, but source quality still matters.

10.47 — AI-Assisted Debugging Workflow

When something fails:

```
reproduce
↓
collect error
↓
identify component
↓
ask agent to analyze evidence
↓
propose smallest fix
↓
apply
↓
retest
```

Avoid:

```
paste vague symptom
→ let agent change everything
```

10.48 — Example Debug Prompt

```
The test `auth refresh rejects expired token` is failing.

Do not edit yet.

Inspect:
- the failing test;
- implementation path;
- relevant types/config;
- recent Git diff.

Explain the likely root cause and propose the smallest fix.
```

Then review before implementation.

10.49 — AI-Assisted Omarchy Troubleshooting

The same workflow applies to the workstation.

Example:

```
My terminal shortcut launches but immediately exits.

Do not modify anything.

Inspect:
- current Omarchy version;
- default terminal;
- relevant Hyprland binding;
- terminal command;
- failed user services;
- relevant logs.

Explain the evidence and propose one next step.
```

This keeps troubleshooting evidence-driven.

10.50 — Current Omarchy Crash Diagnosis

Current Omarchy can watch `systemd-coredump` for process crashes and present a crash notification.

The current official AI documentation says the crash can be handed to the configured default agent together with Omarchy's crash-diagnosis guidance.

This is a good example of:

```
system evidence
→ AI analysis
```

rather than:

```
AI guess
→ random fix
```

10.51 — Remote Development

Current Omarchy shell functions include remote-development helpers.

The current official shell-function documentation includes:

```
rsw [source] [destination]
```

to watch and synchronize files via rsync.

It also includes:

```
fip
dip
lip
```

for SSH port forwarding.

These are more advanced, but they fit naturally into the AI-development workstation.

10.52 — `rsw` : Remote Sync Watching

Current official behavior:

```
rsw [source] [destination]
```

starts a background watcher that syncs source changes to a destination using rsync.

Example from current documentation:

```
rsw ~/Work/app nyc-dev:Work/app
```

This can support a workflow where:

```
edit locally
→ files sync remotely
→ run workload on stronger/remote machine
```

10.53 — List and Stop Rsync Watchers

Current helpers:

```
lsw
```

list active watchers.

```
dsw
```

stop active watchers.

Verify current functions with:

```
type rsw
type lsw
type dsw
```

before relying on exact implementation details.

10.54 — SSH Port Forwarding

Current Omarchy shell functions include:

```
fip
```

for forwarding remote ports to localhost.

Current official example describes forwarding a remote dev server such as:

```
remote:3000
```

to:

```
localhost:3000
```

This is especially useful when browser security features require a localhost secure context.

10.55 — Remote AI Workflow

A possible advanced architecture:

```
Local Omarchy workstation
├── editor
├── Git
├── coding agent
└── browser

        ↓ rsync / SSH

Remote development machine
├── larger compute
├── dev server
├── database
└── test workloads
```

Do not build this unless you need it.

Local development is simpler.

10.56 — Long-Running Remote Work and Tmux

Tmux works extremely well over SSH.

Conceptually:

```
Omarchy
↓
SSH
↓
remote host
↓
Tmux
↓
long-running task
```

If the network drops:

```
Tmux session continues
```

Reconnect and reattach.

This is one reason Tmux remains valuable even with modern graphical terminals.

10.57 — AI and Long-Running Processes

Do not let an agent hide all long-running processes inside its own opaque session.

Prefer visible, controllable processes.

Example:

```
Terminal pane
→ dev server

Agent pane
→ implementation
```

You can then independently:

- stop;
- restart;
- inspect;
- read logs.

10.58 — Keep Development Servers Visible

Good:

```
npm run dev
```

in its own Tmux window/pane.

Why?

Because you can see:

- startup failures;
- request logs;
- rebuilds;
- crashes.

This gives both you and the agent better evidence.

10.59 — A Daily Development Routine

A practical routine:

START

```
cd ~/Projects/YOUR_PROJECT
git status
git branch --show-current
mise ls --current
```

Open/attach Tmux.

Launch:

- editor;
- agent;
- terminal.

DURING WORK

```
task
→ analyze
→ implement
→ test
→ diff
→ review
```

BEFORE COMMIT

```
git status
git diff
```

Run relevant validation.

Then:

```
git add ...
git diff --staged
git commit -m "..."
```

END

```
git status
```

Ensure you know what remains uncommitted.

Push if appropriate.

Detach Tmux if you want processes/session state to remain.

10.60 — End-of-Day Git Check

Before leaving a project:

```
git status
```

You want to know whether the repository is:

```
clean
```

or:

```
contains intentional unfinished work
```

Do not discover tomorrow that you forgot which changes were yours.

10.61 — End-of-Day Tmux Decision

If you want the workspace preserved:

```
Prefix + d
```

detach.

If the project session is finished:

- stop dev servers;
- exit panes;
- end the session.

Persistence is useful, but abandoned sessions create clutter.

10.62 — Capture Decisions Outside the Agent Session

If an important architectural decision was made during AI conversation, put it somewhere durable.

Examples:

```
README
architecture doc
issue
ADR
AGENTS.md
commit message
```

Do not depend on:

```
"I think the agent session from Tuesday explained this."
```

Agent session history is not the project's source of truth.

10.63 — A Good Project Is Self-Describing

Aim for this:

```
clone repository
↓
read README / AGENTS.md
↓
mise install
↓
dependency install
↓
run documented validation
↓
start development
```

If a new developer or agent cannot determine how to work with the project, improve the project documentation.

10.64 — Reproducibility Helps Humans and Agents Equally

When project setup is explicit:

```
new laptop
new developer
CI environment
AI coding agent
remote workstation
```

all benefit.

This is why:

- Git;
- mise;
- instructions;
- tests;

are foundational, not optional ceremony.

10.65 — Avoid Hidden Workstation Magic

Bad workstation:

```
"It works because I installed something globally six months ago."
```

Better workstation:

```
"This project declares its runtime and setup."
```

The AI-native workstation should become **less magical**, not more magical.

10.66 — The Human Approval Boundary

Define which actions should normally require deliberate approval.

A sensible baseline:

```
read files
→ low risk

edit project files
→ review afterwards

run project tests
→ generally low risk

install dependencies
→ inspect package/project first

Git commit
→ deliberate

Git push
→ deliberate

system package changes
→ deliberate

sudo
→ high scrutiny

delete data
→ high scrutiny

production actions
→ explicit approval
```

The exact agent can enforce these differently.

The principle is stable.

10.67 — Do Not Confuse Automation With Autonomy

Automation:

```
repeat a defined task
```

Autonomy:

```
decide the task and execute broadly
```

For development, automation is often easier to trust.

Example:

```
Run tests after each change
```

is clearer than:

```
Make the project better until you think it's done.
```

10.68 — Practical Exercise: Build the Daily Workspace

Create:

```
Workspace 1
→ browser

Workspace 2
→ Tmux development layout

Workspace 3
→ second terminal / dev server
```

Inside the Tmux layout:

```
editor
AI
terminal
```

Practice switching:

```
Workspace 1 ⇔ Workspace 2
```

using:

```
Super + Ctrl + Tab
```

10.69 — Practical Exercise: Bootstrap a Project

Create:

```
mkdir -p ~/Projects/ai-workflow-practice
cd ~/Projects/ai-workflow-practice
git init
```

Add a suitable runtime with mise.

Create:

```
README.md
AGENTS.md
.gitignore
```

Commit:

```
git add .
git diff --staged
git commit -m "chore: bootstrap development environment"
```

You now have a clean baseline.

10.70 — Practical Exercise: Agent Analysis → Implementation

Start:

```
git status
```

Ask your agent to inspect the project without changes.

Then ask it to implement one small task.

Afterwards:

```
git status
git diff
```

Run validation.

Commit only after reviewing.

10.71 — Practical Exercise: Separate Reviewer

After one small implementation, open a second agent session or use a different model/agent if available.

Prompt:

```
Review the current Git diff only.

Do not edit files.

Identify:
- bugs;
- regressions;
- security concerns;
- missing tests;
- unnecessary complexity.
```

Compare the review with your own assessment.

10.72 — Practical Exercise: Worktree Isolation

From a clean repository, inspect:

```
type ga
```

If the current helper matches the documented behavior, create a temporary feature worktree:

```
ga feature/agent-test
```

Verify:

```
pwd
git branch --show-current
git worktree list
```

Use this worktree for a small isolated AI task.

When finished and safe to remove, follow the current documented/helper workflow.

Do not delete worktrees blindly.

10.73 — Practical Exercise: Live Diff Layout

From a test project:

```
type tds
type hunk
```

If dependencies are available:

```
tds
```

Ask the agent to make a small file change.

