

IziBuddy — The Learning Path

IziBuddy — The Learning Path

From first prompt to power user

InZeMobile

TABLE DES MATIÈRES

Partie 1 — Day one (Simple preset territory)

Chapter I — Your first prompt	9
Chapter II — Undo and recover	14
Chapter III — The Stash	17
Chapter IV — The extension in ten minutes	20

Partie 2 — Week one — Organizing as your library grows (Standard preset)

Chapter V — Markers	27
Chapter VI — Folders.....	31
Chapter VII — Tagging	35
Chapter VIII — Filtering and searching.....	39
Chapter IX — Split-pane editing	43
Chapter X — Versioning	47
Chapter XI — Milestone — from Simple to Standard	51

Partie 3 — Month one — Engineering reusable templates (POO²L L1-L3)

Chapter XII — Sections (L1)	57
Chapter XIII — Parameters (L3)	63
Chapter XIV — Constants	68
Chapter XV — Includes	72
Chapter XVI — Attachments	77
Chapter XVII — Parameter sets	81
Chapter XVIII — Rendering, system variables, and injection....	85

Partie 4 — Quarter one — Object-Oriented templates (POO²L L4-L5, Advanced preset)

Chapter XIX — Inheritance (L4)	93
--------------------------------------	----

Chapter XX — `parent` and `self`	98
Chapter XXI — From folder to collection.....	103
Chapter XXII — Virtual templates	108
Chapter XXIII — Collection inheritance (the second OO axis)	114
Chapter XXIV — Aliases.....	118
Chapter XXV — Deprecation.....	121
 Partie 5 — Year one — Mastery and teamwork (Expert preset)	
Chapter XXVI — Diff viewer and rollback workflows	129
Chapter XXVII — Tag administration.....	134
Chapter XXVIII — Groups and shared templates	139
Chapter XXIX — Invitations and onboarding teammates	144
Chapter XXX — Git sync	149
Chapter XXXI — The lab — tests, scoring, experiments.....	155
Chapter XXXII — Discovering Expert mode	159
Chapter XXXIII — Your maintenance routine	164
 Appendices	
Appendix A — Appendix A — Exercise solutions	171
Appendix B — Appendix B — A 30-day plan	179
Appendix C — Appendix C — Reverse index — feature → chapter	184
Appendix D — Appendix D — Crosswalk — tips.json triggers → chapter.....	190
Resources.....	194
Lexique	196
Index thématique.....	205
Index alphabétique	209

Partie 1

*Day one (Simple preset
territory)*

Chapter I — Your first prompt

Welcome. This first chapter walks you from « I have an IziBuddy account and the extension is installed » to « I just sent my first prompt to a chatbot through IziBuddy ». That round trip is the only thing that matters today ; the deeper machinery — folders, tags, parameters, inheritance — can wait until you have felt the basic loop work.

1.1 — Goal

By the end of this chapter you will have written one prompt in the dashboard and injected it into a chat tab through the browser extension. Five minutes of clicking, no jargon.

1.2 — Prerequisites

- An IziBuddy account (the dashboard is open in a tab).
- The browser extension installed and paired with your dashboard.
- One supported chatbot — Claude, ChatGPT, Grok, Gemini or DeepSeek — open in another tab.

If the extension is not yet installed or paired, do that first ; the browser extension is the bridge that takes a prompt from the dashboard and puts it in front of the chatbot, and there is no useful exercise without it.

1.3 — Walkthrough

Open the dashboard. Go to the IziBuddy dashboard in your browser. You land on the list of templates. For a new account the list is empty, which is exactly what we are about to fix.

Click *New prompt*. The button is in the top-left sidebar. The editor opens with two visible fields : a *Name* and a *Content* area.

Name it. Type a short, human-readable handle — for example `Code reviewer`. Naming the prompt first is a habit worth picking up on day one; an unnamed prompt is hard to find in two weeks.

Write the body. Click into the content area and write the actual instructions you want to give the chatbot. For this first prompt, keep it plain prose — no special syntax:

```
You are an expert software code reviewer.  
Review the code I paste below for correctness, performance and  
readability. List the issues in three bullet points and propose  
a concrete fix for each.
```

Save. Click *Save* (or use Ctrl+S). The prompt now appears in the sidebar list. It is yours, stored on the server, ready for use.

Open the chatbot tab. Switch to the ChatGPT (or Claude, or any supported chatbot) tab you have already opened.

Open the IziBuddy floating panel. Look for the IziBuddy launcher icon — a small floating button at the corner of the page. Click it. The floating panel expands and shows the *Templates* tab.

Find your prompt and inject it. Search for `Code reviewer` in the panel's search box, click the result, and click *Inject*. The text you wrote appears in the chatbot's editor as if you had typed it.

Add the code you want reviewed and send. Paste a short Python snippet under the injected prompt and hit the chatbot's send button. The chatbot now plays the role you defined. The loop is complete.

1.4 — Hands-on exercise

Write a 30-line *Code reviewer* prompt that:

- Tells the chatbot it is a code reviewer.
- Asks for at most three issues.
- Asks each issue to be followed by a concrete fix.

- Asks for a one-line overall verdict at the end.

Save it under the name `Code reviewer`, then inject it into the chatbot of your choice with a short Python snippet pasted underneath. Read the response. If the response is structured the way you asked, you have a working baseline you will reuse all week.

Bonus: write a second prompt called `English proofreader` for short text revisions. Two templates is the minimum library that makes the *search* box useful.

1.5 — Pitfall and how to avoid it

The most common day-one mistake is **saving without a name** — or saving with a placeholder name like `New prompt 1`. Two weeks later that prompt is impossible to find by search; you scroll the list, recognize the body, then rename. Skip the chore by naming the prompt *before* you write a single line of content.

The second-most-common mistake is **expecting the extension to appear immediately on a chatbot tab you opened before installing it**. A simple reload of the chatbot tab solves it.

1.6 — Recap and next step

You have written one prompt. You have injected it into a real chatbot. You have seen the loop work end-to-end. That is the entire point of day one.

The next chapter teaches the safety net you will rely on next: how to delete and recover. Two clicks, no drama — but worth knowing *before* you click the wrong thing.

Exercise : a skill and its unpublished revision

Create a skill with a description. Include a published reusable instruction. Save the skill as a local draft, find it with `flag:draft`, and run Preview. Change its included instruction in another local draft. Compare the normal published output with Preview's explicit local-drafts mode. Publish only the instruction, then the skill. Observe that neither UUID changed, and that the skill's previously exported file remains a separate snapshot.

Change the document type in properties and confirm that descendants retain their own type. Test both administrator policies for a mixed inheritance edge.

Templates and document nature

A **template** is the common reusable entity. Its **Nature** is `prompt`, `skill` or a custom identifier such as `mod` or `acme.review`. Create **New prompt** or **New skill**, then edit Nature in Properties when needed. Changing nature preserves the UUID, references, parameters and history. The leaf template owns its type; includes never change it. Mixed inheritance can be allowed with warnings or denied.

```
@@@template review_assistant
@@@type skill
@@@description Reusable review instructions
@@@section rules
Separate observations from assumptions.
```

Use `type:acme.review` to search a custom nature. The **Example** flag replaces the old Template flag; `flag:example` finds these starting examples. `flag:draft` continues to find private drafts. A

prompt copies the rendered text, a skill exports SKILL.md and resources, and other types export composed text without claiming a specialized package format.

Chapter II — Undo and recover

Now that you have written a prompt, the next reflex worth building is the opposite one — deleting it without fear. IziBuddy is built around a forgiving lifecycle: a deleted prompt is not gone, it is in the *Trash*, and you can pull it back with one click. This chapter is tied to the contextual tip `first-deletion-undo`, which fires the first time you delete a prompt — by the end of these few pages, that tip will only confirm what you already know.

2.1 — Goal

By the end of this chapter you will delete a prompt, observe it land in the Trash, restore it, and understand the difference between a trashed prompt and a purged one.

2.2 — Prerequisites

- Chapter I completed — you have at least one prompt in your dashboard.
- A brief understanding of the prompt lifecycle: every prompt is either *Active* (visible in the main list), *Trashed* (visible in the Trash, recoverable in one click) or *Purged* (gone for good, not recoverable). The trash is the safety net between the two.

2.3 — Walkthrough

Pick the prompt to delete. Open the dashboard. In the sidebar, right-click (or tap the three-dot menu on) the `Code reviewer` prompt you created in the previous chapter. The context menu shows *Delete* among other actions.

Delete it. Click *Delete*. The prompt vanishes from the list and a small toast appears at the bottom of the screen — typically «Template sent to trash». You will also notice a *Trash* counter has appeared somewhere in the sidebar (its exact location depends on your current preset).

Open the Trash. Click on the Trash entry. The list now shows your `Code reviewer`. Each entry shows when it was trashed, which matters once you have a few items in there.

Restore. Click the *Restore* action on the row. The prompt returns to the main list, exactly where it was, with the same content, the same name, the same markers — restoration is lossless.

Read the lifecycle. Behind those two clicks, the prompt cycled through four states: Active → Trash → Restored → Active. The middle two are bookkeeping; the dashboard hides them but they exist in the versioning history. If you open the version log later, you will see the restoration entry.

Understand *Purge*. Inside the Trash there is a second, more serious action: *Purge*. Purging removes the prompt for good — no restore button, no undo. Purge is appropriate when you are confident the prompt is dead: never used, replaced by a successor, or genuinely a mistake. The rule of thumb is the *week-of-remorse* rule: leave a prompt in the Trash for at least a week. If you have not missed it by then, purge it.

2.4 — Hands-on exercise

1. Delete your `Code reviewer` prompt.
2. Open the Trash and confirm it is there.
3. Restore it.
4. Now *duplicate* the restored prompt (the context menu has a *Duplicate* action). You now have two copies: the original and `Code reviewer (copy)`.
5. Trash the duplicate. Confirm the original is still in the main list, untouched.

6. Open the Trash and observe that only the duplicate is in there.

You have just rehearsed the move you will use a hundred times : try something destructive on a *duplicate*, not on the original.

2.5 — Pitfall and how to avoid it

The single most common day-two mistake is **confusing Trash with Purge** — especially when working through the Trash quickly to «clean up». A purged prompt is unrecoverable ; if you purge the wrong one, even support cannot restore it from the dashboard. Two defenses :

- Read the action label before clicking. *Delete* puts a prompt in the Trash ; *Purge* removes it permanently. The two confirmations are worded differently — IziBuddy phrases *Purge* with stronger language on purpose.
- Apply the week-of-remorse rule : when in doubt, leave it in the Trash and revisit next week. Disk space is cheap ; lost work is expensive.

A secondary pitfall is **assuming the Trash is empty because the counter shows zero**. The counter only updates on a refresh of the sidebar. If a colleague restored your prompt from another tab, your local counter may lag for a few seconds.

2.6 — Recap and next step

Delete is reversible. Purge is not. The Trash is a one-week holding pen, not a permanent landfill. Once those reflexes are in place, you can move fast without fear.

The next chapter introduces the *Stash*, the extension-side tray for the templates you actually use today — a different idea from the dashboard’s storage, and one that complements it nicely. See Chapter III.

Chapter III — The Stash

The dashboard is where templates *live* ; the extension is where you *use* them. Between the two sits a small, deliberately limited tray called the *Stash* — a hand-picked list of templates that follow you into the chat tab and stay visible at the top of the extension’s Templates panel. Think of it as your daily working set : not the library, just the few templates you reach for today.

3.1 — Goal

By the end of this chapter you will know how to add templates to the Stash from the extension, reorder them, and tell at a glance which templates are stashed versus merely visible.

3.2 — Prerequisites

- Chapters I and II completed.
- The browser extension installed and paired with your dashboard.
- A handful of templates in your library — three to five is enough. If you have only one, duplicate it twice ; this chapter is about *picking* among templates.
- One concept to keep in mind : stashing a prompt tags it `STASH` on the dashboard server. The stash tray is that server-side set surfaced at the top of the extension’s Templates panel — so it follows you across chat tabs and browsers, not just the current extension session.

3.3 — Walkthrough

Open the floating panel on a chat tab. Go to ChatGPT, Claude, Grok, Gemini or DeepSeek — any of the supported chatbots. Click the IziBuddy launcher icon. The panel expands ; the *Templates* tab is the default.

Locate a prompt and stash it. Each row in the Templates tab has a small star or pin icon — the exact glyph depends on your theme. Click it. The row moves up and a *Stash* section appears at the top of the panel, containing your newly stashed prompt.

Stash two more. Repeat for two more templates. The Stash section now lists three rows, and the rest of the library is shown below. You have just declared your « today » list.

Reload the chat tab. Press F5. The extension reloads with the chatbot. Open the panel again — the Stash is still there, in the same order. The Stash survives navigations and tab reloads because it is stored on the server as a `STASH` tag on each prompt.

Reorder. Grab a stashed row by its handle and drag it up or down inside the Stash section. The order persists. Place your most-used prompt at the top — you will save half a second every time you inject it, which adds up.

Inject from the Stash. Click any stashed row, then *Inject*. The text lands in the chatbot exactly as if you had injected it from the full library. The only difference is that you got to it in two clicks instead of searching.

Un-stash. Click the same star/pin icon on a stashed row. The row leaves the Stash and goes back to the main list. The prompt itself is untouched in the dashboard ; only its stash status changed.

3.4 — Hands-on exercise

1. Pick three templates you would conceivably use today.
2. Stash all three.
3. Reorder them by frequency : the one you reach for most often at the top.
4. Reload the chat tab and confirm the Stash survived.
5. Inject the top-of-stash prompt once, then un-stash it.
6. End the exercise with two templates in the Stash — your « core two » for today.

If five minutes from now you find yourself searching the panel for one of the same templates again, that is a signal to stash it.

3.5 — Pitfall and how to avoid it

The dominant pitfall with the Stash is **treating it as your filing system**. The stash tray is a small, curated *today* set — a `STASH` tag the server keeps for you — not a place to organize a library. It has no folder structure and no sub-grouping, and it is meant to stay short. If a prompt matters beyond today, give it a proper name and a folder or a tag so it is findable on its own ; the Stash is the day's working tray, not the filing cabinet.

A related pitfall : **letting the Stash grow past ten items**. The whole point of the Stash is to be small enough to read at a glance. Once it crosses ten or fifteen templates, the search box and the filters in the main library are faster. Un-stash older entries weekly.

3.6 — Recap and next step

The Stash is your « today » tray inside the extension. Add templates with one click, reorder by drag, and trust the dashboard for long-term storage. Stashed templates inject exactly like any other prompt — the Stash is just a faster path to them.

The next chapter is a quick guided tour of the rest of the extension — the three tabs, the panel position, themes, and how not to lose the panel when a chat page reloads. See Chapter IV.

Chapter IV — The extension in ten minutes

You have used the extension three times already : to inject your first prompt in Chapter I, briefly during the trash exercise in Chapter II, and to stash your daily templates in Chapter III. This chapter is the guided tour you skipped on day one. Ten minutes of clicking around the floating panel will save you hours of squinting at it later.

4.1 — Goal

By the end of this chapter you will operate the floating panel with confidence : navigate its three tabs, move it around the screen, collapse and expand it, search inside it, and recover it when it disappears after a chat-page reload.

4.2 — Prerequisites

- Chapters I to III.
- One supported chatbot tab open (Claude, ChatGPT, Grok, Gemini or DeepSeek).
- One concept to keep in mind : the floating panel is *injected* into each chatbot page by the extension. It is not a window of its own — it lives inside the page. A page reload removes it ; the launcher icon brings it back.

4.3 — Walkthrough

Find the launcher. On any supported chatbot page, look for the small circular IziBuddy launcher — by default at the bottom-right corner. If you do not see it, click the IziBuddy icon in your browser's toolbar ; the extension will re-inject the launcher.

Open the panel. Click the launcher. The floating panel expands. The header has three tabs :

- **Templates** — your library, search box, and Stash. This is the tab you spent Chapter III in.
- **Profiles** — visual presets (panel position, theme, button size, accent colour). Where you make the panel feel like *yours*.
- **Config** — settings tied to the current chatbot platform and to the connection with your dashboard.

Switch tabs. Click each tab once. They are stateless from your perspective : switching does not disturb whatever you were doing on another tab. The currently selected tab persists across panel collapses, but a chat-page reload sends you back to Templates.

Search inside Templates. Click into the search box at the top of the Templates tab and type two or three letters. The list narrows. Search is case-insensitive and matches the prompt name first, then the body. This is the fastest path to any prompt — faster than scrolling, faster than using folders.

