Modern Emacs Configuration

YAMASHITA, Takao

February 8, 2026

Contents

1 Overview

A modern, literate Emacs configuration using Org Mode's Babel format, emphasizing performance, language server integration, AI assistance, and productivity.

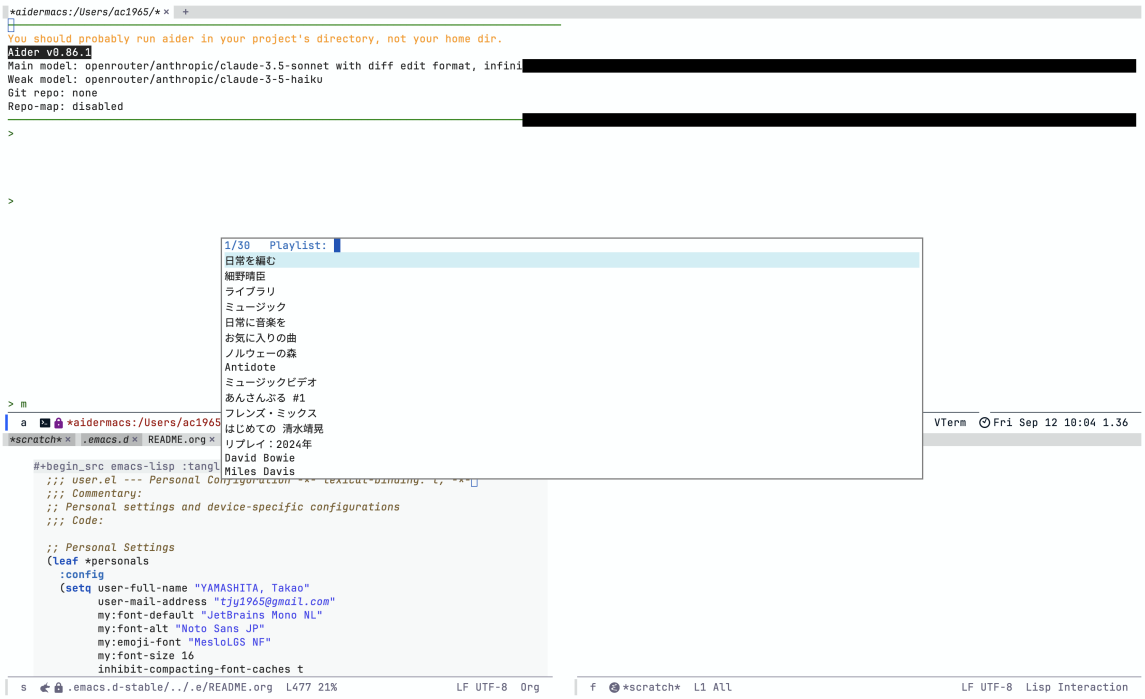
This project focuses on extensibility ☒ and maintainability.

1.1 Design Tenets (TL;DR)

- **Deterministic startup**: no implicit installs, no hidden side effects.
- **Layered ownership**: each layer owns policy; dependencies flow one way.
- **Opt-in power**: advanced tooling must never be required for baseline use.
- **Observable over magical**: favor inspectable state over automation.
- **Deletion-friendly**: removing a layer should degrade convenience, not correctness.

1.2 Features

- **Performance & Native Compilation** — Early-init moves ELN under ‘.cache/‘, silences async warnings, widens GC at startup and restores sane values later, and uses GCMH for idle GC.
- **Language Server Protocol** — Backend-agnostic helpers in ‘core/general.el‘; choose **Eglot** or **lsp-mode** via ‘core/switches.el‘ with presence checks and auto-enable logic.
- **AI Integration** — Aidermacs (vterm backend). Prefers OpenRouter when ‘$\text{OPENROUTER}_{\text{APIKEY}}$‘ is set; otherwise uses OpenAI with ‘$\text{OPENAI}_{\text{APIKEY}}$‘.
- **Modern UI & Editing** — Tree-sitter remaps (‘*-ts-mode‘), ef-themes + spacious-padding, Nerd Icons, Vertico/Orderless/Corfu/CAPE/Embark/Consult, Doom/Nano modeline switchers.
- **Productivity Tools** — Opinionated Org stack (agenda, capture, journal, roam, download, TOC), Magit + diff-hl/Forge, REST client, Docker/dev helpers, tidy backups/autosave-visited.



1.3 Coding Rules

- ☒ ‘lexical-binding: t‘ is **mandatory** for all files.
- ☒ The ‘(provide ’FEATURE)‘ symbol **must match the file’ s logical feature name** (usually derived from the file path).

- Built-in packages MUST explicitly declare ‘:straight nil‘.
- Each ‘leaf‘ form follows a stable, readable structure: ‘:straight‘ → ‘:bind‘ → ‘:hook‘ → ‘:custom‘ → ‘:config‘
- Only documented, public APIs are used.
  - Private, internal, or speculative APIs are intentionally avoided.
- **Compatibility & Forward-Safety Policy**
  - This configuration targets **Emacs 30+**.
  - Code is written with **Emacs 31 and later** in mind.
  - Obsolete APIs are avoided even if still functional.
    - Prefer ‘if-let*‘, ‘when-let*‘, ‘and-let*‘ over deprecated forms.
  - New compiler or runtime warnings are treated as **actionable signals**.
  - The codebase aims to remain warning-free under the latest stable Emacs with default ‘byte-compile-warnings‘.

See also: Modular Loader & Core/Utils design for how these rules are enforced structurally.

1.4 Installation

1.4.1 Prerequisites

- **Required**
  - Emacs **30.0+** with native compilation (‘–with-native-compilation‘)
  - Git
  - GNU Make
  - GCC **10+** with ‘libgccjit‘
- **Optional but Recommended**
  - ripgrep (‘rg‘) → faster project-wide search
  - aspell or hunspell → spell checking
  - pass + GnuPG → password and auth-source integration
  - Homebrew (macOS only) → for consistent toolchain installation

1.4.2 Building Emacs

Use the provided build script: build-emacs.sh

```
build-emacs.sh
```

1.4.3 Quick Start

1. Clone the repository:

   ```
   git clone --depth 1 https://github.com/ac1965/.emacs.d ~/.emacs.d/
   ```

2. Tangle configuration:

   ```
   EMACS=/Applications/Emacs.app/Contents/MacOS/Emacs make -C ~/.emacs.d/ tangle
   ```

1.4.4 Makefile

```
# Makefile — One-pass builder for a modular Emacs config
# - Default / `make all` : onepass-init (tangle -> incremental byte-compile)
# - `make onepass-q`     : -Q (minimal env) tangle -> full byte-compile
# - Paths are absolutized from repo root to avoid "lisp/personal" confusion.

SHELL := /bin/sh

# ---- Repo-root & absolutized dirs ------------------------------------------
ROOT := $(CURDIR)

EMACS  ?= emacs
ORG    ?= README.org
EARLY  ?= early-init.el
INIT   ?= init.el

# Always treat these as top-level under repo root
LISPDIR_REL     ?= lisp
PERSONALDIR_REL ?= personal

LISPDIR     := $(abspath $(ROOT)/$(LISPDIR_REL))
PERSONALDIR := $(abspath $(ROOT)/$(PERSONALDIR_REL))
ORG := $(abspath $(ROOT)/$(ORG))
EARLY := $(abspath $(ROOT)/$(EARLY))
INIT  := $(abspath $(ROOT)/$(INIT))

STRICT_BYTE_WARN ?= 0   # Treat byte-compile warnings as errors
NATIVE_COMPILE   ?= 1   # Prefer native-compile if available

# ---- Emacs runners & common eval snippets ----------------------------------
EMACS_BATCH := "$(EMACS)" --batch
EMACS_Q     := $(EMACS_BATCH) -Q

EVAL_STRICT := $(if $(filter 1,$(STRICT_BYTE_WARN)),--eval "(setq byte-compile-
error-on-warn t)",)
EVAL_NATIVE := $(if $(filter 1,$(NATIVE_COMPILE)),--eval "(setq comp-deferred-
compilation t)",)

# Optional leaf injection for -Q
STRAIGHT_BASE_DIR ?= $(shell \
  if [ -f "$(EARLY)" ]; then \
    $(EMACS_Q) -l "$(EARLY)" \
      --eval "(princ (expand-file-name (or (ignore-errors STRAIGHT_BASE_DIR) \
                                           (ignore-errors (and (boundp 'straight-
base-dir) straight-base-dir)) \
                                           (expand-file-name \"straight\" user-
emacs-directory))))"; \
  else \
    printf "%s" "$$HOME/.emacs.d/straight"; \
  fi)
LEAF_DIR   := $(STRAIGHT_BASE_DIR)/repos/leaf
LEAFKW_DIR := $(STRAIGHT_BASE_DIR)/repos/leaf-keywords
```

```
EVAL_LEAF := \
  --eval "(let* ((ldir \"$(LEAF_DIR)\") (kwdir \"$(LEAFKW_DIR)\")) \
            (when (file-directory-p ldir)  (add-to-list 'load-path ldir)) \
            (when (file-directory-p kwdir) (add-to-list 'load-path kwdir)) \
            (ignore-errors (require 'leaf)) \
            (ignore-errors (require 'leaf-keywords)) \
            (when (featurep 'leaf-keywords) (leaf-keywords-init)))"

# ---- Default target (no args) ------------------------------------------------
.PHONY: all onepass-init onepass-q clean distclean show-files echo-paths tangle
all: onepass-init

# ---- One-pass (early+init env) : tangle -> incremental compile ---------------
onepass-init: $(ORG)
        @echo "[onepass-init] tangle -> incremental byte-compile (init loaded)"
        @$(EMACS_BATCH) -l "$(EARLY)" -l "$(INIT)" \
          $(EVAL_STRICT) $(EVAL_NATIVE) \
          --eval "(setq org-confirm-babel-evaluate nil)" \
          --eval "(require 'org)" \
          --eval "(org-babel-tangle-file \"$(ORG)\")" \
          --eval "(let* ((dirs (delq nil (list (and (file-directory-
p \"$(LISPDIR)\") \"$(LISPDIR)\") \
                                               (and (file-directory-
p \"$(PERSONALDIR)\") \"$(PERSONALDIR)\"))))) \
                     (dolist (d dirs) (byte-recompile-directory d 0)) \
                     (when (and (featurep 'comp) (bound-and-true-p comp-deferred-
compilation)) \
                       (dolist (d dirs) (ignore-errors (native-compile-
async d 'recursively)))))" \
          --eval "(message \"[onepass-init] done\")"

# ---- One-pass (-Q minimal env) : tangle -> full compile ----------------------
onepass-q: $(ORG)
        @echo "[onepass-q] -Q tangle -> full byte-compile (init not loaded)"
        @$(EMACS_Q) \
          $(EVAL_LEAF) $(EVAL_STRICT) $(EVAL_NATIVE) \
          --eval "(setq org-confirm-babel-evaluate nil)" \
          --eval "(require 'org)" \
          --eval "(org-babel-tangle-file \"$(ORG)\")" \
          --eval "(let* ((dirs (delq nil (list (and (file-directory-
p \"$(LISPDIR)\") \"$(LISPDIR)\") \
                                               (and (file-directory-
p \"$(PERSONALDIR)\") \"$(PERSONALDIR)\"))))) \
                     (dolist (d dirs) (byte-recompile-directory d t)) \
                     (when (and (featurep 'comp) (bound-and-true-p comp-deferred-
compilation)) \
                       (dolist (d dirs) (ignore-errors (native-compile-
async d 'recursively)))))" \
          --eval "(message \"[onepass-q] done\")"

# ---- Utilities ---------------------------------------------------------------
show-files:
        @echo "[list] $(LISPDIR)";    { [ -d "$(LISPDIR)" ] && find "$(LISPDIR)" -
```

```
type f -name '*.el' | sort; } || true
        @echo "[list] $(PERSONALDIR)"; { [ -d "$(PERSONALDIR)" ] && find "$(PERSONALDIR)" -
type f -name '*.el' | sort; } || true

echo-paths:
        @echo "ROOT=$(ROOT)"; \
        echo "EARLY=$(EARLY)"; \
        echo "INIT=$(INIT)"; \
        echo "LISPDIR=$(LISPDIR)"; \
        echo "PERSONALDIR=$(PERSONALDIR)"; \
        echo "STRAIGHT_BASE_DIR=$(STRAIGHT_BASE_DIR)"; \
        echo "LEAF_DIR=$(LEAF_DIR)"; \
        echo "LEAFKW_DIR=$(LEAFKW_DIR)"

clean:
        @echo "[clean] remove *.elc under $(LISPDIR) and $(PERSONALDIR)"
        @{ [ -d "$(LISPDIR)" ] && find "$(LISPDIR)" -type f -name '*.elc' -
delete; } 2>/dev/null || true
        @{ [ -d "$(PERSONALDIR)" ] && find "$(PERSONALDIR)" -type f -name '*.elc' -
delete; } 2>/dev/null || true

distclean: clean
        @echo "[distclean] remove stray *.eln"
        @find "$(ROOT)" -type f -name '*.eln' -delete

tangle:
        @echo "[tangle] $(ORG)"
        @$(EMACS_Q) \
          --eval "(require 'org)" \
          --eval "(require 'ob-core)" \
          --eval "(org-babel-do-load-languages 'org-babel-load-languages '((emacs-
lisp . t)))" \
          --eval "(setq org-confirm-babel-evaluate nil noninteractive t)" \
          --eval "(org-babel-tangle-file \"$(ORG)\")"
```

1.4.5 System Information

1. Apple Silicon (Primary)

    - GNU Emacs **31.0.50**

Property	Value
Commit	0dfaa756120f4feecf5f6011ec243741b071e440
Branch	master
System	aarch64-apple-darwin25.2.0
Date	2026-02-05 20:43:39 (JST)
Patch	N/A ns-inline.patch
Features	ACL DBUS GLIB GNUTLS IMAGEMAGICK LCMS2 LIBXML2 MODULES NATIVE$_{\text{COMP}}$
Options	–with-ns –enable-mac-app=yes –with-xwidgets –with-native-compilation –with-json –with-tree

2. Intel (Secondary)

    - GNU Emacs **31.0.50**

Property	Value
Commit	63ea5e5b3a57e7660ece022ba1834002ca2f206d
Branch	master
System	$x86_{64}$-apple-darwin25.1.0
Date	2025-11-01 12:05:25 (JST)
Patch	N/A ns-inline.patch
Features	ACL DBUS GIF GLIB GMP GNUTLS JPEG LCMS2 LIBXML2 MODULES $\text{NATIVE}_{\text{COMP}}$
Options	–with-native-compilation –with-gnutls=ifavailable –with-json –with-modules –with-tree-sitter

1.5 Tools

1.5.1 Graph Capture (Require Dependency Visualization)

1. Purpose Capture and visualize ‘require‘ relationships between Emacs Lisp features during startup or module loading. This helps understanding implicit dependencies, load order, and unwanted coupling between modules.

2. What it does

   - Advises ‘require‘ to record **from → to** edges between features
   - Stores edges in a hash table (no duplicates)
   - Exports the dependency graph as:
     - Graphviz DOT
     - Mermaid (for Org / Markdown)
   - Provides interactive commands to enable/disable capture at runtime

3. Notes

   - Intended for **diagnostics only**, not for normal startup
   - Enable capture **before** loading modules
   - Disable capture after use to avoid overhead
   - Feature names are derived from file names or ‘require‘ symbols

4. Implementation

   (a) tools/tools-graph.el

```
;;; tools/tools-graph.el --- Require dependency graph capture -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: tools
;;
;;; Commentary:
;; Diagnostic tool to capture and visualize `require` dependencies
;; between Emacs Lisp features.
;;
;; This module advises `require` to record feature-to-feature edges
;; during evaluation and exports the dependency graph as:
;;
;; - Graphviz DOT
```

```
;; - Mermaid flowchart
;; - Org Babel mermaid source block
;;
;; Intended for diagnostics only.
;; Do not enable during normal startup.
;;
;;; Code:

;; ---------------------------------------------------------------------------
;; State
;; ---------------------------------------------------------------------------

(defvar tools-graph/require-edges (make-hash-table :test 'equal)
  "Hash table storing recorded require edges as FROM->TO keys.")

;; ---------------------------------------------------------------------------
;; Internal helpers
;; ---------------------------------------------------------------------------

(defun tools-graph--record (from to)
  "Record a require edge FROM -> TO if both are symbols."
  (when (and (symbolp from) (symbolp to))
    (puthash (format "%s->%s" from to) t tools-graph/require-edges)))

(defun tools-graph--edge-list ()
  "Return a sorted list of recorded edges."
  (sort (hash-table-keys tools-graph/require-edges) #'string<))

(defun tools-graph--sanitize (name)
  "Return a Mermaid/Org-safe node NAME."
  (replace-regexp-in-string "[./-]" "_" name))

;; ---------------------------------------------------------------------------
;; require advice
;; ---------------------------------------------------------------------------

(defun tools-graph/require-advice (orig feature &optional filename noerror)
  "Advice around `require` to record dependency edges."
  (let* ((from-file (or load-file-name (buffer-file-name)))
         (from-sym
          (if from-file
              (intern
               (file-name-sans-extension
                (file-name-nondirectory from-file)))
            'init))
         (to-sym (if (symbolp feature)
                     feature
                   (intern (format "%s" feature)))))
    (tools-graph--record from-sym to-sym)
    (funcall orig feature filename noerror)))

;; ---------------------------------------------------------------------------
;; Exporters
```

```
;; ---------------------------------------------------------------------------

;;;###autoload
(defun tools-graph/export-dot ()
  "Render recorded require edges as Graphviz DOT."
  (interactive)
  (let ((buf (get-buffer-create "*Require Graph (DOT)*")))
    (with-current-buffer buf
      (erase-buffer)
      (insert "digraph G {\n  rankdir=LR;\n  node [shape=box, fontsize=10];\n")
      (dolist (e (tools-graph--edge-list))
        (when (string-match "^\\([^->]+\\)->\\(.+\\)$" e)
          (insert
           (format "  \"%s\" -> \"%s\";\n"
                   (match-string 1 e)
                   (match-string 2 e)))))
      (insert "}\n"))
    (pop-to-buffer buf)))

;;;###autoload
(defun tools-graph/export-mermaid ()
  "Render recorded require edges as Mermaid flowchart."
  (interactive)
  (let ((buf (get-buffer-create "*Require Graph (Mermaid)*")))
    (with-current-buffer buf
      (erase-buffer)
      (insert "```mermaid\nflowchart LR\n")
      (dolist (e (tools-graph--edge-list))
        (when (string-match "^\\([^->]+\\)->\\(.+\\)$" e)
          (insert
           (format "  %s --> %s\n"
                   (tools-graph--sanitize (match-string 1 e))
                   (tools-graph--sanitize (match-string 2 e))))))
      (insert "```\n"))
    (pop-to-buffer buf)))

;;;###autoload
(defun tools-graph/export-org ()
  "Render recorded require edges as an Org mermaid source block."
  (interactive)
  (let ((buf (get-buffer-create "*Require Graph (Org)*")))
    (with-current-buffer buf
      (erase-buffer)
      (insert "#+begin_src mermaid\n")
      (insert "flowchart LR\n")
      (dolist (e (tools-graph--edge-list))
        (when (string-match "^\\([^->]+\\)->\\(.+\\)$" e)
          (insert
           (format "  %s --> %s\n"
                   (tools-graph--sanitize (match-string 1 e))
                   (tools-graph--sanitize (match-string 2 e))))))
      (insert "#+end_src\n"))
    (pop-to-buffer buf)))
```

```
;; ---------------------------------------------------------------------------
;; Control
;; ---------------------------------------------------------------------------

(defvar tools-graph--enabled-p nil
  "Non-nil if require capture advice is enabled.")

;;;###autoload
(defun tools-graph/enable-require-capture ()
  "Enable capture of `require` dependency edges.

This advises `require` globally. Intended for diagnostic use only."
  (interactive)
  (unless tools-graph--enabled-p
    (advice-add 'require :around #'tools-graph/require-advice)
    (setq tools-graph--enabled-p t)
    (message "[tools-graph] require capture enabled")))

;;;###autoload
(defun tools-graph/disable-require-capture ()
  "Disable capture of `require` dependency edges."
  (interactive)
  (when tools-graph--enabled-p
    (advice-remove 'require #'tools-graph/require-advice)
    (setq tools-graph--enabled-p nil)
    (message "[tools-graph] require capture disabled")))

(provide 'tools-graph)
;;; tools/tools-graph.el ends here
```

(b) tools/tools-graph-unused.el

```
;;; tools-graph-unused.el --- Detect unused defuns (aware of hooks/bindings) -
*- lexical-binding: t; -*-

;;; Commentary:
;; Static detector for unused top-level defuns.
;; Excludes functions referenced via hooks, keymaps, advice, or autoloads.
;; Intended for diagnostic use only.

;;; Code:

(require 'cl-lib)

;; ---------------------------------------------------------------------------
;; Collectors
;; ---------------------------------------------------------------------------

(defun tools-graph--collect-defuns (dir)
  "Return an alist of (symbol . file) for defuns under DIR."
  (let (defs)
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
```

```
        (goto-char (point-min))
        (while (re-search-forward
                "^(defun\\s-+\\(\\(?:\\sw\\|\\s_\\)+\\)" nil t)
          (push (cons (intern (match-string 1)) file) defs))))
    defs))

(defun tools-graph--collect-called-symbols (dir)
  "Return hash table of symbols that appear in function call position."
  (let ((calls (make-hash-table :test 'eq)))
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (re-search-forward
                "(\\(\\(?:\\sw\\|\\s_\\)+\\)" nil t)
          (puthash (intern (match-string 1)) t calls))))
    calls))

(defun tools-graph--collect-hooked-symbols (dir)
  "Return hash table of functions registered to hooks."
  (let ((hooks (make-hash-table :test 'eq)))
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (re-search-forward
                "(add-hook\\s-+'[^ ]+\\s-+#'\\(\\(?:\\sw\\|\\s_\\)+\\)" nil t)
          (puthash (intern (match-string 1)) 'hook hooks))
        ;; leaf :hook (xxx . func)
        (goto-char (point-min))
        (while (re-search-forward
                ":hook\\s-*(\\(?:[^ .]+\\)\\s-+.\\s-+\\(\\(?:\\sw\\|\\s_\\)+\\))"
                nil t)
          (puthash (intern (match-string 1)) 'hook hooks))))
    hooks))

(defun tools-graph--collect-bound-symbols (dir)
  "Return hash table of functions bound in keymaps."
  (let ((binds (make-hash-table :test 'eq)))
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (re-search-forward
                "(define-key[^)]*#'\\(\\(?:\\sw\\|\\s_\\)+\\)" nil t)
          (puthash (intern (match-string 1)) 'keymap binds))
        ;; leaf :bind ("x" . func)
        (goto-char (point-min))
        (while (re-search-forward
                ":bind\\s-*(\"[^\"]+\"\\s-+.\\s-+\\(\\(?:\\sw\\|\\s_\\)+\\))"
                nil t)
          (puthash (intern (match-string 1)) 'keymap binds))))
    binds))
```

```
(defun tools-graph--collect-advised-symbols (dir)
  "Return hash table of functions used as advice."
  (let ((advs (make-hash-table :test 'eq)))
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (re-search-forward
                "(advice-add[^)]*#'\\(\\(?:\\sw\\|\\s_\\)+\\)" nil t)
          (puthash (intern (match-string 1)) 'advice advs))))
    advs))

(defun tools-graph--collect-autoloaded-symbols (dir)
  "Return hash table of autoloaded defuns."
  (let ((autos (make-hash-table :test 'eq)))
    (dolist (file (directory-files-recursively dir "\\.el\\'"))
      (with-temp-buffer
        (insert-file-contents file)
        (goto-char (point-min))
        (while (re-search-forward
                ";;;###autoload[ \t\n]+(defun\\s-+\\(\\(?:\\sw\\|\\s_\\)+\\)"
                nil t)
          (puthash (intern (match-string 1)) 'autoload autos))))
    autos))

;; ---------------------------------------------------------------------------
;; Public command
;; ---------------------------------------------------------------------------

;;;###autoload
(defun tools-graph/unused-functions (dir)
  "Report unused defuns under DIR, excluding hooks/bindings/advice/autoload."
  (interactive "DDirectory: ")
  (let* ((defs   (tools-graph--collect-defuns dir))
         (calls  (tools-graph--collect-called-symbols dir))
         (hooks  (tools-graph--collect-hooked-symbols dir))
         (binds  (tools-graph--collect-bound-symbols dir))
         (advs   (tools-graph--collect-advised-symbols dir))
         (autos  (tools-graph--collect-autoloaded-symbols dir)))
    (with-current-buffer (get-buffer-create "*Unused Defuns*")
      (erase-buffer)
      (insert (format "Unused function candidates under: %s\n\n" dir))
      (dolist (d defs)
        (let ((sym (car d)))
          (unless (or (gethash sym calls)
                      (gethash sym hooks)
                      (gethash sym binds)
                      (gethash sym advs)
                      (gethash sym autos))
            (insert (format "%-35s  (%s)\n" sym (cdr d))))))
      (pop-to-buffer (current-buffer)))))
```

```
(provide 'tools-graph-unused)
;;; tools-graph-unused.el ends here
```

5. Module Load Summary Helper

```
;;; my-modules-summary.el --- Module loading summary helper -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;;; Commentary:
;; Provide an interactive command to display a concise summary of
;; module loading status and captured require dependency edges.
;;
;;; Code:

(defun my/modules-summary-line ()
  "Display a concise summary of module loading and captured require edges."
  (interactive)
  (let* ((edges (if (and (boundp 'tools-graph/require-edges)
                         (hash-table-p tools-graph/require-edges))
                    (hash-table-count tools-graph/require-edges)
                  "N/A"))
         (skip  (if (boundp 'my-modules-skip)
                    (length (or my-modules-skip '()))
                  0))
         (extra (if (boundp 'my-modules-extra)
                    (length (or my-modules-extra '()))
                  0)))
    (message "[modules] edges=%s, skip=%d, extra=%d"
             edges skip extra)))

(provide 'my-modules-summary)
;;; my-modules-summary.el ends here
```

6. Usage

   Purpose: Capture runtime ‘require‘ dependencies between Emacs Lisp modules and export them as a graph.

   (a) Restart Emacs

   (b) Enable capture (evaluate in order):
   - ‘M-: (require ’tools-graph)‘
   - ‘M-: (tools-graph/enable-require-capture)‘

   (c) Load modules (e.g. ‘(require ’modules)‘ or normal startup)

   (d) Export graph:
   - ‘M-: (tools-graph/export-mermaid)‘
   - or ‘M-: (tools-graph/export-dot)‘

- or ‘M-: (tools-graph/export-org)‘

(e) Paste the result into Org / Markdown

(f) (Optional) ‘M-: (my/modules-summary-line)‘

(g) (Optional) Disable capture:

- ‘M-: (tools-graph/disable-require-capture)‘

```
;; 1) Enable capture
(require 'tools-graph)
(tools-graph/enable-require-capture)

;; 2) Load modules (example)
;; (require 'modules)

;; 3) Export captured edges (choose one)
(tools-graph/export-org)
;; (tools-graph/export-mermaid)
;; (tools-graph/export-dot)

;; 4) Optional: stop capturing
(tools-graph/disable-require-capture)
```

7. Mermaid Example

```
flowchart TD
    core[core]
    ui[ui]
    completion[completion]
    orgx[orgx]
    dev[dev]
    vcs[vcs]
    utils[utils]
    personal[personal]

    core --> ui
    core --> completion
    core --> orgx

    ui --> completion

    completion --> orgx

    orgx --> dev
    completion --> dev
    ui --> dev

    dev --> vcs
    completion --> vcs
    ui --> vcs

    core --> utils
    ui --> utils
    completion --> utils
```

```
orgx --> utils
dev --> utils
vcs --> utils

core --> personal
ui --> personal
completion --> personal
orgx --> personal
dev --> personal
vcs --> personal
utils --> personal
```

2 Configuration Files

This Emacs configuration is **modular by design** and targets **Emacs 30+**. Each layer has a clearly defined responsibility to keep behavior predictable, UI replaceable, and personal customizations isolated.

- `early-init.el` → earliest bootstrap (performance, paths, UI defaults)
- `init.el` → package bootstrap, global defaults, module entrypoint
- `lisp/` → shared, versioned modules (core, ui, completion, orgx, dev, vcs, utils)
- `personal/` → user- and device-specific overlays (not shared policy)

2.1 Core Bootstrap — early-init.el & init.el

2.1.1 Overview

1. Purpose Provide a **clean, fast, and conservative** bootstrap sequence that prepares Emacs before regular initialization.

   The bootstrap is split into two explicit stages:

   - `early-init.el` runs **before package initialization** and establishes directories, performance guards, and flicker-free UI defaults.
   - `init.el` completes package bootstrapping (**straight.el + leaf**), imports the login environment on macOS, applies runtime performance knobs, and exposes a deterministic module loader entrypoint.

   This separation keeps early startup minimal and infrastructure-focused, while deferring all feature logic to later stages.

2. What this configuration does

   - Disables `package.el` early; **straight.el** and **leaf** are the only package managers.
   - Speeds up startup by temporarily widening GC limits and clearing `file-name-handler-alist`, then restoring sane runtime values.
   - Normalizes all state under predictable directories: `.cache/`, `.etc/`, and `.var/` (including native-comp artifacts).
   - On macOS, prefers the Homebrew toolchain by preparing PATH-related variables (e.g. `PATH`, `LIBRARY_PATH`, `CC`) **before** native compilation is triggered.
   - Disables classic backups and auto-save early; higher-level modules may enable `auto-save-visited-mode` later in a controlled way.

- Applies early UI defaults (no menu/tool/scroll bars, stable frame parameters) to avoid startup flicker.
- Bootstraps **straight.el** robustly, with guarded network access and explicit error reporting.
- Initializes **leaf** and its keywords, and enables conservative performance helpers (e.g. GCMH, `read-process-output-max`).
- Sets URL-related state paths **before** `url.el` loads, so downstream consumers (including **straight**) inherit them.
- Provides two stable entrypoints:
  - a per-user personal override file (`personal/<login-name>.el`)
  - a shared module loader (`lisp/modules.el`)

3. Reproducibility Note (Personal Tangling)

   This configuration prioritizes reproducibility for all **shared** layers:

   - `early-init.el`
   - `init.el`
   - `lisp/`

   Personal files are intentionally tangled to user-specific paths:

   `personal/<login-name>.el`

   This design explicitly trades strict reproducibility for:

   - Per-user isolation
   - Safe multi-user sharing of the same repository
   - Zero-conflict personal overrides

   This behavior is intentional and by design.

4. Module map (where things live)

File	Role
`early-init.el`	Pre-init bootstrap (dirs, performance guards, package.el off, macOS toolchain, early UI)
`init.el`	Main init (URL state, straight bootstrap, env import, runtime knobs, module loader)

5. How it works (boot flow)

   (a) Emacs loads `early-init.el`:
   - Directory paths are established.
   - `package.el` is disabled.
   - GC and file-handler pressure is relaxed.
   - Early UI defaults are applied.
   - macOS toolchain variables are prepared when applicable.

   (b) Emacs loads `init.el`:
   - URL state directories are set **before** `url.el`.
   - **straight.el** is bootstrapped.
   - On macOS GUI/daemon sessions, the login environment is imported.
   - **leaf** is initialized and a minimal base of packages is ensured.