Observe the diff watcher.

This shows how Omarchy can make AI-generated changes visible while they happen.

10.74 — Practical Exercise: Daily Shutdown

Before ending:

```
git status
```

Identify:

- committed work;
- uncommitted work;
- ignored/generated files.

Then decide:

```
detach Tmux
or
close the session
```

A professional workflow includes knowing the state you leave behind.

10.75 — Common AI Workflow Mistakes

Mistake 1 — Starting the Agent Before Checking the Repository

Always inspect:

```
pwd
git status
git branch --show-current
```

Mistake 2 — Asking AI to Implement Before Understanding the Task

For non-trivial work:

```
analyze first
```

Mistake 3 — One Huge Prompt for a Huge Refactor

Large vague tasks create large unreviewable diffs.

Mistake 4 — Letting the Agent Own Testing

You should still be able to run and interpret the project's validation.

Mistake 5 — Using the Strongest/Most Expensive Model for Every Tiny Task

Match model capability to task complexity.

Mistake 6 — Starting Multiple Agents Without Isolation

Use worktrees for independent parallel editing.

Mistake 7 — Assuming More Agents Means Faster Delivery

Coordination and merge costs can erase the advantage.

Mistake 8 — Hiding Everything Inside One Agent Session

Keep:

- editor;
- tests;
- Git;
- logs;

independently visible.

Mistake 9 — Relying on Agent Session History for Important Project Knowledge

Put durable knowledge in the repository.

Mistake 10 — Making the Workstation Dependent on One Agent Vendor

Keep the project, runtime, Git, editor, and terminal workflow agent-independent.

10.76 — Checkpoint

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

Workstation Layout

- [ ] I have a sensible workspace layout for research, development, infrastructure, and communication.
- [ ] I understand how Tmux fits inside a Hyprland workspace.
- [ ] I can build an editor + agent + terminal development layout.

Project Bootstrap

- [ ] I can create a new Git project.
- [ ] I can declare runtime requirements with mise.
- [ ] I can create useful README/AGENTS/.gitignore files.
- [ ] I understand how to bootstrap a cloned project safely.

AI Workflow

- [ ] I understand Analyze → Plan → Implement → Verify.
- [ ] I know when a separate planning phase is useful.
- [ ] I keep AI tasks small enough to review.
- [ ] I inspect `git diff` after changes.
- [ ] I run independent validation.

Parallel Work

- [ ] I know when a normal branch is enough.
- [ ] I understand why worktrees are useful for parallel agents.
- [ ] I understand the purpose of `tdl`, `tds`, `tdlm`, and `tsl`.
- [ ] I know a swarm is not automatically a productive workflow.

Cost and Model Use

- [ ] I understand that model strength should match task complexity.
- [ ] I understand that latency and cost matter.

- [ ] I know Omarchy currently provides usage visibility for selected providers.

Reproducibility

- [ ] I understand that important setup belongs in the project.
- [ ] I avoid hidden global workstation dependencies.
- [ ] I capture important decisions in durable project documentation.

10.77 — What You Can Now Do

After completing Module 10, you can now:

- operate Omarchy as an integrated AI-development workstation;
- structure graphical and terminal workspaces;
- bootstrap projects reproducibly;
- create AI-readable project context;
- separate analysis from implementation;
- keep AI changes reviewable through Git;
- run tests independently;
- use live diff watching;
- use branches and worktrees appropriately;
- parallelize AI work only when tasks justify it;
- make sensible model/cost trade-offs;
- preserve long-running work with Tmux;
- build local or remote development workflows;
- end each session with a known project state.

Most importantly:

> *You now have a repeatable AI-assisted software-development process rather than a collection of AI tricks.*

Module 10 Summary

Workstation layout:

```
Workspace 1
→ browser / docs

Workspace 2
→ Tmux development environment

Workspace 3
→ infrastructure / servers / logs

Workspace 4
→ communication
```

Project baseline:

```
project/
├── .git/
├── .gitignore
├── AGENTS.md
├── mise.toml
├── README.md
└── source/
```

Daily start:

```
cd ~/Projects/YOUR_PROJECT
git status
git branch --show-current
mise ls --current
```

AI workflow:

```
Analyze
↓
Plan
↓
Implement
↓
Verify
```

Development loop:

```
git status
↓
agent
↓
test
↓
git diff
↓
review
↓
stage
↓
git diff --staged
↓
commit
```

Tmux helpers currently documented by Omarchy:

```
tdl
→ editor + AI + terminal

tds
→ editor + live diff + terminal + AI

tdlm
→ one dev layout per subdirectory

tsl
→ command/agent swarm
```

Parallel editing:

```
one task
→ branch

multiple independent tasks
→ separate worktrees
```

Remote helpers currently documented by Omarchy:

```
rsw
→ watched rsync

lsw
→ list watchers

dsw
→ stop watchers

fip / dip / lip
→ SSH port-forward helpers
```

And the most important lesson:

> ***AI should accelerate a disciplined development loop, not replace it.***

MODULE 11

11

Hyprland Configuration the Omarchy Way

You now know how to use Omarchy as a development workstation.

Now we begin customizing the workstation itself.

About This Module

You now know how to use Omarchy as a development workstation.

Now we begin customizing the workstation itself.

This is where many Linux users make a costly mistake:

```
find a Hyprland config online
→ copy the entire thing
→ replace the existing setup
→ lose Omarchy integration
→ updates become harder
→ troubleshooting becomes confusing
```

We are not going to do that.

Current Omarchy 4 / Quattro is designed around a layered Hyprland configuration:

```
Omarchy defaults
        ↓
your personal override files
        ↓
your additional rules
```

The official current Omarchy configuration explicitly loads the Omarchy defaults first and then loads your personal files from:

```
~/.config/hypr/
```

That architecture is extremely important.

It means you can customize the workstation heavily without copying or permanently forking the entire Omarchy configuration.

The goal of this module is:

> ***Customize Hyprland while preserving Omarchy's upgrade path.***

What You Will Learn

By the end of Module 11, you will understand:

- how Omarchy loads Hyprland configuration;

- where the package-owned defaults live;
- where your personal Hyprland configuration lives;
- what `hyprland.lua` does;
- what `monitors.lua` controls;
- what `input.lua` controls;
- what `bindings.lua` controls;
- what `looknfeel.lua` controls;
- what `autostart.lua` controls;
- how Omarchy's Lua helpers fit over Hyprland;
- how to inspect active Hyprland state with `hyprctl`;
- how to configure monitors safely;
- how to configure input devices safely;
- how to add window rules;
- how to bind applications to workspaces;
- how to add, replace, and remove bindings;
- how to reload/test changes safely;
- how to recover from a broken config;
- why `omarchy refresh hyprland` is destructive to personal Hyprland configuration.

11.1 — The Most Important Rule

Current official Omarchy documentation draws a clear boundary.

Your files:

```
~/.config
```

Omarchy-owned files:

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

For normal customization:

> ***Edit your files, not Omarchy's package-owned files.***

Files under:

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

are installed and managed by the Omarchy package.

An update can replace them.

11.2 — Why Reading `/usr/share/omarchy` Is Still Useful

Do not normally edit Omarchy's internal files.

But reading them is extremely useful.

For example:

```
rg "scratchpad" /usr/share/omarchy/default/hypr
```

or:

```
rg "SUPER + RETURN" /usr/share/omarchy
```

This helps answer:

```
How does Omarchy currently implement this?
```

The pattern is:

```
read defaults
↓
understand behavior
↓
override in ~/.config
```

not:

```
edit defaults
```

11.3 — The Main Hyprland Configuration

Current Omarchy user configuration begins with:

```
~/.config/hypr/hyprland.lua
```

Inspect it:

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

The current official Quattro template follows this structure:

```
dofile((os.getenv("OMARCHY_PATH") or "/usr/share/omarchy") .. "/default/hypr/
bootstrap.lua")

require("default.hypr.omarchy")

require("hypr.monitors")
require("hypr.input")
require("hypr.bindings")
require("hypr.looknfeel")
require("hypr.autostart")

require("default.hypr.toggles")
```

The exact comments or surrounding lines may change, but the important load order is deliberate.

11.4 — Understand the Load Order

Conceptually:

```
1. Omarchy bootstrap
        ↓
2. Omarchy Hyprland defaults
        ↓
3. your monitors.lua
        ↓
4. your input.lua
        ↓
5. your bindings.lua
        ↓
6. your looknfeel.lua
        ↓
7. your autostart.lua
        ↓
8. Omarchy dynamic toggle flags
        ↓
9. any extra personal config below
```

Your files are loaded **after** Omarchy's defaults.

That gives your own configuration the opportunity to override default behavior.