Inject quickly. Pick any row in the filtered list and click *Inject*. The text lands in the chat editor. If you want to inject without leaving the keyboard, the Enter key on a focused row does the same.

Move the panel. Switch to the *Profiles* tab. The first option group governs the panel's anchor — bottom-right, bottom-left, top-right, top-left. Pick the corner that does not collide with the chatbot's own controls (for ChatGPT, bottom-right collides with the send button ; the top-right or bottom-left are usually better).

Pick a theme. Still in *Profiles*, browse the themes. Each one is a coordinated set of colours, fonts and button sizes. A ChatGPT-pink theme exists if you want the panel to vanish into the chatbot's own chrome ; an IziBuddy-blue theme keeps the panel distinct. The choice is aesthetic ; the function is the same.

Collapse and expand. Click the chevron at the panel's top-right corner to collapse the panel back into the launcher. The chevron flips, the panel shrinks. Click the launcher to expand again. The collapse state persists per chatbot platform : collapsing on Claude does not collapse on ChatGPT.

Lose the panel on purpose. Reload the page (F5). The panel disappears — that is normal, the page re-rendered without the extension's injection. Re-open it by clicking the launcher icon, which is always re-injected automatically. **Do not** reload the page hoping to *recover* a missing panel ; the launcher click is the only correct move.

4.4 — Hands-on exercise

1. Open the panel on your favourite chatbot.
2. Move it to the bottom-left corner via the *Profiles* tab.
3. Pick a theme that contrasts with the chatbot's own chrome — for ChatGPT, a pink or blue theme works ; for Claude, an orange one.
4. Stash one prompt (you may already have one from Chapter III).
5. Collapse the panel.
6. Reload the page ; observe that the panel disappears.
7. Click the launcher to bring it back ; confirm your stashed prompt and your theme survived.

You have just rehearsed the three moves that matter most : open via the launcher, configure via Profiles, recover via the launcher (not via reload).

4.5 — Pitfall and how to avoid it

The dominant pitfall on the extension is **expecting it to survive a chat-page reload as a window**. It does not. The floating panel is content injected into the page ; when the page reloads, the injection restarts and the panel returns to its default collapsed state. If you reload looking to *fix* a panel that is misbehaving, you simply lose

your panel state for no benefit. The right reflex when the panel looks off is to *collapse and re-expand it* via the launcher, which re-renders the panel without reloading the page.

A second, smaller pitfall is **picking a panel anchor that overlaps the chatbot's own send button**. On most chatbots the send button hides in the bottom-right. The bottom-right anchor is therefore the worst default. Top-right or bottom-left avoid the collision.

4.6 — Recap and next step

You now operate the floating panel with intent : three tabs, a search box, four anchor positions, a theme palette, a collapse chevron, and the launcher as the universal « bring me back » button. The dashboard and the extension are now both under your fingers.

The next chapter starts Part II of the book — *Week one*, where you build the lightest organization habits that will pay off as your library grows. We begin with markers : Important, Locked, and WIP.

Partie 2

*Week one — Organizing as
your library grows (Standard
preset)*

Chapter V — Markers

Welcome to Part II. Your library is starting to grow, and with growth comes the need for *signal* — small, fast labels that tell you at a glance which templates are precious, which are frozen, and which are half-finished. This chapter introduces the three markers IziBuddy ships out of the box: **Important**, **Locked** and **WIP**. They are the lightest organizational tool in the application and the single best return on five minutes of investment.

5.1 — Goal

By the end of this chapter you will adopt three habits: flag what matters as *Important*, freeze what is final as *Locked*, and label what is in flight as *WIP*. You will also switch from the *Simple* preset to the *Standard* preset, which is where the matching filter chips become visible.

5.2 — Prerequisites

- Part I (Chapters I to IV).
- A library with at least ten templates. If you have fewer, duplicate some of your existing templates so the exercise feels real.
- One concept to keep in mind: an IziBuddy *preset* is a saved set of UI features. Switching from Simple to Standard does not change any of your templates; it only reveals controls — filter chips, buttons, columns — that were hidden. You can revert at any time without consequence.

5.3 — Walkthrough

Switch to the Standard preset. Open the dashboard’s Options page (the gear icon in the sidebar). Find the *Features* panel at the top. The first row shows four pills — *Simple*, *Standard*, *Advanced*, *Expert*. Click *Standard*. The page redraws; new chips appear in the filter bar, and new badges appear on the prompt list.

Spot the new chips. Look at the top of the prompt list. Three new filter chips are visible: *Important*, *Locked*, *WIP*. They are the user-facing handles of the three markers. They start inactive (greyed out); clicking each one filters the list to templates that carry that marker.

Mark a prompt as Important. Pick the prompt that matters most to you in your current library — your *Code reviewer* is a fair candidate. Open it (or right-click its row), and toggle *Important*. The row shows a small star (or flame, depending on theme), and the *Important* filter chip’s counter increments to **1**.

Mark a prompt as Locked. Pick a prompt you consider *final* — one where you want a clear barrier between «good enough» and «someone edits it by accident». Toggle *Locked*. The editor now shows a lock icon; trying to edit the body will display a confirmation prompt. Locking does not prevent injection — it only prevents *casual* editing.

Mark a prompt as WIP. Pick a prompt you are still drafting. Toggle *WIP*. A small «WIP» badge appears next to its name. The filter chip lets you quickly see all the templates that are not yet ready for prime time.

Combine the markers. A prompt can carry several markers at once. A prompt can be both *Important* and *WIP* — the most common combination during active development. *Important* + *Locked* marks the cornerstones of your library: the templates you rely on every day and never want to risk damaging.

Use the filter chips. Click *Important*. The list narrows to your important templates. Click it again to clear. Click *WIP*: you see only what is in flight. This is your daily standup view — what am I working on, what is precious, what is final.

5.4 — Hands-on exercise

Apply the three markers to your real library. Aim for this rough distribution :

- *Important* : 10-20% of your templates — the ones you reach for daily. Too many *Important* templates and the marker stops being a signal.
- *Locked* : 5-10% — only the templates you have a strong reason not to touch. A locked prompt should justify the lock with a sentence in its description.
- *WIP* : as many as you have in flight, but commit to *un-WIP* a prompt the day it is done.

Then, click each of the three filter chips in turn and confirm the counts match your intuition. If *Important* shows 40% of your library, you are over-marking. If *WIP* shows zero, either you have finished everything (congratulations) or you forgot to mark a draft.

5.5 — Pitfall and how to avoid it

The dominant pitfall with markers is **locking a prompt that still needs edits**. *Locked* is a barrier, not a tag — every edit will ask for confirmation. The right marker for « I care about this » is *Important*. Reserve *Locked* for « this is finished, do not let anyone (including future me) edit it without a deliberate gesture ».

A second pitfall is **over-using *Important***. If 60% of your templates are important, none of them are. Restrict *Important* to the templates you would rebuild from memory if the database were wiped. That is usually a small list.

A third, smaller pitfall is **leaving *WIP* on a finished prompt**. The filter chip then loses its meaning. A weekly five-minute review of the *WIP* list keeps it honest.

5.6 — Recap and next step

You have switched to the Standard preset, adopted three markers, and seen the filter chips that make them useful. *Important* says «this matters», *Locked* says «do not touch», *WIP* says «still cooking». These three habits scale to libraries of any size and cost nothing.

The next chapter introduces *folders* — the first explicit organizational axis. Markers are *flat*; folders are *hierarchical*. Together they cover most of what a single user needs.

Chapter VI — Folders

A library of fifteen templates fits in a single flat list. A library of fifty does not. *Folders* are the first structural answer to growth: named buckets you create, drop templates into, and rearrange as your thinking evolves. This chapter is tied to the contextual tip `promote-folders-50`, which fires when your library reaches fifty templates and you still operate without folders — by reading these pages first, you get to organize on your own terms before the dashboard nudges you.

6.1 — Goal

By the end of this chapter you will create folders, move templates into them by drag-and-drop, nest folders sparingly, and design a small folder taxonomy that fits your real work.

6.2 — Prerequisites

- Chapter V completed; you are on the Standard preset.
- A library with at least twenty templates — folders only earn their keep beyond that point.
- One concept to keep in mind: a *folder* in IziBuddy is a *container* — a parent for templates and for other folders. It has a name, a position in the tree, and nothing else. It is not a collection (that comes much later, in Chapter XXI) and it carries no inheritance.

6.3 — Walkthrough

Open the sidebar's tree view. The dashboard sidebar in Standard mode shows a tree of containers under the prompt list. If the tree is empty, you only have *Root* — the implicit top-level folder.

Create your first folder. Click *New folder* (or right-click in the tree and choose *New folder*). A row appears with an editable name. Type something concrete from your real work — `Code review`, `Email drafts`, `Meeting prep`. Press Enter.

Move a prompt into it. Grab any prompt by its drag handle in the main list and drop it onto the folder you just created. The prompt moves; the folder now shows a count of `1`. Click the folder to filter the list to its contents.

Move several templates at once. Hold Ctrl (Cmd on macOS) and click two more templates to multi-select. Drag the selection onto the folder. Three templates now sit inside.

Rename a folder. Right-click the folder and choose *Rename*. Type a clearer name. The rename is instant; references and filters follow automatically. Renaming a folder is *cheap* — feel free to iterate.

Nest one level deep. Inside `Code review`, create a child folder called `Python`. Move two of your code-reviewer templates into it. The tree now has three levels: *Root* → `Code review` → `Python`. Two levels is the sweet spot; three is the comfortable limit; four is where things start getting forgotten.

Click the breadcrumb. When you are inside a nested folder, the top of the main list shows a breadcrumb (e.g. `Code review / Python`). Click any segment to jump back up. The breadcrumb is the single tool you need to recover from «where am I in the tree».

Empty a folder and observe. Drag one of the Python templates back out to `Code review`. The Python folder is now empty but still exists. Empty folders are not removed automatically — they are placeholders for future content. Right-click and *Delete folder* when you really want them gone.

6.4 — Hands-on exercise

Design a four-folder taxonomy that fits *your* real work — not a generic example. Stay shallow :

1. List on paper the four broadest topics your templates cover. For most users this looks like : one *Code* folder, one *Writing* folder, one *Personal* folder, one *Sandbox* folder.
2. Create the four folders in the dashboard.
3. Move every existing prompt into exactly one folder. If a prompt refuses to fit, leave it at the root for now — the taxonomy is wrong and forcing it is worse than not.
4. Inside one of the folders, create one — and only one — child folder for the most populated subtopic.
5. Spend five minutes browsing the result. If finding any prompt takes more than two clicks, the taxonomy is too deep or too narrow.

6.5 — Pitfall and how to avoid it

The dominant pitfall with folders is **deep nesting nobody remembers**. After the second level, the human cost of finding the right folder exceeds the cost of just searching. Keep the tree at two or three levels and let *tags* (next chapter) do the rest.

A second pitfall is **mirroring an external taxonomy you do not actually use day-to-day**. If your team has a thirty-folder shared drive structure, do not import it. Build the folder taxonomy that matches your *prompt* usage, not your file usage. Templates and documents organize differently.

A third pitfall is **renaming folders avoidance**. The first folder name you pick is usually not the right one. Rename freely ; the sooner you converge on a name that feels obvious, the less you fight with the tree later.

6.6 — Recap and next step

Folders are the first explicit organizational axis : named, hierarchical, two-to-three levels deep. They are cheap to create and cheap to rearrange. The discipline is *shallowness* ; the payoff is finding any prompt in two clicks.

The next chapter introduces *tags*, the *second* organizational axis — flat, parallel, and orthogonal to folders. Together with markers and folders, they make the search bar almost optional.

Chapter VII — Tagging

Folders answer the question *where does this prompt live*. Tags answer a different question : *what kind of prompt is this*. A prompt has exactly one folder but can carry as many tags as you want, and the same tag can label templates that live in very different folders. This chapter is tied to the contextual tip `no-tags-30-templates` , which fires when a library reaches thirty templates and zero tags are in use.

7.1 — Goal

By the end of this chapter you will create a small tag taxonomy that is *orthogonal* to your folder taxonomy, apply tags to existing templates, and filter the list by clicking a tag.

7.2 — Prerequisites

- Chapter VI completed ; you have a folder taxonomy you are happy with.
- One concept to keep in mind : a *tag* in IziBuddy is a flat label attached to a prompt. A prompt can have many tags ; a tag can label many templates. Tags are stored centrally — your library shares one tag dictionary, so every spelling of « draft » must match exactly. Renaming a tag updates every prompt that uses it.

7.3 — Walkthrough

Locate the tag editor. Open any prompt and look at the meta strip at the top of the editor. Among the metadata fields, you will see *Tags* — a multi-select with a small `+` button. Click the `+`.

Create your first tag. A dropdown opens with an empty list (this is your first tag, after all). Type a short label — `marketing`, `internal`, `draft` — and press Enter. The tag is added to the prompt and registered in the library-wide tag dictionary.

Create two more tags. Repeat on two different templates with two different labels. After three tags, the dropdown starts auto-completing ; this is when the central dictionary becomes useful.

Tag a prompt with multiple tags. Open a prompt that fits more than one label — e.g. an *internal draft*. Add both tags. The meta strip now shows two pills.

Filter by tag. Click any tag pill (or open the filter bar's tag chip). The list narrows to templates carrying that tag. Click the chip a second time to clear.

Combine a tag filter with a folder filter. Click into a folder, then click a tag chip. The list now shows the intersection: templates in that folder *and* carrying that tag. Combining axes is the entire point of having two of them.

Rename a tag. Open the dashboard's tag administration view (visible in Standard mode for basic operations, or via the sidebar's tag list). Rename one of your tags. Every prompt that used the old label now shows the new one — automatically.

7.4 — Hands-on exercise

Design a tag taxonomy *orthogonal* to your folder taxonomy. The exercise has two steps.

Step 1 — pick an axis. If your folders are organized by *domain* (Code, Writing, Personal), pick tags that describe *something else*: audience, tone, criticality, or status. A common working set:

- `audience:internal`, `audience:public`, `audience:client`
- `tone:formal`, `tone:friendly`
- `wip-draft`, `ready-for-review`, `archive-candidate`

Step 2 — apply the tags. Spend ten minutes tagging twenty templates according to your taxonomy. Skip any prompt where no tag applies; that is fine — tags are *optional*. At the end, click each tag chip in turn and confirm the resulting list feels right.

If after twenty templates you find yourself wishing tags could nest (e.g. `audience.client.acme`), congratulations: you are ready for tag hierarchies, a feature available in Standard via `tagsHierarchy`. The Advanced preset enables the full tag administration covered in Chapter XXVII.

7.5 — Pitfall and how to avoid it

The dominant pitfall with tags is **duplicating folder names as tags**. If your `Code review` folder spawns a `code-review` tag, you now have two axes that say the same thing — the second one adds noise without information. The rule of thumb: tags and folders must describe *different* aspects of a prompt. If you cannot articulate the difference in one sentence, drop the tag.

A second pitfall is **spelling variants**: `draft`, `Draft`, `drafts` and `draf` are four different tags. The autocomplete in the tag input helps; respecting the existing tag list helps more. When in doubt, type the first two letters and pick from the dropdown rather than typing the whole word.

A third pitfall is **tagging everything**. A prompt with seven tags is filtered by none of them — the tags lose their discriminating power. Most templates deserve one or two tags. Three is the comfortable maximum.

7.6 — Recap and next step

You now have two organizational axes that work together: folders for *where*, tags for *what kind*. A prompt has one folder and a small set of tags. Filter by either or by both, depending on the question you are asking. The two taxonomies should be orthogonal — each one describes something the other does not.

The next chapter combines text search with the marker, tag and folder filters into a single, fast way to find any prompt in your library in under five seconds.

Chapter VIII — Filtering and searching

You now have three independent ways to narrow your library: *markers* (Chapter V), *folders* (Chapter VI), and *tags* (Chapter VII). Add a text search box on top, and you can find any prompt in your library in under five seconds. This chapter is the small skill that ties all three axes together.

8.1 — Goal

By the end of this chapter you will combine text search with marker, tag and folder filters into composite queries that match the way you actually work. You will also adopt the *Clear all* chip as a habit and bookmark three queries you will use weekly.

8.2 — Prerequisites

- Chapters V to VII completed. Your library has markers, folders, and at least a handful of tags.
- One concept to keep in mind: filters in IziBuddy *combine* with *AND*, not with *OR*. Clicking *Important* and the tag chip `internal` shows templates that are important *and* internal. The more chips you click, the narrower the list. The text search box adds yet another *AND*.