- Runtime performance knobs (GCMH, IO buffers) are applied.
- A per-user personal file is loaded safely.
- `modules.el` is required as the canonical feature entrypoint.

(c) After initialization completes, a concise startup summary (elapsed time and GC count) is printed.

6. Key settings (reference)

`package-enable-at-startup nil` — rely exclusively on **straight.el**.

`straight-base-dir` Located under `.cache/` to keep the config root clean.

`native-comp-eln-load-path` Centralized under `.cache/eln-cache`.

`read-process-output-max` Temporarily raised (4 MiB) for better LSP/IO throughput.

`gcmh-high-cons-threshold` 16 MiB; `gcmh-mode` enabled for smoother long sessions.

7. Usage tips

- Treat `early-init.el` as infrastructure only; avoid user behavior or feature logic.
- Put shared behavior in modules loaded via `modules.el`.
- Put identity, device-specific glue, and workflow integrations in `personal/<login-name>.el` or related personal modules.
- After installing or upgrading Homebrew toolchains, restart Emacs so native compilation sees updated paths.
- To relocate the entire setup, move the config directory; Emacs will regenerate `.cache/`, `.etc/`, and `.var/` automatically.

8. Troubleshooting

- **"Native compilation can' t find libgccjit on macOS"** → Ensure Homebrew' s `libgccjit` is installed and visible. The early bootstrap prepares `LIBRARY_PATH` when possible.
- **"Straight bootstrap failed"** → A transient network issue during `url-retrieve-synchronously`. Re-run; failures are reported with a clear `[straight] bootstrap failed` message.
- **"Inhibit startup echo warning"** → `inhibit-startup-echo-area-message` is set to the actual user name string to satisfy Emacs' type requirements.

2.1.2 early-init.el

```
;;; early-init.el --- Early bootstrap and runtime foundations -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Early initialization executed before regular init.el.
;;
;; This file:
;; - disables package.el and quickstart
```

```
;; - performs startup optimizations (GC, file-name-handlers)
;; - defines base configuration directories (.var / .etc / .cache / lisp)
;; - configures native compilation and Tree-sitter paths
;; - prepares macOS Homebrew toolchain environment
;; - establishes early UI defaults and frame parameters
;;
;;; Code:

(eval-when-compile
  (require 'subr-x))

(require 'seq)


;;; Internal utilities ------------------------------------------------------

(defun core--ensure-directory (dir)
  "Ensure DIR exists, creating it recursively if needed."
  (unless (file-directory-p dir)
    (condition-case err
        (make-directory dir t)
      (error
       (warn "early-init: failed to create %s (%s)"
             dir (error-message-string err))))))

(defun core--login-username ()
  "Return login username or nil."
  (ignore-errors (user-login-name)))

(defvar core--orig-file-name-handler-alist nil
  "Original `file-name-handler-alist' saved for restoration.")

(defun core--restore-startup-state ()
  "Restore GC and file handler settings after startup."
  (setq file-name-handler-alist core--orig-file-name-handler-alist
        gc-cons-threshold 16777216
        gc-cons-percentage 0.1))

(defalias 'my/ensure-directory-exists #'core--ensure-directory)


;;; Disable package.el ------------------------------------------------------

(setq package-enable-at-startup nil
      package-quickstart nil)


;;; Base directories --------------------------------------------------------

(defvar my:d
  (file-name-as-directory
   (or (and load-file-name
            (file-name-directory (file-chase-links load-file-name)))
```

```
       user-emacs-directory))
  "Root directory of this Emacs configuration.")

(setq user-emacs-directory my:d)

(defconst my:d:var       (expand-file-name ".var/" my:d))
(defconst my:d:etc       (expand-file-name ".etc/" my:d))
(defconst my:d:lisp      (expand-file-name "lisp/" my:d))

(when (and (boundp 'my:d:lisp)
           (file-directory-p my:d:lisp))
  (dolist (dir (directory-files my:d:lisp t "\\`[^.]"))
    (when (file-directory-p dir)
      (add-to-list 'load-path dir))))

(defconst my:d:cache
  (expand-file-name
   "emacs/"
   (or (getenv "XDG_CACHE_HOME")
       (expand-file-name ".cache/" my:d))))
(defconst my:d:eln-cache
  (expand-file-name "eln-cache/" my:d:cache))

(defconst my:d:treesit
  (expand-file-name "tree-sitter/" my:d:var))
(setq treesit-install-dir my:d:treesit)
(setq treesit-extra-load-path (list my:d:treesit))
(unless (file-directory-p treesit-install-dir)
  (make-directory treesit-install-dir t))

(defconst my:d:url       (expand-file-name "url/" my:d:var))
(defconst my:d:eww       (expand-file-name "eww/" my:d:var))

(dolist (dir (list my:d:var my:d:etc my:d:lisp my:d:cache
                   my:d:eln-cache my:d:treesit my:d:url my:d:eww))
  (core--ensure-directory dir))


;;; macOS Homebrew toolchain ------------------------------------------------

(when (eq system-type 'darwin)
  (when-let* ((brew (or (getenv "HOMEBREW_PREFIX")
                        (and (file-directory-p "/opt/homebrew") "/opt/homebrew")
                        (and (file-directory-p "/usr/local")   "/usr/local")))
              (bin  (expand-file-name "bin" brew)))
    (when (file-directory-p bin)
      (let* ((path  (or (getenv "PATH") ""))
             (parts (split-string path ":" t)))
        (unless (member bin parts)
          (setenv "PATH" (concat bin ":" path)))))

    (let* ((libgccjit (expand-file-name "opt/libgccjit" brew))
           (gcc       (expand-file-name "opt/gcc" brew))
```

```
           (candidates
            (seq-filter
             #'file-directory-p
             (list (expand-file-name "lib/gcc/current" libgccjit)
                   (expand-file-name "lib" libgccjit)
                   (expand-file-name "lib/gcc/current" gcc)))))
      (when candidates
        (setenv "LIBRARY_PATH"
                (string-join
                 (delete-dups
                  (append candidates
                          (when-let* ((old (getenv "LIBRARY_PATH")))
                            (split-string old ":" t))))
                 ":"))))

    (when-let* ((gcc-bin
                 (seq-find
                  #'file-exists-p
                  (mapcar
                   (lambda (n)
                     (expand-file-name (format "gcc-%d" n) bin))
                   (number-sequence 20 10 -1)))))
      (setenv "CC" gcc-bin))))


;;; Native compilation ------------------------------------------------------

(when (and (boundp 'native-comp-eln-load-path)
           (listp native-comp-eln-load-path))
  (setopt native-comp-eln-load-path
          (cons my:d:eln-cache
                (delq my:d:eln-cache native-comp-eln-load-path))
          native-comp-async-report-warnings-errors 'silent))


;;; no-littering compatibility ----------------------------------------------

(defvar no-littering-etc-directory (file-name-as-directory my:d:etc))
(defvar no-littering-var-directory (file-name-as-directory my:d:var))


;;; straight.el base --------------------------------------------------------

(setopt straight-base-dir my:d:cache
        straight-use-package-by-default t
        straight-vc-git-default-clone-depth 1
        straight-build-dir
        (format "build-%d.%d" emacs-major-version emacs-minor-version)
        straight-profiles '((nil . "default.el")))


;;; Startup performance -----------------------------------------------------
```

```
(setq core--orig-file-name-handler-alist file-name-handler-alist)

(setq file-name-handler-alist
      (seq-remove
       (lambda (h)
         (let ((fn (cdr h)))
           (and (symbolp fn)
                (string-match-p "\\`\\(tramp\\|jka-compr\\)"
                                (symbol-name fn)))))
       file-name-handler-alist))

(setq gc-cons-threshold most-positive-fixnum
      gc-cons-percentage 0.6)

(add-hook 'emacs-startup-hook #'core--restore-startup-state)


;;; Backups / auto-save ------------------------------------------------------

(setq make-backup-files nil
      version-control nil
      delete-old-versions nil
      backup-by-copying nil
      auto-save-default nil
      auto-save-list-file-prefix nil)


;;; Early UI defaults --------------------------------------------------------

(setopt frame-resize-pixelwise t
        frame-inhibit-implied-resize t
        cursor-in-non-selected-windows nil
        x-underline-at-descent-line t
        window-divider-default-right-width 16
        window-divider-default-places 'right-only)

;; NOTE:
;; fullscreen is intentionally NOT set via frame alists.
;; It is applied once via `window-setup-hook` to avoid double resize/repaint.
(dolist (it '((internal-border-width . 8)
              (tool-bar-lines . 0)))
  (add-to-list 'default-frame-alist it)
  (add-to-list 'initial-frame-alist it))

(setq default-frame-alist
      (assq-delete-all 'fullscreen default-frame-alist))
(setq initial-frame-alist
      (assq-delete-all 'fullscreen initial-frame-alist))

(defvar core--did-fullscreen nil
  "Non-nil once fullscreen is applied to the initial frame.")

(add-hook 'window-setup-hook
```

```
          (lambda ()
            (unless core--did-fullscreen
              (setq core--did-fullscreen t)
              (set-frame-parameter nil 'fullscreen 'fullboth))))

(when (fboundp 'menu-bar-mode)   (menu-bar-mode -1))
(when (fboundp 'tool-bar-mode)   (tool-bar-mode -1))
(when (fboundp 'scroll-bar-mode) (scroll-bar-mode -1))


;;; Startup echo ------------------------------------------------------------

(when-let* ((u (core--login-username)))
  (setq inhibit-startup-echo-area-message u))


(provide 'early-init)
;;; early-init.el ends here
```

2.1.3 init.el

```
;;; init.el --- Main initialization entry point -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Primary initialization sequence for Emacs 30+.
;;
;; This file:
;; - bootstraps straight.el, leaf, and Org
;; - establishes core performance and runtime defaults
;; - loads personal overlays safely
;; - invokes the modular configuration loader
;; - reports startup metrics after initialization
;;
;;; Code:

(require 'subr-x)
(require 'seq)
(require 'cl-lib)

;; ;; Enable debugger on error (temporary)
;; (setq debug-on-error t)

;;; Internal helpers ---------------------------------------------------------

(defun utils--safe-load-file (file &optional noerror)
  "Load FILE safely.
If NOERROR is non-nil, log instead of raising."
  (when (and (stringp file) (file-exists-p file))
```

```
    (condition-case err
        (load file nil 'nomessage)
      (error
       (funcall (if noerror #'message #'user-error)
                "[load] failed: %s (%s)"
                file (error-message-string err))))))

(defalias 'my/safe-load-file #'utils--safe-load-file)


;;; 0) URL state BEFORE url.el ------------------------------------------------

(defvar core--url-state-dir
  (file-name-as-directory
   (or (bound-and-true-p my:d:url)
       (expand-file-name "url/" user-emacs-directory))))

(setopt url-configuration-directory core--url-state-dir
        url-cookie-file (expand-file-name "cookies" core--url-state-dir)
        url-history-file (expand-file-name "history" core--url-state-dir)
        url-cache-directory (expand-file-name "cache/" core--url-state-dir))

(dolist (d (list url-configuration-directory url-cache-directory))
  (make-directory d t))

(require 'url)


;;; 1) Bootstrap straight.el --------------------------------------------------

(defvar bootstrap-version 7)

(let* ((base (or (bound-and-true-p straight-base-dir)
                 user-emacs-directory))
       (bootstrap-file
        (expand-file-name "straight/repos/straight.el/bootstrap.el" base)))
  (unless (file-exists-p bootstrap-file)
    (let ((buf
           (url-retrieve-synchronously
            "https://raw.githubusercontent.com/radian-software/straight.el/develop/install.el"
            'silent 'inhibit-cookies)))
      (unless (buffer-live-p buf)
        (user-error "[straight] failed to retrieve install.el"))
      (with-current-buffer buf
        (goto-char (point-max))
        (eval-print-last-sexp))))
  (load bootstrap-file nil 'nomessage))


;;; 1.1) leaf / org -----------------------------------------------------------

(dolist (pkg '(leaf leaf-keywords))
  (straight-use-package pkg))
```

```
(require 'leaf)

(eval-when-compile
  (require 'leaf-keywords))

(when (fboundp 'leaf-keywords-init)
  (leaf-keywords-init))

(straight-use-package 'org)
(require 'org)


;;; 1.2) macOS environment ----------------------------------------------------

(leaf exec-path-from-shell
  :straight t
  :when (and (eq system-type 'darwin)
             (or (daemonp) (memq window-system '(mac ns))))
  :config
  (setq exec-path-from-shell-check-startup-files nil
        exec-path-from-shell-arguments '("-l" "-i"))
  (exec-path-from-shell-copy-envs
   '("PATH" "LANG"
     "PASSWORD_STORE_DIR"
     "GPG_KEY_ID"
     "OPENROUTER_API_KEY"
     "OPENAI_API_KEY"))
  (exec-path-from-shell-initialize))


;;; 2) Performance ------------------------------------------------------------

(defvar core--orig-read-process-output-max
  (and (boundp 'read-process-output-max)
       read-process-output-max))

(when (boundp 'read-process-output-max)
  (setq read-process-output-max (* 4 1024 1024)))

(add-hook 'after-init-hook
          (lambda ()
            (when (boundp 'read-process-output-max)
              (setq read-process-output-max
                    core--orig-read-process-output-max))))

(leaf gcmh
  :straight t
  :custom
  ((gcmh-idle-delay . 5)
   (gcmh-high-cons-threshold . 16777216))
  :config
  (gcmh-mode 1))
```

```
;;; 3) Core built-ins ----------------------------------------------------------

(leaf emacs
  :straight nil
  :hook
  ((prog-mode . display-line-numbers-mode))
  :custom
  ((inhibit-startup-screen . t)
   (inhibit-startup-message . t)
   (initial-scratch-message . nil)
   (initial-major-mode . 'fundamental-mode)
   (use-short-answers . t)
   (create-lockfiles . nil)
   (idle-update-delay . 0.2)
   (ring-bell-function . #'ignore)
   (display-line-numbers-type . 'relative)
   (frame-title-format . t)
   (confirm-kill-emacs . #'y-or-n-p))
  :config
  (when (fboundp 'window-divider-mode)
    (window-divider-mode 1))
  (when (fboundp 'pixel-scroll-precision-mode)
    (pixel-scroll-precision-mode 1))
  (when (fboundp 'electric-pair-mode)
    (electric-pair-mode 1))
  (dolist (k '("C-z" "C-x C-z" "C-x C-c"))
    (keymap-global-unset k)))


;;; 4) Modifier keys -----------------------------------------------------------

(leaf my:modifier
  :straight nil
  :config
  (pcase system-type
    ('darwin
     (setq mac-option-modifier 'meta
           mac-command-modifier 'super
           mac-control-modifier 'control))
    ('windows-nt
     (setq w32-lwindow-modifier 'super
           w32-rwindow-modifier 'super))))


;;; 5) Personal overlay --------------------------------------------------------

(let* ((root (or (bound-and-true-p my:d) user-emacs-directory))
       (personal (expand-file-name "personal/" root))
       (user (ignore-errors (user-login-name))))
  ;; Add personal directory to load-path
  (when (file-directory-p personal)
```

```
    (add-to-list 'load-path personal))
  ;; Load personal files
  (my/safe-load-file (expand-file-name "user.el" personal) t)
  (when user
    (my/safe-load-file
     (expand-file-name (concat user ".el") personal) t)))


;;; 6) Modules entrypoint ----------------------------------------------------

(defvar my:modules-extra nil
  "Extra module list appended by optional layers.")

(let* ((root (or (bound-and-true-p my:d) user-emacs-directory))
       (lisp-dir (expand-file-name "lisp/" root)))
  (when (file-directory-p lisp-dir)
    (add-to-list 'load-path lisp-dir))
  (require 'core-custom-ui-extras nil t)
  (require 'modules nil t))


;;; 7) Startup message -------------------------------------------------------

(defun core--announce-startup ()
  "Report startup time and GC count."
  (message "Emacs ready in %.2f seconds with %d GCs."
           (float-time
            (time-subtract after-init-time before-init-time))
           gcs-done))

(add-hook 'after-init-hook
          (lambda ()
            (run-with-idle-timer 0 nil #'core--announce-startup)))


(provide 'init)
;;; init.el ends here
```

2.2 Modular Loader & Core Module Suite

2.2.1 Overview

1. Purpose Provide a **deterministic, explicit, and auditable module loading architecture** for this Emacs configuration.

   This layer defines *where feature activation begins* after bootstrap, and guarantees that module load order, dependency direction, and extension points are fully inspectable and reproducible.

   All shared behavior is activated through a single, explicit entry point, independent of filesystem traversal or implicit load-path side effects.

2. What this layer defines

   This layer establishes:

   - `modules.el` as the **only authoritative loader** for shared modules
   - A directory-based responsibility split with clear semantic boundaries

- An explicit, ordered loading sequence handed off from `init.el`

Modules are grouped by responsibility:

`core/` fundamental runtime, infrastructure, and global policy

`ui/` visual presentation and interaction layer

`completion/` completion frameworks and CAPF orchestration

`orgx/` Org-mode extensions beyond upstream defaults

`dev/` development, build, and language tooling

`vcs/` version control systems and repository interaction

`utils/` small, domain-specific helpers not promoted upward

3. What this layer does **not** do

   This layer intentionally does **not**:

   - Perform bootstrap or environment preparation (handled exclusively by `early-init.el`)
   - Establish runtime foundations (handled by `init.el`)
   - Define configuration logic, advice, hooks, or feature behavior
   - Discover modules via directory scanning or wildcard expansion
   - Encode user-, device-, or host-specific policy

   Its sole responsibility is **orchestration**, not implementation.

4. Design principles

   - `modules.el` contains **only ordered `require` forms**
   - Each module file:
     - provides exactly one feature
     - documents its own assumptions and non-goals
   - Load order is fixed and intentional:
     (a) `core`
     (b) `ui`
     (c) `completion`
     (d) `orgx`
     (e) `dev`
     (f) `vcs`
     (g) `utils`

   This order reflects dependency direction. Any deviation requires explicit documentation and rationale.

   All modules are written to be:

   - Idempotent (safe to load multiple times)
   - Side-effect conscious
   - Safe under batch, byte-compilation, and native-comp contexts

5. Benefits

   This architecture enables:

- Fully inspectable and reproducible startup behavior
- Predictable dependency relationships
- Selective enable/disable during diagnostics
- Long-term maintainability across Emacs versions
- Dependency visualization and auditing from a single file

6. Relationship to bootstrap

   - `early-init.el` prepares infrastructure only
   - `init.el` establishes runtime foundations and environment state
   - `modules.el` is the **first and only layer** where shared features are activated

   This separation ensures that:

   - bootstrap remains minimal, conservative, and side-effect free
   - feature activation is centralized and reviewable
   - personal overlays remain isolated from global policy

2.2.2 modules.el

- Uses only `require` and comments
- No conditional loading based on environment or features
- Load order reflects semantic dependency layers:
  1. core
  2. ui
  3. completion
  4. orgx
  5. dev
  6. utils
- Every required feature must:
  - be provided by exactly one file
  - correspond to a clearly named module

```
;;; modules.el --- Modular configuration loader -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Central entry point for loading modular configuration under lisp/.
;;
;; This module:
;; - defines the ordered list of configuration modules
;; - loads modules safely with error isolation
;; - provides timing, skip, and extension mechanisms for startup control
```

```
;;
;;; Code:

(eval-when-compile (require 'subr-x))
(require 'seq)
(defgroup my:modules nil
  "Loader options for modular Emacs configuration."
  :group 'convenience)

(defcustom my:modules-verbose t
  "If non-nil, print per-module load time and a summary."
  :type 'boolean
  :group 'my:modules)

(defcustom my:modules-skip nil
  "List of module features to skip during loading."
  :type '(repeat symbol)
  :group 'my:modules)

(defcustom my:modules-extra nil
  "List of extra module features to append after `my:modules'."
  :type '(repeat symbol)
  :group 'my:modules)

(defconst my:modules
  '(
    ;; Core
    core-fixes
    core-policy
    core-session
    core-gc
    core-persistence
    core-buffers
    core-general
    core-tools
    core-treesit
    core-history
    core-editing
    core-switches
    core-custom

    ;; UI
    ui-font
    ui-theme
    ui-window
    ui-utils
    ui-health-modeline
    ui-imenu

    ;; Completion
    completion-core
    completion-vertico
    completion-consult
```

```
    completion-embark
    completion-corfu
    completion-icons
    completion-capf
    completion-capf-org-src
    completion-capf-org-src-lang
    completion-corfu-org-src
    completion-orderless-org-src
    completion-lsp

    ;; Org ecosystem (module namespace = )
    orgx-core
    orgx-visual
    orgx-extensions
    orgx-fold
    orgx-export
    orgx-notes-markdown
    orgx-auto-tangle

    ;; VCS (uncomment when needed)
    vcs-magit
    vcs-gutter
    vcs-forge

    ;; Development
    dev-ai
    dev-term
    dev-web-core
    dev-build
    dev-format
    dev-infra-modes
    dev-docker
    dev-sql
    dev-rest
    dev-navigation
    dev-tools

    ;; Utils
    utils-path
    utils-async
    utils-buffers
    utils-edit
    utils-dired
    utils-functions
    utils-org-agenda
    utils-lint
    utils-diagnostics
    )
  "Default set of modules to load in order.")

(defun my/modules--should-load-p (feature)
  "Return non-nil if FEATURE should be loaded (i.e., not in skip list)."
  (not (memq feature my:modules-skip)))
```

```
(defun my/modules--require-safe (feature)
  "Require FEATURE with error trapping. Return non-nil on success.
Errors are reported via `message' but do not abort the whole loader."
  (condition-case err
      (progn (require feature) t)
    (error
     (message "[modules] Failed to load %s: %s"
              feature (error-message-string err))
     nil)))

(defun my:modules--format-seconds (sec)
  "Format SEC (float seconds) in a compact human-readable form."
  (cond
   ((< sec 0.001) (format "%.3fms" (* sec 1000.0)))
   ((< sec 1.0)   (format "%.1fms"  (* sec 1000.0)))
   (t             (format "%.2fs"   sec))))

(defun my/modules-load ()
  "Load all modules defined by `my:modules', respecting options.
- Honors `my:modules-skip' and `my:modules-extra'.
- Prints per-module timing when `my:modules-verbose' is non-nil.
- Prints a final summary including counts *and* the lists of skipped/failed."
  (let* ((all (append my:modules my:modules-extra))
         (final (seq-remove (lambda (m) (not (my/modules--should-load-p m))) all))
         (skipped (seq-remove (lambda (m) (memq m final)) all))
         (ok 0) (ng 0)
         (failed '())
         (loaded '())
         (t0 (and my:modules-verbose (current-time))))
    (dolist (mod final)
      (let ((m0 (and my:modules-verbose (current-time))))
        (if (my/modules--require-safe mod)
            (progn (setq ok (1+ ok)) (push mod loaded))
          (setq ng (1+ ng)) (push mod failed))
        (when my:modules-verbose
          (message "[modules] %-24s %s"
                   mod (my:modules--format-seconds
                        (float-time (time-subtract (current-time) m0)))))))
    (when my:modules-verbose
      ;; Main summary (backward-compatible)
      (message "[modules] loaded=%d skipped=%d failed=%d total=%s"
               ok (length skipped) ng
               (my:modules--format-seconds
                (float-time (time-subtract (current-time) t0))))
      ;; Detail: skipped targets
      (when skipped
        (message "[modules] skipped (%d): %s"
                 (length skipped)
                 (mapconcat #'symbol-name (nreverse skipped) " ")))
      ;; Detail: failed targets
      (when failed
        (message "[modules] failed  (%d): %s"
```

```
                (length failed)
                (mapconcat #'symbol-name (nreverse failed) " "))))
    ok))

(my/modules-load)

(provide 'modules)
;;; modules.el ends here
```

2.2.3 core/

1. Overview

   - `early-init.el` prepares the physical environment and startup invariants
   - `init.el` establishes runtime foundations and hands control to `modules.el`
   - `core/` is the **first feature layer** activated after bootstrap

   This ordering guarantees that all higher layers operate on a stable, predictable, and explicitly defined foundation.

   (a) Purpose Provide the **foundational runtime, policy, and infrastructure layer** required by all other shared modules.

   This layer establishes **hard guarantees** about execution context, availability of primitives, and global defaults that higher layers may rely on **without defensive checks or conditional logic**.

   Core exists to make the rest of the configuration simpler, safer, and more predictable.

   (b) What this layer does Core modules define the **non-negotiable baseline** of the entire configuration.

   This layer is responsible for:

   - Establishing low-level behavior shared globally
   - Defining configuration-wide policy and invariants
   - Initializing essential subsystems with long-lived impact
   - Providing stable primitives consumed by all higher layers

   Typical responsibilities include:

   - Compatibility guards and forward-looking fixes across Emacs versions
   - Session lifecycle orchestration and global health thresholds
   - Global keybinding infrastructure and command scaffolding
   - Editing, history, persistence, and filesystem interaction policy
   - Tree-sitter and parsing infrastructure with explicit grammar control
   - Feature switches that gate downstream activation (UI, LSP, etc.)

   (c) What this layer does **not** do Core modules intentionally do **not**:

   - Implement visual presentation, theming, or UI styling
   - Define completion UX or minibuffer interaction patterns
   - Extend Org semantics beyond upstream defaults
   - Configure language servers, build tools, or VCS workflows
   - Encode user-, device-, or host-specific preferences

   Those responsibilities are explicitly delegated to higher layers or to `personal/` overlays.

   (d) Design constraints

   - Core modules must not depend on:

  - `ui`
  - `completion`
  - `orgx`
  - `dev`
  - `vcs`
  - `utils`
- All side effects must be:
  - explicit
  - documented
  - global in intent
- Failure in this layer is **configuration-critical** and must:
  - surface early
  - fail loudly
  - never degrade silently

(e) Implementation principles

- Loaded **first** by `modules.el`
- Each file:
  - provides exactly one feature
  - documents its scope, assumptions, and non-goals
- Optional behavior is gated behind explicit customization variables
- No module assumes:
  - interactive usage
  - user presence
  - GUI availability

Core must remain safe under batch, byte-compilation, native-compilation, and daemon contexts.

(f) Benefits This separation ensures that:

- Higher layers can rely on stable, unconditional primitives
- Debugging starts from a known, deterministic baseline
- Policy decisions are centralized and reviewable
- Long-running sessions remain predictable and observable

(g) Module map

File	Responsibility
`core/fixes.el`	Compatibility guards and minimal hotfixes scoped by Emacs version
`core/core-session.el`	Long-running session orchestration and global health policy
`core/general.el`	Global keybinding infrastructure and non-modal leader layout
`core/tools.el`	Cross-cutting helper commands for navigation, inspection, and toolin
`core/utils.el`	Core-level utility helpers and global hooks shared across layers
`core/core-treesit.el`	Centralized Tree-sitter infrastructure and grammar policy
`core/core-history.el`	Session persistence (history, places, recent files)
`core/editing.el`	Editing behavior, UX aids, and filesystem interaction policy
`core/switches.el`	Unified switches controlling UI and LSP activation
`core/custom.el`	Routing and management of Customize output (`custom-file`)
`core/custom-ui-extras.el`	Optional UI-related module extensions appended at runtime

(h) Notes

- Core is the **only** layer allowed to define global invariants.

- Any logic that depends on UI, completion, or Org semantics is misplaced here.
- If a change in core feels "convenient" rather than "necessary" , it likely belongs elsewhere.

This layer exists to make everything above it **boring, stable, and trustworthy**.