11.5 — Why This Architecture Is Better

Suppose Omarchy improves its default configuration in an update.

If you copied the entire default configuration six months ago:

```
your giant copied config
→ remains frozen
→ misses improvements
```

With small overrides:

```
new Omarchy defaults
+
your small personal changes
```

You benefit from upstream improvements while preserving your preferences.

This is the correct long-term model.

11.6 — Your Main Hyprland Files

Current official Omarchy documentation identifies these key files:

```
~/.config/hypr/hyprland.lua
```

Main config and load order.

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

Monitor mode, scaling, positioning, and related display rules.

```
~/.config/hypr/input.lua
```

Keyboard, mouse, and trackpad configuration.

```
~/.config/hypr/bindings.lua
```

Your bindings and binding overrides.

```
~/.config/hypr/looknfeel.lua
```

Gaps, borders, animations, and visual/layout behavior.

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

Your session startup processes.

11.7 — Inspect the Directory

Run:

```
ls -la ~/.config/hypr
```

Then:

```
lt ~/.config/hypr
```

if your current Omarchy shell provides the `lt` helper.

The point is to know exactly which files are yours before editing anything.

11.8 — Prefer the Omarchy Setup Menu for Common Configs

Current official Omarchy documentation allows core config files to be opened through:

```
Super + Space
→ Setup
```

Examples include:

```
Setup → Monitors
Setup → Keybindings
Setup → Input
Setup → Config
```

One advantage of opening config this way is that Omarchy can perform relevant reload/restart behavior when you exit the editor.

This is a good beginner workflow.

11.9 — The Lua Configuration Layer

Current Omarchy 4 uses Lua-based helper APIs around Hyprland configuration.

You will see helpers such as:

```
hl.config(...)
hl.monitor(...)
hl.bind(...)
hl.unbind(...)
o.bind(...)
o.rebind(...)
o.window(...)
o.launch_on_start(...)
```

You do not need to understand the implementation of these helper libraries.

Treat them as Omarchy's supported configuration interface.

11.10 — Why Use Omarchy Helpers?

You could try to bypass Omarchy's structure and write raw Hyprland configuration everywhere.

But the Omarchy helper layer gives you:

- consistent syntax;
- integration with Omarchy defaults;
- easier readable configuration;
- a configuration style that matches the current project.

Our course therefore prefers:

> ***Current Omarchy configuration conventions first, raw Hyprland internals when genuinely needed.***

11.11 — Inspect Active Hyprland State

Before changing configuration, inspect reality.

Current Hyprland provides:

```
hyprctl monitors all
```

for monitor state.

Use:

```
hyprctl activewindow
```

for the currently focused window.

You can also inspect configuration errors:

```
hyprctl configerrors
```

These commands are extremely valuable.

11.12 — The Configuration Loop

Use this pattern whenever changing Hyprland:

```
inspect current state
        ↓
edit one small thing
        ↓
reload/apply
        ↓
check config errors
        ↓
verify behavior
```

Not:

```
change 20 things
↓
desktop behaves strangely
↓
no idea which change caused it
```

11.13 — Monitor Configuration

Current official Omarchy documentation uses:

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

for persistent monitor configuration.

Open it:

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

or:

```
Super + Space
→ Setup
→ Monitors
```

11.14 — Identify Monitor Names First

Run:

```
hyprctl monitors all
```

Look for names such as:

```
DP-1
DP-2
HDMI-A-1
eDP-1
```

Do not guess the output name.

Monitor names depend on:

- GPU;
- connection type;
- port;
- laptop hardware.

11.15 — Real-World Monitor Example

On the workstation used to develop this course, the monitor is:

```
DP-2
```

and the preferred configuration is:

```
2560x1440 @ 144 Hz
```

The personal Omarchy monitor rule is:

```
hl.monitor({
  output = "DP-2",
  mode = "2560x1440@144",
  position = "0x0",
  scale = 1
})
```

This is a good example because the display initially worked while running at a lower refresh rate.

Nothing looked obviously broken.

Verification exposed the problem.

11.16 — Never Copy a Monitor Name From Someone Else

This:

```
output = "DP-2"
```

is correct only if **your** monitor is actually `DP-2`.

Always run:

```
hyprctl monitors all
```

first.

11.17 — Monitor Scaling

Current official Omarchy monitor documentation discusses scaling extensively.

The correct scale depends on:

- physical display size;
- resolution;
- pixel density;
- eyesight;
- application behavior.

Broadly:

```
1080p / 1440p desktop monitor
→ often scale 1

very high-DPI display
→ often higher or fractional scaling
```

Do not blindly copy another user's scaling value.

11.18 — Fractional Scaling

Current Omarchy documentation includes recommendations for fractional scaling on displays such as 4K screens that are below extremely high "retina-class" pixel density.

You may encounter combinations such as:

```
Omarchy/GDK scale
+
Hyprland monitor scale
```

Use the current official monitor documentation for the exact recommendation for your display class.

Do not freeze these recommendations in your memory forever; they can evolve.

11.19 — Multiple Monitors

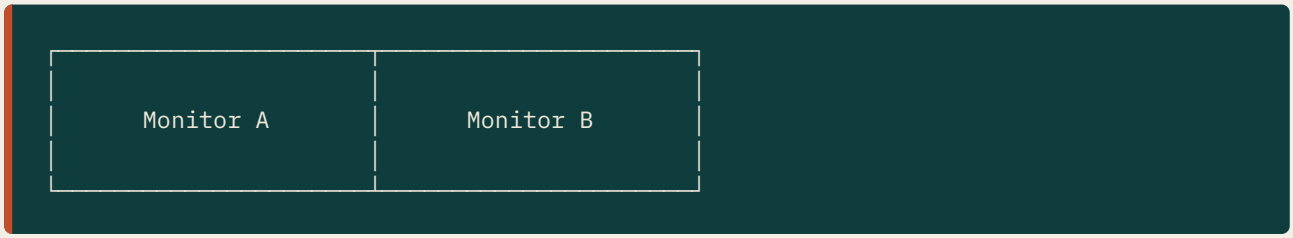
Hyprland can position monitors using coordinates.

Conceptually:

```
Monitor A
position 0x0

Monitor B
position 2560x0
```

means Monitor B begins to the right of a 2560-pixel-wide Monitor A.

Example layout:



Use the current Hyprland monitor documentation for advanced multi-monitor positioning rules.

11.20 — Laptop + External Display

Current official Omarchy monitor documentation includes behavior for laptop external displays.

Current Quattro automatically extends to an external display by default.

The documentation also describes current Omarchy hardware triggers for:

- mirroring;
- toggling internal display behavior.

Exact shortcuts should be verified with:

```
Super + K
```

because hardware bindings may evolve.

11.21 — Verify After a Monitor Change

Run:

```
hyprctl monitors all
```

Confirm:

- correct output;
- resolution;
- refresh rate;
- scale;
- position.

Then:

```
hyprctl configerrors
```

If there are errors, fix those before adding more configuration.

11.22 — Input Configuration

Current official Omarchy documentation uses:

```
~/.config/hypr/input.lua
```

for:

- keyboard layout;
- keyboard options;
- repeat behavior;
- mouse sensitivity;
- acceleration;
- trackpad configuration.

Open:

```
Super + Space
→ Setup
→ Input
```

or:

```
nvim ~/.config/hypr/input.lua
```

11.23 — Current Input Configuration Pattern

Current official documentation shows a pattern such as:

```
hl.config({
  input = {
    kb_layout = "us",
    repeat_rate = 40,
    repeat_delay = 600,
    sensitivity = 0.35,

    touchpad = {
      natural_scroll = true,
    },
  },
})
```

Treat this as a structural example.

Use settings appropriate to your hardware and preferences.

11.24 — Keyboard Layout

Example:

```
kb_layout = "us"
```

Multiple layouts are possible.

The official current documentation shows examples of multiple layouts and keyboard options.

If you need multiple languages, use the current Omarchy/Hyprland input documentation rather than guessing syntax.

11.25 — Keyboard Options

Omarchy may include keyboard options for behaviors such as:

- Compose key;
- layout switching;
- Caps Lock behavior.

During our real workstation setup, the user wanted ordinary Caps Lock behavior.

The personal config included:

```
kb_options = ""
```

This removed the inherited option behavior for that machine.

Do not copy this unless you actually want to clear those options.

11.26 — Mouse Sensitivity

A personal example from the course workstation:

```
sensitivity = -0.3
```

This reduced pointer sensitivity.

The current official Omarchy input documentation notes that:

```
0
```

is the default baseline.

Your correct value is personal.

11.27 — Mouse Acceleration

For some users, particularly gaming or precision desktop work, a flat acceleration profile may feel better.