8.3 — Walkthrough

Find the filter bar. Above the main list, the dashboard shows a row of chips and a search box. From left to right: the *Clear all* chip (only visible when at least one filter is active), the marker chips (*Important*, *Locked*, *WIP*), the tag chip, the folder breadcrumb, and the search box.

Start with a text search. Type two or three letters of a prompt name. The list narrows live as you type. Search is case-insensitive and matches the prompt name first, then the body. No regex is required at this stage; in Standard mode the search is plain substring.

Add a marker. With the text filter still active, click the *Important* chip. The list narrows further: templates whose name contains your text *and* that are marked important. Watch the count next to the chip update.

Add a tag. Click a tag chip. The list narrows again. You now have three active filters: text + marker + tag.

Restrict to a folder. Click into one of your folders in the sidebar. The list now respects the folder *and* keeps all your other filters. You have just expressed a composite query: «show me the important templates tagged internal inside Code review whose name contains the letters pyt ». One line in writing, four clicks in practice.

Inspect the *Clear all* chip. With multiple filters active, the *Clear all* chip appears at the left of the bar. Click it. All filters reset, the folder remains (folder restriction is a navigation choice, not a filter, and is cleared by clicking the breadcrumb's root segment).

Try exclusion (the Advanced preset enables this). The `filtersExclude` feature — part of the Advanced preset, enableable on its own from *Options* → *Features* — lets you long-press a tag chip, or hold Shift while clicking, to flip it into *exclude* mode. The chip now shows a strike-through and the list hides every prompt carrying that tag. Exclusion is essential when you want «every prompt that is not yet ready», expressed as *exclude wip-draft*.

Reuse a filter. There is no formal «saved filter» feature, and the filter state does *not* live in the URL — the dashboard keeps your active filters in the browser's local storage, so they persist across reloads on the same machine automatically. To reuse a composite view, note the exact chips it needs (marker + tag + folder) and

reapply them ; the combination is only a few clicks. The one habit worth building is *Clear all* at the end of a session, so the next one starts clean.

8.4 — Hands-on exercise

Define three composite filters you genuinely use weekly and record their chip recipes. Three practical examples to inspire yours :

1. **Daily standup view** — *WIP* marker + folder `Active projects` .
Shows everything you are mid-work on.
2. **Quality review** — *Important* marker + tag `ready-for-review` .
Shows finals worth a second look.
3. **End-of-week cleanup** — tag `archive-candidate` + folder root (so you see candidates across all projects). Shows what is due for retirement.

For each filter :

1. Apply the filter combination in the dashboard.
2. Confirm the list looks right (between five and twenty rows is the usable range ; if the list is empty, the filters are too strict).
3. Write down the recipe — which marker, which tag, which folder — under a clear name (a sticky note or a personal doc).
4. Reapply it whenever you need that view ; the combination is a few clicks.

You now have three quick, repeatable views of your library.

8.5 — Pitfall and how to avoid it

The dominant pitfall with filtering is **leaving a filter on and wondering why the list looks empty**. The symptom is always the same : you come back to the dashboard, the list shows three rows or none, and you assume something is broken. Look at the filter bar. If any chip is highlighted, click *Clear all*. The full list returns.

A useful habit : end every dashboard session with *Clear all*. That way your next session starts on a clean slate, and the sight of a narrow list always means « I deliberately filtered it », never « I forgot ».

A second pitfall is **building a recurring view on a text search that depends on specific prompt names**. If your recipe is «search `Code reviewer` » and you later rename the prompt to `Strict reviewer`, the view returns zero results. Base recurring views on *markers*, *folders* and *tags* — these are stable ; use text search only for short-lived investigations.

A third, smaller pitfall is **forgetting that a folder click is not a filter**. Folder navigation persists across filter clears. If you keep getting an empty list after clicking *Clear all*, look at the breadcrumb — you are probably inside an empty folder. Click the breadcrumb's root segment to escape.

8.6 — Recap and next step

Finding is solved. With text search, three markers, tags and folders, every prompt in your library is two to four clicks away. The *Clear all* chip is your reset button, and a written chip recipe lets you rebuild any daily view in a few clicks.

The next chapter unlocks parallel work : opening two templates side by side so you can read one while editing the other. See Chapter IX.

Chapter IX — Split-pane editing

Up to this point you have worked on one prompt at a time. That works as long as the prompt is short and self-contained, but the day you start borrowing wording from a *reference* prompt while writing a *draft* prompt, the single-pane editor becomes a juggling act. Split-pane editing puts the two templates side by side so you can read one and write the other in the same viewport.

Tied to tip: `cut-paste-burst-split` — the dashboard fires this suggestion after roughly five cut-paste operations within ten minutes. By reading this chapter first, you turn the tip into a confirmation rather than a surprise.

9.1 — Goal

By the end of this chapter you will have edited a prompt while a second prompt is open on the right, swapped focus between the two panes, and copied content from one to the other without touching the system clipboard.

9.2 — Prerequisites

- VIII — you should already find your way around the list with markers, tags, folders and search.
- You are on the Standard preset (or higher). Split-pane is the feature `viewSplitPane`, which the **Expert** preset turns on by default; to use it now while on Standard, enable that one feature from the *Options* → *Features* → *Display* panel.
- A second feature drives the keyboard moves: `actionCrossPaneF5F6`, also an Expert-preset feature you can enable on its own. It binds F5 to *copy to the other pane* and F6 to *move to the other pane*, exactly like Norton Commander or any classical file manager.

9.3 — Walkthrough

Confirm the feature is on. Open *Options* → *Features* and look for *Split pane* in the *Display* category. If the toggle is on, you will see a *Open in split* action on every prompt row.

Open the first prompt. Click the prompt you want to write in. It opens in the editor in the usual way, occupying the full width.

Open the second prompt on the right. Right-click any other prompt in the list and pick *Open in split*. The viewport divides in two. The pane you started in keeps its content; the new pane appears on the right with the second prompt.

Notice the focus indicator. One pane has an accent-colored border; that is the *focused* pane — the one your keyboard shortcuts target. Click anywhere in the other pane to move focus. The accent border follows.

Swap focus with the keyboard. Tab between editors works inside a pane; to jump *across* panes the explicit moves are F5 (copy the current selection into the other pane at the cursor) and F6 (move the selection across — it leaves the source pane). These two shortcuts are the heart of split-pane work; they let you transplant a paragraph without ever loading it into the OS clipboard, which means you cannot accidentally paste it into a chat window.

Resize the divider. Drag the vertical bar between the two panes if one side needs more room. The split ratio is remembered for the session.

Close the split. Click the *Close right pane* icon in the right pane's header, or pick *Single pane* from the layout menu. The right side disappears and the left pane resumes full width.

9.4 — Hands-on exercise

Pick two templates you already own : one *reference* prompt rich in wording you want to reuse, and one *draft* prompt where you are authoring something new.

1. Open the draft in the main pane.
2. Open the reference in the right pane via *Open in split*.
3. Select one paragraph in the reference and press F5. The paragraph appears at your cursor in the draft. The clipboard is untouched.
4. Select a second paragraph and press F6 this time — it moves rather than copies. The reference loses that paragraph ; verify and then *undo* (Ctrl+Z) inside the reference pane to restore it. F6 is a real edit ; it is reversible exactly like any other edit.
5. Save the draft and close the split.

You should now feel why split-pane is faster than cut-paste even for two paragraphs : no clipboard contamination, no tab switching, no risk of pasting the reference into a chatbot by mistake.

9.5 — Pitfall and how to avoid it

The single most common confusion is **acting on the wrong pane**. Two templates visible at once means twice the surface area for typos, and a keystroke meant for the draft can end up rewriting the reference. The defense is muscle memory around the accent border : a fast glance at the colored frame *before* you start typing or pressing F5/F6. If the border is not on the pane you expected, click once inside the correct pane to move focus.

The second pitfall is **forgetting that F6 mutates the source pane**. F5 is a safe copy ; F6 is a move. If you are in any doubt, use F5, read the result in the destination, and then delete from the source manually — that explicit two-step is what F6 collapses into one shortcut.

9.6 — Recap and next step

Split-pane turns two templates into one workspace, F5/F6 cross-pane gives you a clipboard-free way to shuttle text, and the accent border tells you which side your keystrokes hit. You now have a parallel work surface.

The next chapter introduces versioning — once you start refactoring templates in earnest, you will want a way to mark « this version was good » so a future edit can be undone without panic.

Chapter X — Versioning

A prompt that you actually use evolves. You tighten a sentence, you add a constraint, you swap a verb. Most of these edits are forgettable. A few are landmarks — the moment the prompt finally produced the response you wanted, the version your team agreed to share, the configuration you froze before an important demo. Versioning is the dashboard feature that lets you tell the forgettable apart from the landmark, and recover the landmark months later.

Tied to tip: `first-version-saved` — the dashboard fires this gentle congratulation the first time a single prompt accumulates two versions. The tip exists because the second version is the moment versioning becomes interesting; with one version there is nothing to compare against.

10.1 — Goal

By the end of this chapter you will have saved a new version of a prompt on purpose, opened the version history, previewed a diff between two versions, and rolled back to an earlier one.

10.2 — Prerequisites

- IX — versioning is independent of split-pane, but in practice you will trigger versions while doing serious refactoring, which is where split-pane shines.
- Standard preset enabled (versioning UI is part of Standard).
- One prompt with enough body to make edits visible. The *Code reviewer* prompt from I is a good candidate.

10.3 — Walkthrough

Edit a prompt. Open your *Code reviewer* prompt and change one sentence — for example, add «Limit your answer to three bullet points.» somewhere in the body. Save. By default, the dashboard *may* create a version automatically on save, depending on your settings; either way, the *History* button at the top of the editor now has something to show.

Force a new version explicitly. Click *Save as new version* (or the equivalent menu entry). This is the gesture you use when an edit is significant enough that you want it preserved no matter what your autosave policy is. A short note field accompanies the action — type one line like «limited to three bullets» so the future-you can recognize the version at a glance.

Open the version history. Click *History*. A panel lists every version, most recent first. Each entry shows the timestamp, the note you wrote, and the size delta. The currently active version is highlighted.

Preview a diff. Click any older version. The panel shows the side-by-side diff between that version and the current one — old on the left, new on the right, added spans in green, removed spans in red. Read it; this is exactly the change you made.

Rollback. Decide the older version is what you want again and click *Rollback to this version*. The dashboard templates for confirmation, then makes the older content current. The version you were on before the rollback is preserved in the history — rolling back does not throw work away, it merely demotes it.

Add a deliberate «before refactor» version. Any time you are about to do a big rewrite, do this once: open *Save as new version* and type a note like «before refactor — known-good baseline». You just earned yourself a guaranteed return point.

10.4 — Hands-on exercise

On a prompt of your choice, produce three versions that mean something :

1. **v1 — baseline** : your current content, with the note « baseline 2026-01-15 » (use today's date).
2. **v2 — tighter** : remove all filler words ; aim for half the line count. Save as new version with the note « tightened ».
3. **v3 — restructured** : reorganize the body into three numbered steps. Save as new version with the note « three-step restructure ».

Now open the history, diff v2 against v1 to see the deletions, diff v3 against v2 to see the structural change. Finally, *rollback to v2*. Notice that v3 is still in the history — you can come back to it later.

10.5 — Pitfall and how to avoid it

The classic mistake is **versioning every typo**. After a week of that, the history panel has 130 entries with notes like « fix », « fix2 », « fix the fix », and finding the actual landmark is impossible. Versions are landmarks, not autosaves. Two rules of thumb keep the history readable :

1. If the edit is something a future-you will care about, save a version *and write a note*. No note means no version.
2. If you are doing exploratory edits, save normally (autosave handles it). Promote to a version only when the exploration converges.

The second pitfall is **rolling back without first snapshotting the current state**. If the current state is somewhere you might want to return to, save it as a version *before* you rollback. The history keeps both, and you remain free.

10.6 — Recap and next step

Versioning gives you the audit trail and the safety net of any serious authoring tool : explicit landmarks, a readable diff, a reversible roll-back. Use it with restraint and the history stays useful for years.

The next chapter is a milestone check — you have now seen everything the Standard preset has to offer, and it is time to confirm you are ready to commit to it.

Chapter XI — Milestone — from Simple to Standard

You crossed the line. Without quite noticing, the markers, folders, tags, filters, split-pane and versioning chapters have already pushed you well beyond what the Simple preset was designed for. This chapter is the formal handshake : you stop *behaving* like a Standard user by accident and you *become* one on purpose, by flipping the preset selector once and then living with the new UI for a full week.

Tied to tip: `simple-preset-many-templates` — the dashboard fires this nudge when a Simple-preset user crosses 80 templates. If you read this chapter before that count is reached, the tip will feel like a confirmation ; if it already fired, the chapter explains what the tip suggested.

11.1 — Goal

By the end of this chapter you will have opened the Features panel, moved the preset selector from *Simple* to *Standard*, walked through the UI elements that just appeared, and committed to staying on Standard for at least seven days so muscle memory has time to settle.

11.2 — Prerequisites

- V through X — six chapters that together cover everything the Standard preset adds. If a topic in this list still feels fuzzy, go back to its chapter *before* switching preset.
- A library of at least a few dozen templates. The Standard UI is designed for that volume ; on an empty library the new chips look decorative.
- Five quiet minutes. Switching preset rearranges parts of the sidebar and adds chips ; you want to be able to look around without interruption.

A reminder on what a *preset* is, so you do not have to leave the book: a preset is a named bundle of feature toggles. The Features panel lists every UI capability one by one (filters, view modes, container types, editor extras...), and a preset switches a known subset of them on. You can still hand-tune individual features after picking a preset — the preset is a starting point, not a straitjacket.

11.3 — Walkthrough

Open the Features panel. Click your avatar or the settings icon, then *Options*. The Features panel is the central section of that page. The preset selector sits at the top; the four buttons are *Simple*, *Standard*, *Advanced*, *Expert*. Your current preset is highlighted.

Read the preset summary. Underneath the selector, the page shows a short summary of what *Standard* turns on compared to your current preset. Read it. Nothing surprising should remain — every item in that list is something this book already taught you.

Click *Standard*. The page updates instantly: new chips appear in the prompt list (Important, Locked, WIP filters), the view-mode options expand (Details, Tile sizes, Scoring), the *New folder* button surfaces in the sidebar (alongside *New collection*, which was already there in Simple), and the editor gains the *Notes* and *Attachments* side panels. The body's CSS classes flip without a reload.

Tour each new element once. Hover every new chip and panel. The filter chips at the top of the list now include Important, Locked, WIP, Note, plus the container chips (Folder, Collection, Template) that let you narrow what the list shows. The view switcher above the list now offers Tile sizes and the *Details* density toggle. In the editor, the *Notes* panel is for prose annotations that travel with the prompt; *Attachments* covers files (you will use them properly in XVI). The cut-paste and rubber-band actions become available, and the scoring column appears in tile views.

Open one prompt and look at the editor frame. The *History* button reveals the version timeline (X), and the right-side sidebar shows the Notes and Attachments tabs. (Split-pane editing from IX is an Expert-preset feature; if you enabled `viewSplitPane` individually, its toolbar button shows up here too — Standard alone does not add it.)

Do not touch individual feature toggles yet. The temptation, on day one of Standard, is to scroll down to the granular feature list and start flipping things off because «I do not use that». Resist for a week. The point of the preset is to learn what the designers thought goes together; you cherry-pick *afterwards*, informed by use.

11.4 — Hands-on exercise

Spend an uninterrupted ten minutes on a tour. Set a timer.

1. **List page (3 min)**: open every new filter chip in turn, read what each one shows on your real templates, then click *Clear all*. Repeat in tile and table views — note how dense Tile sizes make the same list feel different.
2. **Editor (3 min)**: open a non-trivial prompt; open and close the Notes panel; open and close the Attachments panel; click *History* and read the timeline.
3. **Sidebar (2 min)**: open *New folder* and create a sandbox folder; drop two templates into it; rename; delete the folder (the templates return to the root).
4. **Commit (2 min)**: pick three things you saw that you do not yet use and write them down. They will become the targets of your next week of practice.

When the ten minutes are up, close the Features panel and go back to your normal workflow. Use Standard the same way you used Simple — write templates, organize them, inject them — but pay attention to which new affordances you naturally reach for.