2. core/core-fixes.el

```
;;; core/core-fixes.el --- Compatibility & hotfix layer -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Minimal and version-scoped compatibility fixes for Emacs.
;;
;; This module:
;; - applies guards only for observed or expected breakage
;; - keeps fixes explicitly version-bounded
;; - avoids architectural or behavioral refactoring
;;
;;; Code:


;;;; Utilities ----------------------------------------------------------

(defun core-fixes--emacs>= (major minor)
  "Return non-nil if running Emacs version is >= MAJOR.MINOR."
  (or (> emacs-major-version major)
      (and (= emacs-major-version major)
           (>= emacs-minor-version minor))))

;;;; Version-scoped fixes -----------------------------------------------
;; All fixes below MUST be guarded by explicit version checks.
;; Never widen a guard range casually.

;;;; Advice guards ------------------------------------------------------
;;
;; Emacs 30.1+:
;; Guard against unsafe or duplicated advice application during
;; early bootstrap and module reload.
;;
;; Observed issues motivating this guard:
;; - Re-entrancy during nested `load'
;; - Duplicate advice when modules are reloaded
;; - Unstable behavior when advice is applied before helper functions exist
;;

(when (core-fixes--emacs>= 30 1)

  ;; Defensive checks:
```

```
  ;; - Only apply advice if *all* required functions exist
  ;; - Keep this block reload-safe
  (when (and (fboundp 'load)
             (fboundp 'require)
             (fboundp 'my:with-thisfile--load)
             (fboundp 'my:with-thisfile--require))

    ;; Avoid duplicate advice on `load'
    (unless (advice-member-p #'my:with-thisfile--load 'load)
      (advice-add 'load :around #'my:with-thisfile--load))

    ;; Avoid duplicate advice on `require'
    (unless (advice-member-p #'my:with-thisfile--require 'require)
      (advice-add 'require :around #'my:with-thisfile--require))))

;;;; Disabled / retired fixes ------------------------------------------------
;;
;; Keep removed fixes here *temporarily* with comments explaining:
;; - Why they were added
;; - Which Emacs version fixed the root cause
;;
;; Safe to delete once the minimum supported Emacs version
;; moves beyond the affected range.
;;

;;;; Forward-compatibility notes ---------------------------------------------
;;
;; - If Emacs 30.2+ resolves the underlying issues guarded above,
;;   this module should be narrowed or partially removed.
;; - Do NOT expand version ranges without a concrete regression.
;; - Prefer deleting fixes over accumulating them.
;;

(provide 'core-fixes)
;;; core/core-fixes.el ends here
```

3. core/core-policy.el

```
;;; core/core-policy.el --- Core runtime policy hooks -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Global runtime policies enforced via core-level hooks.
;;
;; This module:
;; - defines editor-wide behavioral guarantees
;; - installs carefully scoped global hooks
;; - centralizes policy that must not live in utils or dev
```

```
;;
;;; Code:

(defun my/auto-insert-lexical-binding ()
  "Insert `lexical-binding: t` in Emacs Lisp files under `no-littering-var-
directory`."
  (when (and (stringp buffer-file-name)
             (boundp 'no-littering-var-directory)
             (string-prefix-p (expand-file-name no-littering-var-directory)
                              (expand-file-name buffer-file-name))
             (string-match-p "\\.el\\'" buffer-file-name)
             (not (save-excursion
                    (goto-char (point-min))
                    (re-search-forward "lexical-binding" (line-end-
position 5) t))))
    (save-excursion
      (goto-char (point-min))
      (insert ";; -*- lexical-binding: t; -*- \n"))))
(add-hook 'find-file-hook #'my/auto-insert-lexical-binding)

(defun my/enable-view-mode-on-read-only ()
  (if buffer-read-only
      (view-mode 1)
    (view-mode -1)))
(add-hook 'read-only-mode-hook #'my/enable-view-mode-on-read-only)

(provide 'core-policy)
;;; core/core-policy.el ends here
```

4. core/core-session.el

```
;;; core/core-session.el --- Long-running session orchestration -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Central orchestration layer for long-running Emacs sessions.
;;
;; This module:
;; - defines when maintenance tasks run
;; - evaluates session risk heuristics
;; - delegates actual work to utils modules
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))
```

```
;; ------------------------------------------------------------------------
;; Customization
;; ------------------------------------------------------------------------

(defgroup core-session nil
  "Long-running Emacs session orchestration."
  :group 'convenience)

(defcustom core-session-enable-p t
  "Enable long-running session orchestration."
  :type 'boolean
  :group 'core-session)

(defcustom core-session-idle-delay
  (* 30 60)
  "Seconds of idle time before running lightweight maintenance."
  :type 'integer
  :group 'core-session)

(defcustom core-session-periodic-interval
  600
  "Interval in seconds for periodic maintenance tasks."
  :type 'integer
  :group 'core-session)

(defcustom core-session-buffer-threshold
  300
  "Soft threshold for number of live buffers considered risky."
  :type 'integer
  :group 'core-session)

(defcustom core-session-process-threshold
  8
  "Soft threshold for number of live processes considered risky."
  :type 'integer
  :group 'core-session)

;; ------------------------------------------------------------------------
;; Internal helpers
;; ------------------------------------------------------------------------

(defvar core-session--idle-timer nil
  "Idle timer for lightweight session maintenance.")

(defvar core-session--periodic-timer nil
  "Periodic timer for session health checks.")

(defun core-session--buffers-count ()
  "Return the number of live buffers."
  (length (buffer-list)))

(defun core-session--processes-count ()
  "Return the number of live processes."
```

```
  (length (process-list)))

(defun core-session--risky-state-p ()
  "Return non-nil if the current session looks risky."
  (or (> (core-session--buffers-count)
         core-session-buffer-threshold)
      (> (core-session--processes-count)
         core-session-process-threshold)))

;; ---------------------------------------------------------------------------
;; Maintenance actions (delegation only)
;; ---------------------------------------------------------------------------

(defun core-session--lightweight-maintenance ()
  "Run lightweight maintenance tasks.

This function delegates actual work to utils modules and must remain safe."
  (when core-session-enable-p
    ;; GC helpers
    (when (fboundp 'utils-gc--collect)
      (utils-gc--collect))

    ;; Buffer housekeeping
    (when (fboundp 'utils-buffers-cleanup)
      (utils-buffers-cleanup))))

(defun core-session--periodic-check ()
  "Run periodic session health checks.

Currently this only performs maintenance when the session looks risky."
  (when (and core-session-enable-p
             (core-session--risky-state-p))
    (core-session--lightweight-maintenance)))

;; ---------------------------------------------------------------------------
;; Public commands
;; ---------------------------------------------------------------------------

;;;###autoload
(defun core-session-run-health-check ()
  "Run a manual session health check."
  (interactive)
  (core-session--lightweight-maintenance)
  (message "Core session health check completed"))

;;;###autoload
(defun core-session-lightweight-restart ()
  "Perform a safe lightweight restart of the current Emacs session.

This shuts down obsolete LSP servers, cleans buffers, and runs GC.
No buffers with unsaved changes are touched."
  (interactive)
  ;; LSP lifecycle cleanup
```

```
  (when (and core-session-enable-p
             (fboundp 'utils-lsp-on-project-switch))
    (ignore-errors
      (utils-lsp-on-project-switch)))

  ;; Buffers and GC
  (core-session--lightweight-maintenance)

  (clear-image-cache)
  (message "Core session lightweight restart completed"))

;; ---------------------------------------------------------------------------
;; Activation
;; ---------------------------------------------------------------------------

(defun core-session--enable ()
  "Enable core session orchestration."
  ;; Idle maintenance
  (setq core-session--idle-timer
        (run-with-idle-timer
         core-session-idle-delay
         t
         #'core-session--lightweight-maintenance))

  ;; Periodic checks
  (setq core-session--periodic-timer
        (run-with-timer
         core-session-periodic-interval
         core-session-periodic-interval
         #'core-session--periodic-check)))

(defun core-session--disable ()
  "Disable core session orchestration."
  (when (timerp core-session--idle-timer)
    (cancel-timer core-session--idle-timer))
  (when (timerp core-session--periodic-timer)
    (cancel-timer core-session--periodic-timer))
  (setq core-session--idle-timer nil
        core-session--periodic-timer nil))

(when core-session-enable-p
  (core-session--enable))

(provide 'core-session)
;;; core/core-session.el ends here
```

5. core/core-gc.el

```
;;; core/core-gc.el --- Safe garbage collection helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
```

```
;;
;; Category: core
;;
;; Commentary:
;; Safe garbage collection for long-running sessions.
;;
;; This module:
;; - triggers GC only at known-safe moments
;; - avoids synchronous or frequent GC pressure
;; - integrates with session orchestration indirectly
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf nil
  :straight nil
  :init
  (defcustom core-gc-enable-p t
    "Enable GC hooks for long-running sessions."
    :type 'boolean
    :group 'core-gc)

  (defun core-gc--collect ()
    "Run garbage collection safely."
    (when core-gc-enable-p
      (condition-case _err
          (garbage-collect)
        (error nil))))

  (add-hook 'focus-out-hook #'core-gc--collect)
  (add-hook 'minibuffer-exit-hook #'core-gc--collect))

(provide 'core-gc)
;;; core/core-gc.el ends here
```

6. core/core-buffers.el

```
;;; core/core-buffers.el --- Persistent *scratch* buffer helper -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Ensure persistence and safety of the *scratch* buffer.
;;
;; This module:
;; - recreates *scratch* deterministically when killed
;; - avoids user data loss
;; - introduces no interactive surface
```

```
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf my:scratch-auto-recreate
  :straight nil
  :init
  (defun my/create-scratch-buffer ()
    "Create or reset a `*scratch*` buffer."
    (let ((buf (get-buffer-create "*scratch*")))
      (with-current-buffer buf
        (lisp-interaction-mode)
        (erase-buffer)
        (insert ";; This is a new *scratch* buffer\n\n"))
      buf))

  (defun my/kill-scratch-buffer-advice (buf)
    "If BUF is *scratch*, recreate it shortly after kill."
    (when (string= (buffer-name buf) "*scratch*")
      (run-at-time 0 nil #'my/create-scratch-buffer)))

  (add-hook 'kill-buffer-hook
            (lambda ()
              (my/kill-scratch-buffer-advice (current-buffer)))))

(provide 'core-buffers)
;;; core/core-buffers.el ends here
```

7. core/core-persistence.el

```
;;; core/core-persistence.el --- Backup and auto-save helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Backup and persistence hygiene helpers.
;;
;; This module:
;; - enforces retention policies for backups
;; - runs cleanup at controlled lifecycle points
;; - avoids background or continuous scanning
;;
;;; Code:

(defun my/delete-old-backups ()
  "Delete backup files older than 7 days."
  (interactive)
  (let ((backup-dir (concat no-littering-var-directory "backup/"))
```

```
        (threshold (- (float-time (current-time)) (* 7 24 60 60))))
    (when (file-directory-p backup-dir)
      (dolist (file (directory-files backup-dir t))
        (when (and (file-regular-p file)
                   (< (float-time
                       (file-attribute-modification-time
                        (file-attributes file)))
                      threshold))
          (delete-file file))))))

(add-hook 'emacs-startup-hook #'my/delete-old-backups)

(provide 'core-persistence)
;;; core/core-persistence.el ends here
```

8. core/core-general.el

```
;;; core/core-general.el --- General settings & keybindings (NO Meow) -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Global editing defaults and non-modal keybinding infrastructure.
;;
;; This module:
;; - defines a global leader-key system
;; - provides LSP-agnostic editor helpers
;; - centralizes global bindings and authentication helpers
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'leaf-keywords)
  (require 'subr-x))

(defvar plstore-secret-keys)
(defvar plstore-encrypt-to)

(autoload 'magit-status "magit")
(autoload 'magit-blame-addition "magit")
(autoload 'magit-log-current "magit")
(autoload 'magit-diff-buffer-file "magit")
(autoload 'magit-commit "magit")

(autoload 'flymake-goto-next-error "flymake")
(autoload 'flymake-goto-prev-error "flymake")
(autoload 'flymake-show-buffer-diagnostics "flymake")
```

```
(autoload 'project-roots "project")

;;;; Text scaling hydra ------------------------------------------------------
(leaf hydra
  :straight t
  :config
  (defhydra core-hydra-text-scale (:hint nil :color red)
    "
^Text Scaling^
[_+_] increase   [_-_] decrease   [_0_] reset   [_q_] quit
"
    ("+" text-scale-increase)
    ("-" text-scale-decrease)
    ("0" (text-scale-set 0) :color blue)
    ("q" nil "quit" :color blue)))

;;;; Small utilities ---------------------------------------------------------
(leaf my:utils
  :straight nil
  :init
  (defun my/new-frame-with-scratch ()
    "Create a new frame and switch to a fresh buffer."
    (interactive)
    (let ((frame (make-frame)))
      (with-selected-frame frame
        (switch-to-buffer (generate-new-buffer "untitled")))))

  (defun my/restart-or-exit ()
    "Restart Emacs if `restart-emacs' exists; otherwise save & exit."
    (interactive)
    (if (fboundp 'restart-emacs)
        (restart-emacs)
      (save-buffers-kill-emacs)))

  ;; Arrow-based window motions (keeps default muscle memory).
  (windmove-default-keybindings))

;;;; IDE-agnostic helpers (Eglot / lsp-mode) ---------------------------------
(defun my/code-actions ()
  "Run code actions via Eglot or lsp-mode."
  (interactive)
  (cond
   ((fboundp 'eglot-code-actions) (eglot-code-actions))
   ((fboundp 'lsp-execute-code-action) (lsp-execute-code-action))
   (t (user-error "No code action backend (Eglot/LSP) available"))))

(defun my/rename-symbol ()
  "Rename symbol via Eglot or lsp-mode."
  (interactive)
  (cond
   ((fboundp 'eglot-rename) (eglot-rename))
   ((fboundp 'lsp-rename) (lsp-rename))
   (t (user-error "No rename backend (Eglot/LSP) available"))))
```

```
(defun my/format-buffer ()
  "Format buffer via Eglot/LSP; fallback to `indent-region'."
  (interactive)
  (cond
   ((fboundp 'eglot-format-buffer) (eglot-format-buffer))
   ((fboundp 'lsp-format-buffer) (lsp-format-buffer))
   ((fboundp 'indent-region) (indent-region (point-min) (point-max)))
   (t (user-error "No formatter available"))))

(defun my/consult-ripgrep-project ()
  "Run ripgrep in current project; fallback to prompting."
  (interactive)
  (let* ((pr (when (fboundp 'project-current) (project-current)))
         (root (when pr (car (project-roots pr)))))
    (if (and root (fboundp 'consult-ripgrep))
        (consult-ripgrep root)
      (call-interactively 'consult-ripgrep))))

(defun my/toggle-transient-line-numbers ()
  "Toggle line numbers, preserving buffer-local overrides."
  (interactive)
  (if (bound-and-true-p display-line-numbers-mode)
      (display-line-numbers-mode 0)
    (display-line-numbers-mode 1)))

;;;; Global LEADER (non-modal) ------------------------------------------------
;; Define leader keys and prefix maps:
(defconst my:leader-key "C-c SPC"
  "Key sequence used as the global leader key.")

(defconst my:leader-which-prefix "C-c SPC"
  "Human-readable leader prefix string for which-key labels.")

;; Define a local leader key sequence for major mode commands (contextual).
(defconst my:local-leader-key (concat my:leader-key " m")
  "Key sequence used as the local (major-mode) leader key.")

;; Top-level leader prefix map and subgroup prefix maps.
(define-prefix-command 'my/leader-map)
(define-prefix-command 'my/leader-b-map) ;; buffers
(define-prefix-command 'my/leader-w-map) ;; windows
(define-prefix-command 'my/leader-p-map) ;; project
(define-prefix-command 'my/leader-g-map) ;; git
(define-prefix-command 'my/leader-c-map) ;; code
(define-prefix-command 'my/leader-e-map) ;; errors/diagnostics
(define-prefix-command 'my/leader-t-map) ;; toggles
(define-prefix-command 'my/leader-o-map) ;; org/roam
(define-prefix-command 'my/leader-m-map) ;; mode-specific (local leader)
(define-prefix-command 'my/leader-a-map) ;; ai
(define-prefix-command 'my/leader-q-map) ;; session/quit
(define-prefix-command 'my/leader-h-map) ;; help
```

```
;; Bind the global leader key to its prefix map.
(when (fboundp 'keymap-global-set)
  (keymap-global-set my:leader-key 'my/leader-map)
  ;; In older Emacs, use (global-set-key (kbd my:leader-key) 'my/leader-map)
  )

;; Bind group prefixes under the leader map.
(define-key my/leader-map (kbd "b") 'my/leader-b-map)
(define-key my/leader-map (kbd "w") 'my/leader-w-map)
(define-key my/leader-map (kbd "p") 'my/leader-p-map)
(define-key my/leader-map (kbd "g") 'my/leader-g-map)
(define-key my/leader-map (kbd "c") 'my/leader-c-map)
(define-key my/leader-map (kbd "e") 'my/leader-e-map)
(define-key my/leader-map (kbd "t") 'my/leader-t-map)
(define-key my/leader-map (kbd "o") 'my/leader-o-map)
(define-key my/leader-map (kbd "m") 'my/leader-m-map)  ;; "m" for major-
mode leader
(define-key my/leader-map (kbd "a") 'my/leader-a-map)
(define-key my/leader-map (kbd "q") 'my/leader-q-map)
(define-key my/leader-map (kbd "h") 'my/leader-h-map)

;; 1) Top-level leader bindings (LEADER <key>)
(define-key my/leader-map (kbd "SPC") #'execute-extended-command) ;; M-x
(define-key my/leader-map (kbd "/")   #'consult-line)
(define-key my/leader-map (kbd ";")   #'comment-or-uncomment-region)
(define-key my/leader-map (kbd "=")   #'er/expand-region)
(define-key my/leader-map (kbd "`")   #'eval-expression)
(define-key my/leader-map (kbd "z")   #'core-hydra-text-scale/body)
;; frequent file and buffer helpers
(define-key my/leader-map (kbd ".")   #'other-window)
(define-key my/leader-map (kbd "f")   #'find-file)
(define-key my/leader-map (kbd "F")   #'find-file-other-window)
(define-key my/leader-map (kbd "O")   #'find-file-other-frame)
(define-key my/leader-map (kbd "r")   #'consult-recent-file)

;; 2) Buffers (LEADER b ...)
(define-key my/leader-b-map (kbd "b") #'consult-buffer)
(define-key my/leader-b-map (kbd "B") #'consult-project-buffer)
(define-key my/leader-b-map (kbd "k") #'my/kill-buffer-smart)
(define-key my/leader-b-map (kbd "n") #'next-buffer)
(define-key my/leader-b-map (kbd "p") #'previous-buffer)
(define-key my/leader-b-map (kbd "r") #'revert-buffer)

;; 3) Windows (LEADER w ...)
(define-key my/leader-w-map (kbd "w") #'ace-window)
(define-key my/leader-w-map (kbd "s") #'split-window-below)
(define-key my/leader-w-map (kbd "v") #'split-window-right)
(define-key my/leader-w-map (kbd "d") #'delete-window)
(define-key my/leader-w-map (kbd "o") #'delete-other-windows)
(define-key my/leader-w-map (kbd "=") #'balance-windows)
(define-key my/leader-w-map (kbd "2") #'my/toggle-window-split)

;; 4) Project (LEADER p ...)
```

```
(define-key my/leader-p-map (kbd "p") #'project-switch-project)
(define-key my/leader-p-map (kbd "f") #'project-find-file)
(define-key my/leader-p-map (kbd "s") #'my/consult-ripgrep-project)
(define-key my/leader-p-map (kbd "b") #'consult-project-buffer)
(define-key my/leader-p-map (kbd "r") #'project-query-replace-regexp)
(define-key my/leader-p-map (kbd "d") #'project-dired)

;; 5) Search (LEADER s ...) - (placed under main map for convenience)
(define-key my/leader-map (kbd "s s") #'consult-line)
(define-key my/leader-map (kbd "s r") #'consult-ripgrep)
(define-key my/leader-map (kbd "s g") #'my/consult-ripgrep-project)
(define-key my/leader-map (kbd "s m") #'consult-imenu)

;; 6) Git (LEADER g ...)
(define-key my/leader-g-map (kbd "s") #'magit-status)
(define-key my/leader-g-map (kbd "b") #'magit-blame-addition)
(define-key my/leader-g-map (kbd "l") #'magit-log-current)
(define-key my/leader-g-map (kbd "d") #'magit-diff-buffer-file)
(define-key my/leader-g-map (kbd "c") #'magit-commit)

;; 7) Code (LEADER c ...) - LSP-agnostic helpers
(define-key my/leader-c-map (kbd "a") #'my/code-actions)
(define-key my/leader-c-map (kbd "r") #'my/rename-symbol)
(define-key my/leader-c-map (kbd "f") #'my/format-buffer)
(define-key my/leader-c-map (kbd "d") #'xref-find-definitions)
(define-key my/leader-c-map (kbd "D") #'xref-find-definitions-other-window)
(define-key my/leader-c-map (kbd "R") #'xref-find-references)
(define-key my/leader-c-map (kbd "i") #'completion-at-point)

;; 8) Errors/diagnostics (LEADER e ...)
(define-key my/leader-e-map (kbd "n") #'flymake-goto-next-error)
(define-key my/leader-e-map (kbd "p") #'flymake-goto-prev-error)
(define-key my/leader-e-map (kbd "l") #'flymake-show-buffer-diagnostics)

;; 9) Toggles (LEADER t ...)
(define-key my/leader-t-map (kbd "l") #'my/toggle-transient-line-numbers)
(define-key my/leader-t-map (kbd "w") #'whitespace-mode)
(define-key my/leader-t-map (kbd "r") #'read-only-mode)
(define-key my/leader-t-map (kbd "z") #'core-hydra-text-scale/body)
;; Removed toggle for images here, as it's specific to EWW (now local leader in eww-
mode).

;; 10) Org & Roam (LEADER o ...)
(define-key my/leader-o-map (kbd "a") #'org-agenda)
(define-key my/leader-o-map (kbd "c") #'org-capture)
(define-key my/leader-o-map (kbd "i") #'org-roam-node-insert)
(define-key my/leader-o-map (kbd "f") #'org-roam-node-find)
(define-key my/leader-o-map (kbd "s") #'my/org-sidebar)
(define-key my/leader-o-map (kbd "t") #'my/org-sidebar-toggle)

;; 11) Misc/Web - **(Moved to local leader or global C-c w)**
;; (LEADER m ...) previously held EWW (web) commands.
;; We leave my/leader-m-map defined for local leader usage, but no global bindings here n
```

```
;; EWW commands are accessible via global "C-c w" prefix or when in eww-
mode via local leader.

;; 12) AI (LEADER a ...)
(define-key my/leader-a-map (kbd "a") #'aidermacs-transient-menu)

;; 13) Session/quit (LEADER q ...)
(define-key my/leader-q-map (kbd "n") #'my/new-frame-with-scratch)
(define-key my/leader-q-map (kbd "r") #'my/restart-or-exit)
(define-key my/leader-q-map (kbd "q") #'save-buffers-kill-emacs)

;; 14) Help (LEADER h ...)
(define-key my/leader-h-map (kbd "k") #'describe-key)
(define-key my/leader-h-map (kbd "f") #'describe-function)
(define-key my/leader-h-map (kbd "v") #'describe-variable)

;;;; Which-Key integration for leader groups ----------------------------------
(leaf which-key
  :straight t
  :hook (after-init-hook . which-key-mode)
  :custom ((which-key-idle-delay . 0.4))
  :config
  ;; Label leader groups dynamically according to `my:leader-which-prefix`.
  (dolist (it `((,(concat my:leader-which-prefix " b") . "buffers")
                (,(concat my:leader-which-prefix " w") . "windows")
                (,(concat my:leader-which-prefix " p") . "project")
                (,(concat my:leader-which-prefix " s") . "search")
                (,(concat my:leader-which-prefix " g") . "git")
                (,(concat my:leader-which-prefix " c") . "code")
                (,(concat my:leader-which-prefix " e") . "errors")
                (,(concat my:leader-which-prefix " t") . "toggles")
                (,(concat my:leader-which-prefix " o") . "org/roam")
                (,(concat my:leader-which-prefix " m") . "mode")   ;; updated label
                (,(concat my:leader-which-prefix " a") . "ai")
                (,(concat my:leader-which-prefix " q") . "session")
                (,(concat my:leader-which-prefix " h") . "help")))
    (which-key-add-key-based-replacements (car it) (cdr it))))

;;;; Major-mode specific (local leader) bindings ------------------------------
(with-eval-after-load 'dired
  (require 'dired-filter nil t)
  (require 'dired-subtree nil t)
  (add-hook 'dired-mode-hook #'dired-filter-mode)
  (setq dired-subtree-use-backgrounds t)
  ;; ---- Dired Local Keybindings ----
  (define-key dired-mode-map (kbd "TAB") #'dired-subtree-toggle)
  (define-key dired-mode-map (kbd "i")   #'dired-subtree-insert)
  (define-key dired-mode-map (kbd ";")   #'dired-subtree-remove)
  (define-key dired-mode-map (kbd "f")   #'dired-filter-mode)
  (define-key dired-mode-map (kbd "/")   #'dired-filter-map)
  (define-key dired-mode-map (kbd "z")   #'my/dired-view-file-other-window)
  (with-eval-after-load 'which-key
    (which-key-add-key-based-replacements
```

```
      "C-c SPC m TAB" "toggle subtree"
      "C-c SPC m i"   "insert subtree"
      "C-c SPC m ;"   "remove subtree"
      "C-c SPC m f"   "filter mode"
      "C-c SPC m /"   "filter map"
      "C-c SPC m z"   "view in other window")))

(with-eval-after-load 'vterm
  (define-key vterm-mode-map (kbd "C-c SPC m c") #'vterm-send-C-c)
  (define-key vterm-mode-map (kbd "C-c SPC m r") #'vterm-send-return)
  (define-key vterm-mode-map (kbd "C-c SPC m k") #'vterm-reset-cursor-point)
  (define-key vterm-mode-map (kbd "C-c SPC m q") #'vterm-quit)
  (with-eval-after-load 'which-key
    (which-key-add-key-based-replacements
      "C-c SPC m c" "send C-c"
      "C-c SPC m r" "send RET"
      "C-c SPC m q" "quit vterm")))

(with-eval-after-load 'eww
  (define-key eww-mode-map (kbd "s") #'my/eww-search)
  (define-key eww-mode-map (kbd "o") #'eww-open-file)
  (define-key eww-mode-map (kbd "b") #'eww-list-bookmarks)
  (define-key eww-mode-map (kbd "r") #'eww-readable)
  (define-key eww-mode-map (kbd "u") #'my/eww-toggle-images)
  (with-eval-after-load 'which-key
    (which-key-add-key-based-replacements
      "C-c SPC m s" "search"
      "C-c SPC m b" "bookmarks"
      "C-c SPC m u" "toggle images")))

(with-eval-after-load 'org
  (define-key org-mode-map (kbd "C-c SPC m t") #'org-todo)
  (define-key org-mode-map (kbd "C-c SPC m a") #'org-archive-subtree)
  (define-key org-mode-map (kbd "C-c SPC m s") #'org-schedule)
  (define-key org-mode-map (kbd "C-c SPC m d") #'org-deadline)
  (define-key org-mode-map (kbd "C-c SPC m p") #'org-priority)
  (with-eval-after-load 'which-key
    (which-key-add-key-based-replacements
      "C-c SPC m t" "todo"
      "C-c SPC m s" "schedule"
      "C-c SPC m d" "deadline"
      "C-c SPC m p" "priority")))

(with-eval-after-load 'magit
  (define-key magit-mode-map (kbd "C-c SPC m c") #'magit-commit)
  (define-key magit-mode-map (kbd "C-c SPC m p") #'magit-push-current)
  (define-key magit-mode-map (kbd "C-c SPC m f") #'magit-fetch)
  (define-key magit-mode-map (kbd "C-c SPC m l") #'magit-log-buffer-file)
  (define-key magit-mode-map (kbd "C-c SPC m s") #'magit-stage)
  (define-key magit-mode-map (kbd "C-c SPC m u") #'magit-unstage)
  (with-eval-after-load 'which-key
    (which-key-add-key-based-replacements
      "C-c SPC m c" "commit"
```

```
      "C-c SPC m p" "push"
      "C-c SPC m f" "fetch"
      "C-c SPC m s" "stage"
      "C-c SPC m u" "unstage")))

;;;; Notes / Markdown knowledge (C-c n ...) -----------------------------------
;;
;; Markdown-based personal notes (Inkdrop-like workflow).
;;
;; Design:
;; - Kept outside the global leader (C-c SPC)
;; - Lightweight, prose-oriented notes (not tasks)
;; - notes/ is excluded from org-agenda-files (see orgx/org-core.el)
;; - Never override existing user/global bindings

(with-eval-after-load 'utils/utils-notes-markdown
  ;; Define prefix only if it is free
  (unless (lookup-key global-map (kbd "C-c n"))
    (define-prefix-command 'my/notes-prefix)
    (global-set-key (kbd "C-c n") 'my/notes-prefix))

  ;; Sub bindings (guarded)
  (unless (lookup-key global-map (kbd "C-c n f"))
    (define-key my/notes-prefix (kbd "f") #'consult-notes))
  (unless (lookup-key global-map (kbd "C-c n r"))
    (define-key my/notes-prefix (kbd "r") #'my/notes-consult-ripgrep))
  (unless (lookup-key global-map (kbd "C-c n n"))
    (define-key my/notes-prefix (kbd "n") #'my/notes-new-note))
  (unless (lookup-key global-map (kbd "C-c n d"))
    (define-key my/notes-prefix (kbd "d") #'my/notes-open-root)))

;; which-key label (optional, non-fatal)
(with-eval-after-load 'which-key
  (which-key-add-key-based-replacements
    "C-c n" "notes / markdown"))

;;;; Global keybindings (outside leader) ----------------------------------------
;; Define global keys (macOS-like shortcuts, function keys, etc.)
(global-set-key (kbd "<f1>") #'help-command)
(global-set-key (kbd "<f5>") #'my/revert-buffer-quick)   ;; quick revert buffer (if define
(global-set-key (kbd "<f8>") #'treemacs)
(global-set-key (kbd "C-.") #'other-window)
(global-set-key (kbd "C-/") #'undo-fu-only-undo)
(global-set-key (kbd "C-=") #'er/expand-region)
(global-set-key (kbd "C-?") #'undo-fu-only-redo)
(global-set-key (kbd "C-c 0") #'delete-window)
(global-set-key (kbd "C-c 1") #'delete-other-windows)
(global-set-key (kbd "C-c 2") #'my/toggle-window-split)
(global-set-key (kbd "C-c ;") #'comment-or-uncomment-region)
(global-set-key (kbd "C-c M-a") #'align-regexp)
(global-set-key (kbd "C-c V") #'view-file-other-window)
(global-set-key (kbd "C-c a a") #'aidermacs-transient-menu)   ;; global AI menu (duplicate
(global-set-key (kbd "C-c b") #'consult-buffer)
```

```
;; "C-c d ..." org/roam bindings (these may be redundant with leader o):
(global-set-key (kbd "C-c d a") #'org-agenda)
(global-set-key (kbd "C-c d c") #'org-capture)
(global-set-key (kbd "C-c d f") #'org-roam-node-find)
(global-set-key (kbd "C-c d i") #'org-roam-node-insert)
(global-set-key (kbd "C-c d s") #'my/org-sidebar)
(global-set-key (kbd "C-c d t") #'my/org-sidebar-toggle)
(global-set-key (kbd "C-c k") #'my/kill-buffer-smart)
(global-set-key (kbd "C-c l") #'display-line-numbers-mode)    ;; quick toggle line numbers
(global-set-key (kbd "C-c o") #'find-file)
(global-set-key (kbd "C-c r") #'consult-ripgrep)
(global-set-key (kbd "C-c v") #'find-file-read-only)
;; Web/EWW global prefix keys:
(global-set-key (kbd "C-c w b") #'eww-list-bookmarks)
(global-set-key (kbd "C-c w o") #'eww-open-file)
(global-set-key (kbd "C-c w r") #'eww-readable)
(global-set-key (kbd "C-c w s") #'my/eww-search)
(global-set-key (kbd "C-c w u") #'my/eww-toggle-images)
(global-set-key (kbd "C-c w w") #'eww)
(global-set-key (kbd "C-c z") #'core-hydra-text-scale/body)
;; macOS-like Super (s-) keys:
(global-set-key (kbd "C-h") #'backward-delete-char)   ;; Make C-
h backspace (like in terminals/mac)
(global-set-key (kbd "C-s") #'consult-line)           ;; Search in buffer (override isear
(global-set-key (kbd "s-.") #'ace-window)
(global-set-key (kbd "s-<down>") #'end-of-buffer)
(global-set-key (kbd "s-<left>") #'previous-buffer)
(global-set-key (kbd "s-<right>") #'next-buffer)
(global-set-key (kbd "s-<up>") #'beginning-of-buffer)
(global-set-key (kbd "s-b") #'consult-buffer)
(global-set-key (kbd "s-j") #'find-file-other-window)
(global-set-key (kbd "s-m") #'my/new-frame-with-scratch)
(global-set-key (kbd "s-o") #'find-file-other-frame)
(global-set-key (kbd "s-r") #'my/restart-or-exit)
(global-set-key (kbd "s-w") #'ace-swap-window)
(global-set-key (kbd "M-x") #'execute-extended-command)

;;;; Auth / secrets -------------------------------------------------------------
(defvar my:d:password-store
  (or (getenv "PASSWORD_STORE_DIR")
      (concat no-littering-var-directory "password-store/"))
  "Path to the password store.")

(defun my/auth-check-env ()
  "Validate authentication environment and warn if misconfigured."
  (unless (getenv "GPG_KEY_ID")
    (display-warning 'auth "GPG_KEY_ID is not set." :level 'debug))
  (unless (file-directory-p my:d:password-store)
    (display-warning 'auth
                     (format "Password store directory does not exist: %s"
                             my:d:password-store)
                     :level 'warning)))
```

```
(leaf *authentication
  :straight nil
  :init
  (my/auth-check-env)

  (leaf epa-file
    :straight nil
    :commands (epa-file-enable)
    :init
    (setq epa-pinentry-mode
          (if (getenv "USE_GPG_LOOPBACK") 'loopback 'default))
    (add-hook 'emacs-startup-hook #'epa-file-enable))

  (leaf auth-source
    :straight nil
    :init
    (with-eval-after-load 'auth-source
      (let ((key (getenv "GPG_KEY_ID")))
        (if key
            (setq auth-source-gpg-encrypt-to key)
          (display-warning 'auth-source
                           "GPG_KEY_ID is not set. Authentication backends may be limited

  (leaf password-store :straight t)

  (leaf auth-source-pass
    :straight t
    :commands (auth-source-pass-enable)
    :hook (emacs-startup-hook . (lambda ()
                                  (when (executable-find "pass")
                                    (auth-source-pass-enable)))))

  (leaf plstore
    :straight nil
    :init
    (with-eval-after-load 'plstore
      (setq plstore-secret-keys 'silent
            plstore-encrypt-to (getenv "GPG_KEY_ID")))))

(provide 'core-general)
;;; core/core-general.el ends here
```