Our real workstation used:

```
accel_profile = "flat"
```

This is a preference, not an Omarchy requirement.

Try changes gradually.

11.28 — Trackpad Configuration

Laptop users may configure options such as:

```
touchpad = {
  natural_scroll = true,
}
```

Additional current Hyprland options can control:

- tapping;
- click behavior;
- scroll direction;

- sensitivity.

Use the current official Omarchy input documentation plus upstream Hyprland documentation for advanced settings.

11.29 — Keybindings

Personal bindings belong in:

```
~/.config/hypr/bindings.lua
```

Current official Omarchy documentation explicitly directs users here for adding or replacing bindings.

Open:

```
Super + Space
→ Setup
→ Keybindings
```

or:

```
nvim ~/.config/hypr/bindings.lua
```

11.30 — Add a New Binding

Current Omarchy provides:

```
o.bind(...)
```

A conceptual example:

```
o.bind("SUPER + ALT + RETURN", "Tmux", { omarchy = "terminal-tmux" })
```

The workstation used while building this course contains this personal binding.

The important structure is:

```
key combination
description
action
```

11.31 — Replace an Existing Binding

Current Quattro source provides:

```
o.rebind(...)
```

for replacing an existing binding.

The current official dotfiles source explains that `o.rebind` removes the previous binding before adding its replacement.

Conceptual example:

```
o.rebind("SUPER + SHIFT + O", "Joplin", "joplin-desktop")
```

This is cleaner than leaving two actions competing for the same shortcut.

11.32 — Remove a Binding

Current Omarchy supports:

```
hl.unbind(...)
```

Example:

```
hl.unbind("SUPER + SHIFT + O")
```

Use this when you want a default shortcut removed without replacing it.

11.33 — Current Documentation Can Differ Slightly Between Published Manual and Source

Because Omarchy evolves quickly, you may occasionally see the published manual demonstrate:

```
hl.unbind(...)
o.bind(...)
```

while the current Quattro source also provides:

```
o.rebind(...)
```

This is exactly why our documentation policy is:

```
current official manual
+
current official Quattro source
+
installed system
```

When in doubt, inspect the installed/current implementation.

11.34 — Disable All Default Omarchy Bindings

The current official `hyprland.lua` template contains an advanced option:

```
-- omarchy_default_bindings = false
```

If enabled before the default configuration loads, it disables Omarchy's default bindings.

The same template also currently provides:

```
-- omarchy_preinstalled_bindings = false
```

to disable preinstalled app/web-app bindings while keeping the core window-manager bindings.

These are advanced options.

STEVINATOR RECOMMENDATION

> *Do **not** disable all Omarchy defaults while learning the system.*

Override individual bindings first.

11.35 — Verify Bindings Before Editing

Current Omarchy gives you:

```
Super + K
```

for keybinding discovery.

You can also use:

```
omarchy menu keybindings
```

Before adding a shortcut, check whether that combination is already used.

11.36 — Keyboard Hardware Can Lie to You

Sometimes the config is correct but the physical keyboard sends a different key.

This is especially relevant for:

- compact keyboards;
- Fn layers;
- non-standard layouts;
- vendor remapping software.

Use:

```
wev
```

to inspect the actual Wayland key event.

This saved time during the real workstation setup used for this course.

11.37 — Window Rules

Hyprland can apply rules to specific application windows.

Current Omarchy's `hyprland.lua` template includes a personal example:

```
o.window("qemu", { workspace = "5" })
```

Conceptually:

```
when window class matches qemu
→ place it on workspace 5
```

This is powerful for building predictable workspaces.

11.38 — Identify the Actual Window Class

Never guess.

Focus the application and run:

```
hyprctl activewindow
```

Look for:

```
class:
```

Example from the workstation used in this course:

```
com.mitchellh.ghostty
```

That is the current Ghostty window class on that machine.

The visible application title is not necessarily the correct rule identifier.

11.39 — Example Window Rule

Suppose:

```
hyprctl activewindow
```

reports:

```
class: example-app
```

A conceptual rule might be:

```
o.window("example-app", { workspace = "4" })
```

Now the application can be assigned to workspace 4.

Verify behavior after adding it.

11.40 — Do Not Over-Automate Window Placement

It is tempting to create rules for every application.

That can make the desktop feel rigid.

Good candidates:

- virtual machine;
- dedicated monitoring app;
- communication app;

- application that always belongs in one workflow.

Poor candidate:

- every terminal;
- every browser window;
- every transient utility.

Use workspace rules when they remove recurring friction.

11.41 — Real Course Decision: Ghostty Rule

During the workstation setup we identified Ghostty's class:

```
com.mitchellh.ghostty
```

We deliberately **did not** create a rule forcing every Ghostty window to one workspace.

Why?

Because terminal windows can serve many purposes across many workspaces.

This is a good example of not automating something simply because you can.

11.42 — Workspace Rules

Hyprland can also apply rules to workspaces.

For example, advanced configurations may assign:

- a workspace to a monitor;
- layout behavior;
- special properties.

Current Omarchy monitor documentation says workspace-to-monitor rules belong conceptually with the monitor setup.

Use upstream current Hyprland workspace-rule documentation for advanced rules.

11.43 — Verify Workspace Behavior

After configuring a rule:

```
hyprctl workspaces
```

and:

```
hyprctl monitors all
```

can help you inspect current workspace/monitor state.

Do not rely only on what you remember configuring.

11.44 — Look and Feel

Current Omarchy uses:

```
~/.config/hypr/looknfeel.lua
```

for Hyprland visual/layout overrides.

Current official dotfiles documentation says it controls areas such as:

- gaps;
- borders;
- animations;
- other look/feel behavior.

Open:

```
nvim ~/.config/hypr/looknfeel.lua
```

Do not immediately replace the entire visual configuration.

11.45 — Themes vs `looknfeel.lua`

These are related but not identical.

Omarchy themes can control broad visual styling.

`looknfeel.lua` is your Hyprland-level override layer.

Conceptually:

```
Omarchy theme
→ coordinated colors/background/etc.

looknfeel.lua
→ your Hyprland behavior/visual overrides
```

We will cover themes in Module 13.

11.46 — Autostart

Current Omarchy uses:

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

for processes that should start with your graphical session.

Current official documentation provides:

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

as the user-facing pattern.

11.47 — What Belongs in Autostart?

Possible examples:

- sync daemon;
- local helper;
- communication app;
- personal script.

But ask:

> *Does this really need to start every login?*

Unnecessary autostart programs increase:

- startup complexity;
- resource use;
- troubleshooting surface.

11.48 — Autostart vs Systemd Service

These are not the same.

Use autostart when something belongs to:

```
your graphical session
```

A systemd service may be better when the process:

- needs restart policy;
- must run independently of Hyprland;
- has service dependencies;
- needs structured logging.

We will explore services and hooks later.

11.49 — Autostart vs Omarchy Hooks

Also different:

```
autostart
→ run when your session starts

hook
→ run when a particular Omarchy event occurs
```

Example:

```
post-update
→ hook

launch chat app each session
→ autostart
```

Hooks belong to Module 14.

11.50 — Reloading Hyprland Configuration

Hyprland normally supports dynamic configuration reloads.

For current Omarchy, using the Setup menu for supported config files is a good option because Omarchy can perform the required reload/restart action when the editor exits.

If editing manually, check the current Hyprland/Omarchy behavior after saving.

Do not assume every component reacts identically.

11.51 — Check for Configuration Errors

After editing:

```
hyprctl configerrors
```

If the output is empty/no-errors, that is a good sign.

If it reports errors:

```
stop
↓
fix the error
↓
do not keep stacking more changes
```

This should become automatic.

11.52 — Inspect Current Window State

For window rules:

```
hyprctl activewindow
```

Useful fields can include:

```
class
title
workspace
monitor
pid
xwayland
```

This is much better than guessing based on the app name shown in the UI.

11.53 — Inspect Monitor State

For monitor changes:

```
hyprctl monitors all
```

Compare:

```
configured mode
vs
active mode
```

Do not consider the change successful until the active state matches your intention.

11.54 — Inspect Input Devices

For advanced troubleshooting, current Hyprland provides tools to inspect devices.

You can also use:

```
wev
```

for actual keyboard/mouse events.

This is especially useful when:

```
binding looks correct
but key does nothing
```

11.55 — Reload vs Restart vs Reboot

Do not reboot for every Hyprland problem.

Preferred escalation:

```
save/reload config
        ↓
restart affected component if necessary
        ↓
restart Hyprland/session if necessary
        ↓
reboot only when genuinely needed
```

Omarchy provides component-level restart helpers we will cover in detail later.