11.5 — Pitfall and how to avoid it

The single most common mistake at this milestone is **bouncing between presets**. The reader switches to Standard, finds it « noisy » after a few hours, flips back to Simple, then forward to Standard the next morning because they remembered the Important chip is useful. After a week of bouncing, no muscle memory exists for either layout and the Features panel feels broken.

The defense is simple : **commit to Standard for at least seven calendar days** before you change anything. If a specific feature genuinely bothers you, turn *that one* off in the granular list — do not roll the whole preset back. The granular toggles exist for exactly this case.

A secondary pitfall is **expecting the dashboard to teach you the new UI**. It will not ; that is what this book is for. If a chip or panel surprises you and you do not remember the chapter, the chapter still exists — go back and re-read it before you give up on the feature.

11.6 — Recap and next step

You moved from Simple to Standard intentionally, walked the new UI in ten minutes, and committed to a week of practice. The granular feature list is now your fine-tuning tool — you reach for it only when a specific behavior bothers you, not to flip presets.

Part III begins next. You have organized templates as flat texts up to now ; from the next chapter on you will start *engineering* them — breaking prose into named sections so the same prompt can grow without becoming a wall of text.

Partie 3

*Month one — Engineering
reusable templates (POO²L
L1-L3)*

Chapter XII — Sections (L1)

A prompt that grows past a screen of text becomes hard to maintain for the same reason any long document does: there is no map. POO²L's *section* construct gives you that map. You wrap each logical chunk of the prompt in a named section, and from then on both you and the renderer treat the prompt as a small, ordered list of chunks rather than a single wall of prose. This is POO²L Level 1, the first step out of the « monolith » style.

12.1 — Goal

By the end of this chapter you will have taken a prompt you already wrote in plain prose and rewritten it as three named sections plus a pair of dotted sub-sections, observed how the renderer preserves the declaration order, and understood why section identifiers cannot contain spaces.

12.2 — Prerequisites

- XI — you are now on the Standard preset. The editor's section helpers are not a feature: the **Structure** tab appears from ceiling level L1 onwards (*Options* → *Editor* → *Input*), whatever the preset. Sections are L1, so the default ceiling already shows them.
- One existing prompt with at least 15–20 lines of prose. The Code reviewer from I, after a few weeks of edits, is usually a good candidate.

A one-paragraph POO²L summary so you can do the walkthrough without leaving the book: a section is declared with `@@@section <name>` on its own line, and its content is everything that follows until the next declaration (or the end of the prompt). Names start with a letter and then allow letters, digits and underscores — no spaces, no hyphens — and they are case- and diacritic-insensitive

(Intro and intro name the same section). A section can carry a dotted parent.child name to declare a **sub-section** (examples.first), and a : <description> suffix is allowed. The keyword also carries an optional **presence toggle** glued to it: section? turns the section into a user-toggled, default-OFF opt-in (the renderer asks the user at instantiation whether to include it), section! is the symmetric default-ON opt-out; both ? and ! require a description because the description is the text the user reads in the include/exclude dialog. Two inheritance modifiers, written after the keyword — @@@section virtual <name> (abstract, no body, descendants must implement) and @@@section final <name> (sealed, descendants must not redeclare) — belong to the inheritance layer; you will meet them in XIX. Sections are themselves optional: a prompt with zero sections is perfectly valid — that is what every prompt you wrote so far is.

12.3 — Walkthrough

Pick the prompt and read it once. Open it in the editor and read the body top to bottom. As you read, mentally cut it into three or four chunks that each have a clear job: the role, the constraints, the format of the answer, the examples, and so on. Those mental chunks are your section candidates.

Insert the first section declaration. Place the cursor at the beginning of the first chunk and add a line above it:

```
@@@section role
```

The section's content is everything that follows, until the next declaration. A section runs until the next section declaration or the end of the prompt; an explicit @@@end-section terminator is optional — a precision tool for closing a section before a following directive, never required.

Insert the remaining sections. Repeat for each chunk. A typical code-reviewer prompt ends up with sections like :

```
@@@section role
```

```
You are an expert software code reviewer.
```

```
@@@section task
```

```
Review the code I paste below for correctness, performance and readability.
```

```
@@@section format
```

```
List at most three issues. For each issue, propose a concrete fix.  
End with a one-line overall verdict.
```

Each name is short, lowercase, and identifier-safe. Notice that the surface prose is unchanged — sections are *labels* on existing text, not transformations of it. The renderer preserves the declaration order verbatim, and titles each section with its name: `@@@section role` renders as `## Role` above its own prose — and so does `@@@section role : Rôle attendu`, because the description after the colon is dialog prose, never a title. Naming a section is therefore not just an editing convenience: the name reaches the model as its heading.

Add a sub-section. Sub-sections group related material under a parent section, addressed by a dotted `parent.child` name. Add at the end:

```
@@@section examples.first : First example, a simple case  
Snippet A: ...
```

```
@@@section examples.second : Second example, an edge case  
Snippet B: ...
```

The dotted names make `first` and `second` sub-sections of `examples`; the description after `:` is human-readable text the dashboard surfaces in the section navigator; the rendered output titles each sub-section by its name (`First`, `Second`). Descriptions are *not* enclosed in quotes — that quoting rule applies to parameter and constant labels, not to section descriptions.

Render and compare. Click *Render*. The output shows the resolved content, each section under its own markdown heading — the same prose, now with its structure spelled out. Every section is titled by its name — the two sub-sections carry a description, and it appears nowhere in the output:

```
## Role
```

```
You are an expert software code reviewer.
```

```
## Task
```

```
Review the code I paste below for correctness, performance and readability.
```

```
## Format
```

```
List at most three issues. For each issue, propose a concrete fix. End with a one-line overall verdict.
```

```
## Examples
```

```
### 1. First
```

```
Snippet A: ...
```

```
### 2. Second
```

```
Snippet B: ...
```

Sub-sections are numbered in render order, which is what makes an item's rank distinguishable from a numbered list written inside it. One consequence to note now: `#`, `##` and `###` belong to the renderer inside a section, so — now that the prompt is cut into sections — typing one of them in a section's prose is a syntax error: make that line a section of its own, or use `####` or deeper, or a `@@@verbatim` block, when you need a literal heading. Before you sectioned it, the same prose could carry them freely; nothing was

emitted to collide with them. The editor’s section helpers light up : a navigator on the side, click-to-jump on declarations, and a `DUPLICATE_DECLARATION` error if you reuse the same section name twice.

Save. If you want a checkpoint, save as a new version with the note « L1 — sectioned » — a clean return point if you ever decide the prompt was easier to read as prose.

12.4 — Hands-on exercise

Take a real « monolithic » prompt from your library — ideally one you have edited at least three times — and refactor it into a five-section version.

1. Read the prompt once and list five chunks on paper : most commonly *role*, *task*, *constraints*, *format*, *examples*.
2. Declare each section. Names are short, lowercase, no spaces.
3. If *examples* has distinct cases, split them into sub-sections (`examples.first` , `examples.second`).
4. Render and compare the output to the original — verify nothing was lost in translation.
5. Re-read the source. The map should now be visible at a glance : you can describe the prompt’s structure to a colleague in ten seconds.

Bonus exercise : add one `section? notes : Optional reviewer hints` at the end. Because it is optional, the description is mandatory. At injection time the dashboard will offer to include or skip it.

12.5 — Pitfall and how to avoid it

The single most common mistake is **giving sections names with spaces** : `@@@section the role`. The parser refuses it because section names share the same identifier rules as parameters and constants — they start with a letter and then allow letters, digits, and underscore only (no hyphen — the hyphen is reserved for the language’s own keywords). The dashboard’s syntax highlighter flags it red as soon as you type the space ; if you save anyway, the

renderer reports a syntax error and no section is created. Use `snake_case (the_role)` or `camelCase (theRole)` — pick one convention and stick to it.

The second pitfall is **declaring an optional section without a description**: `@@@section? notes`. This is rejected outright (`TOGGLEABLE_SECTION_WITHOUT_DESCRIPTION`); optional sections are required to carry a description so the user knows what they are toggling at injection. Add `: <description>` and you are fine.

A subtler trap is **over-sectioning**. A four-line prompt does not need five sections; sections earn their keep when the prompt exceeds a screen or when you want a clean attachment point for future includes or overrides. Below that threshold, prose is fine.

12.6 — Recap and next step

Sections turn your prose into a small ordered map: every chunk has a name, the renderer preserves order, and the editor gives you a navigator. You stayed at Level 1 — no parameters, no constants, no includes — but the structural seam is now in place for everything that follows.

The next chapter introduces parameters. Once the prompt is sectioned, the next refactor is to identify the *values* that vary between uses (a language, a tone, a maximum number of issues) and pull them out as proper inputs.

Chapter XIII — Parameters (L3)

Look at your library after a few weeks of real use and one pattern shows up everywhere: the same prompt, duplicated three, five, sometimes ten times, each copy differing by a single word — a language code, a tone, a model name, a target audience. Each duplicate is a maintenance bomb: fix a typo in one, the others keep the bug. Parameters are POO²L's answer. You declare the variable parts once, the renderer substitutes them at injection, and the duplicate copies become a single parameterized prompt.

Tied to tip: `duplicate-burst-pool2l` — the dashboard fires this suggestion after three duplications within five minutes. If that tip just appeared, you are exactly in the situation this chapter solves; if it has not, you can read this chapter first and forestall the burst entirely.

13.1 — Goal

By the end of this chapter you will have identified a prompt that exists in several near-identical copies in your library, declared the variable parts as parameters with default values and labels, deleted the redundant copies, and injected the single parameterized version twice with different parameter bindings.

13.2 — Prerequisites

- XII — your candidate prompt is already sectioned, which is what makes the variable parts obvious.
- The editing-level ceiling reaches L3 (`pool_max_level`, Options → Editor → Input). Parameters are not behind a feature toggle. This is part of the *Advanced* preset by default, not *Standard*; switch individual feature on from *Options* → *Features* → *POO²L Edition* without leaving *Standard*, or move to *Advanced* if your library is ready for it.

A one-paragraph POO²L summary so you can follow without leaving the book: a parameter is declared with `@@@param <name> : <type> = <default>, "<label>"` and used in the body with `@@@=<name>`. Types are `text`, `textarea`, `number`, `date`, `boolean`, `select`. Optional constraints (`min`, `max`, `minlen`, `maxlen`, `pattern`) sit between the type and the default. `param?` makes the parameter optional; plain `param` makes it required. The label is the string the injection dialog displays; if omitted, the dialog uses the parameter name.

13.3 — Walkthrough

Pick your most-duplicated prompt. Find the family of templates whose names differ only by a suffix: `Code reviewer FR`, `Code reviewer EN`, `Code reviewer DE`. Pick one as the *canonical* version; you will edit it and delete the others at the end.

Diff two copies side by side. Open two duplicates using IX. Identify what differs: a language code, a number, an audience term. Each difference is a parameter candidate; each commonality stays in the prose unchanged.

Declare the first parameter. At the very top of the canonical prompt (above the first section), add:

```
@@@param lang : select("en", "fr", "de") = "en", "Output language"
```

The `select` type produces a dropdown in the injection dialog limited to the three values you listed. The `= "en"` makes *English* the default; the label appears in the dialog.

Use the parameter in the body. Wherever the canonical prompt mentioned the language as a literal — say, in a *format* section that read « Answer in English. » — replace the literal with the substitution form:

```
Answer in @@@=lang.
```

Substitution happens at render time. The same prompt, rendered with `lang="fr"`, will say « Answer in fr. » Notice that the value flows in verbatim; if you need a full name rather than a code, declare the option set with the full names directly: `select("English", "French", "German")`.

Declare a numeric parameter with constraints. A common duplication is around output length. Add :

```
@@@param max_issues : number min=1 max=10 = 3, "Maximum issues to re
```

Then in the *task* section, replace any hard-coded « three issues » with `@@@=max_issues issues`. A value outside the range raises `CONSTRAINT_VIOLATION` — the engine rejects it rather than clamping — and the dialog guards against `max_issues=99` before it ever renders.

Declare an optional parameter. Use `param?` for genuinely opt-in values :

```
@@@param? tone : select("neutral", "strict", "gentle") = "neutral",
```

The injection dialog lets the user skip the binding; when skipped, the renderer uses the default.

Render and inject. Click *Render* with the defaults — the output matches the original duplicate. Inject, change `lang` to `fr` in the dialog, inject again. One prompt, two outputs.

Delete the duplicates. Send the other copies to the Trash (see II); keep them there a week before purging in case the parameterized version misses something.

13.4 — Hands-on exercise

Replace your most-duplicated prompt by a single parameterized version with three parameters :

1. **One select** for the categorical axis that varies (language, tone, audience, model).
2. **One number with constraints** for the quantitative axis (max words, max issues, max sentences). Include `min` and `max` so the dialog cannot produce nonsense.
3. **One text** for the free-form axis (project name, client name, focus topic). Add `minlen=1` so the dialog cannot inject an empty string.

Inject the result three times with three different combinations and verify each output is what the corresponding duplicate used to say. Only then send the duplicates to the Trash.

13.5 — Pitfall and how to avoid it

The most common over-correction is **making every value a parameter**, even the ones that have never varied and never will. The prompt becomes a wall of `@@@=...` substitutions, the injection dialog grows to a dozen fields, and the user resigns themselves to clicking *OK* on every default. A value belongs as a parameter only if it actually varies across uses ; values that are fixed but you want to surface as named constants belong in XIV, not as parameters.

The second pitfall is **declaring the parameter without a label**. The dialog then shows the raw identifier (`max_issues`) instead of the human phrase («Maximum issues to report»). For your own templates the difference is small ; for shared templates it is the difference between a friendly form and a developer cheat-sheet. Labels also help future-you remember what the parameter is for when you reopen the prompt six months later.

13.6 — Recap and next step

Parameters absorb the *varying* parts of a family of templates into one declaration. You replaced three near-duplicates with a single parameterized version, gave the dialog readable labels, and constrained the values that should not go wild.

The next chapter introduces constants — the right place for values that are fixed across uses but still deserve a name. Together, parameters and constants give every value in your templates a deliberate home.

Chapter XIV — Constants

Parameters absorb the values that *vary* across uses. Constants absorb the values that do *not* vary but still deserve a name — the API endpoint, the project code, the maximum line length your team agreed on. Putting them in a `@@@const` declaration keeps the prose readable, keeps the injection dialog uncluttered, and gives you one obvious place to change the value if it ever needs to change.

14.1 — Goal

By the end of this chapter you will have declared two constants in an existing prompt, observed that the injection dialog does not ask the user to fill them in, and refactored three templates to move would-be parameters into constants — keeping only the truly varying values as parameters.

14.2 — Prerequisites

- XIII — you already declare parameters fluently. Constants share their syntax shape ; the distinction is semantic.
- The editing-level ceiling reaches L3 (`pool_max_level` , Options → Editor → Input). Like parameters, constants are not behind a feature toggle ; switch it on individually from *Options* → *Features* → *POO²L Edition* if you are still on Standard.

A one-paragraph POO²L summary so you can follow without leaving the book: a constant is declared with `@@@const <NAME> : <type> = <value>, "<label>"` and used in the body with `@@@=<NAME>` — exactly the same substitution syntax as a parameter. The difference is that a constant's value is **immutable** after definition and is **not asked at injection time**: the renderer substitutes it silently. Constant names are uppercase by convention (`API_URL` ,

`TIMEOUT`), the type is required, the value is required, and the label is optional. Types match the parameter types: `text`, `textarea`, `number`, `date`, `boolean`, `select`.

14.3 — Walkthrough

Pick a prompt with a hard-coded value the user keeps re-typing. Open one of your real templates where a fixed string appears in the prose more than once: a tool name, a model name, a maximum number, a URL. The redundancy is the smell; the constant is the cure.

Declare the constant at the top. Right under any parameter declarations (and above the first section), add:

```
@@@const API_URL : text = "https://api.example.com/v1", "API endpoint"
```

The uppercase name is convention, not enforcement — the parser accepts any identifier — but it makes constants visually distinct from parameters in the source, and that visual distinction pays off when you reread the prompt later.

Substitute in the body. Anywhere the body had the literal URL, replace it with `@@@=API_URL`. Save and *Render*. The output shows the URL inlined; the injection dialog, when you click *Inject*, does not ask for `API_URL`. That is the whole point: the value is fixed, the user is spared the question.

Add a second constant of a different type. A typical second constant is a numeric tolerance or a boolean switch the prompt encodes:

```
@@@const MAX_LINES : number = 80, "Maximum line length in code review"
@@@const STRICT_MODE : boolean = true
```

Reference them in the prose: «Flag any line longer than `@@@=MAX_LINES` characters.» or, in a constraints section, «If `@@@=STRICT_MODE` is true, escalate every warning to an error.» The renderer substitutes the literal values; there is no boolean logic at the POO²L level beyond substitution.