9. core/core-tools.el

```
;;; core/core-tools.el --- Internal core helper utilities -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
```

```
;; Internal helper primitives for core modules.
;;
;; This module:
;; - provides non-interactive, side-effect-free helpers
;; - avoids hooks, keybindings, and UI concerns
;; - is safe to load at any stage of initialization
;;
;;; Code:

;;;; Version helpers ------------------------------------------------------

(defun core-tools-emacs>= (major minor)
  "Return non-nil if running Emacs version is >= MAJOR.MINOR."
  (or (> emacs-major-version major)
      (and (= emacs-major-version major)
           (>= emacs-minor-version minor))))

;;;; Filesystem helpers ---------------------------------------------------

(defun core-tools-ensure-directory (dir)
  "Ensure directory DIR exists.
Create it recursively if necessary."
  (when (and (stringp dir)
             (not (file-directory-p dir)))
    (make-directory dir t)))

;;;; Feature / function probes --------------------------------------------

(defun core-tools-feature-present-p (feature)
  "Return non-nil if FEATURE can be safely required."
  (or (featurep feature)
      (locate-library (symbol-name feature))))

(defun core-tools-function-present-p (fn)
  "Return non-nil if FN is a callable function."
  (and (symbolp fn) (fboundp fn)))

;;;; Safe require / call --------------------------------------------------

(defun core-tools-require-if-present (feature)
  "Require FEATURE only if it is present.
Return non-nil if successfully loaded."
  (when (core-tools-feature-present-p feature)
    (require feature nil t)))

(defun core-tools-call-if-present (fn &rest args)
  "Call FN with ARGS if FN is defined.
Return the result, or nil if FN is unavailable."
  (when (core-tools-function-present-p fn)
    (apply fn args)))

;;;; Diagnostics helpers --------------------------------------------------
```

```
(defun core-tools-check-provide (feature file)
  "Warn if FILE does not provide FEATURE.
Intended for use from `after-load-functions'."
  (when (and feature file)
    (unless (featurep feature)
      (warn "[core-tools] %s loaded from %s but did not provide `%s`"
            feature file feature))))

;;;; Internal policy helpers ------------------------------------------------

(defun core-tools-user-file-p (file)
  "Return non-nil if FILE belongs to the user configuration."
  (and (stringp file)
       (string-prefix-p
        (expand-file-name user-emacs-directory)
        (expand-file-name file))))

(provide 'core-tools)
;;; core/core-tools.el ends here
```

10. core/core-treesit.el

```
;;; core/core-treesit.el --- Tree-sitter infrastructure layer -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Centralized Tree-sitter integration and policy.
;;
;; This module:
;; - owns Tree-sitter availability and configuration
;; - defines grammar sources and remapping rules
;; - guarantees explicit and non-implicit installation behavior
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))

;; ---------------------------------------------------------------------------
;; Customization
;; ---------------------------------------------------------------------------

(defgroup core-treesit nil
  "Tree-sitter infrastructure layer."
  :group 'convenience)
```

```
(defcustom core-treesit-enable-p t
  "Enable Tree-sitter based major modes when available."
  :type 'boolean
  :group 'core-treesit)

(defcustom core-treesit-auto-install-p nil
  "Automatically install missing Tree-sitter grammars on startup."
  :type 'boolean
  :group 'core-treesit)

(defcustom core-treesit-install-languages
  '(python javascript typescript tsx json css yaml bash toml)
  "Languages considered by `core-treesit-install-all` and auto-install."
  :type '(repeat symbol)
  :group 'core-treesit)

;; ---------------------------------------------------------------------------
;; Availability
;; ---------------------------------------------------------------------------

(defun core-treesit--available-p ()
  "Return non-nil if Tree-sitter is available and enabled."
  (and core-treesit-enable-p
       (fboundp 'treesit-available-p)
       (ignore-errors (treesit-available-p))))

(defun core-treesit--language-available-p (lang)
  "Return non-nil if grammar for LANG is available."
  (and (fboundp 'treesit-language-available-p)
       (ignore-errors (treesit-language-available-p lang))))

;; ---------------------------------------------------------------------------
;; Mode remapping (single source of truth)
;; ---------------------------------------------------------------------------

(defvar core-treesit--mode-remap-alist
  '((python-mode      . python-ts-mode)
    (js-mode          . js-ts-mode)
    (js-json-mode     . json-ts-mode)
    (json-mode        . json-ts-mode)
    (css-mode         . css-ts-mode)
    (typescript-mode  . typescript-ts-mode)
    (tsx-mode         . tsx-ts-mode)
    (yaml-mode        . yaml-ts-mode)
    (sh-mode          . bash-ts-mode)
    (toml-mode        . toml-ts-mode))
  "Explicit major-mode remapping for Tree-sitter.")

(defun core-treesit-apply-remap ()
  "Apply Tree-sitter major-mode remapping when reopening buffers."
  (interactive)
  (when (core-treesit--available-p)
    (dolist (pair core-treesit--mode-remap-alist)
```

```
      (add-to-list 'major-mode-remap-alist pair))))

;; ---------------------------------------------------------------------------
;; Grammar sources (single source of truth)
;; ---------------------------------------------------------------------------

(defvar core-treesit--language-sources
  '((python     . ("https://github.com/tree-sitter/tree-sitter-python"))
    (javascript . ("https://github.com/tree-sitter/tree-sitter-
javascript" "master" "src"))
    (typescript . ("https://github.com/tree-sitter/tree-sitter-
typescript" "master" "typescript/src"))
    (tsx        . ("https://github.com/tree-sitter/tree-sitter-
typescript" "master" "tsx/src"))
    (json       . ("https://github.com/tree-sitter/tree-sitter-json"))
    (css        . ("https://github.com/tree-sitter/tree-sitter-css"))
    (yaml       . ("https://github.com/ikatyang/tree-sitter-yaml"))
    (bash       . ("https://github.com/tree-sitter/tree-sitter-bash"))
    (toml       . ("https://github.com/tree-sitter/tree-sitter-toml")))
  "Alist of (LANG . (REPO [REV] [DIR])) for `treesit-language-source-alist`.")

(defun core-treesit-register-sources ()
  "Register grammar sources into `treesit-language-source-alist`."
  (interactive)
  (when (boundp 'treesit-language-source-alist)
    (dolist (it core-treesit--language-sources)
      (let* ((lang (car it))
             (spec (cdr it))
             (entry (cons lang spec)))
        (add-to-list 'treesit-language-source-alist entry)))))

;; ---------------------------------------------------------------------------
;; Installation (explicit commands)
;; ---------------------------------------------------------------------------

(defun core-treesit-install-grammar (lang)
  "Install Tree-sitter grammar for LANG using registered sources."
  (interactive
   (list
    (intern
     (completing-read
      "Install grammar: "
      (mapcar (lambda (x) (symbol-name (car x))) core-treesit--language-
sources)
      nil t))))
  (unless (core-treesit--available-p)
    (user-error "Tree-sitter is not available or disabled"))
  (unless (fboundp 'treesit-install-language-grammar)
    (user-error "treesit-install-language-grammar is not available in this Emacs"))
  (core-treesit-register-sources)
  (if (core-treesit--language-available-p lang)
      (message "[treesit] already available: %s" lang)
    (message "[treesit] installing: %s" lang)
```

```
    (treesit-install-language-grammar lang)
    (if (core-treesit--language-available-p lang)
        (message "[treesit] installed: %s" lang)
      (message "[treesit] install finished but not detected: %s (check treesit-
extra-load-path)" lang))))

(defun core-treesit-install-all (&optional force)
  "Install all grammars in `core-treesit-install-languages`.
If FORCE is non-nil, attempt installation even if Emacs reports available."
  (interactive "P")
  (unless (core-treesit--available-p)
    (user-error "Tree-sitter is not available or disabled"))
  (core-treesit-register-sources)
  (dolist (lang core-treesit-install-languages)
    (when (or force (not (core-treesit--language-available-p lang)))
      (ignore-errors
        (core-treesit-install-grammar lang)))))

(defun core-treesit-report ()
  "Report Tree-sitter availability and language status."
  (interactive)
  (message
   "[treesit] available=%s, extra-load-path=%S"
   (if (core-treesit--available-p) "yes" "no")
   (when (boundp 'treesit-extra-load-path) treesit-extra-load-path))
  (when (core-treesit--available-p)
    (dolist (lang core-treesit-install-languages)
      (message "[treesit] %s: %s"
               lang
               (if (core-treesit--language-available-p lang) "ok" "missing")))))

;; ---------------------------------------------------------------------------
;; Startup behavior
;; ---------------------------------------------------------------------------

(defun core-treesit--startup-init ()
  "Apply Tree-sitter remapping and optional installation after startup."
  (when (core-treesit--available-p)
    (core-treesit-apply-remap)
    (when core-treesit-auto-install-p
      (ignore-errors (core-treesit-install-all)))))

(add-hook 'emacs-startup-hook #'core-treesit--startup-init)

(provide 'core-treesit)
;;; core/core-treesit.el ends here
```

11. core/core-history.el

```
;;; core/core-history.el --- Session persistence & autorevert -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
```

```
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Persistence of navigation and buffer history.
;;
;; This module:
;; - enables saveplace, recentf, and savehist
;; - stores history under controlled directories
;; - avoids aggressive or implicit restoration
;;
;;; Code:

(eval-when-compile (require 'leaf))

;; ---------------------------------------------------------------------------
;; saveplace
;; ---------------------------------------------------------------------------

(leaf saveplace :straight nil
  :init
  (setq save-place-file (concat no-littering-var-directory "saveplace")))

;; ---------------------------------------------------------------------------
;; recentf
;; ---------------------------------------------------------------------------

(leaf recentf :straight nil
  :init
  (setq recentf-max-saved-items 100
        recentf-save-file (concat no-littering-var-directory "recentf")))

;; ---------------------------------------------------------------------------
;; savehist
;; ---------------------------------------------------------------------------

(leaf savehist
  :straight nil
  :config
  (setq savehist-file (concat no-littering-var-directory "history"))
  (my/ensure-directory-exists (file-name-directory savehist-file))

  ;; `savehist-additional-variables' is defined in savehist.el
  (with-eval-after-load 'savehist
    (add-to-list 'savehist-additional-variables
                 'my:desktop-ask-on-restore)))

;; ---------------------------------------------------------------------------
;; Startup activation
;; ---------------------------------------------------------------------------

(defun core-history--startup-init ()
```

```
  "Enable history-related minor modes after startup."
  (save-place-mode +1)
  (recentf-mode +1)
  (savehist-mode +1))

(add-hook 'emacs-startup-hook #'core-history--startup-init)

(provide 'core-history)
;;; core/history.el ends here
```

12. core/core-editing.el

```
;;; core/core-editing.el --- Editing helpers & UX aids -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;;; Code:

(leaf tramp
  :straight nil
  :pre-setq
  `((tramp-persistency-file-name . ,(concat no-littering-var-
directory "tramp"))
    (tramp-auto-save-directory   . ,(concat no-littering-var-directory "tramp-
autosave")))
  :custom
  '((tramp-default-method . "scp")
    (tramp-verbose        . 3)))

(setopt auto-save-visited-interval 1
        auto-save-default        nil)
(when (fboundp 'auto-save-visited-mode)
  (auto-save-visited-mode 1))

(leaf paredit :straight t
  :hook (emacs-lisp-mode . (lambda ()
                             (enable-paredit-mode)
                             (electric-pair-local-mode -1))))

(leaf paren :straight nil
  :custom ((show-paren-delay . 0)
           (show-paren-style . 'expression)
           (show-paren-highlight-openparen . t))
  :global-minor-mode show-paren-mode)

(leaf puni :straight t
  :global-minor-mode puni-global-mode
  :hook ((minibuffer-setup . (lambda () (puni-global-mode -1)))))

(leaf undo-fu :straight t
```

```
  :custom ((undo-fu-allow-undo-in-region . t)))

(leaf vundo :straight t)

(leaf ace-window :straight t
  :custom ((aw-keys . '(?a ?s ?d ?f ?g ?h ?j ?k ?l))
           (aw-scope . 'frame)
           (aw-background . t))
  :config (ace-window-display-mode 1))

(leaf visual-line-mode :straight nil
  :hook (text-mode . visual-line-mode))

(leaf dired-filter :straight t)
(leaf dired-subtree :straight t :after dired)

(leaf dired :straight nil
  :config
  (if (and (eq system-type 'darwin) (executable-find "gls"))
      (progn
        (setq insert-directory-program "gls"
              dired-use-ls-dired t
              dired-listing-switches "-aBhl --group-directories-first"))
    (setq dired-use-ls-dired nil
          dired-listing-switches "-alh")))

(leaf expand-region :straight t :after treesit)
(leaf aggressive-indent :straight t :hook (prog-mode . aggressive-indent-mode))
(leaf delsel :straight nil :global-minor-mode delete-selection-mode)

(leaf autorevert :straight nil
  :custom ((auto-revert-interval . 2)
           (auto-revert-verbose . nil))
  :global-minor-mode global-auto-revert-mode)

(leaf transient
  :straight t
  :config
  (setq transient-history-file (concat no-littering-var-directory "transient/history.el")
        transient-levels-file  (concat no-littering-var-directory "transient/levels.el")
        transient-values-file  (concat no-littering-var-directory "transient/values.el"))
  (my/ensure-directory-exists (concat no-littering-var-directory "transient/")))

(provide 'core-editing)
;;; core/core-editing.el ends here
```

13. core/core-switches.el

```
;;; core/core-switches.el --- Unified feature switches (UI/LSP) -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
```

```
;;
;; Category: core
;;
;; Commentary:
;; Centralized feature switches for UI bundles and LSP backends.
;;
;; This module:
;; - defines mutually exclusive feature choices
;; - resolves availability at runtime
;; - avoids hard dependencies on optional subsystems
;;
;;; Code:

(eval-when-compile (require 'subr-x))

(when (boundp 'my:use:modules)
  (when (or (not (boundp 'my:use-ui)) (eq my:use-ui 'none))
    (setq my:use-ui my:use:modules)))
(define-obsolete-variable-alias 'my:use:modules 'my:use-ui "2025-10-11")

(defgroup my:switches nil "Unified switches for UI and LSP." :group 'convenience)

(defcustom my:use-lsp 'eglot
  "Which LSP client to use. One of: `eglot`, `lsp`."
  :type '(choice (const eglot) (const lsp))
  :group 'my:switches)

(defcustom my:use-ui 'none
  "Which UI bundle to use. One of: `none`, `doom`, `nano`."
  :type '(choice (const none) (const doom) (const nano))
  :group 'my:switches)

(autoload 'my/ui-enable-doom  "ui/ui-doom-modeline" "Enable Doom UI bundle." t)
(autoload 'my/ui-enable-nano  "ui/ui-nano-modeline" "Enable Nano UI bundle." t)
(autoload 'my/lsp-enable-eglot   "dev/dev-lsp-eglot" "Enable Eglot LSP." t)
(autoload 'my/lsp-enable-lspmode "dev/dev-lsp-mode"  "Enable lsp-mode LSP." t)

(defun my/sw--present-p (kind choice)
  (pcase kind
    ('ui (pcase choice
           ('doom (or (fboundp 'my/ui-enable-doom)
                      (locate-library "ui/ui-doom-modeline")
                      (locate-library "doom-modeline")))
           ('nano (or (fboundp 'my/ui-enable-nano)
                      (locate-library "ui/ui-nano-modeline")
                      (locate-library "nano-modeline")))
           (_ t)))
    ('lsp (pcase choice
            ('eglot (or (fboundp 'my/lsp-enable-eglot)
                        (locate-library "dev/dev-lsp-eglot")
                        (locate-library "eglot")))
            ('lsp   (or (fboundp 'my/lsp-enable-lspmode)
                        (locate-library "dev/dev-lsp-mode")
```

```
                        (locate-library "lsp-mode")))
            (_ nil)))
    (_ nil)))

(defun my/sw--enable-ui (choice)
  (pcase choice
    ('doom (cond
            ((fboundp 'my/ui-enable-doom) (my/ui-enable-doom) t)
            ((locate-library "ui/ui-doom-modeline")
             (load (locate-library "ui/ui-doom-modeline") nil 'nomessage)
             (when (fboundp 'my/ui-enable-doom) (my/ui-enable-doom) t))
            (t (message "[switches] Doom UI not found.") nil)))
    ('nano (cond
            ((fboundp 'my/ui-enable-nano) (my/ui-enable-nano) t)
            ((locate-library "ui/ui-nano-modeline")
             (load (locate-library "ui/ui-nano-modeline") nil 'nomessage)
             (when (fboundp 'my/ui-enable-nano) (my/ui-enable-nano) t))
            (t (message "[switches] Nano UI not found.") nil)))
    ('none (message "[switches] UI bundle disabled.") t)
    (_ (message "[switches] Unknown UI choice: %s" choice) nil)))

(defun my/sw--enable-lsp (choice)
  (pcase choice
    ('eglot (cond
             ((fboundp 'my/lsp-enable-eglot) (my/lsp-enable-eglot) t)
             ((locate-library "dev/dev-lsp-eglot")
              (load (locate-library "dev/dev-lsp-eglot") nil 'nomessage)
              (when (fboundp 'my/lsp-enable-eglot) (my/lsp-enable-eglot) t))
             (t (message "[switches] Eglot setup not found.") nil)))
    ('lsp (cond
           ((fboundp 'my/lsp-enable-lspmode) (my/lsp-enable-lspmode) t)
           ((locate-library "dev/dev-lsp-mode")
            (load (locate-library "dev/dev-lsp-mode") nil 'nomessage)
            (when (fboundp 'my/lsp-enable-lspmode) (my/lsp-enable-lspmode) t))
           (t (message "[switches] lsp-mode setup not found.") nil)))
    (_ (message "[switches] Unknown LSP choice: %s" choice) nil)))

(when (not (eq my:use-ui 'none))
  (let ((present (my/sw--present-p 'ui my:use-ui)))
    (cond
     ((my/sw--enable-ui my:use-ui) (message "[switches] UI bundle: %s" my:use-
ui))
     (present (message "[switches] UI seems present but could not enable: %s" my:use-
ui))
     (t (message "[switches] UI bundle unavailable: %s" my:use-ui)))))

(let ((present (my/sw--present-p 'lsp my:use-lsp)))
  (cond
   ((my/sw--enable-lsp my:use-lsp) (message "[switches] LSP backend: %s" my:use-
lsp))
   (present (message "[switches] LSP seems present but could not enable: %s" my:use-
lsp))
   (t (message "[switches] LSP backend unavailable: %s" my:use-lsp))))
```

```
(provide 'core-switches)
;;; core/core-switches.el ends here
```

14. core/core-custom.el

```
;;; core/core-custom.el --- custom-file helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Centralized handling of Customize output and helpers.
;;
;; This module:
;; - routes customize output to a controlled location
;; - provides explicit commands to inspect or dump state
;; - never applies automatic or implicit persistence
;;
;;; Code:

(eval-when-compile (require 'subr-x))

(defconst my:f:custom
  (or (bound-and-true-p my:f:custom)
      (expand-file-name "custom.el"
                        (or (bound-and-true-p my:d:etc)
                            (expand-file-name ".etc" user-emacs-directory))))
  "Path to the custom-file (Customize output).")

(defun my/custom--ensure-file ()
  "Ensure `custom-file` exists and has a small header."
  (let* ((dir (file-name-directory my:f:custom)))
    (unless (file-directory-p dir)
      (condition-case err
          (make-directory dir t)
        (error
         (warn "[custom] failed to create %s: %s"
               dir (error-message-string err)))))
    (unless (file-exists-p my:f:custom)
      (with-temp-file my:f:custom
        (insert
         ";;; custom.el --- Customize output -*- lexical-binding: t; -*-\n"
         ";; This file is generated by Customize. Edit with care.\n\n")))))

;; Route Customize output
(setq custom-file my:f:custom)
(my/custom--ensure-file)

(when (file-readable-p custom-file)
  (ignore-errors
```

```
    (load custom-file nil 'nomessage)))

;;;###autoload
(defun my/custom-open ()
  "Open the `custom-file`."
  (interactive)
  (my/custom--ensure-file)
  (find-file my:f:custom))

;;;###autoload
(defun my/custom-dump-current ()
  "Persist a curated snapshot of current settings/faces into `custom-file`.
This does not run automatically."
  (interactive)
  (my/custom--ensure-file)

  ;; Variables to persist
  (dolist (pair
           `((inhibit-startup-screen        . ,inhibit-startup-screen)
             (frame-resize-pixelwise        . ,(bound-and-true-p frame-resize-
pixelwise))
             (completion-styles             . ,(and (boundp 'completion-styles)
                                                    completion-styles))
             (completion-category-overrides . ,(and (boundp 'completion-
category-overrides)
                                                    completion-category-
overrides))
             (org-startup-indented           . ,(and (boundp 'org-startup-
indented)
                                                    org-startup-indented))
             (org-hide-leading-stars         . ,(and (boundp 'org-hide-leading-
stars)
                                                    org-hide-leading-stars))
             (org-tags-column                . ,(and (boundp 'org-tags-column)
                                                    org-tags-column))
             (org-agenda-tags-column         . ,(and (boundp 'org-agenda-tags-
column)
                                                    org-agenda-tags-column))))
    (when (car (last pair))
      (customize-save-variable (car pair) (cdr pair))))

  ;; Faces to persist
  (let ((faces
         '((org-modern-date-active
            ((t (:background "#373844" :foreground "#f8f8f2"
                             :height 0.75 :weight light :width condensed))))
           (org-modern-time-active
            ((t (:background "#44475a" :foreground "#f8f8f2"
                             :height 0.75 :weight light :width condensed))))
           (org-modern-tag
            ((t (:background "#44475a" :foreground "#b0b8d1"
                             :height 0.75 :weight light :width condensed)))))))
    (dolist (f faces)
```

```
      (custom-set-faces `(,(car f) ,(cadr f)))))

  (custom-save-all)
  (message "[custom] Wrote snapshot to %s" my:f:custom))

(provide 'core-custom)
;;; core/core-custom.el ends here
```

15. core/core-custom-ui-extras.el

```
;;; core/core-custom-ui-extras.el --- User UI module extensions -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: core
;;
;; Commentary:
;; Append user-specific UI extension modules.
;;
;; This module:
;; - extends the active module list without mutation
;; - remains optional and non-intrusive
;; - performs no configuration itself
;;
;;; Code:

;; Append without touching your default module list.
(setq my:modules-extra
      (delete-dups
       (append my:modules-extra
               '(ui-visual-aids
                 orgx-typography
                 ui-macos))))

(provide 'core-custom-ui-extras)
;;; core/core-custom-ui-extras.el ends here
```

2.2.4 completion/

1. Overview

   - Loaded after `core` and `ui`
   - Each file:
     - provides exactly one completion feature
     - documents its assumptions and non-goals
   - Disabling this entire layer must leave Emacs functional, with only basic ‘completion-at-point‘ behavior

   (a) Purpose Provide a **coherent, explicit, and debuggable completion architecture** for this Emacs configuration.

This layer centralizes completion behavior to avoid fragmentation across modes and subsystems, and to keep completion flow **fully inspectable** via standard Emacs mechanisms (CAPFs, completion categories, minibuffer UI).

(b) What this layer does Completion modules are responsible for defining and wiring together the completion stack used by this configuration.

Specifically, this layer:

- Defines the minibuffer completion UI and navigation/search commands
- Defines in-buffer completion UI and its default behavior
- Establishes matching policy and completion category overrides
- Orchestrates ‘completion-at-point-functions‘ (CAPFs) per major mode
- Provides targeted, buffer-local tweaks for Org SRC edit buffers
- Provides optional session hygiene around project switches (Eglot lifecycle)

Typical responsibilities include:

- Minibuffer completion (selection UI, narrowing, annotations)
- In-buffer completion (popup UI, auto completion defaults)
- Matching styles (orderless vs basic, file partial completion)
- CAPF ordering and safe fallbacks across editing contexts

(c) What this layer does **not** do Completion modules intentionally do **not**:

- Define global keybinding schemes or leader layouts
- Implement UI themes or palette policy (owned by `ui`)
- Introduce language- or project-specific semantics
- Own LSP configuration or server selection policy (owned by `dev`)
- Extend Org workflows beyond completion mechanics (owned by `orgx`)
- Provide VCS features (owned by `vcs`) or general utilities (owned by `utils`)

(d) Design constraints

- Completion modules may depend on:
  - `core`
  - `ui`
- Completion modules must not depend on:
  - `orgx`
  - `dev`
  - `vcs`
  - `utils`
- Completion behavior must be:
  - deterministic
  - composable
  - debuggable using standard Emacs facilities (e.g. inspecting CAPFs, ‘completion-category-overrides‘, and the active minibuffer frontend)

(e) Design principles

- There is a **single completion flow** per context:
  - minibuffer completion (Vertico + Marginalia + Consult)
  - in-buffer completion (CAPF + Corfu)
- CAPF order and fallback rules are explicit and reviewable
- Frontend (UI) and backend (sources/CAPFs) are clearly separated
- Category-based policy is preferred over mode-specific special cases
- Org SRC edit buffers are treated as a controlled exception: buffer-local overrides apply, without changing global policy

(f) Module map

File	Responsibility
completion/completion-core.el	Global completion policy: ‘completion-styles‘ an
completion/completion-vertico.el	Minibuffer completion UI (Vertico) and annotat
completion/completion-consult.el	Search/navigation integration, including xref dis
completion/completion-embark.el	Contextual actions and prefix help integration (
completion/completion-corfu.el	In-buffer completion UI (Corfu) + candidate vis
completion/completion-icons.el	Nerd icons integration for minibuffer annotation
completion/completion-capf.el	Mode-specific CAPF presets and shared categor
completion/completion-capf-org-src.el	CAPF switching when entering Org SRC edit b
completion/completion-capf-org-src-lang.el	Language-specific CAPF presets inside Org SRC
completion/completion-corfu-org-src.el	Buffer-local Corfu behavior tuned for Org SRC
completion/completion-orderless-org-src.el	Buffer-local category overrides for permissive Or
completion/completion-lsp.el	Optional cleanup helper: shut down obsolete Eg

(g) Notes

- Matching policy is defined centrally via ‘completion-styles‘ and ‘completion-category-overrides‘; Org SRC overrides are buffer-local only.
- CAPF presets are applied via major-mode hooks; exceptions are limited to Org SRC edit buffers where ‘org-src-mode-hook‘ controls local overrides.
- LSP server lifecycle policy is not implemented here; only safe cleanup is provided, and only when Eglot is present and enabled.

2. completion/completion-core.el

```
;;; completion/completion-core.el --- Completion core settings -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Core completion style and category configuration.
;;
;; This module:
;; - defines global completion styles
;; - sets category overrides for common symbols
;; - serves as the foundation for Corfu/CAPE
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf orderless
  :straight t
  :custom
  ((completion-styles . '(orderless basic))
   (completion-category-overrides
    . '((file    (styles . (partial-completion)))
        (symbol  (styles . (orderless)))
```

```
        (command (styles . (orderless)))))))

(provide 'completion-core)
```

3. completion/completion-vertico.el

```
;;; completion/completion-vertico.el --- Vertico minibuffer UI -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Vertico-based minibuffer completion UI.
;;
;; This module:
;; - enables Vertico, Marginalia, and vertico-posframe
;;   after window geometry becomes stable
;; - guarantees posframe usage from the very first minibuffer
;;
;;; Code:

(eval-when-compile
  (require 'leaf))

;; ------------------------------------------------------------------------
;; Initialization state
;; ------------------------------------------------------------------------

(defvar completion-vertico--initialized nil
  "Non-nil once Vertico, Marginalia, and vertico-posframe are initialized.")

;; ------------------------------------------------------------------------
;; Initialization
;; ------------------------------------------------------------------------

(defun completion-vertico--init ()
  "Initialize Vertico, Marginalia, and vertico-posframe once after window setup."
  (unless completion-vertico--initialized
    (setq completion-vertico--initialized t)

    (require 'vertico)
    (vertico-mode 1)

    (require 'marginalia)
    (marginalia-mode 1)

    (when (display-graphic-p)
      (require 'vertico-posframe)
      (vertico-posframe-mode 1))))
```

```
(add-hook 'window-setup-hook #'completion-vertico--init)

;; ----------------------------------------------------------------------------
;; Packages
;; ----------------------------------------------------------------------------

(leaf vertico
  :straight t
  :custom
  ((vertico-count . 15)))

(leaf vertico-posframe
  :straight t
  :if (display-graphic-p)
  :after vertico
  :custom
  ((vertico-posframe-border-width . 2)))

(leaf marginalia
  :straight t)

(provide 'completion-vertico)
;;; completion/completion-vertico.el ends here
```

4. completion/completion-consult.el

```
;;; completion/completion-consult.el --- Consult search and navigation -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Consult integration for xref-based navigation.
;;
;; This module:
;; - routes xref UI through Consult
;; - avoids redefining navigation commands
;; - remains backend-agnostic
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf consult
  :straight t
  :custom
  ((xref-show-xrefs-function        . #'consult-xref)
   (xref-show-definitions-function . #'consult-xref)))

(provide 'completion-consult)
```

5. completion/completion-embark.el

```
;;; completion/completion-embark.el --- Embark actions -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Embark integration for contextual actions and previews.
;;
;; This module:
;; - enables Embark as the primary action framework
;; - integrates Embark collections with Consult previews
;; - introduces no global keybindings or policy
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf embark
  :straight t
  :custom ((prefix-help-command . #'embark-prefix-help-command)))

(leaf embark-consult
  :straight t
  :after (embark consult)
  :hook (embark-collect-mode . consult-preview-at-point-mode))

(provide 'completion-embark)
```

6. completion/completion-corfu.el

```
;;; completion/completion-corfu.el --- Corfu popup completion module -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Corfu-based in-buffer completion with sensible defaults.
;;
;; This module:
;; - enables global Corfu completion
;; - integrates icon margins and CAPE backends
;; - provides a lightweight, non-intrusive UI
;;
;;; Code:
```

```
(eval-when-compile (require 'leaf))

;; ------------------------------------------------------------------------
;; Lazy initialization
;; ------------------------------------------------------------------------

(defvar completion-corfu--initialized nil
  "Non-nil once Corfu, kind-icon, and CAPE have been initialized.")

(defun completion-corfu--init ()
  "Initialize Corfu completion UI and related backends on first use."
  (unless completion-corfu--initialized
    (setq completion-corfu--initialized t)

    ;; corfu
    (require 'corfu)
    (setq tab-always-indent 'complete)
    (global-corfu-mode 1)

    ;; kind-icon
    (require 'kind-icon)
    (add-to-list 'corfu-margin-formatters
                 #'kind-icon-margin-formatter)

    ;; cape
    (require 'cape)
    (add-to-list 'completion-at-point-functions #'cape-elisp-symbol)
    (add-to-list 'completion-at-point-functions #'cape-file)
    (add-to-list 'completion-at-point-functions #'cape-dabbrev)

    ;; One-shot initialization
    (remove-hook 'completion-at-point-functions #'completion-corfu--init)))

(add-hook 'completion-at-point-functions #'completion-corfu--init)

;; ------------------------------------------------------------------------
;; Packages
;; ------------------------------------------------------------------------

(leaf corfu
  :straight t
  :custom
  ((corfu-auto  . t)
   (corfu-cycle . t)))

(leaf kind-icon
  :straight t
  :after corfu
  :require t
  :custom ((kind-icon-default-face . 'corfu-default)))

(leaf cape
```

```
  :straight t
  :after corfu)

(provide 'completion-corfu)
;;; completion/completion-corfu.el ends here
```