11.56 — Back Up Before Major Changes

Before substantial customization, your personal Hyprland config should be version-controlled.

At minimum:

```
~/.config/hypr/hyprland.lua
~/.config/hypr/monitors.lua
~/.config/hypr/input.lua
~/.config/hypr/bindings.lua
~/.config/hypr/looknfeel.lua
~/.config/hypr/autostart.lua
```

should be part of your dotfiles strategy once intentionally modified.

We will build that properly in Module 22.

11.57 — `omarchy refresh config`

Current Omarchy includes configuration refresh functionality.

The important behavior is:

```
refresh a user config
→ backup the user's current version
→ replace it with the shipped Omarchy version
```

This can be valuable for recovery.

It is not something to run casually on working customized files.

11.58 — `omarchy refresh hyprland` Is Destructive to Personal Hyprland Files

Current official Quattro source for:

```
omarchy refresh hyprland
```

shows that it refreshes/overwrites the user Hyprland Lua configs including:

```
hyprland.lua
autostart.lua
bindings.lua
input.lua
looknfeel.lua
monitors.lua
```

That means:

> ***Do not run `omarchy refresh hyprland` merely because one shortcut or monitor setting is wrong.***

It is a broad recovery action.

11.59 — `omarchy reinstall configs` Is Even Broader

Current Omarchy provides:

```
omarchy reinstall configs
```

to reset user configuration.

This is destructive to custom configuration.

Use it only when you intentionally want a reset/recovery.

Our later troubleshooting hierarchy will be:

```
inspect
↓
fix specific line/file
↓
reload
↓
refresh one config
↓
refresh Hyprland
↓
reinstall configs
```

Use the smallest recovery action that solves the problem.

11.60 — Practical Exercise: Map Your Hyprland Config

Run:

```
cd ~/.config/hypr
```

Then:

```
ls -la
```

Open:

```
bat hyprland.lua
```

Identify:

```
Omarchy bootstrap
Omarchy defaults
personal config requires
toggle config
```

Do not edit yet.

11.61 — Practical Exercise: Verify Monitor Reality

Run:

```
hyprctl monitors all
```

Write down:

```
output name
resolution
refresh rate
scale
position
```

Then open:

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

Compare configuration with reality.

11.62 — Practical Exercise: Make One Safe Monitor Change

Only if your display actually needs a change, modify one property.

Example structure:

```
hl.monitor({
  output = "YOUR_OUTPUT",
  mode = "YOUR_RESOLUTION@YOUR_REFRESH",
  position = "0x0",
  scale = 1
})
```

Save.

Then:

```
hyprctl configerrors
```

and:

```
hyprctl monitors all
```

If your display is already correct, do **not** change it just for the exercise.

Inspection alone is valid.

11.63 — Practical Exercise: Inspect Input

Open:

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

Then read current device behavior.

If you want to test pointer sensitivity, change only the sensitivity value.

Afterwards:

```
hyprctl configerrors
```

If the original setting was already good, leave it unchanged.

11.64 — Practical Exercise: Identify a Window Class

Open an application.

Focus it.

Run:

```
hyprctl activewindow
```

Record:

```
class
title
workspace
```

Repeat with your terminal.

Observe that the human-visible app name and Hyprland class may differ.

11.65 — Practical Exercise: Add a Harmless Window Rule

Choose an application where automatic placement makes sense.

Example structure:

```
o.window("ACTUAL_CLASS", { workspace = "5" })
```

Add it below the existing personal-load section in:

```
~/.config/hypr/hyprland.lua
```

Save.

Run:

```
hyprctl configerrors
```

Close and reopen the application.

Verify the behavior.

Remove the rule afterwards if you do not actually want it.

11.66 — Practical Exercise: Inspect Your Bindings File

Open:

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

Then open:

```
Super + K
```

Compare:

```
personal overrides
vs
effective binding list
```

This demonstrates how your small file participates in a much larger default configuration.

11.67 — Practical Exercise: Find a Default Binding

Suppose you want to know where Omarchy defines a scratchpad binding.

Search:

```
rg "scratchpad" /usr/share/omarchy/default/hypr
```

or search the broader current Omarchy tree:

```
rg "Scratchpad" /usr/share/omarchy
```

Read the implementation.

Do not edit it.

This is one of the best ways to learn how the current system really works.

11.68 — Practical Exercise: Verify Key Hardware

If a keybinding does not respond:

```
wev
```

Press the intended key.

Check whether the expected event appears.

Exit `wev` when done.

This separates:

```
keyboard problem
```

from:

```
Hyprland binding problem
```

11.69 — Practical Exercise: Safe Recovery Thought Experiment

Imagine:

```
bindings.lua
```

now contains a syntax error.

What should you do?

Correct order:

```
1. inspect hyprctl configerrors
2. reopen bindings.lua
3. undo/fix the last change
4. save
5. check configerrors again
```

Not:

```
omarchy reinstall configs
```

for one typo.

11.70 — Common Hyprland Configuration Mistakes

Mistake 1 — Editing `/usr/share/omarchy`

Those files belong to Omarchy and can be overwritten by package updates.

Mistake 2 — Copying an Entire Stranger's Hyprland Config

You lose the advantages of Omarchy's layered configuration.

Mistake 3 — Guessing Monitor Output Names

Run:

```
hyprctl monitors all
```

Mistake 4 — Assuming a Display Is Correct Because It Shows an Image

Verify resolution and refresh rate.

Mistake 5 — Guessing Window Classes

Run:

```
hyprctl activewindow
```

Mistake 6 — Changing Five Config Files Before Testing

Change one thing at a time.

Mistake 7 — Ignoring `hyprctl configerrors`

A config syntax problem should be resolved before further customization.

Mistake 8 — Creating Rules for Every Application

Automation can become rigidity.

Mistake 9 — Assuming a Keybinding Problem Is Always Hyprland

Check actual hardware events with:

```
wev
```

Mistake 10 — Running `omarchy refresh hyprland` for a Tiny Problem

That action replaces the user's Hyprland configuration files with the shipped defaults.

Use targeted repair first.

11.71 — Checkpoint

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

Architecture

- [ ] I understand the Hyprland configuration load order.
- [ ] I know the difference between `/usr/share/omarchy` and `~/.config/hypr`.
- [ ] I understand why small overrides are better than copied full configs.

Files

- [ ] I know what `hyprland.lua` does.
- [ ] I know what `monitors.lua` does.
- [ ] I know what `input.lua` does.
- [ ] I know what `bindings.lua` does.
- [ ] I know what `looknfeel.lua` does.
- [ ] I know what `autostart.lua` does.

Inspection

- [ ] I can use `hyprctl monitors all`.
- [ ] I can use `hyprctl activewindow`.
- [ ] I can use `hyprctl configerrors`.
- [ ] I know when to use `wev`.

Configuration

- [ ] I understand `hl.monitor(...)`.
- [ ] I understand `o.bind(...)` conceptually.
- [ ] I understand `o.rebind(...)` conceptually.
- [ ] I understand `hl.unbind(...)`.
- [ ] I understand `o.window(...)`.
- [ ] I understand `o.launch_on_start(...)`.

Safety

- [ ] I test one change at a time.
- [ ] I do not edit package-owned defaults.
- [ ] I understand that `omarchy refresh hyprland` is a broad reset action.
- [ ] I understand that `omarchy reinstall configs` is destructive to personal config.

11.72 — What You Can Now Do

After completing Module 11, you can now:

- understand Omarchy's Hyprland configuration architecture;
- safely customize monitors;
- customize keyboard and pointer behavior;
- add or replace keybindings;
- remove unwanted bindings;
- identify real application window classes;
- create intentional window/workspace rules;
- configure graphical-session autostart;
- inspect active Hyprland state;
- validate configuration changes;
- diagnose input-event problems;
- recover from configuration mistakes without wiping your setup;
- preserve Omarchy's upgrade path while customizing your workstation.