Refactor a parameter into a constant. Open XIII's exercise prompt. Look at the three parameters you declared. For each one, ask: *does this actually vary across uses, or have I never injected it with a non-default value?* If you always inject the default, the value is a constant in disguise. Convert it:

```
- @@@param max_issues : number min=1 max=10 = 3, "Maximum issues"
+ @@@const MAX_ISSUES : number = 3, "Maximum issues"
```

Update the body substitutions from `@@@=max_issues` to `@@@=MAX_ISSUES`. Reinject; the dialog is now shorter by one field.

14.4 — Hands-on exercise

Refactor three of your templates to move three would-be parameters into constants. The selection rule is the test:

1. **Open the prompt's history** (X). Look at the last ten injections (or your memory of them, if the history does not log bindings yet).
2. **Identify every parameter that was always injected with its default.** That is the candidate.
3. **Convert it to a constant.** Rename to UPPERCASE, change the keyword from `param` to `const`, drop the constraint clauses (constants do not need range checks since the value is fixed in source), update all substitutions in the body.
4. **Reinject** with the same usage you had before. The output is identical; the dialog is shorter.

After the three refactors, count the parameters left in those templates. If a prompt has zero remaining parameters, that is fine — plenty of well-engineered templates are pure constants and prose.

14.5 — Pitfall and how to avoid it

The trap with the highest cost is **turning a team-disputed value into a constant**. The convention says «max 80 characters per line»; one teammate insists it should be 100. Whoever wins the debate hard-codes their preference as a constant and ships the prompt to the group. Two weeks later the other teammate edits the constant; two weeks after that, the first one edits it back. The constant has become a battleground.

The rule of thumb is the team test: **if two operators would disagree on the value, it is a parameter, not a constant**. Let the injection dialog ask the question; give each user their own answer. Constants are for values nobody will ever argue about — the API URL, the file extension, the company name.

A smaller pitfall is **mixing case styles**. You declare some constants in UPPERCASE and others in lowercase, and the body becomes inconsistent. Pick UPPERCASE (the convention encoded in POO²L examples and the dashboard’s syntax highlighter) and stay with it.

14.6 — Recap and next step

Constants give every *fixed* value a named home. The prose stops repeating itself, the injection dialog stays short, and your future self has a single line to edit when a value finally needs to change. Together with parameters from XIII, constants cover every value-with-a-name your templates will ever need.

The next chapter introduces includes — the right tool for sharing *fragments of prose* across many templates, not just values. After sections, parameters, constants and includes, your toolkit through L3 will be complete.

Chapter XV — Includes

You have parameters for varying values and constants for fixed values. The third axis of reuse is *prose* — the safety preamble that every customer-support prompt repeats, the glossary that every analyst prompt references, the legal disclaimer the team requires at the top of every draft. Copy-pasting that prose into each prompt works exactly as well as copy-pasting code: it works until the prose needs to change, at which point you discover twelve copies that drifted apart. Includes solve this by letting one prompt pull in the body of another at render time.

15.1 — Goal

By the end of this chapter you will have extracted a fragment of prose that exists in several of your templates into its own dedicated prompt, replaced the copies with an `include` reference, observed that editing the included prompt changes every caller's output, and seen the renderer reject a circular include with the `CIRCULAR_INCLUDE` error.

15.2 — Prerequisites

- XIV — you can already structure a prompt with sections, parameters and constants. Includes operate at the same layer as those declarations but reuse *content*, not values.
- A working library where the same paragraph appears literally in more than one prompt. If you are not sure, search your library for a phrase you know you repeat (a safety preamble, a tone instruction, a closing disclaimer); the search results will show the candidates.

A one-paragraph POO²L summary so you can do the walkthrough without leaving the book: an include is declared with `@@@{include <reference_path>}`, where `<reference_path>` is the hierarchical

path to another prompt (forward slashes separate folder levels: `common/safety_preamble`). At render time, the referenced prompt is resolved and its body spliced in at the include's position. Includes can pass parameter bindings: `@@@{include common/greeting(name="Alice", formal=true)}`; only parameters declared on the target can be passed. The `self` and `parent` keywords for section-level access are introduced in XX — you do not need them today. The renderer detects circular include chains and reports `CIRCULAR_INCLUDE` rather than looping forever.

15.3 — Walkthrough

Pick the fragment to extract. Open three of your templates that share a paragraph — for instance, a safety preamble that reads « Do not invent facts; if unsure, say so. » Copy that paragraph (just once); the three templates will lose it in a moment.

Create the new prompt for the fragment. Click *New prompt* and place it in a folder named `common` (create the folder if needed — see VI). Name the prompt `safety_preamble`. Paste the copied paragraph as the body. Save.

The fragment is now a first-class prompt with its own name, id and version history.

Reference it from a caller. Open one of the three original templates. Delete the now-duplicated paragraph from the body. In its place, write :

```
@@@include common/safety_preamble
```

The path is the prompt's location inside your library. Save and *Render*. The output is identical to what it was before the extraction — the renderer fetched `common/safety_preamble` and spliced its body in. Repeat for the other two callers.

Edit the fragment and observe the propagation. Open `common/safety_preamble` and tighten the wording. Save. Now open any of the three callers and *Render* it: the new wording appears. One edit, three updates. This is the entire reason includes exist.

Add a parameterized include. Suppose your `safety_preamble` takes an audience parameter :

```
@@@param audience : select("dev", "exec", "public") = "dev"  
@@@section preamble  
This output is intended for the @@@=audience audience.  
Do not invent facts; if unsure, say so.
```

A caller can pass the value at the include site :

```
@@@include common/safety_preamble(audience="exec")
```

Only parameters declared on the target may be passed ; passing an undeclared name raises `UNDECLARED_INCLUDE_PARAMETER`. The renderer reports it when it resolves the include, before substitution begins.

Trigger a circular include, then fix it. Add a self-reference `@@@{include common/safety_preamble}` to the preamble. Render: the renderer refuses with `CIRCULAR_INCLUDE` and the cycle path. Remove the offending line ; the prompt renders again. Cycles are detected at every depth — $A \rightarrow B \rightarrow A$, $A \rightarrow B \rightarrow C \rightarrow A$, and so on.

15.4 — Hands-on exercise

Build a glossary and include it from two consumer templates.

1. **Create `common/glossary`** . The body is a sectioned list of definitions :

`@@@section terms - SLA: Service Level Agreement. - RT0: Recovery Time Objective. - RP0: Recovery Point Objective.`

1. **Create or pick two consumer templates** that need the glossary — say, an *Incident report* prompt and a *Postmortem* prompt.
2. **Include the glossary** from each consumer with `@@@{include common/glossary}` placed in a `definitions` section at the top.
3. **Render both consumers.** Verify the glossary appears in each.
4. **Edit the glossary** to add a fourth term. Re-render both consumers. The new term appears in both without you touching them.

Bonus: protect the rename. If you ever decide to rename `common/glossary` to `common/terms`, see XXIV for the rename-without-breaking-callers move.

15.5 — Pitfall and how to avoid it

The textbook failure is the **circular include**: A pulls in B, B pulls in A. The renderer catches it and returns `CIRCULAR_INCLUDE` with the cycle path, so the worst-case behavior is a refused render — no infinite loop, no stack overflow, just an error you can read and fix. The fix is always the same: break the cycle by extracting the shared part of A and B into a *third* prompt C that neither of them re-includes back into.

The subtler pitfall is **inlining a target that you should parameterize instead**. Three templates include the same fragment but each tweaked a single word in their original copy. The naive fix is three forked fragments next to their callers. The right fix is one fragment with a parameter (a tone, an audience, a domain) and three includes that pass different bindings.

A logistical pitfall is **putting fragments outside a clearly named folder**. Adopt a `common/` (or `shared/`, `fragments/`) folder convention from day one and your library stays legible.

15.6 — Recap and next step

Includes give your prose the same reuse story that sections gave your structure and parameters gave your values: declare a fragment once, reference it from many callers, edit it in one place. The renderer enforces consistency (circular detection, undeclared parameter passing rejected), and a `common/` folder convention keeps fragments findable.

The next chapter moves from text fragments to binary payloads — attachments. Some templates need a PDF, an image, a CSV file; the attachment construct gives those files a clean place to live inside the prompt definition.

Chapter XVI — Attachments

Some templates need more than text. A «describe this picture» prompt needs a picture; a «follow the style guide» prompt needs the style guide PDF; a «review this CSV» prompt needs the CSV the user is holding in their hand. POO²L's *attachment* construct gives you a clean place to declare those payloads — fixed once at definition time, or supplied at injection — without smuggling them into the prose.

16.1 — Goal

By the end of this chapter you will have authored two templates: one with a *fixed* attachment (a file that travels with the prompt forever) and one with a *to-include* attachment (a file the user attaches at injection). You will know which binding time to pick and where attachments appear in the prompt body.

16.2 — Prerequisites

- Chapter XV: you can extract a fragment into its own prompt and reuse it from several callers. Attachments are the binary counterpart of that idea — instead of reusing a text fragment, you bind a file (text or binary) to the prompt.
- A working Standard or Advanced preset (attachments are visible from Standard upward).

Quick POO²L summary, so you do not have to leave the book: an attachment is a named file associated with a prompt, described by two orthogonal axes. The *when* axis is the binding time: *fixed* binds the file content at definition time and ships with the prompt forever; *to-include* requires the user to provide a file at injection. The *source* axis applies only to *fixed* and says where the bytes live: `file` (a path), `blob` (an inline Data URI), or `record` (an external store). A *to-include* attachment carries no source. Only *to-include*

attachments can be optional (suffix the keyword with `?`). Attachments belong at the *end* of the prompt body so the readable content stays at the top.

16.3 — Walkthrough

Open a prompt and scroll to the bottom. Attachments live below the content. The convention is not enforced by the parser — declarations may appear anywhere — but the recommended order keeps the prose readable and the heavy binary metadata out of the way.

Declare a fixed attachment. Add the following line at the end of a prompt :

```
@@@attachment manual : pdf fixed file "docs/style-guide.pdf"
```

The parts are: the attachment name (`manual`), the file type (`pdf`), the binding time (`fixed`), and the source (`file`) followed by a quoted path. The path is relative to the prompt's storage location. To embed the file inline instead, use the `blob` source with a Data URI (`@@@attachment manual : pdf fixed blob "data:application/pdf;base64,..."`); a third source, `record`, references an external store. Embedded data ships with the prompt; a path or ref is resolved at render time.

Declare a to-include attachment. Now add a second one :

```
@@@attachment data : csv to-include, "Spreadsheet to analyze"
```

The binding time is `to-include`, followed by a comma and a *description* (not a path). A `to-include` attachment carries no source — the user supplies the content at injection. The description is what the dashboard shows the user when asking them to pick a file. Because the keyword has no `?`, this attachment is *required* — no file, no injection.

Make a to-include optional. If the file may or may not be there, put the `?` *after the keyword*, not after the name:

```
@@@attachment? notes : txt to-include, "Optional reviewer notes"
```

The `?` placement is the same convention as for parameters (`@@@param?`). It signals that omitting the file is a valid state.

Inject and observe. When you inject this prompt, the dashboard templates you only for the to-include attachments (the fixed one is already bound). Required ones must be picked; optional ones may be skipped. The injection then includes the file in the chat tab in whatever way the target chatbot supports — most accept a drag-and-drop upload alongside the typed prompt.

16.4 — Hands-on exercise

Build a «describe this picture» prompt with one to-include image attachment:

```
@@@template describe_picture
@@@name Describe this picture
@@@description Asks the chatbot to describe a user-supplied image

You are an art critic. Describe the image attached to this message
in three short paragraphs: subject, composition, mood.

@@@attachment image : png to-include, "Image to describe"
```

Save it, inject it into Claude or ChatGPT, attach a real picture from your computer when the dialog asks, and read the response. The type after the colon (`png`) is a free-form hint; a narrower value nudges the file-picker filter, a broader one loosens it.

Bonus: add a *fixed* attachment that ships your art-critic style guide, so every injection of this prompt carries the same reference document alongside the user's picture.

16.5 — Pitfall and how to avoid it

The most common attachment mistake is **embedding a private file as a fixed Data URI inside a prompt you later share with a group**. The binary travels with the prompt; export, copy, or share operations propagate it. The recovery move is to open the prompt before sharing, scroll to the attachment block, and check every `fixed` entry. If the file should not leave your laptop, switch the binding to `to-include` so each user supplies their own copy at injection.

The second-most-common mistake is putting the `?` in the wrong place: `@@@attachment name? ...` looks plausible but is not the form the parser expects. The marker is on the *keyword*, not the *name*: `@@@attachment? name ...`. The validator flags this clearly, but a single-character typo two clicks away from «Inject» can still cost a minute.

16.6 — Recap and next step

Attachments give your templates a slot for binary or text payloads without polluting the prose. Fixed bindings ship with the prompt; to-include bindings ask the user at injection time; `?` makes a to-include optional.

The next chapter introduces *parameter sets* — a way to name a bundle of parameter bindings so you stop typing the same five values every time you inject your favourite prompt.

Chapter XVII — Parameter sets

The day your *Client report* prompt has six parameters, and you instantiate it three times a day for the same three clients, you notice you are typing the same six values over and over. Parameter sets are POO²L's answer to that drudgery: a *named* bundle of parameter bindings you pick once from a dropdown at injection time instead of filling the dialog by hand.

17.1 — Goal

By the end of this chapter you will have declared two parameter sets on the same prompt, picked one at injection time, and seen the parameter dialog pre-fill the values for you. You will also know the one rule that catches every first-time author: parameter sets on collection members live at the collection scope, not in the prompt body.

17.2 — Prerequisites

- Chapter XIII: you have at least one prompt with two or three real parameters. Parameter sets bind the parameters you already declared; there is nothing to bind if the prompt has none.
- Chapter XVI: not strictly required, but the same «declare at the end of the prompt» habit applies.

Quick POO²L summary, so you do not have to leave the book: a parameter set is a named collection of parameter bindings, declared as a block of `key: value` pairs between `@@@parameter-set <name>` and `@@@end-parameter-set`. A prompt may declare several. The user picks one at injection time, and its bindings populate the parameter dialog before any manual edit. *Restriction*: parameter sets in the prompt body are allowed only if

the prompt does *not* belong to a collection. Collection members carry their parameter sets at the collection level — covered later in Chapter XXI.

17.3 — Walkthrough

Pick a prompt with several parameters. A typical candidate is the «Client report» prompt with parameters `client_name`, `region`, `audience`, `tone`, `length`.

Declare a first parameter set. At the bottom of the prompt body, just above the attachments (if any), add :

```
@@@parameter-set marketing_tone
client_name: "Acme"
region: "EMEA"
audience: "executives"
tone: "warm"
length: 300
@@@end-parameter-set
```

The block is delimited by an opening tag with the set name and a matching closing tag. The set name is a single identifier — no spaces (`marketing_tone`, not `Marketing tone`). Inside, every line binds one parameter that the prompt already declares.

Declare a second one for contrast. Right below :

```
@@@parameter-set engineering_tone
client_name: "Acme"
region: "EMEA"
audience: "engineers"
tone: "precise"
length: 600
@@@end-parameter-set
```

Save the prompt.

Inject and pick a set. Open the extension, find the prompt, and click *Inject*. The parameter dialog now sports a *Parameter set* drop-down above the fields. Pick *marketing_tone* — the five fields fill in instantly. You can still tweak any value by hand; the set is a *starting point*, not a lock. Click *Inject* and watch the chatbot receive the rendered prose.

Switch sets in the dialog. Re-open the same prompt, pick *engineering_tone* this time, and confirm the dialog repopulates. That is the daily payoff: one click instead of five typed values.

17.4 — Hands-on exercise

Find a real prompt you customize the same way every time — a weekly status update, a code-review prompt aimed at a recurring repository, a customer email template aimed at a recurring customer. Identify the three or four parameters whose values are stable for that recurring usage, then declare a parameter set named after the context — a single identifier, no spaces (`project_phoenix`, `customer_acme`, `friday_standup`).

Inject the prompt twice: once by picking the set, once by filling the values by hand, and confirm the rendered output is identical.

Bonus: declare a second set for a contrasting context and switch between them in the dialog. Two sets is the minimum that makes the dropdown earn its place.