7. completion/completion-icons.el

```
;;; completion/completion-icons.el --- Icons for completion UI -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Icon integration for completion and buffer lists.
;;
;; This module:
;; - decorates completion candidates with Nerd Icons
;; - integrates with Marginalia and Ibuffer
;; - avoids altering completion behavior itself
;;
;;; Code:

(eval-when-compile (require  'leaf))

(leaf nerd-icons-ibuffer
  :straight t
  :hook (ibuffer-mode-hook . nerd-icons-ibuffer-mode))

(leaf nerd-icons-completion
  :straight t
  :hook (marginalia-mode-hook
         . nerd-icons-completion-marginalia-setup)
  :config
  (nerd-icons-completion-mode))

(setq marginalia-align 'right)

(provide 'completion-icons)
```

8. completion/completion-capf.el

```
;;; completion/completion-capf.el --- Mode-specific CAPF configuration -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
```

```
;;
;; Commentary:
;; Central definition of completion-at-point-functions by major mode.
;;
;; This module:
;; - defines CAPF presets for code, text, Org, and REPL buffers
;; - installs mode hooks to activate presets
;; - configures completion-category-overrides for Corfu/CAPE
;;
;;; Code:

;;;; Utilities

(defun completion--set-capfs (capfs)
  "Set buffer-local CAPFs to CAPFS."
  (setq-local completion-at-point-functions capfs))

;;;; CAPF presets by mode

(defun completion--elisp-capfs ()
  "CAPFs for Emacs Lisp buffers."
  (completion--set-capfs
   '(cape-elisp-symbol
     cape-file
     cape-dabbrev)))

(defun completion--prog-capfs ()
  "CAPFs for programming modes (LSP-friendly)."
  (completion--set-capfs
   (append completion-at-point-functions
           '(cape-file
             cape-dabbrev))))

(defun completion--text-capfs ()
  "CAPFs for generic text buffers."
  (completion--set-capfs
   '(cape-dabbrev
     cape-file)))

(defun completion--org-capfs ()
  "CAPFs specialized for Org buffers."
  (completion--set-capfs
   '(cape-dabbrev
     cape-file
     cape-tex)))

(defun completion--shell-capfs ()
  "CAPFs for shell and eshell buffers."
  (completion--set-capfs
   '(cape-file
     cape-dabbrev)))

(defun completion--repl-capfs ()
```

```
  "CAPFs for REPL-like buffers (IELM, comint)."
  (completion--set-capfs
   (append completion-at-point-functions
           '(cape-dabbrev))))

;;;; Hooks

(add-hook 'emacs-lisp-mode-hook #'completion--elisp-capfs)
(add-hook 'lisp-interaction-mode-hook #'completion--elisp-capfs)

(add-hook 'prog-mode-hook #'completion--prog-capfs)
(add-hook 'text-mode-hook #'completion--text-capfs)

(add-hook 'org-mode-hook #'completion--org-capfs)

(add-hook 'sh-mode-hook #'completion--shell-capfs)
(add-hook 'eshell-mode-hook #'completion--shell-capfs)

(add-hook 'ielm-mode-hook #'completion--repl-capfs)
(add-hook 'comint-mode-hook #'completion--repl-capfs)

;;;; completion-category-overrides

(setq completion-category-overrides
      '((file (styles basic partial-completion))
        (symbol (styles orderless))
        (keyword (styles orderless))
        (tex (styles basic))
        (dabbrev (styles basic))))


(provide 'completion-capf)
;;; completion/completion-capf.el ends here
```

9. completion/completion-capf-org-src.el

```
;;; completion/completion-capf-org-src.el --- CAPF switching for Org SRC blocks -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Automatic CAPF selection inside Org Babel source edit buffers.
;;
;; This module:
;; - switches CAPFs based on detected source block major mode
;; - runs only in org-src-mode buffers
;; - delegates actual CAPF definitions to completion-capf
;;
;;; Code:
```

```
(defun completion--org-src-capfs ()
  "Apply appropriate CAPFs inside Org SRC edit buffers.

This function runs in `org-src-mode` buffers and selects CAPFs
based on the detected major mode of the source block."
  (cond
   ;; Emacs Lisp blocks
   ((derived-mode-p 'emacs-lisp-mode)
    (completion--elisp-capfs))

   ;; Programming language blocks
   ((derived-mode-p 'prog-mode)
    (completion--prog-capfs))

   ;; Fallback for text-like blocks
   (t
    (completion--text-capfs))))

(add-hook 'org-src-mode-hook #'completion--org-src-capfs)

(provide 'completion-capf-org-src)
;;; completion/completion-capf-org-src.el ends here
```

10. completion/completion-capf-org-src-lang.el

```
;;; completion/completion-capf-org-src-lang.el --- Language-
specific CAPFs for Org SRC -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Language-aware CAPF dispatch inside Org Babel SRC buffers.
;;
;; This module:
;; - applies shell, SQL, and Python-specific CAPFs
;; - runs after generic Org SRC CAPF setup
;; - preserves fallback behavior when no match exists
;;
;;; Code:

(defun completion--org-src-shell-capfs ()
  "CAPFs for shell-like Org SRC blocks."
  (completion--set-capfs
   '(cape-file
     cape-dabbrev)))

(defun completion--org-src-sql-capfs ()
  "CAPFs for SQL Org SRC blocks."
  (completion--set-capfs
```

```
   '(cape-dabbrev
     cape-file)))

(defun completion--org-src-python-capfs ()
  "CAPFs for Python Org SRC blocks."
  (completion--set-capfs
   (append completion-at-point-functions
           '(cape-file
             cape-dabbrev))))

(defun completion--org-src-lang-capfs ()
  "Dispatch language-specific CAPFs for Org SRC edit buffers."
  (cond
   ;; Shell blocks
   ((derived-mode-p 'sh-mode 'shell-mode 'eshell-mode)
    (completion--org-src-shell-capfs))

   ;; SQL blocks
   ((derived-mode-p 'sql-mode)
    (completion--org-src-sql-capfs))

   ;; Python blocks
   ((derived-mode-p 'python-mode)
    (completion--org-src-python-capfs))

   ;; Fallback: keep CAPFs set by completion-capf-org-src
   (t nil)))

(add-hook 'org-src-mode-hook #'completion--org-src-lang-capfs)

(provide 'completion-capf-org-src-lang)
;;; completion/completion-capf-org-src-lang.el ends here
```

11. completion/completion-corfu-org-src.el

```
;;; completion/completion-corfu-org-src.el --- Corfu tweaks for Org SRC -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Buffer-local Corfu behavior adjustments for Org SRC edit buffers.
;;
;; This module:
;; - enables aggressive auto completion in SRC buffers
;; - keeps settings strictly buffer-local
;; - never affects normal Org or code buffers
;;
;;; Code:
```

```
(defun completion--org-src-corfu-setup ()
  "Apply buffer-local Corfu settings for Org SRC edit buffers."
  (setq-local corfu-auto t)
  (setq-local corfu-auto-delay 0.1)
  (setq-local corfu-auto-prefix 1)
  (setq-local corfu-cycle nil))

(add-hook 'org-src-mode-hook #'completion--org-src-corfu-setup)

(provide 'completion-corfu-org-src)
;;; completion/completion-corfu-org-src.el ends here
```

12. completion/completion-orderless-org-src.el

```
;;; completion/completion-orderless-org-src.el --- Orderless tweaks for Org SRC -
*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Buffer-local Orderless matching rules for Org SRC edit buffers.
;;
;; This module:
;; - overrides completion-category-overrides locally
;; - relaxes matching for code-oriented categories
;; - never affects non-Org buffers
;;
;;; Code:

(defun completion--org-src-orderless-setup ()
  "Apply Org SRC-specific completion category overrides.

Use more permissive orderless matching for code-oriented categories
inside Org Babel source edit buffers."
  (setq-local completion-category-overrides
              '((symbol  (styles orderless))
                (keyword (styles orderless))
                (function (styles orderless))
                (variable (styles orderless))
                (file     (styles basic partial-completion))
                (dabbrev  (styles basic)))))

(add-hook 'org-src-mode-hook #'completion--org-src-orderless-setup)

(provide 'completion-orderless-org-src)
;;; completion/completion-orderless-org-src.el ends here
```

13. completion/completion-lsp.el

```
;;; completion/completion-lsp.el --- LSP lifecycle cleanup helpers -*- lexical-
binding: t; -*-
```

```
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: completion
;;
;; Commentary:
;; Defensive cleanup of obsolete Eglot servers on project switches.
;;
;; This module:
;; - tracks project root transitions
;; - shuts down stale LSP servers safely
;; - integrates with project.el hooks only
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf nil
  :straight nil
  :init
  (defcustom completion-lsp-enable-p t
    "Enable LSP lifecycle cleanup."
    :type 'boolean
    :group 'completion)

  (defvar completion-lsp--current-project-root nil)

  (defun completion-lsp--project-root ()
    (when-let* ((project (project-current nil))
                (roots (project-roots project)))
      (car roots)))

  (defun completion-lsp-on-project-switch ()
    "Shutdown obsolete eglot servers on project switch."
    (when completion-lsp-enable-p
      (let ((new-root (completion-lsp--project-root)))
        (when (and completion-lsp--current-project-root
                   new-root
                   (not (string-equal completion-lsp--current-project-root new-
root))
                   (featurep 'eglot))
          (dolist (server eglot--servers)
            (when (string-prefix-p completion-lsp--current-project-root
                                   (eglot--project-root (cdr server)))
              (ignore-errors
                (eglot-shutdown (cdr server))))))
        (setq completion-lsp--current-project-root new-root))))

  (add-hook 'find-file-hook #'completion-lsp-on-project-switch)
  (add-hook 'project-switch-project-hook #'completion-lsp-on-project-switch))
```

```
(provide 'completion-lsp)
;;; completion/completion-lsp.el ends here
```

2.2.5 ui/

1. Overview

   - ui modules are loaded **after `core` and before functional layers**
   - Each file:
     - provides exactly one feature
     - contains no non-visual side effects
   - UI bundle modules:
     - are never loaded implicitly
     - are activated explicitly through `core/switches.el`
   - Removing or disabling all ui modules must leave Emacs fully functional, albeit visually minimal

   (a) Purpose Define the **visual presentation and interaction layer** of this Emacs configuration.

   This layer governs *how Emacs looks and feels*—colors, spacing, typography, and visual feedback—while deliberately avoiding any influence on semantic behavior, data flow, or feature logic.

   UI modules exist to improve **clarity, comfort, and legibility**, never correctness.

   (b) What this layer does UI modules are responsible for presentation-only concerns that shape the day-to-day user experience without altering functional meaning.

   Specifically, this layer:

   - Configures visual appearance:
     - theme selection and palette coordination
     - face definitions and typography (fixed / variable pitch)
     - line spacing, margins, and window layout hints
   - Adjusts interaction-related presentation:
     - cursor style and visibility
     - modeline content, density, and signaling
     - subtle visual feedback for focus, state, and context
   - Applies UI customizations that are:
     - orthogonal to functionality
     - safe to toggle or replace independently
     - reversible at runtime

   (c) What this layer does **not** do UI modules intentionally do **not**:

   - Define global keybinding schemes or leader layouts (handled by `core/general.el`)
   - Select, configure, or orchestrate functional backends (completion, LSP, VCS, Org logic, etc.)
   - Introduce workflow logic or mode-specific behavior
   - Encode user-, device-, or host-specific policy (handled by `personal/` overlays)

   Those responsibilities belong to other layers.

   (d) Design constraints

   - UI modules may depend on:
     - `core`

- UI modules must not depend on:
  - `completion`
  - `orgx`
  - `dev`
  - `vcs`
  - `utils`
- All UI changes must be:
  - presentation-only (no semantic side effects)
  - reversible at runtime
  - safe in GUI, terminal, daemon, and batch contexts
  - tolerant of partial availability (fonts, icons, faces)

(e) Design principles

- Presentation and behavior are strictly separated
- UI modules must be safe to disable wholesale for debugging
- Defaults favor readability and low visual noise
- Visual signaling should be subtle, not distracting
- Optional UI bundles are treated as **overlays**, not foundations

(f) Optional UI bundles Theme and modeline bundles (e.g. Doom-style or Nano-style) are treated as **optional overlays**.

Selection and activation are controlled exclusively via `core/switches.el`, ensuring that:

- the base UI remains minimal and stable
- bundles can be compared or disabled safely
- UI experimentation never destabilizes core behavior

(g) Benefits This separation ensures that:

- visual experimentation does not affect correctness
- alternative UI styles can be swapped without refactoring logic
- debugging can be performed with a minimal, presentation-free setup
- long-running sessions remain visually predictable

(h) Module map

File	Responsibility
`ui/ui-font.el`	Font family, size, and typography defaults
`ui/ui-theme.el`	Theme selection and color palette coordination
`ui/ui-window.el`	Frame, window, margin, and spacing presentation rules
`ui/ui-utils.el`	Small visual helpers and UI-safe conveniences
`ui/ui-health-modeline.el`	Session health indicators and visual status signaling
`ui/ui-icons.el`	Icon and glyph integration (GUI-safe, optional)
`ui/ui-macos.el`	macOS-specific visual tuning (fonts, frames, rendering)
`ui/ui-doom-modeline.el`	Doom-style modeline bundle (optional overlay)
`ui/ui-nano-modeline.el`	Nano-style modeline bundle (optional overlay)

(i) Notes

- UI modules must never introduce functional dependencies.
- Any visual adjustment that affects behavior belongs elsewhere.
- If a UI change feels “clever” or “opinionated” , it is likely misplaced.

This layer should remain **boring, predictable, and comfortable**.

2. ui/ui-font.el

```
;;; ui/ui-font.el --- Font configuration -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2025
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Font configuration for graphical UI frames.
;;
;; This module:
;; - configures default, variable-pitch, and emoji fonts
;; - applies OS-aware fallbacks and daemon-safe hooks
;; - optionally enables programming ligatures
;;
;;; Code:

(eval-when-compile
  (require 'leaf))

;;;; Custom variables

(defgroup ui-font nil
  "Font configuration for UI."
  :group 'ui)

(defcustom ui-font-default nil
  "Default monospace font family.
If nil, an OS-dependent fallback is used."
  :type '(choice (const :tag "Auto" nil)
                 (string :tag "Font family"))
  :group 'ui-font)

(defcustom ui-font-variable-pitch nil
  "Variable-pitch font family.
If nil, the default monospace font is reused."
  :type '(choice (const :tag "Auto" nil)
                 (string :tag "Font family"))
  :group 'ui-font)

(defcustom ui-font-emoji nil
  "Emoji font family.
If nil, an OS-dependent fallback is used."
  :type '(choice (const :tag "Auto" nil)
                 (string :tag "Font family"))
  :group 'ui-font)

(defcustom ui-font-size 18
  "Default font size in points."
  :type 'integer
  :group 'ui-font)
```

```
;;;; Internal helpers

(defun ui-font--system-default ()
  "Return a default monospace font family depending on OS."
  (cond
   ((eq system-type 'darwin)     "Menlo")
   ((eq system-type 'gnu/linux)  "Monospace")
   ((eq system-type 'windows-nt) "Consolas")
   (t "Monospace")))

(defun ui-font--system-emoji ()
  "Return a default emoji font family depending on OS."
  (cond
   ((eq system-type 'darwin)     "Apple Color Emoji")
   ((eq system-type 'gnu/linux)  "Noto Color Emoji")
   ((eq system-type 'windows-nt) "Segoe UI Emoji")
   (t "Noto Color Emoji")))

;;;; Core setup

(defun ui-font-apply ()
  "Apply font settings to the current frame."
  (when (display-graphic-p)
    (set-face-attribute
     'default nil
     :family (or ui-font-default (ui-font--system-default))
     :height (* 10 ui-font-size))
    (set-face-attribute
     'variable-pitch nil
     :family (or ui-font-variable-pitch
                 ui-font-default
                 (ui-font--system-default)))
    (set-fontset-font
     t 'emoji
     (font-spec :family
                (or ui-font-emoji (ui-font--system-emoji))))
    (ui-font-describe)))

;;;###autoload
(defun ui-font-show-current ()
  "Echo the current default font family and point size."
  (interactive)
  (let ((family (face-attribute 'default :family))
        (height (face-attribute 'default :height)))
    (message "Current font: %s, %.1f pt"
             family (/ height 10.0))))

;;;###autoload
(defun ui-font-describe ()
  "Display font information applied by `ui-font-apply`."
  (interactive)
  (unless (display-graphic-p)
    (user-error "Fonts are only meaningful in graphical frames"))
```

```
  (let* ((default-family (face-attribute 'default :family))
         (default-height (face-attribute 'default :height))
         (variable-family (face-attribute 'variable-pitch :family))
         (emoji-font (font-get
                      (font-spec :script 'emoji)
                      :family)))
    (message
     (concat
      "Fonts applied:\n"
      "  Default        : %s (%.1f pt)\n"
      "  Variable-pitch : %s\n"
      "  Emoji          : %s")
     default-family
     (/ default-height 10.0)
     variable-family
     (or emoji-font "unknown"))))

;;;; Startup handling

(if (daemonp)
    (add-hook 'after-make-frame-functions
              (lambda (frame)
                (with-selected-frame frame
                  (ui-font-apply))))
  (add-hook 'after-init-hook #'ui-font-apply))

;;;; Optional ligatures

(leaf ligature
  :straight (ligature :type git :host github :repo "mickeynp/ligature.el")
  :when (display-graphic-p)
  :hook (prog-mode-hook . global-ligature-mode)
  :config
  (ligature-set-ligatures
   'prog-mode
   '("->" "=>" "::" "===" "!=" "&&" "||")))

(provide 'ui-font)
;;; ui-font.el ends here
```

3. ui/ui-nano-palette.el

```
;;; ui/ui-nano-palette.el --- Nano-style color palette -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Single source of truth for Nano-style UI colors.
;;
```

```
;; This module:
;; - defines all palette entries as defcustom variables
;; - applies faces through a single normalization function
;; - avoids duplicated literal colors across the UI layer
;;
;;; Code:

(defgroup my:nano nil
  "Minimal nano-style light palette."
  :group 'faces)

(defcustom nano-color-background "#fafafa"
  "UI background for buffers/panels."
  :type 'string :group 'my:nano)

(defcustom nano-color-foreground "#374151"
  "Default body text color."
  :type 'string :group 'my:nano)

(defcustom nano-color-salient "#2563eb"
  "Accent for links/keywords/standout choices."
  :type 'string :group 'my:nano)

(defcustom nano-color-popout "#6b7280"
  "Neutral notice / muted highlight."
  :type 'string :group 'my:nano)

(defcustom nano-color-critical "#dc2626"
  "Critical error / danger color."
  :type 'string :group 'my:nano)

(defcustom nano-color-strong "#111827"
  "Strong emphasis (headings, key mode-line parts)."
  :type 'string :group 'my:nano)

(defcustom nano-color-faded "#9ca3af"
  "De-emphasized info (comments/secondary/disabled)."
  :type 'string :group 'my:nano)

(defcustom nano-color-subtle "#e5e7eb"
  "Subtle backgrounds (mode/header lines, gentle selections)."
  :type 'string :group 'my:nano)

(defun my/nano-apply-faces ()
  "Apply faces based on the nano-style palette defined above."
  (set-face-attribute 'default nil
                      :background nano-color-background
                      :foreground nano-color-foreground)
  (set-face-attribute 'bold nil :foreground nano-color-strong :weight 'bold)
  (set-face-attribute 'italic nil :slant 'italic)

  (set-face-attribute 'font-lock-comment-face nil :foreground nano-color-faded)
  (set-face-attribute 'font-lock-keyword-face nil
```

```
                      :foreground nano-color-salient :weight 'semi-bold)
  (set-face-attribute 'font-lock-string-face nil :foreground nano-color-popout)
  (set-face-attribute 'font-lock-warning-face nil
                      :foreground nano-color-popout :weight 'bold)

  (set-face-attribute 'link nil :foreground nano-color-salient :underline t)
  (set-face-attribute 'button nil :foreground nano-color-salient :underline t)

  (set-face-attribute 'error nil
                      :foreground nano-color-critical :weight 'bold)
  (set-face-attribute 'warning nil
                      :foreground nano-color-popout :weight 'bold)
  (set-face-attribute 'success nil :foreground "#10b981")

  (set-face-attribute 'region nil :background nano-color-subtle)

  (let ((ml-bg nano-color-subtle)
        (ml-fg nano-color-strong))
    (set-face-attribute 'mode-line nil
                        :background ml-bg :foreground ml-fg
                        :box `(:line-width 1 :color ,ml-bg))
    (set-face-attribute 'mode-line-inactive nil
                        :background nano-color-background
                        :foreground nano-color-faded
                        :box `(:line-width 1 :color ,nano-color-background)))

  (set-face-attribute 'minibuffer-prompt nil
                      :foreground nano-color-salient :weight 'semi-bold))

(defun my/nano-set-palette-and-apply (&rest plist)
  "Override palette entries via PLIST and apply faces."
  (when plist
    (let ((map '((:background . nano-color-background)
                 (:foreground . nano-color-foreground)
                 (:salient    . nano-color-salient)
                 (:popout     . nano-color-popout)
                 (:critical   . nano-color-critical)
                 (:strong     . nano-color-strong)
                 (:faded      . nano-color-faded)
                 (:subtle     . nano-color-subtle))))
      (while plist
        (let* ((k (pop plist))
               (v (pop plist))
               (sym (cdr (assq k map))))
          (when sym (set sym v))))))
  (my/nano-apply-faces))

(provide 'ui-nano-palette)
;;; ui/ui-nano-palette.el ends here
```

4. ui/ui-theme.el

```
;;; ui/ui-theme.el --- Theme configuration -*- lexical-binding: t; -*-
;;
```

```
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Theme orchestration and face normalization.
;;
;; This module:
;; - applies Nano-based themes and palettes
;; - re-normalizes faces after theme changes
;; - adjusts UI faces for dired-subtree and padding
;;
;;; Code:

(eval-when-compile (require 'leaf))

(require 'ui-nano-palette)

(setq-default line-spacing 0.24)

(defvar ui-theme--initialized nil
  "Non-nil once the Nano theme and faces have been initialized.")

(defun ui-theme--apply ()
  "Apply Nano theme and related face customizations once."
  (unless ui-theme--initialized
    (setq ui-theme--initialized t)

    ;; Reapply faces after theme change
    (with-eval-after-load 'cus-theme
      (if (boundp 'enable-theme-functions)
          (add-hook 'enable-theme-functions #'my/nano--reapply-after-theme)
        (advice-add 'enable-theme :after #'my/nano--reapply-after-theme)))

    ;; Dired subtree faces
    (with-eval-after-load 'dired-subtree
      (set-face-attribute 'dired-subtree-depth-1-face nil :background "#e6f4ea")
      (set-face-attribute 'dired-subtree-depth-2-face nil :background "#dff0e3")
      (set-face-attribute 'dired-subtree-depth-3-face nil :background "#d8ecdc")
      (set-face-attribute 'dired-subtree-depth-4-face nil :background "#d1e8d5"))

    ;; Nano theme stack
    (require 'nano-layout)
    (require 'nano-faces)
    (nano-faces)

    (set-face-attribute 'nano-face-strong nil
                        :foreground (face-foreground 'nano-face-default)
                        :weight 'bold)

    (load-theme 'nano t)
```

```
    (my/nano-apply-faces)

    (set-face-attribute 'bold nil :weight 'bold)
    (set-face-attribute 'italic nil :slant 'italic)

    (with-eval-after-load 'elec-pair
      (custom-set-faces
       '(electric-pair-overlay-face ((t (:background "#e5e7eb"))))
       '(show-paren-match ((t (:background "#e5e7eb"
                                           :foreground "#111827"
                                           :weight bold))))
       '(show-paren-mismatch ((t (:background "#dc2626"
                                              :foreground "white"
                                              :weight bold))))))

    ;; Padding
    (when (display-graphic-p)
      (spacious-padding-mode 1)
      (my/nano-apply-faces))))

;; Nano packages (definitions only; no eager work)
(leaf nano-emacs
  :straight (nano-emacs :type git :host github :repo "rougier/nano-emacs"))

(leaf nano-theme
  :straight (nano-theme :type git :host github :repo "rougier/nano-theme"))

(leaf spacious-padding
  :straight t
  :if (display-graphic-p)
  :custom
  ((spacious-padding-widths . '((left . 15) (right . 15)))
   (spacious-padding-subtle-mode-line . t)))

;; Apply theme after first frame is ready
(add-hook 'emacs-startup-hook #'ui-theme--apply)

(provide 'ui-theme)
;;; ui/ui-theme.el ends here
```

5. ui/ui-doom-modeline.el

```
;;; ui/ui-doom-modeline.el --- Doom modeline UI bundle -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Doom-modeline based mode-line UI bundle.
;;
```

```
;; This module:
;; - configures and enables doom-modeline
;; - integrates Nerd Icons for the mode-line
;; - exposes an explicit interactive enable command
;;
;;; Code:

(eval-when-compile (require 'leaf))
(declare-function doom-modeline-mode "doom-modeline")

(leaf nerd-icons :straight t)

(leaf doom-modeline
  :straight t
  :custom ((doom-modeline-height . 28)
           (doom-modeline-buffer-file-name-style
            . 'truncate-with-project)
           (doom-modeline-minor-modes . nil)
           (doom-modeline-enable-word-count . t))
  :config
  (doom-modeline-mode 1))

;;;###autoload
(defun my/ui-enable-doom ()
  "Enable Doom UI modeline bundle."
  (interactive)
  (doom-modeline-mode 1)
  (message "[ui] Doom modeline enabled."))

(provide 'ui-doom-modeline)
;;; ui/ui-doom-modeline.el ends here
```

6. ui/ui-nano-modeline.el

```
;;; ui/ui-nano-modeline.el --- Nano modeline UI bundle -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Nano-modeline based lightweight mode-line bundle.
;;
;; This module:
;; - initializes nano-modeline lazily and safely
;; - activates mode-specific modeline hooks
;; - exposes an explicit interactive enable command
;;
;;; Code:

(eval-when-compile (require 'leaf))
```

```
(defvar ui--nano-modeline-initialized nil)

(defun my/ui--nano-available-p ()
  (require 'nano-modeline nil 'noerror))

(defun my/ui--nano-setup ()
  (when (and (not ui--nano-modeline-initialized)
             (my/ui--nano-available-p))
    (setopt nano-modeline-padding '(0.20 . 0.25))
    (add-hook 'prog-mode-hook #'nano-modeline-prog-mode)
    (add-hook 'text-mode-hook #'nano-modeline-text-mode)
    (setq ui--nano-modeline-initialized t)
    (message "[ui] nano-modeline initialized.")))

(leaf nano-modeline
  :straight (nano-modeline :type git :host github :repo "rougier/nano-
modeline")
  :after nano-emacs
  :require nil
  :init
  (add-hook 'after-init-hook #'my/ui--nano-setup))

;;;###autoload
(defun my/ui-enable-nano ()
  "Enable Nano UI modeline bundle."
  (interactive)
  (my/ui--nano-setup))

(provide 'ui-nano-modeline)
;;; ui/ui-nano-modeline.el ends here
```

7. ui/ui-health-modeline.el

```
;;; ui/ui-health-modeline.el --- Session health indicators -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Mode-line indicators for Emacs session health.
;;
;; This module:
;; - displays buffer, process, and LSP counts
;; - updates indicators reactively via lightweight hooks
;; - introduces no control or policy decisions
;;
;;; Code:

(eval-when-compile
```

```
  (require 'leaf)
  (require 'subr-x))

(defgroup ui-health-modeline nil
  "Mode-line indicators for session health."
  :group 'mode-line)

(defcustom ui-health-show-buffers-p t
  "Show buffer count in the mode-line."
  :type 'boolean
  :group 'ui-health-modeline)

(defcustom ui-health-show-processes-p t
  "Show process count in the mode-line."
  :type 'boolean
  :group 'ui-health-modeline)

(defcustom ui-health-show-eglot-p t
  "Show Eglot workspace count in the mode-line when available."
  :type 'boolean
  :group 'ui-health-modeline)

(defvar ui-health--mode-line-string ""
  "Mode-line string representing current session health.")

(defun ui-health--buffers-count ()
  "Return the number of live buffers."
  (length (buffer-list)))

(defun ui-health--processes-count ()
  "Return the number of live processes."
  (length (process-list)))

(defun ui-health--eglot-count ()
  "Return the number of active Eglot workspaces, or nil."
  (when (and ui-health-show-eglot-p
             (featurep 'eglot)
             (boundp 'eglot--managed-buffers))
    (length eglot--managed-buffers)))

(defun ui-health--update ()
  "Update `ui-health--mode-line-string`."
  (setq ui-health--mode-line-string
        (concat
         (when ui-health-show-buffers-p
           (format " B:%d" (ui-health--buffers-count)))
         (when ui-health-show-processes-p
           (format " P:%d" (ui-health--processes-count)))
         (when-let* ((n (ui-health--eglot-count)))
           (format " LSP:%d" n))))
  (force-mode-line-update))

(add-hook 'buffer-list-update-hook #'ui-health--update)
```

```
(add-hook 'process-status-change-hook #'ui-health--update)

(with-eval-after-load 'eglot
  (add-hook 'eglot-managed-mode-hook #'ui-health--update)
  (add-hook 'eglot-server-stopped-hook #'ui-health--update))

(unless (member 'ui-health--mode-line-string global-mode-string)
  (setq global-mode-string
        (append global-mode-string '(ui-health--mode-line-string))))

(ui-health--update)

(provide 'ui-health-modeline)
;;; ui/ui-health-modeline.el ends here
```

8. ui/ui-window.el

```
;;; ui/ui-window.el --- Window management helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Window and layout management helpers.
;;
;; This module:
;; - enables zoom, winner, and desktop persistence
;; - provides interactive window layout save/restore
;; - avoids enforcing rigid window policies
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf zoom
  :straight t
  :hook (after-init-hook . zoom-mode)
  :custom
  ((zoom-size . '(0.62 . 0.62))
   (zoom-ignored-major-modes . '(dired-mode treemacs-mode))
   (zoom-ignored-buffer-names . '("*Messages*" "*Help*"))))

(leaf desktop
  :straight nil
  :config
  (let ((dir (concat no-littering-var-directory "desktop/")))
    (setq desktop-dirname dir
          desktop-path (list dir)
          desktop-base-file-name "desktop"
          desktop-base-lock-name "lock"
          desktop-restore-eager 10
```

```
          desktop-save t
          desktop-load-locked-desktop nil
          desktop-auto-save-timeout 300)
    (my/ensure-directory-exists dir)
    (desktop-save-mode 1)))

(leaf winner
  :straight nil
  :global-minor-mode t)

(defvar my:saved-window-config nil
  "Saved window configuration state.")

(defun my/save-window-layout ()
  "Save current window layout."
  (interactive)
  (setq my:saved-window-config (window-state-get nil t))
  (message "Window configuration saved."))

(defun my/restore-window-layout ()
  "Restore last saved window layout."
  (interactive)
  (if my:saved-window-config
      (window-state-put my:saved-window-config)
    (message "No saved window configuration found.")))

(provide 'ui-window)
;;; ui/ui-window.el ends here
```

9. ui/ui-utils.el

```
;;; ui/ui-utils.el --- UI utilities and Treemacs configuration -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Assorted UI-related utilities and panels.
;;
;; This module:
;; - configures mode-line helpers and system indicators
;; - integrates Treemacs and Dired icon helpers
;; - provides platform-specific clipboard support
;;
;;; Code:

(leaf minions :straight t
  :custom ((minions-mode-line-lighter . "☒"))
  :hook (after-init-hook . minions-mode))
```

```
(setq display-time-interval 30
      display-time-day-and-date t
      display-time-24hr-format t)
(display-time-mode 1)
(when (fboundp 'display-battery-mode) (display-battery-mode 1))

(leaf treemacs :straight t
  :if (display-graphic-p)
  :custom ((treemacs-filewatch-mode . t)
           (treemacs-follow-mode . t)
           (treemacs-indentation . 2)
           (treemacs-missing-project-action . 'remove)))

(leaf nerd-icons-dired :straight t
  :hook (dired-mode . nerd-icons-dired-mode))

(leaf pbcopy
  :if (memq window-system '(mac ns))
  :straight t
  :config (turn-on-pbcopy))

(provide 'ui-utils)
;;; ui/ui-utils.el ends here
```

10. ui/ui-imenu.el

```
;;; ui/ui-imenu.el --- Imenu / outline UI integration -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; UI-level integration for Imenu-based navigation.
;;
;; This module:
;; - installs and autoloads imenu-list as an optional UI component
;; - does NOT define commands or workflows itself
;; - provides a stable dependency surface for utils-layer helpers
;;
;; Design notes:
;; - utils may opportunistically (require 'imenu-list nil 'noerror)
;; - this module owns the decision to install the package
;; - loading is deferred until explicitly used
;;
;;; Code:

(eval-when-compile
  (require 'leaf))

;;;; Built-in ----------------------------------------------------------------
```

```
(require 'imenu)

;;;; imenu-list ----------------------------------------------------------------
;; Optional sidebar-style Imenu UI.
;; Loaded only on demand via commands.

(leaf imenu-list
  :straight t
  :commands
  (imenu-list-minor-mode
   imenu-list-display-dwim
   imenu-list-stop-timer)
  :custom
  ((imenu-list-position . 'left)
   (imenu-list-size . 36)
   (imenu-list-focus-after-activation . t)))

(provide 'ui-imenu)
;;; ui/ui-imenu.el ends here
```

11. ui/ui-visual-aids.el

```
;;; ui/ui-visual-aids.el --- Subtle visual helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Non-intrusive visual aids for editing and navigation.
;;
;; This module:
;; - highlights cursor movement and TODO keywords
;; - adds rainbow delimiters and indent guides
;; - avoids persistent or disruptive visual effects
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf pulsar
  :straight t
  :init
  (setq pulsar-delay 0.04
        pulsar-pulse t
        pulsar-face 'pulsar-generic)
  :config
  (pulsar-global-mode 1)
  (dolist (cmd '(recenter-top-bottom
                 other-window
                 next-buffer
```

```
                previous-buffer))
    (add-to-list 'pulsar-pulse-functions cmd)))

(leaf hl-todo
  :straight t
  :hook ((prog-mode-hook text-mode-hook) . hl-todo-mode)
  :init
  (setq hl-todo-keyword-faces
        '(("TODO" . "#d97706")
          ("FIXME" . "#dc2626")
          ("NOTE" . "#2563eb")
          ("HACK" . "#9333ea"))))

(leaf rainbow-delimiters
  :straight t
  :hook (prog-mode-hook . rainbow-delimiters-mode))

(leaf indent-bars
  :straight t
  :hook (prog-mode-hook . indent-bars-mode)
  :init
  (setq indent-bars-prefer-character t
        indent-bars-spacing-override 2))

(provide 'ui-visual-aids)
;;; ui/ui-visual-aids.el ends here
```

12. ui/ui-macos.el

```
;;; ui/ui-macos.el --- macOS-specific UI niceties -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: ui
;;
;; Commentary:
;; Small macOS-specific UI adjustments.
;;
;; This module:
;; - enables native title-bar integration on macOS
;; - avoids any effect on non-darwin systems
;;
;;; Code:

(eval-when-compile (require 'leaf))

(when (eq system-type 'darwin)
  (add-to-list 'default-frame-alist
               '(ns-transparent-titlebar . t))
  (add-to-list 'default-frame-alist
               '(use-title-bar . t)))
```

```
(provide 'ui-macos)
;;; ui/ui-macos.el ends here
```

2.2.6 orgx/

1. Overview

   - Loaded after `completion` and before `dev`
   - Each file:
     - provides exactly one Org-related feature
     - documents its scope and assumptions clearly
   - Disabling this layer must leave:
     - basic Org editing functional
     - Org files readable and editable without loss

   (a) Purpose Extend Org-mode **beyond upstream defaults** while preserving its core semantics, file format, and data model.