Most importantly:

> *You can now customize Hyprland **as an Omarchy user**, rather than turning Omarchy into an unmanaged personal fork.*

Module 11 Summary

Configuration hierarchy:

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

~/.config/hypr
→ your overrides
```

Current load order:

```
bootstrap
↓
default.hypr.omarchy
↓
monitors
↓
input
↓
bindings
↓
looknfeel
↓
autostart
↓
toggles
↓
additional personal rules
```

Key files:

```
hyprland.lua
→ main load structure

monitors.lua
→ display configuration

input.lua
→ keyboard/mouse/trackpad

bindings.lua
→ shortcuts

looknfeel.lua
→ gaps/borders/animations/etc.

autostart.lua
→ session startup processes
```

Core inspection tools:

```
hyprctl monitors all
```

```
hyprctl activewindow
```

```
hyprctl configerrors
```

```
wev
```

Current Omarchy helper concepts:

```
hl.monitor(...)
o.bind(...)
o.rebind(...)
hl.unbind(...)
o.window(...)
o.launch_on_start(...)
```

Recovery hierarchy:

```
fix specific config
↓
reload
↓
refresh specific config
↓
refresh Hyprland
↓
reinstall configs
```

And the most important rule:

> ***Override Omarchy. Do not overwrite Omarchy.***

MODULE 12

12

Custom Keybindings on Omarchy

You already know how to **use** Omarchy's keyboard-first workflow.

Now you are going to make that workflow yours.

About This Module

You already know how to **use** Omarchy's keyboard-first workflow.

Now you are going to make that workflow yours.

This is one of the best parts of Omarchy.

You can add shortcuts for:

- applications;
- terminal workflows;
- scripts;
- AI tools;
- workspace actions;
- personal utilities;

without replacing Omarchy's whole Hyprland configuration.

Current Omarchy 4 / Quattro uses:

```
~/.config/hypr/bindings.lua
```

for your own bindings and overrides.

The current official Omarchy dotfiles documentation also provides a clean distinction between three actions:

```
o.bind(...)
→ add a new binding

o.rebind(...)
→ replace an existing binding

hl.unbind(...)
→ remove a binding
```

That gives us a simple and upgrade-friendly customization model.

This module is intentionally practical.

The goal is not to teach every Hyprland binding feature.

The goal is:

> ***Create reliable shortcuts, avoid collisions, debug keyboard problems correctly, and preserve Omarchy's defaults.***

What You Will Learn

By the end of Module 12, you will understand:

- where personal Omarchy keybindings live;
- how current Omarchy keybindings are discovered;
- how to check whether a key combination is already used;
- how to add a new shortcut;
- how to replace a default shortcut;
- how to disable a shortcut;
- the difference between `o.bind`, `o.rebind`, and `hl.unbind`;
- why printed shortcut notation and Lua notation may differ;
- how to verify actual key events with `wev`;
- how compact keyboards and Fn layers can cause false troubleshooting signals;
- how to bind commands and Omarchy actions;
- how to test bindings safely;
- how to diagnose a binding that does nothing;
- how to recover from a broken `bindings.lua`;
- how to design a personal shortcut system that remains understandable.

12.1 — Where Your Keybindings Live

Current official Omarchy documentation identifies:

```
~/.config/hypr/bindings.lua
```

as the file for:

> *your own keybindings and overrides of the defaults.*

Open it:

```
nvim ~/.config/hypr/bindings.lua
```

or through Omarchy:

```
Super + Space
→ Setup
→ Keybindings
```

Using the Omarchy Setup path is convenient because Omarchy can handle relevant reload behavior after you exit the editor.

12.2 — Do Not Edit Default Binding Files

Omarchy's default bindings live under its package-owned configuration.

Conceptually:

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

Those files belong to Omarchy.

They are useful to:

```
read
search
understand
```

but not for normal personal customization.

Your changes belong in:

```
~/.config/hypr/bindings.lua
```

12.3 — The Effective Binding Model

Think of the final binding set like this:

```
Omarchy default bindings
        ↓
your removals
        ↓
your replacements
        ↓
your additions
        ↓
effective keyboard map
```

This lets you preserve the majority of Omarchy's defaults while changing only the parts that matter to you.

12.4 — Discover Before You Customize

Before adding or replacing anything, first inspect the current binding map.

The current official Omarchy hotkey documentation recommends:

```
Super + K
```

for the main keyboard-binding view.

This is the fastest visual method.

You can also use the CLI:

```
omarchy menu keybindings
```

Current versions may also expose a printable form:

```
omarchy menu keybindings --print
```

Check current syntax with:

```
omarchy menu keybindings --help
```

12.5 — Why Discovery Comes First

Suppose you want:

```
Super + Shift + D
```

for a custom development command.

You add it.

But Omarchy already uses that key combination.

Now one of three things may happen:

- your binding replaces/conflicts with the old behavior;
- both definitions exist in an unexpected way;
- behavior becomes version-dependent or confusing.

Better:

```
1. search current bindings
2. identify existing action
3. decide whether to add, replace, or choose another key
```

12.6 — Search the Binding List

Use:

```
Super + K
```

and search by:

- key combination;
- application;
- action name.

From the terminal, you can also inspect the printed list.

Example:

```
omarchy menu keybindings --print | grep -i terminal
```

or:

```
omarchy menu keybindings --print | grep -i tmux
```

12.7 — Important Syntax Detail: Display Format vs Lua Format

Current Omarchy 4 users should be aware of a subtle issue.

The printable keybinding list may display combinations like:

```
SUPER SHIFT + S
```

while the Lua binding APIs expect the style:

```
SUPER + SHIFT + S
```

Do **not** blindly paste the printed visual form into:

```
o.bind(...)
```

or:

```
hl.unbind(...)
```

Use the Lua syntax expected by the current configuration:

```
MODIFIER + MODIFIER + KEY
```

Example:

```
SUPER + SHIFT + S
```

This is a valuable "verify, don't assume" lesson.

12.8 — Adding a New Binding with `o.bind`

Current official Omarchy documentation recommends:

```
o.bind(...)
```

to add a binding.

Conceptual example:

```
o.bind("SUPER + ALT + RETURN", "Tmux", { omarchy = "terminal-tmux" })
```

This structure means:

```
key combination
description
action
```

12.9 — Real Course Example: Tmux Shortcut

The workstation used while developing this course contains this personal binding:

```
o.bind("SUPER + ALT + RETURN", "Tmux", { omarchy = "terminal-tmux" })
```

The intention is simple:

```
Super + Alt + Return
→ launch the Omarchy Tmux terminal
```

This is a good custom-binding example because:

- the action has a clear purpose;
- the description is readable;
- it uses the Omarchy helper layer;
- it belongs in the user config.

12.10 — Use a Clear Description

Bad:

```
o.bind("SUPER + ALT + X", "Thing", "some-command")
```

Better:

```
o.bind("SUPER + ALT + X", "Open project dashboard", "project-dashboard")
```

Descriptions matter because they can appear in keybinding discovery.

Your shortcut file should be understandable six months later.

12.11 — Bind a Normal Command

A binding can launch a command.

Conceptual example:

```
o.bind("SUPER + ALT + B", "Open btop", "btop")
```

For terminal applications, whether the command needs a terminal wrapper depends on how the action is launched.

Prefer current Omarchy launch helpers when a built-in integration exists.

12.12 — Prefer Omarchy Helpers for Omarchy Actions

If Omarchy already has a high-level command/helper for the behavior, prefer that over reproducing low-level internals.

Example concept:

```
launch terminal
→ use Omarchy terminal helper

restart audio
→ use Omarchy restart helper

open configured editor
→ use Omarchy default/editor integration
```

This reduces coupling to implementation details.

12.13 — Replace a Binding with `o.rebind`

Current official Omarchy dotfiles documentation provides:

```
o.rebind(...)
```

for replacing an existing shortcut.

Example from the current official documentation:

```
o.rebind("SUPER + SHIFT + O", "Joplin", "joplin-desktop")
```

The current docs explain that `o.rebind` removes the existing binding first, then adds the replacement.

This is cleaner than manually unbinding and rebinding when a direct replacement is your intent.

12.14 — When to Use `o.rebind`

Use `o.rebind` when:

```
same shortcut
→ different action
```

Example:

```
Super + Shift + O

Old:
→ Obsidian

New:
→ Joplin
```

That is a replacement.

12.15 — Remove a Shortcut with `hl.unbind`

Current official Omarchy documentation uses:

```
hl.unbind(...)
```

to remove a binding without replacing it.

Example:

```
hl.unbind("SUPER + SHIFT + O")
```

Use this when you want:

```
shortcut
→ nothing
```

rather than:

```
shortcut
→ new action
```

12.16 — `bind`, `rebind`, or `unbind`?

Use this decision tree:

```
Does the key combination already exist?
        ↓
no
→ o.bind

yes
        ↓
Do I want a different action on the same key?
        ↓
yes
→ o.rebind

no, I want it disabled
→ hl.unbind
```

This keeps intent explicit.

12.17 — Do Not Disable Omarchy's Entire Shortcut System Too Early

The current Omarchy Hyprland template includes advanced flags that can disable:

- all default Omarchy bindings;
- preinstalled app/web-app bindings.

These are advanced customization options.

For this course:

> *Keep the default binding system and override individual shortcuts.*

Why?

Because Omarchy's keyboard-driven workflow is one of its strongest integrations.

Replacing everything immediately creates unnecessary maintenance.

12.18 — A Good Personal Shortcut Strategy

A useful approach is to preserve categories.

For example:

```
Super
→ window/workspace management

Super + Shift
→ applications

Super + Ctrl
→ system panels/actions

Super + Alt
→ custom/developer workflows
```

You do not have to follow this exactly.

But a consistent mental system is much better than random shortcuts.

12.19 — Avoid Modifier Soup

Technically, you can create shortcuts like:

```
Super + Ctrl + Shift + Alt + P
```

But that does not make them good.

A good shortcut should be:

- physically comfortable;
- easy to remember;
- unlikely to conflict;
- semantically related to the action.

Reserve complex chords for infrequent or risky actions.

12.20 — Keyboard Layout Matters

A shortcut that feels natural on:

```
US ANSI
```

may be awkward on:

```
German ISO
```

or another layout.

Symbols can move.

Some keys require:

- Shift;
- AltGr;
- Fn.

Therefore shortcut design should match the learner's real keyboard.

12.21 — Compact Keyboards Add Another Layer

The course workstation uses a compact 75% keyboard.

On compact keyboards, functions such as:

- Print Screen;
- Home;
- End;
- Insert;
- media controls;

may live behind:

```
Fn layers
```

The operating system does not necessarily see the label printed on the key.

It sees the event produced after the keyboard firmware/Fn layer processes it.

12.22 — Use `wev` to See What the System Receives

When a shortcut should work but does not:

```
wev
```

Press the key combination.

`wev` shows Wayland input events.

You can inspect:

- key code;

- key symbol;
- modifiers;
- press/release behavior.

This separates:

```
physical keyboard / firmware issue
```

from:

```
Hyprland binding issue
```

12.23 — Real Course Example: Print Screen

The course workstation's compact keyboard did not have a normal dedicated Print Screen key.

The Fn layer had to be configured so that the operating system actually received:

```
Print
```

Only then could the Omarchy screenshot binding work as expected.

That was not a Hyprland problem.

It was an input-event problem.

This distinction is extremely important.

12.24 — Test the Key Before Editing the Config

If a shortcut involving an unusual key fails:

```
1. run wev
2. press the key
3. confirm the system receives what you expect
4. only then inspect Hyprland bindings
```

This avoids wasting time editing a correct configuration.

12.25 — Physical Key Codes

Hyprland can also work with physical key codes.

You may see syntax such as:

```
code:113
```

in advanced configurations.

This can be useful when:

- layouts differ;
- symbolic names are unreliable;
- you intentionally want a physical-key mapping.

But it is more advanced and less readable.

Prefer named keys when they work reliably.

12.26 — Press vs Release Actions

Some bindings may execute:

```
when key is pressed
```

while others can execute:

```
when key is released
```

Current Omarchy Lua bindings can support binding options for behavior such as release actions.

This can be useful for tools such as:

- dictation;
- push-to-talk;
- temporary overlays.

Do not use advanced binding options until you have a real use case.

12.27 — Mouse Bindings

Hyprland also supports mouse events.

Current Omarchy already uses combinations such as:

```
Super + mouse wheel
```

for workspace movement.

Advanced personal mappings can use mouse buttons or scroll directions.

Again:

> *First inspect existing bindings.*

Do not create collisions with Omarchy's current defaults.

12.28 — Bindings Can Trigger Scripts

A powerful pattern is:

```
keybinding
→ personal script
→ multi-step behavior
```

Example:

```
Super + Alt + P
→ ~/.local/bin/open-project
```

The script could:

- choose a project;
- open a workspace;
- start Tmux;
- launch your editor.

This becomes especially useful later when we cover automation.

12.29 — Keep Complex Logic Out of `bindings.lua`

If a binding requires many shell operations, prefer:

```
bindings.lua
→ call script
```

instead of embedding a huge command string.

Why?

Scripts are easier to:

- test;
- version;
- debug;
- reuse.

Keep `bindings.lua` focused on:

```
key → action
```

12.30 — Example: Project Launcher Pattern

Conceptually:

```
o.bind(
  "SUPER + ALT + P",
  "Open project workspace",
  "~/.local/bin/open-project-workspace"
)
```

Then put the real logic in:

```
~/.local/bin/open-project-workspace
```

This is a clean separation of responsibilities.

12.31 — Use Absolute/Resolvable Commands

If a binding runs:

```
my-script
```

make sure the command is actually available in the environment Hyprland uses.

Check:

```
type my-script
```

and:

```
command -v my-script
```

A command that works only because of an interactive Bash alias may not behave the same when launched from Hyprland.

This is a subtle but important distinction.

12.32 — Shell Aliases Are Not Always Safe Binding Targets

Suppose:

```
alias proj='cd ~/Projects/project'
```

works in Bash.

That does not mean Hyprland can launch:

```
proj
```

as a normal executable.

Aliases belong to the interactive shell.

For bindings, prefer:

- real executables;
- scripts;
- Omarchy launch helpers.

12.33 — Environment Differences Matter

A process launched from Hyprland may not inherit exactly the same environment as your interactive terminal.

If a binding command works manually but fails from the shortcut:

check:

- executable path;
- required environment variables;
- shell-only functions;
- current working directory assumptions.

Do not immediately blame Hyprland.

12.34 — Working Directory Assumptions

A keyboard shortcut should generally not assume:

```
current terminal directory
```

because Hyprland is launching the action globally.

If the action requires a specific folder, make the script explicitly choose it.

Example:

```
cd "$HOME/Projects/my-project" || exit 1
```

inside the script.

12.35 — Reload and Validate After Editing

After changing:

```
~/.config/hypr/bindings.lua
```

check:

```
hyprctl configerrors
```

If errors appear:

```
fix them before continuing
```

If you opened the file through Omarchy Setup, the required reload may happen automatically when the editor exits.

12.36 — One Binding at a Time

Do not add ten shortcuts and then test.

Use:

```
add one
↓
save
↓
check configerrors
↓
test
↓
continue
```

This is faster than debugging a batch of changes.

12.37 — Binding Does Nothing: Diagnostic Sequence

If a new shortcut does nothing:

```
1. hyprctl configerrors
2. confirm key exists with wev
3. check current binding map
4. verify the command exists
5. run the command manually
6. inspect whether the action needs a terminal/environment
```

This sequence solves most binding problems.

12.38 — Step 1: Check Config Errors

Run:

```
hyprctl configerrors
```

If you see a Lua/config error, fix that first.

A broken config means the shortcut may never have loaded.

12.39 — Step 2: Verify the Key Event

Run:

```
wev
```

Press the exact combination.

Confirm:

- expected key;
- expected modifiers.

If the keyboard does not emit the expected event, changing `bindings.lua` will not fix the physical/input-layer issue.

12.40 — Step 3: Check for Conflicts

Open:

```
Super + K
```

Search the shortcut.

Or:

```
omarchy menu keybindings --print
```

Confirm whether the combination is already used.

12.41 — Step 4: Verify the Command

Run:

```
type COMMAND
```

Then:

```
command -v COMMAND
```

If nothing resolves, the binding cannot launch it.

12.42 — Step 5: Run It Manually

Run the action from a normal terminal.

If the command itself fails:

```
the binding is not the root problem
```

Fix the command first.

12.43 — Step 6: Check Environment Assumptions

If it works manually but not as a binding, ask:

- Is it an alias?
- Does it depend on `.bashrc`?
- Does it expect a terminal?
- Does it expect a current working directory?
- Does it need a graphical environment variable?
- Does it need a script wrapper?

This is where many "Hyprland shortcut bugs" actually come from.

12.44 — Use Descriptions as Documentation

Good `bindings.lua`:

```
o.bind("SUPER + ALT + P", "Project launcher", "open-project-workspace")
o.bind("SUPER + ALT + RETURN", "Tmux", { omarchy = "terminal-tmux" })
```

This is self-documenting.

Bad:

```
o.bind("SUPER + ALT + P", "X", "foo")
o.bind("SUPER + ALT + RETURN", "Y", "bar")
```

Descriptions are part of maintainability.

12.45 — Group Personal Bindings by Purpose

A clean file can be organized conceptually like:

```
-- Development
...

-- Applications
...

-- System utilities
...