17.5 — Pitfall and how to avoid it

The pitfall the spec itself spells out is **declaring a parameter set inside a prompt that belongs to a collection**. The parser refuses, because parameter sets on collection members live at the collection scope so they can be inherited and overridden along the collection chain. If you see the error after promoting a folder to a collection (Chapter XXI), the recovery move is: cut the parameter-set blocks out of the prompt body, open the collection's settings panel, and re-declare them there.

The second-most-common mistake is **misalignment**: declaring `length: 300` in the set when the prompt declares `@@@param length : text` — the type does not match the value. The renderer raises `TYPE_MISMATCH` rather than guessing; the fix is to check that each binding’s literal type matches the parameter’s declared type (text in quotes, numbers bare, booleans `true / false`).

17.6 — Recap and next step

Parameter sets give your templates a named starting point. Two or three well-named sets on a heavily-injected prompt save more time than any other POO²L feature in this part of the book.

The next chapter pulls all of the L1-L3 mechanics together by looking at what *actually* happens when you click *Inject*: the rendering pipeline, the system variables, and the precedence between user input and parameter-set defaults.

Chapter XVIII — Rendering, system variables, and injection

You have spent five chapters teaching POO²L to your templates — sections, parameters, constants, includes, attachments, parameter sets. Every single one of them is silent text in the editor and live behaviour the moment you click *Inject*. This chapter opens the black box. It walks you through what the server actually does between your click and the text that appears in the chatbot’s input field, in the order the operations happen. Once you have seen the pipeline once, every puzzling render result you ever produce becomes diagnosable in seconds.

18.1 — Goal

By the end of this chapter you will be able to predict, from reading a prompt’s source, what the *Render preview* will show — including which value of a parameter wins, which include is expanded first, and which system variable lands where. You will also have used at least two system variables (`@@@=DATE` , `@@@=USER`) in a real prompt and observed them change between two consecutive injections.

18.2 — Prerequisites

- Chapter XVII: you know how to bind parameters in bulk and you have at least one prompt that uses parameter sets.
- All L1-L3 mechanics from Chapters XII through XVII: sections, parameters, constants, includes, attachments.

Quick conceptual summary, so you do not have to leave the book. Rendering a prompt is a pipeline of **eleven phases** that always run in the same order. In outline: the source is (1) parsed; then, for a collection member, (2) the collection context is applied and (3) prompt inheritance is resolved — with a call-contract check (phase 3.b) — so the *effective definition* is known; (4) sections are selected

according to their toggles ; then instantiation begins — (5) parameters are bound (constants are resolved at the very start of this phase, since a bound value may depend on them), (6) constant resolution is confirmed, (7) attachments are bound, (8) includes are resolved (the included prompt is rendered in turn, recursively), (9) `@@@=` substitutions are performed, (10) any assertions are evaluated, and (11) the final text is emitted. The result is what the *Render preview* shows you and what the extension sends to the chatbot. The order matters because each phase reads the output of the previous one — a parameter cannot be used to choose a constant's name, because a constant's value is a fixed literal that depends on no instantiation input.

18.3 — Walkthrough

Open a prompt that exercises several features. A good candidate is the « Client report » prompt from Chapter XVII : it has parameters, at least one constant, and you can add an include and a system variable for this exercise.

Add a constant. At the top of the prompt body declare :

```
@@@const COMPANY : text = "Acme Corp"
```

Constants are resolved first and never appear in the parameter dialog. They are how you fix a value at definition time.

Add an include. Below the constant, reference a small « safety preamble » prompt you authored in Chapter XV :

```
@@@include safety_preamble
```

Includes are resolved late in the pipeline (phase 8), after the current prompt's parameters are bound and its constants resolved. The expansion is recursive — the included prompt is rendered in turn, through its own pipeline.

Add a system variable. Drop this line near the top of the content area :

```
Report generated on @@@=DATE by @@@=USER.
```

System variables are substituted during the substitution phase, near the end of the pipeline. They are not parameters — the user is never asked for them — and they are not constants — you cannot rebind them.

Open the Render preview. In the editor, click the *Preview* button (or its keyboard shortcut). The preview panel shows the result of every phase up to and including system variable substitution, with the parameter values currently in the dialog. As you tweak a parameter, the preview re-renders in real time. The preview is the safest place to debug a prompt — it costs zero chatbot tokens.

Inject and compare. Click *Inject*. The chatbot tab receives exactly what the preview showed — same constants, same include, same parameter values, same `@@@=DATE` substituted with today's date and same `@@@=USER` substituted with your username. The extension renders **locally**: it runs the very same POO²L engine (compiled to WebAssembly) that the dashboard uses, so the injected text matches the preview without a server round-trip.

Inject again later in the day. Re-inject the same prompt two hours later. The body is identical except `@@@=DATE` may have the same value (still today) and `@@@=TIME` — if you added it — will be different. Every system variable resolves *at render time*, not at *save time*; this is what makes a prompt that says «today's date is X» stay accurate forever.

18.4 — Hands-on exercise

Pick one of your most-injected templates. Add a header line at the top of the body :

```
@@@const AUTHOR : text = "Your Name"  
Rendered on @@@=DATE at @@@=TIME by @@@=AUTHOR (@@@=USER).
```

Inject the prompt twice in the same day, separated by at least one minute. Compare the two rendered headers in the chat tab: the `AUTHOR` constant is identical because constants are render-time literals from your editor; `DATE` is identical because it is the same day; `TIME` is different by at least a minute; `USER` is identical because it is your account name.

Bonus: replace `@@@=DATE` with `@@@=WEEKDAY` and re-inject on two different days of the week to see the substitution change.

18.5 — Pitfall and how to avoid it

The single most common rendering surprise is **expecting a `{{X}}`-style placeholder to work as a POO²L variable**. Many other tools and templating docs use `{{X}}` for substitution, so it is an easy habit to carry over — but POO²L has no `{{...}}` form. Its only substitution form is `@@@=...`, and the parser does not interpret `{{X}}` as a variable at all: it treats those braces as literal text. When you copy-paste a snippet from another tool into a prompt body, you sometimes inherit the wrong syntax. The recovery move is simple: for POO²L authoring the variable form is always `@@@=...`. The render preview will show any stray `{{X}}` as literal braces in the output, which is your visible warning.

The second-most-common surprise is **expecting a parameter to be visible to a constant**. Phase ordering forbids it: constants resolve first, parameters do not exist yet. The recovery move is to turn the constant into a parameter, or to compose the value differently — typically by interpolating the parameter where you needed it instead.

18.6 — A note on assertions

One last thing can happen at the very end of a render, after every substitution is done: an **assertion** can block it. If a prompt carries `@@@assert` lines — checks like «the rendered text must contain this disclaimer» or «must stay under 500 characters» — the renderer evaluates them against the finished text just before injection. If one fails, the *Render preview* shows the failure and the injection is blocked, so a prompt that violates its own guarantees never reaches the chatbot. You do not author assertions in this beginner track; they are an authoring-side guard rail covered in the reference book's *Assertions and Scopes* chapter. For now, just know that a blocked render with an «assertion failed» message means the prompt's content broke a rule its author wrote on purpose.

18.7 — Recap and next step

The rendering pipeline has eleven phases in a fixed order — parse, collection context, inheritance (with a call-contract check at phase 3.b), section selection, parameter binding (constants hoisted to its start), constant resolution, attachment binding, include resolution, substitution, assertion evaluation, and output. The assertion pass, if any, can block the result. The *Render preview* is the trusted mirror of that pipeline and your fastest debugging tool. System variables resolve at injection time, so today's prompt is always fresh tomorrow.

The next part of the book is the big one: OO² — object-oriented templates. The next chapter introduces inheritance, one of the pipeline phases you have not yet seen in action.

Partie 4

*Quarter one — Object-
Oriented templates (POO²L
L4-L5, Advanced preset)*

Chapter XIX — Inheritance (L4)

The earlier POO²L levels gave you structure (sections, L1), reuse (includes, L2), and variation (parameters, L3). The next level changes the shape of the game. With inheritance you stop thinking of templates as files and start thinking of them as classes — a base prompt defines the skeleton, a child prompt declares «I am like the base, except...». The child writes only the *exception*; everything else is inherited. This is how a fleet of related templates stays consistent when you fix a typo or rewrite a paragraph in the base.

19.1 — Goal

By the end of this chapter you will have a three-template inheritance chain (a generic `code_reviewer`, a `reviewer_with_checklist` middle layer, and a leaf `python_reviewer` that overrides exactly one section). You will know how to declare an `inherits` relationship on the `@@@template` header, what an *override* does, and why inheritance is the right tool for *structure* and the wrong tool for *values*.

19.2 — Prerequisites

- Chapter XVIII: you understand the rendering pipeline and you can point to the *inheritance* phase on the list. This chapter populates that phase.
- A working Advanced preset (the *inherits* picker and the chain visualizer become visible from Advanced upward).

Quick conceptual summary, so you do not have to leave the book. An inheriting prompt declares one parent with an inline `inherits <id>` clause on its `@@@template` header. At render time the inheritance resolver builds the chain bottom-up (child → parent → grandparent), then walks it top-down, merging section by section. For every section name encountered, the most-derived (latest in the

chain) definition wins; sections that only the parent defines are kept as-is. The same merge rule applies to constants, parameters, and parameter-set bindings. The child never has to repeat the parent's content — the merge handles it. The rule of thumb: use inheritance when you are changing *what sections exist* or *what one section says*; use a parameter when you are changing *a value inside a section*.

19.3 — Walkthrough

Open the base prompt. Reuse your `code_reviewer` from Chapter I — or, if you have rewritten it since, any prompt with three or four named sections (`role`, `focus`, `output_format`, for example).

Create a child that inherits from it. Click *New prompt* and, as the entity-opening header, declare both its id and its parent on the same `@@@template` line:

```
@@@template reviewer_with_checklist inherits code_reviewer
@@@name Reviewer with checklist
```

That is enough. Save and open the render preview — you will see the full rendered content of `code_reviewer`. The child currently adds nothing, so it renders identically to the parent. This is the zero-override baseline.

Add a new section in the child. Below the header, add:

```
@@@section checklist
Verify, in this order:
1. The code compiles or runs.
2. The tests pass.
3. The diff is minimal.
@@@end-section
```

Save and re-preview. The rendered prompt now contains every section the parent declared *plus* the new `checklist` section. The new section took its place at the end because the child added it; if the parent had a `checklist` section already, the child's definition would have replaced it.

Override an existing section. Add this above the new section :

```
@@@section output_format
Three bullet points, each one sentence, no preamble.
@@@end-section
```

Save and re-preview. The parent's `output_format` content is gone from the output; the child's version takes its place. This is a pure override.

Build the third level. Now create `python_reviewer` with:

```
@@@template python_reviewer inherits reviewer_with_checklist
@@@name Python reviewer
@@@section focus
Pay special attention to: PEP 8 violations, idiom mismatches, and
implicit type confusion.
@@@end-section
```

Save and preview. You now see: the original `role` (from `code_reviewer`), the overridden `output_format` (from `reviewer_with_checklist`), the new `checklist` (from `reviewer_with_checklist`), and the overridden `focus` (from `python_reviewer`). Three levels, each contributing exactly the delta the author wrote.

Visualize the chain. The Advanced preset offers an inheritance chain panel that lists, for every section, where it came from. Open it and confirm the resolver agrees with the mental model above.

19.4 — Hands-on exercise

Pick a real prompt family in your library — usually two or three templates that share 80 % of their text and differ in one or two sections. Refactor them into a chain :

1. Create the base by keeping the common content of all of them.
2. Create a middle level if at least two of your real templates share a delta the base does not have.
3. Make the remaining specializations into leaves that inherit from the appropriate level and override only what differs.

Inject each leaf at the end of the exercise and verify the rendered content is identical to what the original copy-pasted templates produced. The win is invisible at injection time ; you only feel it the day a typo in the base disappears from all three leaves at once.

19.5 — Pitfall and how to avoid it

The single most common inheritance mistake is **using inheritance where a parameter would do**. If the only thing that differs between two templates is the value of a noun (« Python » vs « Rust »), the right tool is a parameter, not an `inherits` relationship. Inheritance changes *structure* (which sections exist, what they say); parameters change *values* (the words inside a section). The recovery move is to flatten the leaf back into the base and turn the diverging text into a parameter — the parent now serves both specializations.

The second-most-common mistake is **declaring a cycle** : `A inherits B`, `B inherits C`, `C inherits A`. The resolver detects the cycle and refuses to render. The remedy is to break the cycle at the weakest link — the one whose « is-a » relationship was the least convincing in the first place.

A side note for when you start writing base templates that other people will extend : a section declared `@@@section final <name>` in the base is *sealed* — no descendant may redeclare it under any

form (not the body, not the toggle, not the description). Any attempt raises `VERRIDE_OF_FINAL_SECTION`. Use it sparingly, for sections that genuinely must not drift across the chain (boilerplate, license notices, a regulatory clause); use plain sections everywhere else so descendants keep the freedom to override.

19.6 — Recap and next step

Inheritance lets a child write only the delta from its parent. The resolver walks the chain top-down, applying overrides section by section. The right call is « inheritance for structure, parameters for values » — the day you internalize that rule, your templates stop duplicating.

The next chapter introduces the two pronouns that make inheritance practical day to day: `parent` and `self`. They are how a child says « keep what the parent said, *and* add this » — instead of replacing.

Chapter XX — `parent` and `self`

Pure override has a sharp edge. The first time you redefine a section in a child to add one bullet point, you also have to re-paste every bullet point the parent already wrote, then keep both copies in sync forever. That is exactly the duplication inheritance was supposed to eliminate. POO²L offers two pronouns that fix it: `parent`, which means «the parent’s content for this name», and `self`, which means «the most specialized content for this name — including the very override I am writing». Together they let a child *extend* the parent instead of replacing it.

20.1 — Goal

By the end of this chapter you will have produced three variants of the same child section — *override*, *extend with parent*, and *reuse with self* — and you will know, from the rendered output, which one to reach for when. You will also recognize the most common mistake: forgetting `parent` and silently dropping the parent’s content.

20.2 — Prerequisites

- Chapter XIX: you have a working inheritance chain with at least one override.

Quick conceptual summary, so you do not have to leave the book. The two pronouns are **include references** along the inheritance chain — you reach them through `@@@include` (or the inflowing `@@@{include ...}`), never through the `@@@=` substitution form. `parent.<section>` resolves to *the parent’s* rendered content for that section, computed as if the current prompt did not exist; `parent.<prompt>.<section>` reaches a named ancestor in the chain; `parent` on its own resolves to the parent’s entire rendered content. `self.<section>` re-

solves to the *most specialized* version available — that is, the same answer the resolver would give when rendering the leaf. Both keywords are on the **prompt** inheritance axis; the collection axis uses the `^parent` / `^self` prefix, covered in XXIII. Includes of `parent` / `self` can appear anywhere in a section body, including inside section overrides.

20.3 — Walkthrough

Open the leaf. Reuse the `python_reviewer` prompt from Chapter XIX, the one that inherits from `reviewer_with_checklist` which itself inherits from `code_reviewer`. You will rewrite its `focus` section three different ways and compare.

Variant 1 — pure override. This is what you already wrote :

```
@@@section focus
Pay special attention to: PEP 8 violations, idiom mismatches, and
implicit type confusion.
@@@end-section
```

Open the render preview and copy the resulting `focus` text into a scratchpad. Notice that anything the parent had in its `focus` section is gone — the override replaced it wholesale.

Variant 2 — extend with parent . Edit the same section to call the parent's content first, then append your specialization :

```
@@@section focus
@@@include parent.focus

In addition, for Python specifically: PEP 8 violations, idiom
mismatches, and implicit type confusion.
@@@end-section
```

Save and re-preview. The rendered `focus` now contains the parent's text first, a blank line, and your appended paragraph. You wrote two lines; the resolver reused twelve. The day someone improves the parent's `focus`, your leaf inherits the improvement automatically.

Variant 3 — reuse with `self`. Suppose another section of the child wants to *quote* the focus text without redefining it. From a neighbour section, write :

```
@@@section reminder
Reminder to the reviewer:

@@@include self.focus

Please use the list above as the table of contents for your review.
@@@end-section
```

`@@@include self.focus` always resolves to whatever the leaf would render for `focus` — including any `parent` calls that section makes. It is the right pronoun to use when you want the final, fully-resolved text rather than a hard-coded copy.