   This layer encapsulates all Org-specific customization, workflows, and assumptions, ensuring that Org behavior remains isolated from non-Org contexts and other subsystems.

   Orgx exists to make Org usage **explicit, auditable, and modular**, without turning Org into a bespoke or opaque system.

   (b) What this layer does Orgx modules are responsible for **Org-mode – scoped behavior only**.

   This layer:

   - Extends Org-mode in a structured, non-invasive manner
   - Defines Org-specific workflows, commands, and helper utilities
   - Integrates Org with other subsystems **only at explicit boundaries**
   - Centralizes agenda, capture, and navigation policy

   Typical responsibilities include:

   - Agenda file discovery, scoping rules, and caching
   - Capture templates, inbox flows, and refiling strategy
   - Org-specific navigation, folding, and structural helpers
   - Visual and UX enhancements limited strictly to Org buffers
   - Optional integration with roam-style or graph-based note systems

   (c) What this layer does **not** do Orgx modules intentionally do **not**:

   - Reimplement, fork, or override core Org internals
   - Affect behavior in non-Org buffers
   - Define global editing, completion, or UI policy
   - Introduce development, VCS, or infrastructure concerns
   - Encode user-, device-, or host-specific paths directly

   Those responsibilities belong to other layers or to `personal/` overlays.

   (d) Design constraints

   - Orgx modules may depend on:
     - `core`
     - `ui`
     - `completion`
   - Orgx modules must not depend on:

- `dev`
- `vcs`
- `utils`

- All Org-specific assumptions must be:
  - confined to Org buffers
  - discoverable through module boundaries
  - safe to disable as a group without data loss

(e) Design principles

- Org remains **plain Org** on disk (no proprietary formats, sidecar metadata, or irreversible transforms)
- Extensions are additive, reversible, and optional
- Agenda and capture behavior is explicit and reviewable
- Org-specific UX must degrade gracefully when extensions are disabled
- Configuration favors transparency over clever automation

(f) Benefits This separation ensures that:

- Org files remain portable and future-proof
- Non-Org workflows are unaffected by Org customization
- Debugging Org behavior is localized and predictable
- Org extensions can evolve without destabilizing the rest of Emacs

(g) Module map

File	Responsibility
`orgx/org-core.el`	Core Org defaults and shared Org policy
`orgx/org-agenda.el`	Agenda discovery, scoping, and views
`orgx/org-capture.el`	Capture templates and inbox workflows
`orgx/org-navigation.el`	Navigation, folding, and structural helpers
`orgx/org-roam.el`	Roam-style linking and graph-based notes
`orgx/org-export.el`	Export configuration and backends
`orgx/org-ui.el`	Org-specific visual and UX enhancements

(h) Notes

- Orgx is the **only** layer allowed to assume Org semantics.
- Any logic that could apply outside Org likely belongs elsewhere.
- If an Org change affects global behavior, it is misplaced.

This layer exists to make Org powerful **without making it fragile**.

2. orgx/orgx-core.el

```
;;; orgx/orgx-core.el --- Org Mode core configuration -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Core Org Mode configuration and directory layout.
;;
;; This module:
```

```
;; - defines canonical Org directory structure
;; - configures agenda, capture, refile, and TODO workflows
;; - avoids UI presentation or export concerns
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'cl-lib))

(defvar my:d:org (expand-file-name "org" my:d:var))
(defvar my:d:org-journal (expand-file-name "journal" my:d:org))
(defvar my:d:org-roam (expand-file-name "org-roam" my:d:org))
(defvar my:d:org-pictures (expand-file-name "pictures" my:d:org))
(defvar my:f:capture-blog-file (expand-file-name "blog.org" my:d:org))

(my/ensure-directory-exists my:d:org)
(my/ensure-directory-exists my:d:org-journal)
(my/ensure-directory-exists my:d:org-roam)
(my/ensure-directory-exists my:d:org-pictures)

(defun my/org-buffer-files ()
  "Return a list of *.org files currently visited in live buffers."
  (cl-loop for buf in (buffer-list)
           for file = (buffer-file-name buf)
           when (and file (string-match-p "\\.org\\'" file))
           collect file))

(leaf org
  :straight nil
  :custom
  ((org-directory . my:d:org)
   (org-default-notes-file . "notes.org")
   (org-log-done . 'time)
   (org-support-shift-select . t)
   (org-return-follows-link . t))
  :config
  ;; NOTE: my:d:org/notes/ is intentionally excluded from org-agenda-
files (prose-only Markdown domain).
  (setq org-agenda-files
        (seq-filter (lambda (file)
                      (and (string-match-p "\\.org$" file)
                           (not (string-match-p "archives" file))))
                    (directory-files-recursively org-directory "\\.org$")))
  (unless org-agenda-files
    (setq org-agenda-files (list (expand-file-name "inbox.org" org-
directory))))
  (setq org-todo-keywords
        '((sequence "TODO(t)" "SOMEDAY(s)" "WAITING(w)" "|" "DONE(d)" "CANCELED(c@)")))
  (setq org-refile-targets
        '((nil :maxlevel . 3)
          (my/org-buffer-files :maxlevel . 1)
          (org-agenda-files :maxlevel . 3)))
```

```
  (setq org-capture-templates
        `(("t" "Todo" entry (file+headline ,(expand-file-name "gtd.org" org-
directory) "Inbox")
           "* TODO %?\n %i\n %a")
          ("n" "Note" entry (file+headline ,(expand-file-name "notes.org" org-
directory) "Notes")
           "* %?\nEntered on %U\n %i\n %a")
          ("j" "Journal" entry (function org-journal-find-location)
           "* %(format-time-string org-journal-time-format)%^{Title}\n%i%?")
          ("m" "Meeting" entry (file ,(expand-file-name "meetings.org" org-
directory))
           "* MEETING with %? :meeting:\n  %U\n  %a"))))

(with-eval-after-load 'org
  (let* ((central (expand-file-name "archive.org" (or (bound-and-true-p org-
directory)
                                                      (expand-file-
name "~/org")))))
    (setopt org-archive-location (concat central "::"))))

(provide 'orgx-core)
;;; orgx/org-core.el ends here
```

3. orgx/orgx-visual.el

```
;;; orgx/orgx-visual.el --- Org visual enhancements -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Visual enhancements for Org buffers and agenda views.
;;
;; This module:
;; - enables org-modern for improved Org visuals
;; - customizes headings, lists, TODO states, and priorities
;; - adjusts Org Agenda visual presentation only
;;
;;; Code:

(eval-when-compile (require 'leaf))

(defvar orgx-visual--initialized nil
  "Non-nil once org-modern visual variables are initialized.")

(defun orgx-visual--apply ()
  "Apply org-modern visual configuration."
  (unless orgx-visual--initialized
    (setq orgx-visual--initialized t)

    (setq
```

```
      org-startup-indented t
      org-hide-leading-stars t
      org-auto-align-tags nil
      org-tags-column 0
      org-catch-invisible-edits 'show-and-error
      org-special-ctrl-a/e t
      org-insert-heading-respect-content t
      org-hide-emphasis-markers t
      org-pretty-entities t

      org-modern-todo-faces
      '(("TODO"     :background "#673AB7" :foreground "#f8f8f2")
        ("SOMEDAY"  :background "#6b7280" :foreground "#f8f8f2")
        ("WAITING"  :background "#6272a4" :foreground "#f8f8f2")
        ("DONE"     :background "#373844" :foreground "#b0b8d1")
        ("CANCELED" :background "#4b5563" :foreground "#e5e7eb"))

      org-modern-list '((?+ . "◦") (?- . "–") (?* . "• "))
      org-modern-checkbox '((?X . " ") (?- . " ") (?\s . " "))
      org-modern-priority '((?A . " ") (?B . " ") (?C . " "))
      org-modern-replace-stars "     "

      org-agenda-tags-column 0
      org-agenda-block-separator ?—
      org-agenda-time-grid
      '((daily today require-timed)
        (800 1000 1200 1400 1600 1800 2000)
        " ---------- " " ---------- ")
      org-agenda-current-time-string
      "⌧ now ──────────────────────────────────────────────────────────────────────────────
—"))

    (require 'org-modern)
    (org-modern-mode 1))

  (leaf org-modern :straight t)

  (add-hook 'org-mode-hook #'orgx-visual--apply)

  (provide 'orgx-visual)
  ;;; orgx/orgx-visual.el ends here
```

4. orgx/orgx-extensions.el

```
;;; orgx/orgx-extensions.el --- Org Mode extensions -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Optional but production-safe Org extensions.
```

```
;;
;; This module:
;; - enables org-journal and org-roam with defensive guards
;; - delays org-roam autosync to avoid IO races
;; - integrates org-download, TOC, and link helpers
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf org-journal
  :straight t
  :custom ((org-journal-dir . my:d:org-journal)))

(leaf org-roam
  :straight t
  :custom ((org-roam-directory . my:d:org-roam))
  :config
  (setq org-roam-db-location
        (expand-file-name "org-roam.db" my:d:org-roam))

  (defun orgx--org-roam-file-hash-guard (orig file)
    "Guard `org-roam-db--file-hash' against file read errors."
    (condition-case err
        (funcall orig file)
      (error
       (message "[org-roam] skip hash: %s (%s)"
                file (error-message-string err))
       nil)))

  (with-eval-after-load 'org-roam
    (when (fboundp 'org-roam-db--file-hash)
      (advice-add 'org-roam-db--file-hash
                  :around #'orgx--org-roam-file-hash-guard)))

  (run-with-idle-timer
   5 nil
   (lambda ()
     (when (fboundp 'org-roam-db-autosync-mode)
       (org-roam-db-autosync-mode 1)
       (message "[org-roam] autosync enabled (delayed)")))))

(leaf org-download
  :straight t
  :custom ((org-download-image-dir . my:d:org-pictures)))

(leaf toc-org
  :straight t
  :hook ((org-mode . toc-org-enable)
         (markdown-mode . toc-org-mode)))

(leaf org-cliplink :straight t)
```

```
(provide 'orgx-extensions)
;;; orgx/orgx-extensions.el ends here
```

5. orgx/orgx-fold.el

```
;;; orgx/orgx-fold.el --- Extra Org folding helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Convenience helpers for Org subtree folding.
;;
;; This module:
;; - wraps org-fold primitives with interactive helpers
;; - installs local keybindings in org-mode only
;; - avoids global keybinding pollution
;;
;;; Code:

(with-eval-after-load 'org
  (require 'org-fold)

  (defun my/org-fold-subtree ()
    "Fold current Org subtree."
    (interactive)
    (org-fold-subtree t))

  (defun my/org-unfold-subtree ()
    "Unfold current Org subtree."
    (interactive)
    (org-show-subtree))

  (defun my/org-toggle-fold ()
    "Toggle fold state of current Org subtree."
    (interactive)
    (save-excursion
      (org-back-to-heading t)
      (if (org-fold-folded-p (point))
          (org-show-subtree)
        (org-fold-subtree t))))

  ;; Local bindings only (no global pollution)
  (define-key org-mode-map (kbd "C-c f") #'my/org-fold-subtree)
  (define-key org-mode-map (kbd "C-c e") #'my/org-unfold-subtree)
  (define-key org-mode-map (kbd "C-c t") #'my/org-toggle-fold))

(provide 'orgx-fold)
;;; org/orgx-fold.el ends here
```

6. orgx/orgx-export.el

```
;;; orgx/orgx-export.el --- Org export configuration -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Export and publishing helpers for Org documents.
;;
;; This module:
;; - configures Hugo, Markdown, and preview exporters
;; - enables Org Babel languages for diagrams
;; - avoids altering core Org editing behavior
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf ox-hugo
  :straight t
  :after ox
  :custom ((org-hugo-front-matter-format . "toml")))

(leaf markdown-mode :straight t)
(leaf markdown-preview-mode :straight t)
(leaf edit-indirect :straight t)

(leaf ob-mermaid
  :straight t
  :after org
  :config
  (when (executable-find "mmdc")
    (setq ob-mermaid-cli-path (executable-find "mmdc")))
  (org-babel-do-load-languages
   'org-babel-load-languages
   (append org-babel-load-languages '((mermaid . t)))))

(leaf ob-dot
  :straight nil
  :after org
  :config
  (org-babel-do-load-languages
   'org-babel-load-languages
   (append org-babel-load-languages '((dot . t)))))

(provide 'orgx-export)
;;; orgx/orgx-export.el ends here
```

7. orgx/orgx-notes-markdown.el

```
;;; orgx/orgx-notes-markdown.el --- Markdown notes under Org -*- lexical-
```

```
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Prose-oriented Markdown notes as a sibling domain under Org.
;;
;; This module:
;; - manages a Markdown notes directory outside agenda scope
;; - provides note creation, navigation, and search helpers
;; - configures Markdown UX optimized for prose
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))

;;; Customization ------------------------------------------------------------

(defgroup my:notes nil
  "Prose-oriented Markdown notes under Org directory."
  :group 'convenience)

(defcustom my:notes-directory-name
  "notes"
  "Directory name for Markdown notes under `my:d:org`."
  :type 'string)

(defcustom my:notes-default-notebooks
  '("inbox" "work" "life" "ideas" "archive")
  "Default notebook subdirectories under the notes directory."
  :type '(repeat string))

;;; Derived paths ------------------------------------------------------------

(defvar my:d:notes
  (expand-file-name my:notes-directory-name my:d:org)
  "Root directory for Markdown notes.

This directory is a sibling of Org files, not part of agenda scope.")

;;; Internal helpers ---------------------------------------------------------

(defun my/notes--ensure-root ()
  "Ensure `my:d:notes` and default notebooks exist."
  (unless (file-directory-p my:d:notes)
    (make-directory my:d:notes t))
  (dolist (name my:notes-default-notebooks)
```

```
    (let ((dir (expand-file-name name my:d:notes)))
      (unless (file-directory-p dir)
        (make-directory dir t)))))

(defun my/notes--slugify (title)
  "Return a filesystem-safe slug derived from TITLE."
  (let* ((lower (downcase title))
         (repl  (replace-regexp-in-string "[^[:alnum:]]+" "-" lower)))
    (string-trim repl "-+" "-+")))

;;; Note creation & navigation ----------------------------------------------

(defun my/notes-new-note (notebook title)
  "Create a new Markdown note in NOTEBOOK with TITLE.

NOTEBOOK is a subdirectory under `my:d:notes`."
  (interactive
   (progn
     (my/notes--ensure-root)
     (let* ((choices (directory-files my:d:notes nil "^[^.]"))
            (notebook (completing-read "Notebook: " choices nil nil "inbox"))
            (title (read-string "Title: ")))
       (list notebook title))))
  (my/notes--ensure-root)
  (let* ((dir (expand-file-name notebook my:d:notes))
         (slug (my/notes--slugify title))
         (timestamp (format-time-string "%Y%m%d-%H%M%S"))
         (filename (format "%s-%s.md" timestamp slug))
         (path (expand-file-name filename dir)))
    (unless (file-directory-p dir)
      (make-directory dir t))
    (find-file path)
    (when (= (buffer-size) 0)
      (insert
       (format
        "---\n\
title: %s\n\
notebook: %s\n\
tags: []\n\
created: %s\n\
updated: %s\n\
---\n\n"
        title
        notebook
        (format-time-string "%Y-%m-%d")
        (format-time-string "%Y-%m-%d")))
      (save-buffer))))

(defun my/notes-open-root ()
  "Open the Markdown notes root under Org in Dired."
  (interactive)
  (my/notes--ensure-root)
  (dired my:d:notes))
```

```
;;; Markdown UX ------------------------------------------------------------

(leaf markdown-mode
  :straight t
  :mode (("\\.md\\'" . gfm-mode)
         ("README\\.md\\'" . gfm-mode))
  :hook
  ((markdown-mode . visual-line-mode)
   (markdown-mode . variable-pitch-mode)
   (markdown-mode . my/notes-markdown-visual-fill))
  :custom
  ((markdown-command . "pandoc")
   (markdown-fontify-code-blocks-natively . t))
  :config
  (defun my/markdown-toggle-live-preview ()
    "Toggle `markdown-live-preview-mode` in the current buffer."
    (interactive)
    (if (bound-and-true-p markdown-live-preview-mode)
        (markdown-live-preview-mode -1)
      (markdown-live-preview-mode 1)))
  (define-key markdown-mode-map (kbd "C-c C-p")
              #'my/markdown-toggle-live-preview))

(leaf visual-fill-column
  :straight nil
  :commands (visual-fill-column-mode)
  :init
  (defun my/notes-markdown-visual-fill ()
    "Configure `visual-fill-column` for prose Markdown buffers."
    (setq-local visual-fill-column-width 100)
    (setq-local visual-fill-column-center-text t)
    (visual-fill-column-mode 1)))

;;; Search & navigation ----------------------------------------------------

(leaf consult-notes
  :straight t
  :after consult
  :require t
  :init
  (defun my/notes-consult-ripgrep ()
    "Run `consult-ripgrep` scoped to Markdown notes under Org."
    (interactive)
    (my/notes--ensure-root)
    (let ((default-directory my:d:notes))
      (consult-ripgrep)))
  :config
  (setq consult-notes-file-dir-sources
        (list
         (list "Org Notes (MD)" ?m my:d:notes))))

;;; Image handling ---------------------------------------------------------
```

```
(with-eval-after-load 'org-download
  (defun my/notes-markdown-org-download-setup ()
    "Enable `org-download` for Markdown notes under Org."
    (setq-local org-download-link-format "![](%s)")
    (setq-local org-download-image-dir "./images")
    (org-download-enable))
  (add-hook 'markdown-mode-hook
            #'my/notes-markdown-org-download-setup))

(provide 'orgx-notes-markdown)
;;; orgx/orgx-notes-markdown.el ends here
```

8. orgx/orgx-auto-tangle.el

```
;;; orgx/orgx-auto-tangle.el --- Automatic tangling helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Automatic tangling of Org source blocks in README.org.
;;
;; This module:
;; - tangles only when saving README.org
;; - suppresses confirmation prompts *locally*
;; - scopes hooks strictly to Org buffers
;; - prevents recursive save / auto-revert loops
;;
;;; Code:

(require 'org)
(require 'ob-tangle)

(defconst my/orgx-auto-tangle-target-file "README.org"
  "File name eligible for automatic tangling.")

(defvar-local my/orgx-auto-tangle--in-progress nil
  "Non-nil while auto-tangle is running in this buffer.
Used to prevent recursive save / revert loops.")

(defun my/orgx-auto-tangle--eligible-p ()
  "Return non-nil if current buffer should trigger auto-tangle."
  (and (derived-mode-p 'org-mode)
       buffer-file-name
       (string=
        (file-name-nondirectory buffer-file-name)
        my/orgx-auto-tangle-target-file)))

(defun my/orgx-auto-tangle--maybe ()
```

```
  "Automatically tangle Org source blocks when saving README.org.

This function is intentionally conservative:

- No global variables are mutated.
- Confirmation suppression is buffer-local and ephemeral.
- Recursive save / auto-revert loops are explicitly prevented.
- Tangling is skipped if already in progress."
  (when (and (my/orgx-auto-tangle--eligible-p)
             (not my/orgx-auto-tangle--in-progress))
    (let ((my/orgx-auto-tangle--in-progress t)
          (org-confirm-babel-evaluate nil)
          ;; Prevent save hooks or file notifications from retriggering
          (inhibit-modification-hooks t))
      (condition-case err
          (org-babel-tangle)
        (error
         (message "[orgx-auto-tangle] error: %s"
                  (error-message-string err)))))))

(add-hook 'org-mode-hook
          (lambda ()
            (add-hook 'after-save-hook
                      #'my/orgx-auto-tangle--maybe
                      nil
                      'local)))

(provide 'orgx-auto-tangle)
;;; orgx/orgx-auto-tangle.el ends here
```

9. orgx/orgx-typography.el

```
;;; orgx/orgx-typography.el --- Org typography enhancements -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: org
;;
;; Commentary:
;; Typography and prose readability helpers for Org buffers.
;;
;; This module:
;; - complements orgx-visual without overlapping responsibilities
;; - improves table alignment and emphasis visibility
;; - focuses on long-form prose readability
;;
;;; Code:

(eval-when-compile
  (require 'leaf))
```

```
;; Optional prose setting:
;; Use variable-pitch for text while keeping code blocks monospaced.
;; Disabled by default to avoid surprising font changes.
;;
;; (leaf org
;;   :straight nil
;;   :hook (org-mode-hook . variable-pitch-mode))

;; Align tables and inline images for better visual flow
(leaf valign
  :straight t
  :hook (org-mode-hook . valign-mode))

;; Contextual reveal of emphasis, links, and markers
(leaf org-appear
  :straight t
  :hook (org-mode-hook . org-appear-mode)
  :init
  (setq org-appear-autoemphasis t
        org-appear-autolinks t
        org-appear-autosubmarkers t))

(provide 'orgx-typography)
;;; orgx/orgx-typography.el ends here
```

2.2.7 dev/

1. Overview

   - Loaded after `orgx`
   - Activation is controlled via:
     - `core/switches.el`
     - explicit user or maintainer intent
   - Each file:
     - provides exactly one dev-oriented feature
     - documents activation conditions and non-goals

   (a) Purpose Support **development, diagnostics, and build-oriented workflows** for this Emacs configuration.

   This layer exists primarily for maintainers and advanced users who develop, inspect, or evolve the configuration itself, rather than for routine day-to-day editing.

   Dev modules are strictly **auxiliary**: useful when needed, invisible when not.

   (b) What this layer does Dev modules provide tooling that helps **understand, debug, and extend** the running Emacs environment and its configuration.

   This layer is responsible for:

   - Tooling for configuration development and internal inspection
   - Support for build, compile, and debug workflows
   - Diagnostics and introspection of the live Emacs instance
   - Maintenance-oriented helpers for evolving the configuration safely

   Typical responsibilities include:

   - Language Server Protocol (LSP) client wiring and toggles

- Debugging, profiling, tracing, and performance analysis helpers
- Compilation, test execution, and task orchestration
- Developer-facing inspection commands and reports

(c) What this layer does **not** do Dev modules intentionally do **not**:

- Define global editing behavior or keybinding policy
- Establish startup-critical behavior or core invariants
- Introduce visual presentation or theming concerns
- Encode user-, device-, or host-specific preferences
- Affect Org workflows or semantic editing behavior

Those responsibilities belong to other layers.

(d) Design constraints

- Dev modules may depend on:
  - `core`
  - `ui`
  - `completion`
- Dev modules must not depend on:
  - `orgx`
  - `vcs`
  - `utils`
- All side effects must be:
  - explicit
  - opt-in via customization variables or switches
  - safe to disable as a group

(e) Design principles

- Dev tooling is **additive**, never required for normal operation
- Failure in this layer must degrade gracefully
- Disabling this layer must not affect:
  - startup success
  - baseline editing behavior
  - Org workflows or persistence
- Debugging aids should favor observability over automation

(f) Benefits This separation ensures that:

- Development aids never compromise configuration stability
- Power-user tooling can evolve rapidly without risk
- Production usage remains lightweight and predictable
- Debugging can be enabled surgically when needed

(g) Module map

File	Responsibility
`dev/dev-lsp-eglot.el`	Eglot-based LSP configuration and integration
`dev/dev-lsp-mode.el`	lsp-mode – based LSP configuration (alternative backend)
`dev/dev-dap.el`	Debug Adapter Protocol integration
`dev/dev-compile.el`	Compile, build, and task helpers
`dev/dev-profile.el`	Profiling and performance diagnostics
`dev/dev-inspect.el`	Runtime inspection and developer-facing debug utilities

(h) Notes

- Dev is the **only** layer allowed to assume a maintainer mindset.

- Any tool required for basic usage does not belong here.
- If disabling this layer breaks normal editing, it is misplaced.

This layer exists to make the configuration **easier to evolve**, not harder to use.