-- Personal scripts
...
```

Keep the structure simple.

The goal is to make future edits easy.

12.46 — Comment Why, Not What

Bad comment:

```
-- Bind Super Alt P
o.bind("SUPER + ALT + P", ...)
```

The code already says that.

Better:

```
-- Keep project launchers under Super+Alt to avoid collisions with Omarchy app
bindings.
```

Comments should explain intent.

12.47 — Do Not Recreate Existing Omarchy Actions Manually

If Omarchy already has:

```
Super + Ctrl + A
→ audio
```

do not write a custom low-level PipeWire script just to duplicate it unless you have a specific reason.

Reuse the integrated Omarchy action where possible.

This keeps the workstation easier to update.

12.48 — Build Personal Shortcuts Around Real Friction

Good reason:

```
I open my Tmux project workspace 30 times per day.
```

Good reason:

```
My keyboard lacks a convenient key for this important action.
```

Weak reason:

```
I want 40 custom shortcuts because a dotfiles screenshot looked cool.
```

Every custom shortcut creates:

- something to remember;
- something to maintain;
- something that can conflict later.

12.49 — Version-Aware Keybinding Maintenance

Omarchy's default keybindings can change between releases.

After a major update:

```
custom shortcut suddenly behaves differently
```

check:

```
Super + K
```

and:

```
omarchy menu keybindings --print
```

before assuming your config broke.

A new default may now use your chosen chord.

12.50 — Keybinding Migration Between Omarchy Generations

Omarchy 4 / Quattro moved the active personal Hyprland configuration to Lua.

If you come from an older Omarchy setup, a file such as:

```
~/.config/hypr/bindings.conf
```

may still exist on disk but may no longer be the active personal bindings file.

Current configuration uses:

```
~/.config/hypr/bindings.lua
```

This is a critical migration lesson.

A preserved file is not necessarily a loaded file.

12.51 — Verify What Is Actually Loaded

Inspect:

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

Look for:

```
require("hypr.bindings")
```

That confirms the current Lua binding module is part of the load chain.

Do not infer active config solely from filenames that happen to exist.

12.52 — Recovery: Syntax Error in `bindings.lua`

If a binding edit breaks config:

```
hyprctl configerrors
```

Then reopen:

```
nvim ~/.config/hypr/bindings.lua
```

Undo the last edit.

Save.

Re-check:

```
hyprctl configerrors
```

This is the correct first recovery action.

12.53 — Recovery: Use Git if Your Dotfiles Are Versioned

Later in the course, your personal config will live in Git.

Then recovery can be as simple as:

```
git diff
```

and:

```
git restore bindings.lua
```

or reverting the specific commit.

This is one reason dotfile version control is valuable.

12.54 — Recovery: Do Not Jump Straight to Config Reset

Do **not** start with:

```
omarchy refresh hyprland
```

for one broken keybinding.

That is a broad reset of personal Hyprland configuration.

Also do not start with:

```
omarchy reinstall configs
```

for a single mistake.

Use the smallest repair.

12.55 — Practical Exercise: Inspect Your Current Binding Map

Press:

```
Super + K
```

Search for:

```
Terminal
```

Then:

```
Tmux
```

Then:

```
Capture
```

Then:

```
Clipboard
```

The goal is to become comfortable discovering current bindings rather than memorizing a static sheet.

12.56 — Practical Exercise: Inspect `bindings.lua`

Run:

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

Identify any existing personal changes.

Do not edit yet.

12.57 — Practical Exercise: Add a Safe Custom Binding

Choose an unused key combination.

For example, if verified free on your system:

```
Super + Alt + B
```

Bind a harmless action such as launching the activity monitor or another known command.

Example structure:

```
o.bind("SUPER + ALT + B", "Activity monitor", "btop")
```

If the command needs a terminal wrapper on your current system, use an appropriate Omarchy terminal-launch method instead of assuming raw `btop` will open graphically.

Test one change only.

12.58 — Practical Exercise: Replace a Binding

Choose a default application binding you do not use.

Verify it first with:

```
Super + K
```

Then replace it using:

```
o.rebind("KEY COMBINATION", "New description", "new-command")
```

Save.

Check:

```
hyprctl configerrors
```

Test.

If you do not want to keep the change, restore the original afterward.

12.59 — Practical Exercise: Disable a Binding

Choose a non-essential binding.

Verify it.

Then:

```
hl.unbind("KEY COMBINATION")
```

Save.

Check:

```
hyprctl configerrors
```

Test that the action is gone.

Restore afterward if the change was only for practice.

12.60 — Practical Exercise: Debug a Keyboard Key with `wev`

Run:

```
wev
```

Press:

```
Super
Shift
Alt
Return
```

and another unusual key on your keyboard.

Observe:

- modifier state;
- key symbol;
- press/release.

This will make future keyboard debugging dramatically easier.

12.61 — Practical Exercise: Create a Real Developer Shortcut

Create one shortcut you will actually use.

Recommended course example:

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

If your current Omarchy version already provides it, do **not** duplicate it.

Instead choose another real need.

Examples:

```
project picker
notes
log viewer
AI workspace
custom development script
```

The important part is that the shortcut solves a real repeated action.

12.62 — Practical Exercise: Validate the Full Chain

For your new binding:

```
1. verify key event with wev if necessary
2. verify no collision
3. verify command with type
4. add binding
5. hyprctl configerrors
6. execute shortcut
7. verify expected result
```

This is the professional binding workflow.

12.63 — Common Keybinding Mistakes

Mistake 1 — Editing the Wrong File

Current Omarchy 4 personal binding file:

```
~/.config/hypr/bindings.lua
```

Mistake 2 — Copying Printed Key Syntax Directly

The printed keybinding list may format modifiers differently from Lua.

Lua example:

```
SUPER + SHIFT + S
```

Mistake 3 — Adding a Binding Without Checking Existing Shortcuts

Use:

```
Super + K
```

first.

Mistake 4 — Assuming the Physical Key Sends What Its Label Says

Use:

```
wev
```

Mistake 5 — Binding an Interactive Shell Alias

A Hyprland launch is not the same as an interactive Bash prompt.

Use real commands/scripts.

Mistake 6 — Using a Command That Needs a Terminal Without Launching a Terminal

Terminal applications need a terminal context.

Mistake 7 — Putting Complex Shell Logic Directly in `bindings.lua`

Call a script instead.

Mistake 8 — Adding Many Shortcuts at Once

Test incrementally.

Mistake 9 — Disabling All Omarchy Defaults Too Early

Override individual bindings first.

Mistake 10 — Resetting All Hyprland Config for One Binding Error

Fix the specific file first.

12.64 — Checkpoint

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

Discovery

- [ ] I know how to open the keybinding browser with `Super + K`.
- [ ] I know how to inspect bindings from the CLI.
- [ ] I understand that displayed shortcut syntax may differ from Lua syntax.

Configuration

- [ ] I know that personal bindings live in `~/.config/hypr/bindings.lua`.
- [ ] I understand `o.bind`.
- [ ] I understand `o.rebind`.
- [ ] I understand `hl.unbind`.
- [ ] I know how to choose between them.

Debugging

- [ ] I can use `hyprctl configerrors`.
- [ ] I can use `wev`.
- [ ] I can verify a command with `type`.
- [ ] I can run the target command manually before blaming the binding.
- [ ] I understand shell-environment differences.

Design

- [ ] I avoid random shortcut collisions.
- [ ] I use meaningful descriptions.
- [ ] I keep complex logic in scripts.
- [ ] I create shortcuts based on real repeated friction.

Recovery

- [ ] I know how to undo a broken Lua edit.
- [ ] I understand that old `.conf` files may exist without being loaded.
- [ ] I do not use broad config resets as the first recovery step.

12.65 — What You Can Now Do

After completing Module 12, you can now:

- inspect the live Omarchy keybinding system;
- add personal bindings safely;
- replace default application shortcuts cleanly;
- disable unwanted shortcuts;
- diagnose keyboard-event problems;
- distinguish hardware/Fn-layer problems from Hyprland problems;

- launch scripts and development tools through personal shortcuts;
- avoid shell-alias and PATH pitfalls;
- maintain a readable personal shortcut system;
- recover from binding errors without resetting your workstation.

Most importantly:

> *You can now turn Omarchy's keyboard-first design into a keyboard workflow tailored to the way you actually work.*

Module 12 Summary

Personal binding file:

```
~/.config/hypr/bindings.lua
```

Discovery:

```
Super + K
```

CLI:

```
omarchy menu keybindings
```

Current helper model:

```
o.bind(...)
```
```
add
```
```
o.rebind(...)
```
```
replace
```
```
hl.unbind(...)
```
```
remove
```

Important syntax:

```
Lua:
SUPER + SHIFT + S
```

Input verification:

```
wev
```

Config verification:

```
hyprctl configerrors
```

Command verification:

```
type COMMAND
command -v COMMAND
```

Recommended troubleshooting flow:

```
config errors
↓
actual key event
↓
binding collision
↓
command availability
↓
manual command test
↓
environment/terminal assumptions
```

And the most important rule:

> ***A keybinding is a chain: physical key → input event → Hyprland binding → command/action. Debug the chain in that order.***