Combine deliberately. A common pattern is to inherit the parent's section, then call `parent` at the top of the override :

```
@@@section output_format
@@@include parent.output_format
For Python, also include the line numbers next to each issue.
@@@end-section
```

The parent's contract is preserved ; the leaf simply adds the Python-specific bit at the end. This is the daily verb of object-oriented templates.

Reach a specific ancestor. When the chain is three or more levels deep, you can skip the immediate parent with `@@@include parent.code_reviewer.output_format` — that picks the `output_format` defined by `code_reviewer` rather than the one in the immediate parent. Use this rarely; it couples the leaf to a specific ancestor’s id.

20.4 — Hands-on exercise

Take the leaf prompt you refactored in Chapter XIX and produce three sibling versions of its main overridden section :

1. **Override** : replace the parent’s content entirely.
2. **Extend** : `@@@include parent.<section>` first, then append your delta.
3. **Quote** : from a neighbour section, `@@@include self.<section>` inside a sentence («The above list, repeated for emphasis: ... »).

Save each version under a temporary name, render the three previews side by side, and write one sentence each about when *override*, when *extend*, and when *quote* feels right. The point of the exercise is not memorizing the syntax — it is feeling the difference in your hands.

20.5 — Pitfall and how to avoid it

The single most common mistake with `parent` is **forgetting it and accidentally overriding the parent**. You open a section in the leaf, retype most of the parent’s content, change one line, and ship — never realizing that the parent’s section had a clause you never saw because you wrote your version from scratch. Two months later the parent’s clause changes upstream and your leaf is silently out of date.

The recovery move is the diff viewer. Open the leaf in the version panel and compare the rendered output of `python_reviewer` before and after your override. Any clause that appeared in the *before* and

not in the *after* is a candidate for `parent`. Once you see the gap, the fix is a one-line edit: prepend `@@@include parent.<section>` to the override and delete the duplicated text.

The second-most-common mistake is **self recursion**. Writing `@@@include self.focus` inside the `focus` section creates an unbounded reference — the resolver detects this and refuses to render. The fix is to use `parent.focus` instead when you need the *parent's* content from within `focus`.

20.6 — Recap and next step

`parent` is «the parent's version of this thing»; `self` is «the final, fully-resolved version of this thing». Use `parent` inside an override to *extend* rather than replace; use `self` from a neighbour section to *quote* without duplicating. Together they turn inheritance from a copy-and-rename mechanism into a real composition tool.

The next chapter moves from template-level inheritance to a bigger unit of organization: the *collection*. You will learn how to promote a folder into something closer to a class, with members, shared parameters, and a single inheritance axis of its own.

Chapter XXI — From folder to collection

A folder is a label. It groups templates by where they live in your mental file system, and the dashboard treats them as a tidy list. The day you find yourself duplicating a parameter declaration across half the folder, or repeating the same `parent` chain in every member, the folder is asking to be promoted. A *collection* is what a folder becomes when it acquires its own identity: shared parameters, shared inheritance, shared virtual members. In OO² terms, the folder is a directory; the collection is a class.

Tied to tip: `promote-collections-3-folders` (fires when you have at least three folders and start duplicating templates across them). By the time the tip surfaces in the dashboard, this chapter will already have taught you the move it suggests.

21.1 — Goal

By the end of this chapter you will have promoted one real folder to a collection, observed the new collection-level options (the parent-collection link, the member listing with virtual/final markers, collection-scope parameter sets), and you will know the smell that tells you a folder is *not yet* ready to be promoted.

21.2 — Prerequisites

- Chapter XX: you are fluent with template-level inheritance and the `parent` / `self` pronouns. Collections add a second inheritance axis on top of the prompt axis you already know.
- A folder in your library that contains at least three related templates (the bar is «they could share a parameter», not «they share text»). A folder with one or two members is not yet a collection candidate.

Quick conceptual summary, so you do not have to leave the book. A collection is a named container whose members are templates. Its `@@@collection` block declares the collection's public interface: an optional parent collection (`inherits`, collection inheritance, covered in Chapter XXIII), the listing of its public members, and optional collection-level parameter sets that every member can select. A member becomes *virtual* by carrying the `virtual` modifier on its own `@@@template` header (covered in Chapter XXII); derived collections must implement it. The collection itself does not carry constants or parameters — values shared across members come from a base prompt those members inherit, or from a collection-level parameter set. Promoting a folder to a collection is a one-way operation in spirit — nothing prevents demoting, but the act expresses an intent: «these templates belong together, and we will maintain them together».

21.3 — Walkthrough

Pick the folder. Open your library and find a folder that matches the smell: at least three templates, at least one parameter or constant that appears in two or more of them, at least one sentence of prose that is starting to drift between members. If none of your folders qualify, this is *good news* — your library is still small enough to live with copies. Stop the chapter here and come back when a folder hits the bar.

Promote. Right-click the folder (or open its context menu in the sidebar) and pick *Promote to collection*. The dialog asks for a collection id (the same naming rules as a prompt id: starts with a letter, letters/digits/underscores only, no spaces and no hyphens; case- and diacritic-insensitive). The folder's templates become the collection's members; their ids are unchanged.

Inspect the new collection panel. A collection has its own settings page, distinct from the prompt editor. The panel shows what a folder never had :

1. *Inherits* — the collection's parent collection, if any (you set this in Chapter XXIII).
2. *Members* — the listing of public members, and which of them are `virtual` (abstract, to be implemented by derived collections) or `final` (sealed).
3. *Collection parameter sets* — named bundles of parameter bindings at the collection scope (this is where the parameter sets you could not put inside collection members in Chapter XVII now live).

Hoist a shared constant into a base prompt. A collection does not carry constants or parameters of its own. When several members repeat the same constant — say `COMPANY = "Acme Corp"` — the way to share it is a small base prompt that declares it once, with the members inheriting from that base (Chapter XIX). Cut the constant from the members, declare it once on the base, and re-render any member. The value now flows in through inheritance and one definition is canonical.

Define a collection-scope parameter set. If two or three members are injected the same way for the same recurring context, declare a parameter set in the *Collection parameter sets* panel. At injection time, the parameter dialog of any member will let the user pick from collection-scope sets in addition to its own.

21.4 — Hands-on exercise

Pick the real folder you identified at the start of the chapter and promote it. Then perform the *minimum* hoist: pull *exactly one* recurring binding into a collection-level parameter set, or one shared constant into a base prompt the members inherit. Re-render each member and confirm the output is unchanged.

The point of the «exactly one» rule is to feel the move without restructuring everything. Promotion is reversible in spirit; a day-one over-hoist where you move ten declarations and break three members is the easiest way to abandon collections forever. One declaration today, one tomorrow, and within a week the collection earns its keep on its own.

Bonus: introduce a `parameter-set` named after your team's most frequent injection context, declared at the collection scope, and verify it appears in the *Parameter set* dropdown of every member.

21.5 — Pitfall and how to avoid it

The single most common collection mistake is **promoting too early**. A folder with one or two members is not a collection — it is a folder pretending to be a class. The smell is clear: no shared parameter set, no virtual members, nothing the members actually have in common. You paid the cost of an extra abstraction and bought nothing. The recovery move is to demote the collection back to a plain folder (the option lives in the same context menu) and revisit the question in a month.

The second-most-common mistake is **promoting a folder whose members are unrelated** — three templates about three different domains that happen to live in the same folder for convenience. Promotion implies «these belong together»; if half the members will never share a parameter or a parameter set with the other half, split the folder first, then promote each half on its own merit.

21.6 — Recap and next step

A collection is a folder that earned its identity. Promote when at least one declaration is begging to be shared; hoist incrementally; never promote a folder with one member or three unrelated members. The collection-scope surface — the parent link, the member

listing with virtual/final markers, and collection-level parameter sets — is where the day-to-day OO² authoring of the next two chapters happens.

The next chapter introduces the construct that gives a collection a *contract*: virtual templates, the abstract members that derived collections must implement.

Chapter XXII — Virtual templates

A collection that says «I am a reviewer» is a label. A collection that says «I am a reviewer and every implementation of me *must* provide a `summarize`, a `score`, and a `suggest`» is a contract. Virtual templates are how a collection writes that contract: *placeholder* members the base collection marks abstract and that any derived collection must *implement*. They are the abstract methods of OO², and they are the trick that lets a team share a uniform interface across very different domains.

22.1 — Goal

By the end of this chapter you will have declared a base `reviewer` collection with three virtual members — `summarize`, `score`, `suggest` — and you will have derived a `code_reviewer` collection that implements all three. You will know how a prompt becomes virtual (the `virtual` modifier on its `@@@template` header — the single declaration site), and you will recognize the warning the linter raises when a virtual is never implemented.

22.2 — Prerequisites

- Chapter XXI: you have at least one promoted collection and you understand the three collection-scope panels (member listing, parameter sets, inheritance).
- Familiarity with the template-level inheritance vocabulary from Chapters XIX and XX.

Quick conceptual summary, so you do not have to leave the book. A member of a collection is *virtual* when its `@@@template` header carries the `virtual` modifier — that is the single declaration site (a collection never lists its virtuals as a metadata field). A virtual member is **abstract: it carries no body** — a virtual prompt with any rendered material raises `VIRTUAL_TEMPLATE_HAS_BODY`. The re-

solver tracks virtuals along the collection chain : rendering a member whose only definition in the chain is still virtual raises `UNIMPLEMENTED_VIRTUAL_TEMPLATE` . The dashboard surfaces the still-unmet virtuals as a warning list. Derived collections clear it by providing a concrete member with the same id.

22.3 — Walkthrough

Open the base collection. Reuse the collection you promoted in Chapter XXI, or create a new one called `reviewer` for this exercise. The base collection holds the interface ; the derived ones will hold the implementations.

Declare the contract. A collection’s contract is the set of its members whose `@@@template` header carries the `virtual` modifier. Create three placeholder members in the `reviewer` collection — `summarize` , `score` , `suggest` — each carrying the modifier on its header. For `summarize` :

```
@@@template virtual reviewer.summarize
@@@name Summarize
@@@description Three-sentence summary of the artifact under review
```

A virtual member stops at its header and metadata — it carries **no body**. A note about what the override should produce belongs in `@@@description` (metadata), never in rendered content. Do the same for `score` and `suggest` , then save. The collection now expects every derived collection to provide concrete members with those three ids. The base collection itself holds only the abstract placeholders ; the resolver flags them as virtual *in the base*, which is fine, because nobody is supposed to render the base directly.

The `virtual` modifier takes no argument : the prompt’s own parameters *are* its signature — the contract every concrete override must match. There is no collection-level list to keep in sync ; the modifier on the member’s header is the single source of truth.

Derive a concrete collection. Right-click `reviewer`, pick *Create derived collection*, and name it `code_reviewer`. In the new collection's *Inherits* field, `reviewer` is already filled in. Open the *Virtual templates* warning panel — it lists `summarize`, `score`, `suggest` as still-unmet virtuals. The collection is incomplete and the linter says so; rendering any of them would raise `UNIMPLEMENTED_VIRTUAL_TEMPLATE`.

Implement the contract. Inside `code_reviewer`, create three members with the matching ids:

```
@@@template code_reviewer.summarize
@@@name Summarize
Read the diff. Produce a three-sentence summary that names what
changed, why it changed, and what risk it carries.
```

```
@@@template code_reviewer.score
@@@name Score
On a 1-5 scale, rate the diff for clarity, correctness, and
test coverage. Output one line per axis.
```

```
@@@template code_reviewer.suggest
@@@name Suggest
List up to three concrete improvements, each in the form
"Replace X with Y because Z".
```

Save. Refresh the `code_reviewer` validation panel. The unmet-virtuals list is empty. The collection now satisfies the contract.

Inject a concrete member. Open the extension, find the `summarize` member of `code_reviewer`, and inject it into a chat tab with a real diff pasted underneath. The body that arrives in the chatbot is the concrete one you just wrote, not the placeholder from the base.

22.4 — Hands-on exercise

Derive a second concrete collection from the same `reviewer` base — `prose_reviewer` this time. Implement `summarize`, `score`, and `suggest` for prose rather than code:

- `summarize` : a one-paragraph précis of the document.
- `score` : clarity, structure, and prose quality on a 1-5 scale.
- `suggest` : up to three concrete edit recommendations.

Inject `prose_reviewer.summarize` and `code_reviewer.summarize` in two chat tabs with the same source material. The two templates share the same *interface* — same member id, same role in the contract — and produce two genuinely different outputs because the bodies are written for different domains. That contrast is the whole point of the abstraction.

22.5 — Pitfall and how to avoid it

The single most common virtual-template mistake is **declaring a virtual that nobody ever implements**. The base collection marks `audit` `virtual` on its `@@@template` header, three derived collections inherit without providing it, and the dashboard’s warning panel quietly accumulates unmet-virtual entries that everybody learns to ignore.

The recovery move is to ask: is `audit` *really* part of the contract? If every plausible reviewer provides it, keep the virtual and fix the derived collections. Otherwise delete the virtual from the base — a contract that no implementation honours is documentation pretending to be a check.

The second-most-common mistake is **leaving a placeholder as a concrete member**. A member with the text « (Override me. Replace with a real summary.) » and no modifier passes the validator — it is just a short concrete prompt. To make it a real contract,

delete the placeholder body and add the `virtual` modifier to its `@@@template` header: a virtual member is bodiless, and the linter starts protecting you again.

22.6 — The symmetric modifier — `final`

Just as `virtual` says «*some descendant must implement me*», the symmetric modifier `final` says «*no descendant may override me*». A prompt whose `@@@template` header carries `final` is *sealed* at the collection level: any derived collection that declares a member with the same id raises `OVERRIDE_OF_FINAL_TEMPLATE` at validation time.

```
@@@template final reviewer.legal_notice
@@@name Legal Notice
```

The output of this reviewer is licensed under CC BY-SA 4.0.

`final` takes no argument (the prompt's id is the thing being sealed) and behaves like `virtual` for the renderer: it is stripped from the output. Both modifiers are valid only on a collection-qualified `@@@template` header; the two are mutually exclusive on the same header — a header bearing both is `INVALID_SYNTAX`. A useful idiom: a derived collection takes a `virtual` member from its parent, provides a concrete override, and marks the override `final`, fixing the implementation for every further descendant.

Use `final` sparingly. A sealed member trades flexibility for guarantee — appropriate for boilerplate, regulatory text, or a contract clause that genuinely must not drift; almost never appropriate for content that the team might want to specialize later.

22.7 — Recap and next step

Virtual templates give a collection a contract. Declare them with the `virtual` modifier on the member's `@@@template` header (the single declaration site); derive a collection, implement every vir-

tual, and watch the unmet-virtuals list go to zero. Virtuals are how a team writes interfaces — same member ids across collections, different concrete bodies behind them. The symmetric `final` modifier seals a member against override when the content must not drift across the chain.

This is also the last chapter you complete before the Advanced preset's second OO axis comes into focus. The next chapter walks through what happens when *collections themselves* inherit from one another, so that today's `reviewer` becomes tomorrow's `bilingual_reviewer` without any duplicated members.

Chapter XXIII — Collection inheritance (the second OO axis)

Tied to tip: `standard-using-collections` (fires when a Standard user is making heavy use of collections — at that point the dashboard nudges you toward Advanced and toward this chapter).

In Chapter XIX you learned that one *prompt* can inherit from another. POO²L has a second, parallel axis: an entire *collection* can inherit from another collection. That is what turns a folder of vaguely related templates into a real family of reusable classes.

23.1 — Goal

By the end of this chapter you will have derived a third collection from your `reviewer` collection and observed `^parent.<template_id>` being resolved from a member of the derived collection.

23.2 — Prerequisites

- Chapters XXI and XXII done — you have at least one collection with declared virtual members.
- A short recap of the relevant POO²L rule: at the collection level, an inline `inherits <parent>` clause on the `@@@collection` block makes a collection a child of another collection. Children inherit the parent's templates (which become *inherited methods*), can override any of them, and can reach the parent collection's version of a member through the **collection axis** operator `^parent` — written `@@@include ^parent.<template_id>`, the collection-axis counterpart of the template-axis `parent` from XX. The two axes are disjoint: bare `parent` reaches parent *sections* on the prompt axis, `^parent` reaches parent-collection *members* on the collection axis.

23.3 — Walkthrough

Pick a base collection. Open the `reviewer` collection you built in Chapter XXII. It declares three virtual templates (`summarize`, `score`, `suggest`) and has a `code_reviewer` child that implements them. We are going to add a third sibling.