2. dev/dev-lsp-eglot.el

```
;;; dev/dev-lsp-eglot.el --- Eglot LSP setup -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Baseline Eglot configuration with safe auto-enable.
;;
;; This module:
;; - detects whether an LSP server is applicable
;; - enables Eglot lazily in programming buffers
;; - exposes an explicit interactive enable command
;;
;;; Code:

(eval-when-compile (require 'leaf))

(defun my/eglot-guessable-p ()
  "Return non-nil if current buffer seems to have an LSP server we can start."
  (cond
   ((fboundp 'eglot--guess-contact)
    (ignore-errors (eglot--guess-contact)))
   ((fboundp 'eglot-guess-contact)
    (ignore-errors (eglot-guess-contact)))
   (t nil)))

(leaf eglot
  :straight t
  :commands (eglot eglot-ensure)
  :custom ((eglot-autoreconnect . t))
  :hook ((prog-mode . (lambda ()
                        (when (my/eglot-guessable-p)
                          (eglot-ensure))))))

;;;###autoload
(defun my/lsp-enable-eglot ()
  "Enable Eglot-based LSP setup."
  (interactive)
  (add-hook 'prog-mode-hook
            (lambda ()
              (when (my/eglot-guessable-p)
                (eglot-ensure))))
  (message "[lsp] Eglot enabled."))

(provide 'dev-lsp-eglot)
```

```
;;; dev-lsp-eglot.el ends here
```

3. dev/dev-lsp-mode.el

```
;;; dev/dev-lsp-mode.el --- lsp-mode setup -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; lsp-mode baseline with optional lsp-ui integration.
;;
;; This module:
;; - enables lsp-mode with deferred startup
;; - delegates completion to Corfu
;; - provides an explicit interactive enable command
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf lsp-mode
  :straight t
  :commands (lsp lsp-deferred)
  :custom ((lsp-keymap-prefix . "C-c l")
           (lsp-enable-file-watchers . t)
           (lsp-file-watch-threshold . 5000)
           (lsp-response-timeout . 5)
           (lsp-diagnostics-provider . :auto)
           (lsp-completion-provider . :none))
  :hook ((prog-mode . lsp-deferred)))

(leaf lsp-ui
  :straight t
  :after lsp-mode
  :custom ((lsp-ui-doc-enable . t)
           (lsp-ui-doc-delay . 0.2)
           (lsp-ui-sideline-enable . t)))

;;;###autoload
(defun my/lsp-enable-lspmode ()
  "Enable lsp-mode-based LSP setup."
  (interactive)
  (add-hook 'prog-mode-hook #'lsp-deferred)
  (message "[lsp] lsp-mode enabled."))

(provide 'dev-lsp-mode)
;;; dev-lsp-mode.el ends here
```

4. dev/dev-ai.el

```
;;; dev/dev-ai.el --- AI-assisted development helpers -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; AI-assisted coding via Aider / Aidermacs.
;;
;; This module:
;; - centralizes Aider runtime files and history
;; - configures environment variables consistently
;; - supports OpenRouter and OpenAI backends
;; - prevents duplication in `aidermacs-extra-args`
;; - selects vterm backend safely and defensively
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))

;; ------------------------------------------------------------------------
;; Base directory
;; ------------------------------------------------------------------------

(defvar my:d:aider
  (expand-file-name "aideremacs/"
                    (or (bound-and-true-p my:d:var)
                        (expand-file-name "~/.var/")))
  "Base directory to store aider runtime files.")

(my/ensure-directory-exists my:d:aider)

;; ------------------------------------------------------------------------
;; Environment variables (officially supported by aider)
;; ------------------------------------------------------------------------

(setenv "AIDER_INPUT_HISTORY_FILE"
        (expand-file-name "input.history" my:d:aider))
(setenv "AIDER_CHAT_HISTORY_FILE"
        (expand-file-name "chat.history.md" my:d:aider))
(setenv "AIDER_LLM_HISTORY_FILE"
        (expand-file-name "llm.history" my:d:aider))
(setenv "AIDER_ANALYTICS_LOG"
        (expand-file-name "analytics.log" my:d:aider))

;; ------------------------------------------------------------------------
;; Helper: deduplicated arg append
;; ------------------------------------------------------------------------
```

```
(defun my/aidermacs--append-unique-args (args)
  "Append ARGS to `aidermacs-extra-args`, avoiding duplicates.

ARGS must be a list of strings. Order is preserved:
existing entries are kept, and only missing elements are appended."
  (setq aidermacs-extra-args
        (append aidermacs-extra-args
                (seq-remove
                 (lambda (arg)
                   (member arg aidermacs-extra-args))
                 args))))

;; ---------------------------------------------------------------------------
;; Aidermacs integration (args & backend selection)
;; ---------------------------------------------------------------------------

(with-eval-after-load 'aidermacs
  ;; Defensive declaration for older versions
  (defvar aidermacs-extra-args nil
    "Extra CLI arguments passed to the `aider` command.")

  ;; Optional per-user .env support
  (let ((env-file (expand-file-name ".env" my:d:aider)))
    (when (file-exists-p env-file)
      (my/aidermacs--append-unique-args
       (list "--env-file" env-file))))

  ;; Force history files via CLI flags (redundant but safe)
  (my/aidermacs--append-unique-args
   (list "--input-history-file" (getenv "AIDER_INPUT_HISTORY_FILE")
         "--chat-history-file"  (getenv "AIDER_CHAT_HISTORY_FILE")
         "--llm-history-file"   (getenv "AIDER_LLM_HISTORY_FILE")))

  ;; Backend selection must be delayed and guarded
  (cond
   ((require 'vterm nil 'noerror)
    (setopt aidermacs-backend 'vterm))
   (t
    (setopt aidermacs-backend 'comint)
    (display-warning
     'aidermacs
     "vterm not available; falling back to comint backend"))))

;; ---------------------------------------------------------------------------
;; Aidermacs package configuration
;; ---------------------------------------------------------------------------

(leaf aidermacs
  :straight t
  :init
  (cond
   ((getenv "OPENROUTER_API_KEY")
```

```
     (setenv "OPENAI_API_BASE" "https://openrouter.ai/api/v1")
     (setenv "OPENAI_API_KEY"  (getenv "OPENROUTER_API_KEY"))
     (setopt aidermacs-default-model
             "openrouter/anthropic/claude-3.5-sonnet"))
    ((getenv "OPENAI_API_KEY")
     (setenv "OPENAI_API_BASE" "https://api.openai.com/v1")
     (setopt aidermacs-default-model
             "gpt-4o-mini"))
    (t
     (display-warning
      'aidermacs
      "No API keys set. Set OPENROUTER_API_KEY or OPENAI_API_KEY.")))
  :custom
  ((aidermacs-retry-attempts . 3)
   (aidermacs-retry-delay   . 2.0)))

(provide 'dev-ai)
;;; dev/dev-ai.el ends here
```

5. dev/dev-term.el

```
;;; dev/dev-term.el --- Terminal integration helpers -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Terminal and shell workflow integration.
;;
;; This module:
;; - configures vterm and vterm-toggle
;; - applies UI palette consistently to terminal faces
;; - defines window display rules for terminal buffers
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'cl-lib))

;; Palette is provided by the caller's load-path setup.
(require 'ui-nano-palette)

(defun my/vterm-apply-palette ()
  "Apply nano-style palette to vterm faces safely and compatibly.
Some vterm builds don't define `vterm-color-default`; use `vterm` face instead."
  ;; Fallbacks (in case palette wasn't set yet)
  (defvar nano-color-foreground "#374151")
  (defvar nano-color-background "#fafafa")
  (defvar nano-color-salient    "#2563eb")
```

```
  (defvar nano-color-popout     "#6b7280")
  (defvar nano-color-critical   "#dc2626")
  (defvar nano-color-strong     "#111827")
  (defvar nano-color-faded      "#9ca3af")
  (defvar nano-color-subtle     "#e5e7eb")

  (cl-labels
      ((safe-face-set (face &rest props)
         (when (facep face)
           (apply #'set-face-attribute face nil props))))

    ;; Default fg/bg: prefer vterm-color-default; fall back to vterm
    (if (facep 'vterm-color-default)
        (safe-face-set 'vterm-color-default
                       :foreground nano-color-foreground
                       :background nano-color-background)
      (safe-face-set 'vterm
                     :foreground nano-color-foreground
                     :background nano-color-background))

    ;; 8-color palette (keep backgrounds unobtrusive)
    (safe-face-set 'vterm-color-black   :foreground nano-color-
strong   :background 'unspecified)
    (safe-face-set 'vterm-color-red     :foreground nano-color-
critical :background 'unspecified)
    (safe-face-set 'vterm-color-green   :foreground "#10b981"           :background 'unspe
500
    (safe-face-set 'vterm-color-yellow  :foreground nano-color-
popout   :background 'unspecified)
    (safe-face-set 'vterm-color-blue    :foreground nano-color-
salient  :background 'unspecified)
    (safe-face-set 'vterm-color-magenta :foreground "#a21caf"           :background 'unspe
700
    (safe-face-set 'vterm-color-cyan    :foreground "#0891b2"           :background 'unspe
600
    (safe-face-set 'vterm-color-white   :foreground nano-color-
subtle   :background 'unspecified)))

(defun my/vterm-buffer-p (buf)
  "Return non-nil if BUF is a vterm buffer."
  (with-current-buffer buf
    (or (eq major-mode 'vterm-mode)
        (string-prefix-p "*vterm" (buffer-name buf)))))

(leaf vterm
  :doc  "Emacs libvterm integration"
  :url  "https://github.com/akermu/emacs-libvterm"
  :straight t
  :config
  ;; Apply once vterm is loaded
  (my/vterm-apply-palette)
  ;; Re-apply after theme activation to keep colors consistent
  (with-eval-after-load 'cus-theme
```

```
    (if (boundp 'enable-theme-functions)
        (add-hook 'enable-theme-functions #'my/vterm-apply-palette)
      (advice-add 'enable-theme :after (lambda (&rest _) (my/vterm-apply-
palette))))))

(leaf vterm-toggle
  :doc  "Toggle between vterm and the current buffer"
  :url  "https://github.com/jixiuf/vterm-toggle"
  :straight t
  :custom
  (vterm-toggle-cd-auto-create-buffer . t)
  (vterm-toggle-fullscreen-p          . nil)
  (vterm-toggle-scope                 . 'project))

;; Buffer display rule must be set after vterm is available
(with-eval-after-load 'vterm
  (add-to-list 'display-buffer-alist
               `(my/vterm-buffer-p
                 (display-buffer-reuse-window display-buffer-in-direction)
                 (direction       . bottom)
                 (dedicated       . t)
                 (reusable-frames . visible)
                 (window-height   . 0.3))))

(provide 'dev-term)
;;; dev/dev-term.el ends here
```

6. dev/dev-web-core.el

```
;;; dev/dev-web-core.el --- Project core helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Core project configuration and editor hygiene.
;;
;; This module:
;; - configures project.el and editor defaults
;; - enforces trailing whitespace and newline policy
;; - explicitly excludes Tree-sitter policy ownership
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf project :straight nil)

(leaf files :straight nil
  :custom ((require-final-newline . t)
           (delete-trailing-lines . t))
```

```
  :hook ((before-save-hook . delete-trailing-whitespace)))

(leaf editorconfig
  :straight t
  :global-minor-mode t)

(provide 'dev-web-core)
;;; dev/dev-web-core.el ends here
```

7. dev/dev-build.el

```
;;; dev/dev-build.el --- Build and Makefile tooling -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Build system helpers focused on Makefile workflows.
;;
;; This module:
;; - configures Makefile editing modes
;; - sets sane defaults for compilation buffers
;; - preserves minimal global impact
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf make-mode :straight nil
  :mode (("\\`Makefile\\'"    . makefile-gmake-mode)
         ("\\`GNUmakefile\\'" . makefile-gmake-mode)
         ("\\`makefile\\'"    . makefile-gmake-mode))
  :hook
  ((makefile-mode . my/set-make-compile-command)
   (makefile-mode . (lambda ()
                      (setq-local indent-tabs-mode t
                                  tab-width 8
                                  show-trailing-whitespace t))))
  :config

  (leaf compile :straight nil
    :custom ((compilation-scroll-output . t)
             (compilation-skip-threshold . 2)))

  (leaf ansi-color :straight nil
    :hook
    (compilation-filter . (lambda ()
                            (let ((inhibit-read-only t))
                              (ansi-color-apply-on-region
                               compilation-filter-start
                               (point-max)))))))
```

```
(defun my/set-make-compile-command ()
  "Set `compile-command' to `make -k' in Makefile buffers."
  (setq-local compile-command "make -k"))

(provide 'dev-build)
;;; dev/dev-build.el ends here
```

8. dev/dev-format.el

```
;;; dev/dev-format.el --- Code formatting via Apheleia -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Automatic code formatting for web-centric development.
;;
;; This module:
;; - enables Apheleia globally
;; - configures Prettier / prettierd formatters
;; - applies formatting to common web and markup modes
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf apheleia
  :straight t
  :require t
  :config
  (setf (alist-get 'prettierd apheleia-formatters)
        '("prettierd" filepath))
  (setf (alist-get 'prettier apheleia-formatters)
        '("npx" "prettier" "--stdin-filepath" filepath))
  (dolist (pair '((typescript-ts-mode . prettierd)
                  (tsx-ts-mode        . prettierd)
                  (json-ts-mode       . prettierd)
                  (css-ts-mode        . prettierd)
                  (markdown-mode      . prettierd)))
    (add-to-list 'apheleia-mode-alist pair))
  (apheleia-global-mode +1))

(provide 'dev-format)
;;; dev-format.el ends here
```

9. dev/dev-infra-modes.el

```
;;; dev/dev-infra-modes.el --- Infrastructure file modes -*- lexical-
binding: t; -*-
```

```
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Syntax support for infrastructure-related files.
;;
;; This module:
;; - enables modes for Docker Compose, .env, TOML, and Makefiles
;; - focuses purely on syntax and editing support
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf docker-compose-mode :straight t :mode ("docker-compose\\.*ya?ml\\'"))
(leaf dotenv-mode :straight t
  :mode (("\\.env\\..*\\'" . dotenv-mode)
         ("\\.env\\'"      . dotenv-mode)))
(leaf toml-mode :straight t :mode ("\\.toml\\'" . toml-mode))

(provide 'dev-infra-modes)
;;; dev/dev-infra-modes.el ends here
```

10. dev/dev-docker.el

```
;;; dev/dev-docker.el --- Docker integration helpers -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Docker and container-oriented development helpers.
;;
;; This module:
;; - provides major modes for Dockerfile and Compose files
;; - integrates Docker CLI management commands
;; - configures TRAMP container access
;; - supplies Dockerfile templates via Tempel
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf dockerfile-mode :straight t
  :mode (("Dockerfile\\(\\..*\\)?\\'" . dockerfile-mode)
         ("\\.dockerfile\\'"          . dockerfile-mode))
```

```
  :custom ((dockerfile-mode-command . "docker")))

(leaf yaml-mode :straight t
  :mode (("\\`docker-compose.*\\.ya?ml\\'" . yaml-mode)
         ("\\.ya?ml\\'"                    . yaml-mode)))

(leaf docker :straight t
  :commands (docker docker-containers docker-images docker-volumes docker-
networks))

(leaf tramp-container :straight nil
  :after tramp
  :init
  (setq tramp-container-method "docker"))

(leaf tempel :straight t
  :commands (tempel-insert)
  :init
  (with-eval-after-load 'tempel
    (defvar my:tempel-docker-templates
      '((dockerfile "FROM " p n
                    "WORKDIR /app" n
                    "COPY . /app" n
                    "RUN " p n
                    "CMD [" p "]" n)))
    (add-to-list 'tempel-user-elements my:tempel-docker-templates)))

(provide 'dev-docker)
;;; dev/dev-docker.el ends here
```

11. dev/dev-sql.el

```
;;; dev/dev-sql.el --- SQL and PostgreSQL helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; SQL editing and formatting helpers.
;;
;; This module:
;; - configures SQL modes for PostgreSQL
;; - enables indentation and formatting helpers
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf sql :straight nil
  :custom ((sql-product . 'postgres)))
```

```
(leaf sql-indent :straight t
  :hook (sql-mode . sqlind-minor-mode))

(leaf sqlformat
  :straight t
  :custom ((sqlformat-command . 'pgformatter)
           (sqlformat-args . '("--nostyle")))
  :hook (sql-mode . sqlformat-on-save-mode)
  :hook (sql-ts-mode . sqlformat-on-save-mode))

(provide 'dev-sql)
;;; dev-sql.el ends here
```

12. dev/dev-rest.el

```
;;; dev/dev-rest.el --- REST client helpers -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; REST client integration for testing HTTP APIs inside Emacs.
;;
;; This module:
;; - enables restclient for .http files
;; - integrates jq-based JSON inspection
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf restclient :straight t :mode ("\\.http\\'" . restclient-mode))
(leaf restclient-jq :straight t :after restclient)

(provide 'dev-rest)
;;; dev/dev-rest.el ends here
```

13. dev/dev-navigation.el

```
;;; dev/dev-navigation.el --- Search and navigation helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: dev
;;
;; Commentary:
;; Project-wide search and navigation helpers.
;;
```

```
;; This module:
;; - integrates ripgrep-based jumping via dumb-jump
;; - provides lightweight EWW search helpers
;; - avoids intrusive global navigation policy
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))

(leaf dumb-jump
  :straight t
  :hook (xref-backend-functions . dumb-jump-xref-activate)
  :custom ((dumb-jump-force-searcher  . 'rg)
           (dumb-jump-prefer-searcher . 'rg)))

(leaf multiple-cursors :straight t)

(leaf eww
  :straight nil
  :custom ((eww-search-prefix . "https://duckduckgo.com/html/?kl=jp-jp&k1=-
1&kc=1&kf=-1&q=")
           (eww-download-directory . "~/Downloads"))
  :init
  (defvar my:d:eww
    (expand-file-name "eww/"
                      (or (bound-and-true-p my:d:var)
                          user-emacs-directory)))
  (setopt eww-cache-directory (expand-file-name "cache/" my:d:eww))
  (unless (file-directory-p eww-cache-directory)
    (make-directory eww-cache-directory t))
  (setq eww-history-limit 200)

  (defvar eww-hl-search-word nil)

  (defun my/eww-search (term)
    "Search TERM with EWW and start isearch."
    (interactive "sSearch terms: ")
    (setq eww-hl-search-word term)
    (eww-browse-url (concat eww-search-prefix term)))

  (add-hook 'eww-after-render-hook
            (lambda ()
              (when eww-hl-search-word
                (isearch-mode t)
                (isearch-yank-string eww-hl-search-word)
                (setq eww-hl-search-word nil))))

  (defun my/eww-toggle-images ()
    (interactive)
    (setq shr-inhibit-images (not shr-inhibit-images))
    (eww-reload)))
```

```
(provide 'dev-navigation)
;;; dev/dev-navigation.el ends here
```

14. dev/dev-tools.el

```
;;; dev/dev-tools.el --- Development helper tools -*- lexical-binding: t; -*-
;;
;; Category: dev
;;
;;; Code:

(defun my/open-by-vscode ()
  "Open current file at point in VS Code."
  (interactive)
  (when (buffer-file-name)
    (async-shell-command
     (format "code -r -g %s:%d:%d"
             (buffer-file-name)
             (line-number-at-pos)
             (current-column)))))

(defun my/show-env-variable (var)
  "Show environment variable VAR."
  (interactive "sEnvironment variable: ")
  (message "%s = %s" var (or (getenv var) "Not set")))

(defun my/print-build-info () (interactive)
    (let ((buf (get-buffer-create "*Build Info*")))
      (with-current-buffer buf
        (let ((inhibit-read-only t))
          (erase-buffer)
          (insert (format "- GNU Emacs *%s*\n\n" emacs-version))
          (insert "|Property|Value|\n|--------|-----|\n")
          (insert (format "|Commit|%s|\n" (if (fboundp 'emacs-repository-get-
version)
                                               (emacs-repository-get-
version) "N/A")))
          (insert (format "|Branch|%s|\n" (if (fboundp 'emacs-repository-get-
branch)
                                               (emacs-repository-get-
branch) "N/A")))
          (insert (format "|System|%s|\n" system-configuration))
          (insert (format "|Date|%s|\n" (format-time-string "%Y-%m-
%d %T (%Z)" emacs-build-time)))
          (insert (format "|Patch|%s ns-inline.patch|\n" (if (boundp 'mac-ime--
cursor-type) "with" "N/A")))
          (insert (format "|Features|%s|\n" system-configuration-features))
          (insert (format "|Options|%s|\n" system-configuration-options)))
        (view-mode 1))
      (switch-to-buffer buf)))

(provide 'dev-tools)
;;; dev/dev-tools.el ends here
```

2.2.8 vcs/

1. Overview

   - Loaded after `dev`
   - Each file:
     - provides exactly one VCS-related feature
     - documents backend assumptions and failure modes
   - Disabling this layer must leave Emacs fully usable for non-version-controlled files

   (a) Purpose Provide a **consistent, explicit, and inspectable version control interaction layer** within Emacs.

   This layer encapsulates all interaction with external version control systems — primarily Git — while treating those systems as the authoritative source of truth.

   Emacs is used as a **control surface**, not as a replacement for VCS tooling.

   (b) What this layer does VCS modules integrate Emacs with external version control systems in a way that is observable, reversible, and failure-tolerant.

   Specifically, this layer is responsible for:

   - Bridging Emacs with VCS backends (primarily Git)
   - Exposing repository state, history, and diffs inside Emacs buffers
   - Providing interactive commands for common VCS workflows
   - Surfacing VCS information in buffers without mutating editing semantics

   Typical responsibilities include:

   - Repository status inspection and change overview
   - Commit, diff, hunk, and blame navigation
   - File- and project-scoped VCS operations
   - Lightweight review, history browsing, and annotation

   (c) What this layer does **not** do VCS modules intentionally do **not**:

   - Define global editing behavior or keybinding policy
   - Enforce a specific branching, commit, or review workflow
   - Replace, fork, or reimplement external VCS tools
   - Assume the presence of a repository or remote

   Those responsibilities belong to user workflows or external tooling.

   (d) Design constraints

   - VCS modules may depend on:
     - `core`
     - `ui`
     - `completion`
   - VCS modules must not depend on:
     - `orgx`
     - `dev`
     - `utils`
   - Interaction with external tools must be:
     - explicit
     - inspectable
     - resilient to tool absence or failure

   (e) Design principles

- External VCS tools are the **source of truth**
- Emacs acts as an interface layer, not an abstraction leak
- All commands must degrade gracefully when:
  - no repository is present
  - required binaries are missing
  - network access is unavailable
- User workflow choices remain:
  - opt-in
  - reversible
  - discoverable

(f) Observed implementation characteristics Based on the current configuration:

- Git integration is centered on **Magit** for interactive workflows
- Change visualization relies on lightweight gutter-style indicators
- No assumptions are made about:
  - hosting provider
  - branching model
  - collaboration workflow
- VCS features can be enabled or disabled independently

(g) Module map

File	Responsibility
`vcs/vcs-magit.el`	Magit-based interactive Git workflows
`vcs/vcs-gutter.el`	In-buffer change indicators (diff-hl)
`vcs/vcs-forge.el`	Optional GitHub / GitLab integration via Forge

(h) Notes

- This layer is intentionally thin and compositional.
- Any logic that **depends** on VCS state belongs elsewhere.
- If disabling this layer breaks core editing, the dependency is misplaced.

This layer exists to make version control **visible and accessible**, not opinionated or mandatory.

2. vcs/vcs-magit.el

```
;;; vcs/vcs-magit.el --- Magit Git integration -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: vcs
;;
;; Commentary:
;; Core Magit configuration for Git operations.
;;
;; This module:
;; - enables Magit commands and status buffers
;; - adjusts refresh and revert behavior conservatively
;; - keeps Git interaction explicit and user-driven
;;
;;; Code:
```

```
(eval-when-compile (require 'leaf))

(leaf magit
  :straight t
  :commands (magit-status magit-dispatch)
  :init
  (setq magit-auto-revert-mode nil)
  :config
  (setq magit-refresh-status-buffer nil
        magit-diff-refine-hunk 'all))

(provide 'vcs-magit)
;;; vcs/vcs-magit.el ends here
```

3. vcs/vcs-gutter.el

```
;;; vcs/vcs-gutter.el --- Git change indicators -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: vcs
;;
;; Commentary:
;; Visual indicators for Git working tree changes.
;;
;; This module:
;; - shows added, modified, and deleted lines in the fringe
;; - integrates diff-hl with dired and magit buffers
;; - avoids altering Git workflows or commands
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf diff-hl
  :straight t
  :commands (diff-hl-mode diff-hl-dired-mode diff-hl-magit-post-refresh)
  :hook ((prog-mode . diff-hl-mode)
         (text-mode . diff-hl-mode)
         (dired-mode . diff-hl-dired-mode))
  :config
  (with-eval-after-load 'magit
    (add-hook 'magit-post-refresh #'diff-hl-magit-post-refresh))
  (customize-set-variable 'diff-hl-draw-borders nil))

(provide 'vcs-gutter)
;;; vcs/vcs-gutter.el ends here
```

4. vcs/vcs-forge.el

```
;;; vcs/vcs-forge.el --- Forge integration for Git forges -*- lexical-
binding: t; -*-
```

```
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: vcs
;;
;; Commentary:
;; GitHub and GitLab integration via Forge.
;;
;; This module:
;; - enables issue and pull/merge request workflows
;; - configures Forge database location explicitly
;; - avoids implicit database or network side effects
;;
;;; Code:

(eval-when-compile (require 'leaf))

(require 'utils-path nil t) ;; my/ensure-directory-exists

(leaf forge
  :straight t
  :after magit
  :init
  (with-eval-after-load 'emacsql
    (when (boundp 'emacsql-sqlite3-executable)
      (setq emacsql-sqlite3-executable nil)))
  :config
  (let* ((db-dir (expand-file-name "forge" no-littering-var-directory))
         (db (expand-file-name "forge-database.sqlite" db-dir)))
    (my/ensure-directory-exists db-dir)
    (setq forge-database-file db)))

(provide 'vcs-forge)
;;; vcs/vcs-forge.el ends here
```

2.2.9 utils/

1. Overview

   - Loaded last by `modules.el`
   - Each file:
     - provides exactly one utility feature
     - remains safe to load in isolation
   - Removing this layer must not break any higher-level architectural guarantees

   (a) Purpose Host **small, domain-specific utilities** that do not justify a dedicated architectural layer.

   This layer exists as a **pragmatic utility shelf**: narrowly scoped helpers live here to avoid contaminating core abstractions, while remaining easy to audit, extract, or delete.

   (b) What this layer does Utils modules provide **localized assistance** to other layers without owning workflows, policy, or long-lived behavior.

They are responsible for:

- Small, focused helper functions and interactive commands
- Solving localized or cross-cutting problems that span layers
- Supplying glue code that would be inappropriate in higher layers

Typical responsibilities include:

- Buffer, window, and process housekeeping helpers
- Lightweight automation and defensive cleanup logic
- One-off integrations with external tools or Emacs internals
- Quality-of-life helpers that do not define behavior on their own

(c) What this layer does **not** do Utils modules intentionally do **not**:

- Define global policy, defaults, or invariants
- Perform orchestration or lifecycle management
- Introduce new architectural concepts or abstractions
- Replace, override, or shadow behavior from higher layers
- Encode user-, device-, or environment-specific configuration

Those responsibilities are explicitly owned by other layers.

(d) Design constraints

- Utils modules may depend on:
  - `core`
  - `ui`
  - `completion`
  - `orgx`
  - `dev`
  - `vcs`
- Utils modules must **not** be depended on by:
  - `core`
  - `ui`
  - `completion`
  - `orgx`
  - `dev`
  - `vcs`
- All dependencies must be:
  - minimal
  - explicit
  - non-circular

This makes utils a **leaf layer** in the dependency graph.

(e) Design principles

- Keep modules **small, single-purpose, and disposable**
- Avoid accumulation of loosely related helpers in one file
- Prefer promotion to a dedicated layer if scope or importance grows
- Deletion must always remain a valid and low-risk refactoring option
- Utilities should be easy to reason about in isolation

(f) Observed implementation characteristics Based on the current configuration:

- Utilities are defensive and side-effect conscious
- Many helpers are interactive diagnostics or maintenance tools

- No utils module is required for startup correctness
- Disabling the entire utils layer should degrade convenience, not correctness

(g) Module map

File	Responsibility
`utils/utils-path.el`	Safe directory and path helpers
`utils/utils-async.el`	Minimal asynchronous task helpers
`utils/utils-buffers.el`	Buffer lifecycle and cleanup helpers
`utils/utils-edit.el`	Editing and save-time helper utilities
`utils/utils-dired.el`	Dired convenience helpers
`utils/utils-org-agenda.el`	Cached org-agenda file discovery
`utils/utils-lint.el`	Static lint helpers for Emacs Lisp
`utils/utils-diagnostics.el`	Configuration and runtime diagnostics
`utils/utils-functions.el`	Small general-purpose helper functions

(h) Notes

- This layer is intentionally heterogeneous and low-ceremony.
- Utilities should **never** become implicit dependencies.
- If a utils module feels "important" , it likely belongs elsewhere.

This layer exists to make the rest of the system **simpler**, not smarter.

2. utils/utils-functions.el

```
;;; utils/utils-functions.el --- General-purpose small utilities -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Small, safe utility functions shared across the configuration.
;;
;; This module:
;; - provides lightweight editing and navigation helpers
;; - includes optional Org and UI convenience functions
;; - avoids owning long-lived workflows or global policy
;;
;;; Code:

;;;; Built-ins ----------------------------------------------------------------

(require 'cl-lib) ;; built-in
(require 'imenu) ;; built-in2

;;;; Optional deps & vars (for byte-compiler) ---------------------------------

(eval-when-compile
  (declare-function imenu-list-minor-mode  "imenu-list")
  (declare-function imenu-list-stop-timer  "imenu-list")
  (declare-function imenu-list-display-dwim "imenu-list")
  (declare-function nano-modeline-render   "nano-modeline"))
```

```
(defvar imenu-list-after-jump-hook nil)
(defvar imenu-list-position 'left)
(defvar imenu-list-size 30)
(defvar imenu-list-focus-after-activation t)
(defvar imenu-list--displayed-buffer nil)

;;;; Editing helpers ------------------------------------------------------------

(defun my/kill-buffer-smart ()
  "Kill buffer and window when there are multiple windows; otherwise kill buffer."
  (interactive)
  (if (one-window-p)
      (kill-buffer)
    (kill-buffer-and-window)))

(defalias 'my/smart-kill-buffer #'my/kill-buffer-smart)

;;;; Nano/Modeline helper -------------------------------------------------------

(defun my/nano-headerline (buf subtitle)
  "Return a header-line string; prefer nano-modeline when available."
  (let* ((name (if (and buf (buffer-live-p buf))
                   (buffer-name buf)
                 (buffer-name)))
         (extra ""))
    (if (fboundp 'nano-modeline-render)
        (nano-modeline-render nil name subtitle extra)
      (concat "  "
              (propertize name 'face 'mode-line-buffer-id)
              "  " subtitle))))

;;;; Org helpers ----------------------------------------------------------------

(defun my/org-tree-to-indirect-buffer ()
  "Show current Org subtree in an indirect buffer and reveal its content."
  (interactive)
  (when (derived-mode-p 'org-mode)
    (cond
     ((fboundp 'org-fold-show-all) (org-fold-show-all))
     ((fboundp 'org-show-all)      (org-show-all))))
  (org-tree-to-indirect-buffer))

(defun my/org-sidebar ()
  "Open an imenu-list sidebar with safe fallbacks."
  (interactive)

  (when (require 'imenu-list nil 'noerror)
    (setq imenu-list-after-jump-hook #'recenter
          imenu-list-position 'left
          imenu-list-size 36
          imenu-list-focus-after-activation t)
```

```
    (when (buffer-base-buffer)
      (switch-to-buffer (buffer-base-buffer)))

    (imenu-list-minor-mode 1)

    (when (fboundp 'imenu-list-stop-timer)
      (imenu-list-stop-timer))

    (hl-line-mode 1)
    (when (facep 'nano-subtle)
      (face-remap-add-relative 'hl-line :inherit 'nano-subtle))

    (setq header-line-format
          `(:eval
            (my/nano-headerline
             ,(when (boundp 'imenu-list--displayed-buffer)
                'imenu-list--displayed-buffer)
             "(outline)")))

    (setq-local cursor-type nil)

    (when (fboundp 'imenu-list-display-dwim)
      (imenu-list-display-dwim))))

(defun my/org-sidebar-toggle ()
  "Toggle the imenu-list sidebar."
  (interactive)
  (let ((win (get-buffer-window "*Ilist*")))
    (if win
        (progn
          (quit-window nil win)
          (when (buffer-base-buffer)
            (switch-to-buffer (buffer-base-buffer))))
      (my/org-sidebar))))

(provide 'utils-functions)
;;; utils/utils-functions.el ends here
```

3. utils/utils-path.el

```
;;; utils/utils-path.el --- Minimal filesystem helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Small helpers for safe filesystem operations.
;;
;; This module:
;; - provides defensive directory creation utilities
```

```
;; - avoids introducing global filesystem policy
;; - acts as a fallback when higher-level helpers are absent
;;
;;; Code:

(unless (fboundp 'my/ensure-directory-exists)
  (defun my/ensure-directory-exists (dir)
    "Create DIR if it does not exist."
    (unless (file-directory-p dir)
      (make-directory dir t))))

(provide 'utils-path)
;;; utils/utils-path.el ends here
```

4. utils/utils-async.el

```
;;; utils/utils-async.el --- Safe asynchronous helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Minimal helpers for running asynchronous tasks safely.
;;
;; This module:
;; - executes tasks asynchronously with error isolation
;; - reports failures without interrupting Emacs
;; - avoids background scheduling policy
;;
;;; Code:

(defun my/safe-run-async (task)
  "Run TASK asynchronously, catching and reporting any errors."
  (run-at-time 0 nil
               (lambda ()
                 (condition-case err
                     (funcall task)
                   (error (message "[async] error: %s" err))))))

(provide 'utils-async)
;;; utils/utils-async.el ends here
```

5. utils/utils-dired.el

```
;;; utils/utils-dired.el --- Small Dired navigation helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
```

```
;;
;; Category: utils
;;
;; Commentary:
;; Minor helper commands extending Dired navigation.
;;
;; This module:
;; - provides convenience commands for file viewing
;; - avoids redefining or overriding core Dired behavior
;; - remains local to interactive use
;;
;;; Code:

(declare-function dired-get-file-for-visit "dired")
(declare-function dired-goto-subdir "dired")

(defun my/dired-view-file-other-window ()
  "Open Dired file or directory in another window."
  (interactive)
  (let ((file (dired-get-file-for-visit)))
    (if (file-directory-p file)
        (or (and (cdr dired-subdir-alist)
                 (dired-goto-subdir file))
            (dired file))
      (view-file-other-window file))))

(provide 'utils-dired)
;;; utils/utils-dired.el ends here
```

6. utils/utils-buffers.el

```
;;; utils/utils-buffers.el --- Temporary buffer housekeeping -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Automatic cleanup of temporary and dead-process buffers.
;;
;; This module:
;; - periodically removes unused helper buffers
;; - never touches visible, modified, or file-backed buffers
;; - runs conservatively with explicit opt-in customization
;;
;;; Code:

(eval-when-compile (require 'leaf))

(leaf nil
  :straight nil
```

```
  :init
  (defcustom utils-buffers-enable-p t
    "Enable automatic buffer housekeeping."
    :type 'boolean
    :group 'utils)

  (defvar utils-buffers--temporary-regexp
    (rx string-start "*"
        (or "Help" "Warnings" "Compile-Log" "Backtrace"
            "Async-native-compile-log" "eglot-events")
        (* any) string-end))

  (defun utils-buffers--temporary-p (buffer)
    (with-current-buffer buffer
      (and (string-match-p utils-buffers--temporary-regexp (buffer-name))
           (not (buffer-file-name))
           (not (buffer-modified-p))
           (not (get-buffer-window buffer 'visible)))))

  (defun utils-buffers--dead-process-p (buffer)
    (let ((proc (get-buffer-process buffer)))
      (and proc
           (memq (process-status proc) '(exit signal)))))

  (defun utils-buffers-cleanup ()
    "Clean up temporary and dead-process buffers."
    (interactive)
    (when utils-buffers-enable-p
      (dolist (buf (buffer-list))
        (when (or (utils-buffers--temporary-p buf)
                  (utils-buffers--dead-process-p buf))
          (ignore-errors
            (kill-buffer buf))))))

  (run-with-timer 900 900 #'utils-buffers-cleanup))

(provide 'utils-buffers)
;;; utils/utils-buffers.el ends here
```

7. utils/utils-edit.el

```
;;; utils/utils-edit.el --- Lightweight editing helpers -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Small editing-related helpers for day-to-day workflows.
;;
;; This module:
```

```
;; - provides quick buffer revert utilities
;; - optionally updates lightweight timestamps on save
;; - never enforces global editing policy by default
;;
;;; Code:

;; ----------------------------------------------------------------------------
;; Customization
;; ----------------------------------------------------------------------------

(defgroup utils-edit nil
  "Lightweight editing helpers."
  :group 'editing)

(defcustom utils-edit-enable-p nil
  "Enable utils-edit save-time helpers.

When non-nil, `my/save-buffer-wrapper` is installed into
`before-save-hook`.

This option is opt-in by design, to avoid defining global editing policy
from the utils layer."
  :type 'boolean
  :group 'utils-edit)

;; ----------------------------------------------------------------------------
;; Editing helpers
;; ----------------------------------------------------------------------------

(defun my/revert-buffer-quick ()
  "Revert current buffer without confirmation."
  (interactive)
  (revert-buffer :ignore-auto :noconfirm))

(defun my/save-buffer-wrapper ()
  "Insert or update a `$Lastupdate` timestamp at the top of the buffer.

This runs only for:
- file-backed buffers
- non-Org text buffers

It is intentionally conservative and avoids semantic file formats."
  (when (and buffer-file-name
             (derived-mode-p 'text-mode)
             (not (derived-mode-p 'org-mode)))
    (let ((timestamp
           (concat "$Lastupdate: "
                   (format-time-string "%Y/%m/%d %H:%M:%S")
                   " $")))
      (save-excursion
        (goto-char (point-min))
        (while (re-search-forward
                "\\$Lastupdate: [0-9/: ]*\\$" nil t)
```

```
          (replace-match timestamp t nil))))))

;; ----------------------------------------------------------------------
;; Hook installation (opt-in)
;; ----------------------------------------------------------------------

(defun utils-edit--maybe-enable ()
  "Install or remove utils-edit hooks based on `utils-edit-enable-p`."
  (if utils-edit-enable-p
      (add-hook 'before-save-hook #'my/save-buffer-wrapper)
    (remove-hook 'before-save-hook #'my/save-buffer-wrapper)))

;; Apply once at load time
(utils-edit--maybe-enable)

;; React to customization changes
(add-variable-watcher
 'utils-edit-enable-p
 (lambda (_sym _newval _op _where)
   (utils-edit--maybe-enable)))

(provide 'utils-edit)
;;; utils/utils-edit.el ends here
```

8. utils/utils-org-agenda.el

```
;;; utils/utils-org-agenda.el --- Cached org-agenda-files builder -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Build org-agenda-files using a persistent on-disk cache.
;;
;; This module:
;; - scans org-directory defensively
;; - stores agenda file lists in a reusable cache
;; - avoids runtime penalties during startup and agenda refresh
;;
;;; Code:

(require 'subr-x)
(require 'seq)

(defgroup utils-org-agenda nil
  "Cached org-agenda-files builder."
  :group 'org)

(defcustom utils-org-agenda-cache-file
  (expand-file-name "org-agenda-files.cache"
```

```
                    (or (bound-and-true-p no-littering-var-directory)
                        user-emacs-directory))
  "Cache file for org-agenda-files."
  :type 'file
  :group 'utils-org-agenda)

(defcustom utils-org-agenda-exclude-regexp
  "archives"
  "Regexp used to exclude files from org-agenda."
  :type 'regexp
  :group 'utils-org-agenda)

(defun utils-org-agenda--valid-org-directory-p ()
  "Return non-nil if org-directory is defined and readable."
  (and (boundp 'org-directory)
       (stringp org-directory)
       (file-directory-p org-directory)))

(defun utils-org-agenda--scan (org-directory)
  "Recursively scan ORG-DIRECTORY and return a list of agenda files."
  (when (file-directory-p org-directory)
    (seq-filter
     (lambda (file)
       (and (string-match-p "\\.org\\'" file)
            (not (file-symlink-p file))
            (not (string-match-p utils-org-agenda-exclude-regexp file))))
     (directory-files-recursively
      org-directory "\\.org\\'" nil nil))))

(defun utils-org-agenda--load-cache ()
  "Load cached agenda files from disk."
  (when (file-readable-p utils-org-agenda-cache-file)
    (with-temp-buffer
      (insert-file-contents utils-org-agenda-cache-file)
      (read (current-buffer)))))

(defun utils-org-agenda--save-cache (files)
  "Save FILES to the agenda cache."
  (with-temp-file utils-org-agenda-cache-file
    (prin1 files (current-buffer))))

;;;###autoload
(defun utils-org-agenda-build (&optional force)
  "Return agenda files using the cache.
If FORCE is non-nil, rebuild the cache."
  (if (not (utils-org-agenda--valid-org-directory-p))
      nil
    (let ((cached (and (not force)
                       (utils-org-agenda--load-cache))))
      (if (and cached (listp cached))
          cached
        (let ((files (utils-org-agenda--scan org-directory)))
          (utils-org-agenda--save-cache files)
```

```
          files)))))

;;;###autoload
(defun utils-org-agenda-rebuild ()
  "Force a rebuild of the org-agenda-files cache."
  (interactive)
  (setq org-agenda-files (utils-org-agenda-build t))
  (message "org-agenda-files cache rebuilt (%d files)"
           (length org-agenda-files)))

(provide 'utils-org-agenda)
;;; utils/utils-org-agenda.el ends here
```

9. utils/utils-lint.el

```
;;; utils/utils-lint.el --- Static lint helpers for Emacs Lisp -*- lexical-
binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Interactive static lint helpers for Emacs Lisp buffers.
;;
;; This module:
;; - integrates checkdoc and package-lint when available
;; - provides buffer- and directory-scoped commands
;; - never modifies files or editor state
;;
;;; Code:

(eval-when-compile
  (require 'leaf)
  (require 'subr-x))

;; ------------------------------------------------------------------------
;; Optional dependencies (declared defensively)
;; ------------------------------------------------------------------------

(declare-function checkdoc-current-buffer "checkdoc")
(declare-function checkdoc-file           "checkdoc")
(declare-function package-lint-current-buffer "package-lint")
(declare-function package-lint-buffer     "package-lint")

;; ------------------------------------------------------------------------
;; Customization
;; ------------------------------------------------------------------------

(defgroup utils-lint nil
  "Static lint helpers for Emacs Lisp."
  :group 'lisp)
```

```
(defcustom utils-lint-enable-package-lint-p t
  "Enable package-lint checks when available."
  :type 'boolean
  :group 'utils-lint)

;; ----------------------------------------------------------------------
;; Helpers
;; ----------------------------------------------------------------------

(defun utils-lint--elisp-file-p (file)
  "Return non-nil if FILE looks like an Emacs Lisp file."
  (and (stringp file)
       (string-match-p "\\.el\\'" file)
       (file-regular-p file)))

(defun utils-lint--ensure-elisp-buffer ()
  "Signal an error unless current buffer is visiting an Emacs Lisp file."
  (unless (and buffer-file-name
               (utils-lint--elisp-file-p buffer-file-name))
    (user-error "Not visiting an Emacs Lisp file")))

;; ----------------------------------------------------------------------
;; Public commands
;; ----------------------------------------------------------------------

;;;###autoload
(defun utils-lint-checkdoc-current-buffer ()
  "Run checkdoc on the current Emacs Lisp buffer."
  (interactive)
  (utils-lint--ensure-elisp-buffer)
  (require 'checkdoc)
  (checkdoc-current-buffer)
  (message "checkdoc completed"))

;;;###autoload
(defun utils-lint-package-lint-current-buffer ()
  "Run package-lint on the current Emacs Lisp buffer, if available."
  (interactive)
  (utils-lint--ensure-elisp-buffer)
  (unless utils-lint-enable-package-lint-p
    (user-error "package-lint is disabled by customization"))
  (unless (require 'package-lint nil t)
    (user-error "package-lint is not installed"))
  (package-lint-current-buffer)
  (message "package-lint completed"))

;;;###autoload
(defun utils-lint-run-all ()
  "Run checkdoc and package-lint on the current Emacs Lisp buffer."
  (interactive)
  (utils-lint-checkdoc-current-buffer)
  (when (and utils-lint-enable-package-lint-p
```

```
            (require 'package-lint nil t))
    (utils-lint-package-lint-current-buffer)))

;; ---------------------------------------------------------------------
;; Batch / CI helper
;; ---------------------------------------------------------------------

;;;###autoload
(defun utils-lint-run-on-directory (directory)
  "Run checkdoc and package-lint on all .el files under DIRECTORY."
  (interactive "DDirectory: ")
  (let ((files (directory-files-recursively directory "\\.el\\'")))
    (unless files
      (message "No .el files found"))
    (dolist (file files)
      (when (utils-lint--elisp-file-p file)
        (with-current-buffer (find-file-noselect file)
          (condition-case err
              (progn
                (checkdoc-current-buffer)
                (when (and utils-lint-enable-package-lint-p
                           (require 'package-lint nil t))
                  (package-lint-current-buffer)))
            (error
             (message "[lint] %s: %s"
                      (file-name-nondirectory file)
                      (error-message-string err))))
          ;; IMPORTANT: must be inside with-current-buffer
          (kill-buffer (current-buffer))))))
  (message "Lint run completed"))

(provide 'utils-lint)
;;; utils-lint.el ends here
```

10. utils/utils-diagnostics.el

```
;;; utils/utils-diagnostics.el --- Diagnostic helpers -*- lexical-binding: t; -
*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: utils
;;
;; Commentary:
;; Diagnostic helpers for inspecting configuration consistency.
;;
;; This module:
;; - defines interactive diagnostic commands only
;; - introduces no global hooks
;; - never alters runtime behavior
;;
;;; Code:
```

```
(defun my/find-keybinding-conflicts ()
  "Find conflicting keybindings in active keymaps."
  (interactive)
  (let ((conflicts (make-hash-table :test 'equal)))
    (dolist (map (current-active-maps t))
      (map-keymap
       (lambda (key cmd)
         (when (commandp cmd)
           (let* ((desc (key-description (vector key)))
                  (lst (gethash desc conflicts)))
             (puthash desc (delete-dups (cons cmd lst)) conflicts))))
       map))
    (with-current-buffer (get-buffer-create "*Keybinding Conflicts*")
      (erase-buffer)
      (maphash
       (lambda (k v)
         (when (> (length v) 1)
           (insert (format "%s → %s\n"
                           k (mapconcat #'symbol-name v ", ")))))
       conflicts)
      (view-mode 1)
      (pop-to-buffer (current-buffer)))))

(defun my/check-provide-matches-file (&optional buffer)
  "Check whether BUFFER provides a feature matching its file name.

If BUFFER is nil, use the current buffer.
This command is purely diagnostic and has no side effects."
  (interactive)
  (let* ((buf (or buffer (current-buffer)))
         (file (buffer-file-name buf)))
    (unless file
      (user-error "Current buffer is not visiting a file"))
    (unless (string-match-p "\\.el\\'" file)
      (user-error "Not an Emacs Lisp file"))
    (let* ((base (file-name-base file))
           (feat (intern base)))
      (if (featurep feat)
          (message "[provide] OK: `%s` is provided" feat)
        (message "[provide] WARNING: `%s` is not provided" feat)))))

(defun my/check-provide-in-directory (directory)
  "Check provide/filename consistency for all .el files under DIRECTORY."
  (interactive "DDirectory: ")
  (let ((files (directory-files-recursively directory "\\.el\\'")))
    (dolist (file files)
      (with-current-buffer (find-file-noselect file)
        (condition-case err
            (my/check-provide-matches-file (current-buffer))
          (error
           (message "[provide] %s: %s"
                    (file-name-nondirectory file)
```

```
                    (error-message-string err))))
        (kill-buffer (current-buffer))))
    (message "Provide check completed")))

(provide 'utils-diagnostics)
;;; utils/utils-diagnostics.el ends here
```

2.3 Personal Profile & Device Integrati

2.3.1 Overview

1. Personal Layer Philosophy The `personal/` layer provides **user- and device-specific overlays** on top of the shared, version-controlled configuration.

   Its role is to express **identity, environment, and personal workflow glue** without influencing global policy, architectural decisions, or cross-user behavior.

   This layer exists explicitly to keep the shared system:

   - reproducible
   - auditable
   - portable across machines and users

   while still allowing the configuration to feel **native** on a specific device.

2. Scope definition

   This layer MAY contain:

   - Identity information (user name, email address)
   - Personal feature flags and thresholds (UI bundle selection, LSP preference, timing knobs)
   - Device- and OS-specific glue (IME behavior, mouse/scroll tuning, platform quirks)
   - Personal keybindings and workflow integrations
   - Optional integrations with external, user-owned services

   This layer MUST NOT contain:

   - Core architectural or cross-user decisions
   - Shared defaults or global policy
   - Modules that other layers depend on
   - Assumptions that affect startup correctness

   Hooks and timers are permitted **only** when they are:

   - strictly local to the user' s device or workflow
   - defensive (safe to no-op when unavailable)
   - non-invasive to global behavior

3. Purpose Provide **personal overlays** that adapt the configuration to a specific user and machine, without compromising the modularity, determinism, or reproducibility of shared layers.

   Concretely, this layer exists to:

   - encode identity and safety-related editor defaults
   - select between alternative shared stacks (UI / LSP)

- bind shared infrastructure to local filesystem layout
- integrate OS- and device-specific affordances
- host convenience glue that would be inappropriate elsewhere

4. What this configuration does

   (a) Identity & safety
   - Sets `user-full-name` and `user-mail-address`
   - Applies safety-related editor preferences:
     - disables font cache compaction on macOS
     - enables in-memory passphrase caching for `plstore`

   (b) Look & feel switches
   - Declares personal font preferences and default size
   - Selects UI and LSP stacks explicitly:
     - `my:use-ui` (e.g. `nano`)
     - `my:use-lsp` (e.g. `eglot`)
   - Delegates actual behavior to shared UI and dev layers

   (c) Directories & Org wiring
   - Defines a cloud-rooted workspace (default: `~/Documents`)
   - Derives:
     - `my:d:org`
     - `my:d:blog`
   - Ensures required directories exist
   - Sets `org-directory`
   - Computes `org-agenda-files` dynamically by scanning non-archive `.org` files
   - Removes sensitive or excluded paths from `load-path`

   (d) macOS input method integration (`sis`)
   - Configures ABC ⇄ Kotoeri (Romaji) switching via `macism`
   - Enables cursor-color, respect, and inline modes when available
   - All functionality is guarded with `fboundp` checks

   (e) Cursor color keep-alive
   - Reapplies the frame' s cursor color to the `cursor` face after theme reloads
   - Prevents cursor desynchronization across theme switches

   (f) Device profile (MX Ergo S)
   - Applies conservative, smooth scrolling defaults
   - Preserves screen position and margins
   - Enables tilt scrolling
   - Mouse bindings:
     - `mouse-2` → yank
     - `mouse-4/5` → previous / next buffer

   (g) Apple Music integration (macOS)
   - Defines asynchronous and synchronous AppleScript helpers
   - Exposes interactive commands:
     - play / pause
     - next / previous track
     - playlist selection

- current track info
- Provides a Hydra bound to `C-c M`
- Optionally integrates with Meow leader keys when available

5. Module map

Module file	Role
`personal/user.el`	Identity, fonts, UI/LSP switches, Org paths
`personal/device-darwin.el`	macOS-specific device and IME glue
`personal/apple-music.el`	Apple Music control (AppleScript + Hydra)

6. Execution flow

(a) **Personal bootstrap**
- Identity, fonts, and UI/LSP switches are established
- Cloud, Org, and blog directories are defined and ensured

(b) **Org wiring**
- `org-directory` is set
- `org-agenda-files` is derived by filtering non-archive Org files

(c) **Hygiene**
- Sensitive or excluded directories are removed from `load-path`

(d) **macOS-only glue**
- `sis` is configured defensively
- Cursor color synchronization hook is installed

(e) **Device profile**
- Mouse and scroll tuning for MX Ergo S is applied

(f) **Apple Music integration**
- AppleScript runners are defined
- Interactive commands and Hydra are exposed

7. Key settings (reference)

`my:font-default` "JetBrains Mono"

`my:font-variable-pitch` "Noto Sans JP"

`my:font-size` 18

`my:use-ui` 'nano

`my:use-lsp` 'eglot

`org-directory` `~/Documents/org`

`org-agenda-files` all `*.org` under `org-directory`, excluding `archives`

MX Ergo scroll profile
- `mouse-wheel-scroll-amount`'(1 ((shift) . 5) ((control) . 10))=
- `scroll-conservatively=10000`
- `scroll-margin=2`
- `scroll-preserve-screen-position=t`

Cursor color keep-alive reapply `(set-face-background 'cursor (frame-parameter nil 'cursor-color))` on `after-load-theme`

8. Usage tips

- **Switch UI or LSP stacks**
  - Change `my:use-ui` to `'nano`, `'doom`, or `'none`
  - Change `my:use-lsp` to `'eglot` or `'lsp`
- **Change fonts**
  - Adjust font variables here
  - Shared UI modules consume them automatically
- **Apple Music control**
  - `C-c M` opens the Hydra
  - Keys:
    * `p` play/pause
    * `n` next
    * `b` back
    * `l` playlist
    * `i` track info
- **Agenda scope control**
  - Place files under an `archives/` directory
  - Or include "archives" in filenames to exclude them

9. Troubleshooting

- **sis does not switch input methods**
  - Verify input source IDs:
    * "com.apple.keylayout.ABC"
    * "com.apple.inputmethod.Kotoeri.RomajiTyping.Japanese"
  - All calls are guarded; missing functions will no-op
- **Cursor color incorrect after theme change**
  - Ensure the theme sets the frame' s `cursor-color` parameter
- **Hydra key missing**
  - Confirm `hydra` is installed and loaded
  - Binding is added via `with-eval-after-load`
- **Meow leader binding missing**
  - Requires both `meow` and `hydra`
  - Binding is added defensively

10. Related source blocks

```
;; See:
;; - personal/user.el
;; - personal/device-darwin.el
;; - personal/apple-music.el
```

2.3.2 user.el

```
;;; user.el --- Personal configuration -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
```

```
;; Category: personal
;;
;; Commentary:
;; Personal configuration overrides.
;;
;; This module:
;; - defines user identity and safety-related variables
;; - provides personal feature toggles and threshold overrides
;; - supplies directory paths and exclusions for runtime hygiene
;;
;;; Code:

;; ------------------------------------------------------------------------
;; Runtime requirements (do NOT rely on byte-compile)
;; ------------------------------------------------------------------------
(require 'leaf)
(require 'seq)

;; ------------------------------------------------------------------------
;; Personal Information
;; ------------------------------------------------------------------------
(leaf *personals
  :straight nil
  :init
  ;; Identity & safety
  (setq user-full-name "YAMASHITA, Takao"
        user-mail-address "tjy1965@gmail.com"
        inhibit-compacting-font-caches t
        plstore-cache-passphrase-for-symmetric-encryption t)

  ;; Fonts / UI
(setq ui-font-default "JetBrains Mono"
      ui-font-variable-pitch "Noto Sans JP"
      ui-font-size 18)

  (setq my:use-ui 'nano
        my:use-lsp 'eglot)

  ;; Cloud / Org / Blog directories
  (defvar my:d:cloud
    (expand-file-name "Documents" (getenv "HOME")))
  (defvar my:d:org
    (expand-file-name "org" my:d:cloud))
  (defvar my:d:blog
    (expand-file-name "devel/repos/mysite" my:d:cloud))
  (defvar my:f:capture-blog-file
    (expand-file-name "all-posts.org" my:d:blog))

  ;; Safety: excluded paths
  (defvar my:d:excluded-directories
    (list (expand-file-name "Library/Accounts" (getenv "HOME"))))

  ;; Ensure directories exist
```

```
  (mapc #'my/ensure-directory-exists
        (list my:d:cloud my:d:org my:d:blog))

  ;; core/core-treesit
  (setq core-treesit-enable-p t)
  (setq core-treesit-auto-install-p t)

  ;; Org wiring
  (setq org-directory my:d:org)
  (setq org-roam-db-node-include-function
      (lambda ()
        (let ((file (buffer-file-name)))
          (if (null file)
              t
            (not (string-match-p "/chatgpt/" file))))))

  (setq org-agenda-files
        (when (fboundp 'utils-org-agenda-build)
          (utils-org-agenda-build)))


  ;; load-path hygiene
  (setq load-path
        (seq-remove
         (lambda (dir)
           (member dir my:d:excluded-directories))
         load-path)))

;; ------------------------------------------------------------------------
;; Personal editing preferences
;; ------------------------------------------------------------------------

(add-hook 'before-save-hook #'my/save-buffer-wrapper)

;; ------------------------------------------------------------------------
;; Core session orchestration knobs
;; ------------------------------------------------------------------------

;; Master switch
(setq core-session-enable-p t)

;; Timing overrides
(setq core-session-idle-delay
      (* 30 60))                    ;; 30 minutes idle

(setq core-session-periodic-interval
      600)                          ;; 10 minutes

;; Risk thresholds (personal tolerance)
(setq core-session-buffer-threshold
      350)

(setq core-session-process-threshold
```

```
      8)

;; ------------------------------------------------------------------------
;; UI: session health modeline knobs
;; ------------------------------------------------------------------------

(setq ui-health-show-buffers-p t)
(setq ui-health-show-processes-p t)
(setq ui-health-show-eglot-p t)

;; ------------------------------------------------------------------------
;; Optional personal safety preferences
;; ------------------------------------------------------------------------

;; Avoid font cache compaction on long-running sessions
(setq inhibit-compacting-font-caches t)

;; Cache passphrase in memory for encrypted plstore
(setq plstore-cache-passphrase-for-symmetric-encryption t)

;; ------------------------------------------------------------------------
;; Load personal optional modules
;; ------------------------------------------------------------------------

(when (eq system-type 'darwin)
  (require 'device-darwin nil t)
  (require 'apple-music nil t))

(provide 'user)
;;; personal/user.el ends here
```

2.3.3 device-darwin.el

```
;;; personal/device-darwin.el --- macOS device profile -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: personal
;;
;; Commentary:
;; macOS-specific device and input configuration.
;;
;; This module:
;; - configures macOS IME integration and cursor behavior
;; - defines mouse and scroll characteristics for a specific device profile
;; - applies device-local settings without affecting global policy
;;
;;; Code:

(eval-when-compile
  (require 'leaf))
```

```
(when (eq system-type 'darwin)

  ;; ------------------------------------------------------------------------
  ;; IME integration (sis)
  ;; ------------------------------------------------------------------------

  (leaf sis
    :straight t
    :commands (sis-ism-lazyman-config
               sis-global-cursor-color-mode
               sis-global-respect-mode
               sis-global-inline-mode)
    :hook
    (emacs-startup .
                   (lambda ()
                     (when (fboundp 'sis-ism-lazyman-config)
                       (sis-ism-lazyman-config
                        "com.apple.keylayout.ABC"
                        "com.apple.inputmethod.Kotoeri.RomajiTyping.Japanese"
                        'macism))
                     (when (fboundp 'sis-global-cursor-color-mode)
                       (sis-global-cursor-color-mode t))
                     (when (fboundp 'sis-global-respect-mode)
                       (sis-global-respect-mode t))
                     (when (fboundp 'sis-global-inline-mode)
                       (sis-global-inline-mode t)))))

  ;; ------------------------------------------------------------------------
  ;; Cursor color keep-alive after theme load
  ;; ------------------------------------------------------------------------

  (add-hook 'after-load-theme-hook
            (lambda ()
              (when (facep 'cursor)
                (let ((c (frame-parameter nil 'cursor-color)))
                  (when (stringp c)
                    (set-face-background 'cursor c))))))

  ;; ------------------------------------------------------------------------
  ;; Mouse / scroll profile (MX Ergo S)
  ;; ------------------------------------------------------------------------

  (leaf device-mx-ergo-s
    :straight nil
    :init
    (setq mouse-wheel-scroll-amount '(1 ((shift) . 5) ((control) . 10))
          mouse-wheel-progressive-speed nil
          scroll-conservatively 10000
          scroll-margin 2
          scroll-preserve-screen-position t
          mac-mouse-wheel-smooth-scroll t
          mouse-wheel-tilt-scroll t
          mouse-wheel-flip-direction nil)
```

```
    (global-set-key [mouse-2] #'yank)
    (global-set-key [mouse-4] #'previous-buffer)
    (global-set-key [mouse-5] #'next-buffer)))

(provide 'device-darwin)
;;; personal/device-darwin.el ends here
```

2.3.4 apple-music.el

Purpose: Control Apple Music from Emacs on macOS.

What it does:

- Provides async/sync AppleScript helpers
- Defines interactive playback commands
- Exposes a hydra and optional meow leader binding

Notes:

- This module is strictly optional and macOS-only.
- No core/session or utils logic is used here.

```
;;; personal/apple-music.el --- Apple Music integration -*- lexical-binding: t; -*-
;;
;; Copyright (c) 2021-2026
;; Author: YAMASHITA, Takao
;; License: GNU GPL v3 or later
;;
;; Category: personal
;;
;; Commentary:
;; Control Apple Music using AppleScript.
;;
;; This module:
;; - provides asynchronous and synchronous AppleScript helpers
;; - defines interactive commands to control Apple Music playback
;; - integrates playback control with hydra and modal key systems
;;
;;; Code:

(when (eq system-type 'darwin)

  ;; ---------------------------------------------------------------------
  ;; AppleScript helpers
  ;; ---------------------------------------------------------------------

  (defun my/apple-music--osascript-async (script &optional callback)
    "Run AppleScript SCRIPT asynchronously.
If CALLBACK is non-nil, call it with the trimmed output."
    (let* ((proc-name "apple-music-async")
           (buffer-name "*Apple Music Async*")
           (proc (apply #'start-process
                        proc-name buffer-name
```

```
                        (list "osascript" "-e" script))))
      (when callback
        (set-process-sentinel
         proc
         (lambda (process event)
           (when (string= event "finished\n")
             (with-current-buffer (process-buffer process)
               (funcall callback
                        (string-trim (buffer-string))))
             (kill-buffer (process-buffer process)))))))))

  (defun my/apple-music--osascript-sync (script)
  "Run AppleScript SCRIPT synchronously and return trimmed output."
  (with-temp-buffer
    (let ((exit (process-file "osascript" nil t nil "-e" script)))
      (if (eq exit 0)
          (string-trim (buffer-string))
        (error "osascript failed: %s" (string-trim (buffer-string)))))))

  ;; ---------------------------------------------------------------------------
  ;; Interactive commands
  ;; ---------------------------------------------------------------------------

  ;;;###autoload
  (defun my/apple-music-play-pause ()
    "Toggle play/pause in Apple Music."
    (interactive)
    (my/apple-music--osascript-async
     "tell application \"Music\" to playpause"))

  ;;;###autoload
  (defun my/apple-music-next-track ()
    "Skip to the next track in Apple Music."
    (interactive)
    (my/apple-music--osascript-async
     "tell application \"Music\" to next track"))

  ;;;###autoload
  (defun my/apple-music-previous-track ()
    "Return to the previous track in Apple Music."
    (interactive)
    (my/apple-music--osascript-async
     "tell application \"Music\" to previous track"))

  ;;;###autoload
  (defun my/apple-music-current-track-info ()
    "Display current track information."
    (interactive)
    (message "%s"
             (my/apple-music--osascript-sync
              "tell application \"Music\" \
to (get name of current track) & \" — \" & (get artist of current track) \
& \" [\" & (get album of current track) & \"]\"")))
```

```
  (defun my/apple-music-get-playlists ()
    "Return a list of playlist names."
    (split-string
     (my/apple-music--osascript-sync
      "tell application \"Music\" to get name of playlists")
     ", "))

  ;;;###autoload
  (defun my/apple-music-play-playlist (playlist)
    "Prompt for PLAYLIST and play it."
    (interactive
     (list (completing-read
            "Playlist: "
            (my/apple-music-get-playlists))))
    (my/apple-music--osascript-async
     (format "tell application \"Music\" to play playlist \"%s\""
             (replace-regexp-in-string "\"" "\\\\\"" playlist))))

  ;; ---------------------------------------------------------------------------
  ;; Hydra / meow integration
  ;; ---------------------------------------------------------------------------

  (with-eval-after-load 'hydra
    (defhydra my/hydra-apple-music (:hint nil)
      "
Apple Music
-----------
_p_: Play/Pause   _n_: Next   _b_: Back
_l_: Playlist     _i_: Info   _q_: Quit
"
      ("p" my/apple-music-play-pause)
      ("n" my/apple-music-next-track)
      ("b" my/apple-music-previous-track)
      ("l" my/apple-music-play-playlist)
      ("i" my/apple-music-current-track-info)
      ("q" nil "quit"))
    (global-set-key (kbd "C-c M") #'my/hydra-apple-music/body))

  (with-eval-after-load 'meow
    (with-eval-after-load 'hydra
      (when (fboundp 'meow-leader-define-key)
        (meow-leader-define-key
         '("M" . my/hydra-apple-music/body)))))

(provide 'apple-music)
;;; personal/apple-music.el ends here
```

```
(provide 'README)
;;; README.org ends here
```