Create the new collection. From the sidebar, *New collection* → name it `prose_reviewer`. Declare its parent inline on the `@@@collection` header:

```
@@@collection prose_reviewer inherits reviewer
@@@template summarize
@@@template score
@@@template suggest
@@@end-collection
```

The dashboard validates the link immediately. If the parent is not found you see `UNDEFINED_REFERENCE` — the same error you saw at the prompt level. Rendering is delegated to the POO²L engine, which walks collection chains and prompt chains on their two disjoint axes, with cycle detection built in.

Override one prompt. Inside `prose_reviewer`, create three templates named `summarize`, `score`, `suggest` — the same ids the parent declared as virtual. They are now the *implementations* of those virtual contracts for this branch of the family.

Use `^parent` to reuse the base. The `summarize`, `score` and `suggest` members of `reviewer` are *virtual* (bodiless contracts, Chapter XXII), so there is nothing of theirs to extend — a derived member must *implement* them outright. What a child *does* reuse from the base is its **concrete** shared members. Say `reviewer` also carries a concrete `preamble` member stating the reviewing ground rules. Open `prose_reviewer.summarize` and pull it in on the collection axis, then add your prose-specific content :

```
@@@include ^parent.preamble
```

Add a one-sentence note on tone and register.

Render the prompt and observe the output: the base `preamble` content is rendered first, your prose-specific paragraph appended. `^parent.preamble` resolves the member exactly as the *parent collection* would render it.

Compare with the sibling. Open `code_reviewer.summarize` (the sibling from Chapter XXII). Same parent, same contract, different implementation — that is the entire point of collection inheritance.

Verify with the diff viewer. Pick `code_reviewer.summarize` and `prose_reviewer.summarize` and open a side-by-side diff. The two implementations share the inherited preamble (via `^parent`) and diverge only where each child adds value.

23.4 — Hands-on exercise

In your own dashboard, identify two collections that already share 80% of their behaviour — typically a generic version and a domain-specific version of the same task. Examples :

- A generic `code_reviewer` and a stricter `security_reviewer`.
- A generic `translator` and a `legal_translator` that adds jurisdiction caveats.
- A `meeting_notes` and an `action_item_meeting_notes`.

Promote the generic one to a parent, declare the specific one with an inline `inherits <parent>` clause, move the shared templates to the parent, and use `@@@include ^parent.<template_id>` in the child wherever you want to *extend* a concrete inherited member instead of replacing it. Render two real outputs and check that the chain behaves as expected.

23.5 — Pitfall and how to avoid it

The most common mistake at this level is **using collection inheritance for what is plain include**. If you find yourself inheriting just to reuse one fragment, you do not want a parent collection — you want a fragment prompt that both consumers include via `@@@{include other_template}`. Collection inheritance is the heaviest tool in the POO²L toolbox; it pays off only when the relationship between the two collections is genuinely *is-a* (a `prose_reviewer` is a `reviewer`), not *uses-a*. The rule of thumb: if you can write a one-sentence « X is a Y » without flinching, inherit; otherwise, include.

The second pitfall is **silent override**. If your child collection defines a prompt with the same id as a concrete parent prompt and you forget `^parent`, you have just replaced the parent’s body entirely — your edit, only your edit, and the parent contribution is gone from the rendered output. The diff viewer in Chapter XXVI will catch it, but the cheaper habit is to *always* start a collection-axis override with `@@@include ^parent.<id>` and remove it deliberately if you really want a clean slate.

23.6 — Recap and next step

Your dashboard now has a family of three collections — `reviewer`, `code_reviewer`, `prose_reviewer` — related by inheritance, each member of each child either inheriting silently, overriding fully, or extending via `^parent`. That is OO² in your hands: two orthogonal axes (templates and collections), one mental model.

The next chapter introduces aliases — the safety net that lets you rename heavily-referenced templates without breaking the network of includes and `inherits` links you just built.

Chapter XXIV — Aliases

A prompt that nobody references is easy to rename — you change the name and you are done. A prompt that ten other templates include, three collections inherit from, and two parameter sets bind to is a different matter. Renaming it the naïve way breaks every one of those references. This chapter teaches the safety net.

24.1 — Goal

By the end of this chapter you will have renamed a heavily-referenced prompt and declared its old name as an alias, leaving every existing `include` and `inherits` link working.

24.2 — Prerequisites

- Chapter XXIII done — you have a small network of templates that reference each other.
- A short recap of the POO²L rule: a prompt may declare one or more aliases via the inline declaration `@@@alias <old_id>`. The resolver treats `<old_id>` as a valid identity for the prompt, so any `@@@{include <old_id>}` or inline `inherits <old_id>` clause keeps resolving after the rename. Aliases live *on the renamed prompt itself*, not on a stub at the old path.

24.3 — Walkthrough

Pick a renamable prompt. From your library, choose one that is referenced from at least three other templates. A typical candidate is your earliest «safety preamble» or a glossary you built in Chapter XV. Note the current id — let us call it `safety_preamble`.

Find the references. In the dashboard’s filter bar, search for `safety_preamble` across the corpus. Note every prompt that includes or inherits it. Three or four hits is the minimum that makes this exercise worth doing.

Rename in the editor. Open the prompt itself. Edit its name to the new id, say `house_safety_rules`. *Do not save yet* — saving at this point produces the broken state we want to avoid.

Declare the alias. In the prompt body, add one line at the top :

```
@@@alias safety_preamble
```

Save. The dashboard re-indexes the alias map ; the inline parser at `src/core/inline_declarations_parser.py` strips the declaration from the rendered output and registers `safety_preamble` as a secondary identity for this prompt.

Verify the network. Re-open the three referring templates and render each. The includes resolve, the inheritance link resolves, the rendered text is byte-for-byte identical to what it was before the rename. Search for `safety_preamble` once more — it should still point to the same (now renamed) prompt.

Migrate references at your own pace. Aliases are not deprecation warnings ; the dashboard does not nag you to update callers. Edit each caller when you happen to be in it, replace `@@@{include safety_preamble}` by `@@@{include house_safety_rules}`, and move on. The alias remains as a permanent safety net for any caller you missed.

24.4 — Hands-on exercise

Perform one alias-protected rename in your real library :

1. Identify one prompt with at least three callers.

2. Note the new name you want and the number of callers (write both down — this will tell you whether the rename was worth it).
3. Rename, add `@@@alias <old_id>`, save.
4. Render two of the callers and confirm nothing changed in their output.
5. Optionally migrate one caller to the new id by hand, leave the others, and observe that both forms keep working.

24.5 — Pitfall and how to avoid it

The most common mistake is **using aliases as a synonym dictionary**. Aliases are not a thesaurus — they are a *rename receipt*. Declaring `@@@alias short_name` on a prompt that has never been called `short_name` just to support a «convenient shortcut» pollutes the identity namespace and makes future renames riskier. The rule of thumb: an alias maps a *historical* id to the current prompt; if no one has ever written that id, there is nothing historical and you do not need an alias.

The second mistake is **alias collisions**. Two templates cannot claim the same alias — the parser detects the duplicate and the second prompt refuses to save with an identity error. If you wanted to free an old id for a new purpose, that is a delete-then-create operation, not an alias.

24.6 — Recap and next step

You can now rename without fear. The takeaway: rename freely, drop an alias each time, migrate callers opportunistically, and never break a network. The cost of the alias declaration is one line; the cost of a broken include is a quiet bug your readers will discover in a chatbot session a week later.

The next chapter introduces the second lifecycle tool — deprecation. Where aliases say «this prompt has a new name», deprecation says «this prompt has a successor». Same axis, different verb.

Chapter XXV — Deprecation

Aliases handled the *rename* case : same prompt, new name. Deprecation handles the *retirement* case : a prompt that still works but should no longer be the one your colleagues reach for. It is the difference between «I moved the keys» and «I bought a new car». This chapter teaches the latter — gracefully, without breaking the callers that have not yet switched.

25.1 — Goal

By the end of this chapter you will have marked a prompt as deprecated, attached a migration message to it, observed the warning surface at instantiation and at inherit/include time, and migrated two callers to its successor.

25.2 — Prerequisites

- Chapter XXIV done — you understand the lifecycle axis and the difference between *renaming* and *retiring*.
- A short recap of the POO²L rule : a prompt may declare exactly one deprecation via the inline declaration `@@@deprecated "migration message"`. The parser strips the declaration from the rendered body and sets the `deprecated` field on the prompt metadata. Anywhere the deprecated prompt is reached — direct instantiation, `@@@{include }`, or an inline `inherits` clause — the resolver surfaces a non-blocking warning whose text embeds your migration message verbatim.

25.3 — Walkthrough

Pick a retiring candidate. From your library, identify one prompt that has a successor : a tighter rewrite, a parameterized version, a member of a newer collection. A common case is the «v1» of a

prompt you have since rebuilt as a three-section parameterized version. Note both ids — say `code_reviewer_legacy` (the retiring one) and `code_reviewer` (the successor).

Open the editor on the retiring prompt. At the very top of the body, add one line :

```
@@@deprecated "Use code_reviewer instead – same contract, three parameters"
```

Save. The dashboard re-parses the prompt; the inline declaration is stripped from the rendered output and the `deprecated` field on the parsed prompt is now set to your message. The relevant code lives in `src/core/inline_declarations_parser.py` (functions `extract_deprecated` and `inject_deprecated`, regex `DEPRECATED_PATTERN`).

Observe the warning at instantiation. Open the deprecated prompt in the editor and trigger a render preview. The preview banner shows the warning from `src/lang/strings_en.json` under `deprecated.templateDeprecated` :

```
Template 'code_reviewer_legacy' is deprecated:  
Use code_reviewer instead – same contract, three parameters
```

The output renders normally underneath. Deprecation is informative, not blocking — your colleagues can keep using the prompt while they migrate.

Observe the warning at include time. Pick any other prompt that contains `@@@{include code_reviewer_legacy}`. Render it. The warning surfaces a second time, this time with the `deprecated.includeDeprecated` wording: « Included prompt 'code_reviewer_legacy' is deprecated... ». The caller sees the migration hint without having to leave the rendered output.

Observe the warning at inherit time. If any prompt or collection declares `inherits code_reviewer_legacy`, the validator in `src/core/inheritance_validator.py` emits the `deprecated_inherits` warning using the `deprecated.inheritsDeprecated` template. Two parts of the dashboard surface it: the inheritance chain panel in the editor, and the collection diagnostics panel.

Migrate the first caller. Open one of the includers. Replace `@@@{include code_reviewer_legacy}` by `@@@{include code_reviewer}`. Render. The warning is gone for this caller; the others still show it.

Migrate the second caller. Repeat. The warning count goes down by one each time. The deprecation has done its job: it pulled two callers off the retiring prompt without any forced flag day.

Leave the retired prompt alive. Do not delete it. Its very presence — with the deprecation message — is the documentation that the migration is intentional. Three months from now, when no caller remains, you can revisit and decide whether to retire the prompt itself.

25.4 — Hands-on exercise

Pick one prompt in your library that you would no longer recommend to a colleague but that you have not deleted:

1. Identify the successor. If there is none, this is the wrong exercise — a deprecation without a successor is just a delete in slow motion. Build the successor first.
2. Add the `@@@deprecated "..."` declaration to the retiring prompt; in the message, name the successor and the reason in one short sentence.
3. List the callers (search by id in the filter bar). Pick two.
4. Migrate those two callers to the successor.
5. Leave any remaining callers alone — they will discover the deprecation the next time they render the prompt, which is the intended behaviour.

You have now performed your first graceful retirement.

25.5 — Pitfall and how to avoid it

The most common mistake is **using deprecated as a documentation field**. The migration message is not a place for changelogs, authorship credits, or general commentary — it is a *migration hint*. One actionable sentence pointing to the successor: that is all. The shorter and more imperative the message, the more useful it is at 4pm when a teammate is staring at the warning and needs to know what to do next.

The second pitfall is **deprecating a prompt whose successor does not yet exist**. If the successor is still a draft, the warning points your readers nowhere; the prompt is half-retired and unsupported. Build the successor first, validate it on real callers, *then* deprecate the predecessor. The two steps are sequential, not simultaneous.

The third pitfall is **stacking aliases and deprecation on the same prompt to « rename and retire » at once**. The two lifecycle verbs solve different problems and combining them confuses readers: is this prompt being renamed (use the new name) or retired (use the successor)? Pick one verb per change. Rename first with an alias, let the dust settle, then deprecate later if you also want to retire.

25.6 — Recap and next step

You can now retire templates gracefully. The takeaway: a deprecation is a non-blocking warning with a one-sentence migration hint that surfaces at the three reach-points — instantiation, include, and inherits. Callers migrate at their own pace; the retiring prompt keeps working; nothing breaks suddenly. With aliases (Chapter XXIV) and deprecation in your toolbox, the lifecycle axis is fully covered: you can rename without breaking, and you can retire without abandoning.

The next chapter consolidates the safety net you started in Chapter X: how to read a diff fluently, and how to rollback under stress without throwing the current state away.

Partie 5

*Year one — Mastery and
teamwork (Expert preset)*

Chapter XXVI — Diff viewer and rollback workflows

You met versioning in Chapter X as a safety net: landmarks you save deliberately, a history panel, a rollback button. That was enough for week one. Now that you maintain a real library — inheritance chains, includes, parameter sets, deprecations — the diff viewer and the rollback button are no longer occasional tools. They become the verbs you use when something *was* working and is not anymore. This chapter teaches you to read a diff fluently and to rollback under stress, without throwing the current state away.

26.1 — Goal

By the end of this chapter you will read the side-by-side diff in both Line and Word modes without thinking, perform a rollback that preserves the current state as a new version, and diagnose a regression in one of your templates using only the version history.

26.2 — Prerequisites

- Chapter X: you have already saved versions deliberately and you have rolled back at least once.
- Chapter XXV: not strictly required for the diff viewer, but you will sometimes diff a « before deprecation » version against the current one to confirm the migration was clean.
- One real prompt with at least four meaningful versions. If you do not have one, build it now — there is nothing to compare without history.

26.3 — Walkthrough

Open the version history. Pick your most-edited prompt. Click the *History* button in the editor toolbar. The drop-down lists every version, most recent first; each entry shows the timestamp, the note you wrote when you saved that version, and a size delta. The currently active version is highlighted.

Pick two versions to compare. Click an older version. The panel renders the side-by-side diff between the version you clicked and the current one — old on the left, new on the right, added spans in green, removed spans in red, unchanged context in grey. This is exactly what the `POST /api/diff/compare` endpoint in `src/api/diff.py` returns: a list of blocks tagged `unchanged`, `modified`, `added`, or `removed`.

Read the diff fluently. Three reading habits make this fast. First, scan the *removed* (left-only) spans top to bottom — they are the intent of the change. Second, scan the *added* (right-only) spans — they are the implementation. Third, look at the modified blocks where old and new sit across from each other; this is where you confirm the change preserved meaning. With practice, an experienced reader processes a 50-line diff in under twenty seconds.

The layout is always side-by-side. There is no unified mode; the old version sits on the left, the new on the right, lines aligned. What the toolbar *does* toggle is the *granularity* — Line mode versus Word mode — covered next.

Switch to Word mode for surgical changes. For diffs where one sentence was rephrased, flip the Line/Word toggle to Word mode. The endpoint returns `old_spans` and `new_spans` arrays — the modified words highlighted within each line — instead of marking the whole line as changed. Trivial typo fixes shrink to a single highlighted token, and the panel remembers your Line/Word choice between diffs.

Diagnose a regression. Suppose the rendered output of a prompt suddenly feels off. Open the history; identify the last version that was known-good (the one with the note «before refactor — known good» you saved in Chapter X). Diff it against the current version. The change is right there in green and red. Read both sides; identify the *one* edit that introduced the regression.

Snapshot the current state before rolling back. This is the single most important habit of this chapter. Even if the current state is broken, it contains intent — the partial work you were doing when you noticed the regression. Click *Save as new version* and write a note like «before rollback — broken but contains 2026-05-17 work in progress». The history will now keep this state forever.

Rollback. Select the known-good version in the history dropdown and click *Restore version*. The dashboard templates you to confirm. Note the wording: «*This action will create a new version (copy) based on this old version*» — the restore is not destructive. It copies the old version forward as the new current version. Your broken-but-saved state is still in the history, one click away. The relevant code is `restoreVersion()` in `src/static/index-versioning.js` and the `POST /api/versions/{template_id}/copy` endpoint in `src/api/versioning.py`.

Verify. Render the prompt. The regression is gone. Compare the new current version against the known-good one — they should be byte-for-byte identical except for the version metadata.

26.4 — Hands-on exercise

Pick a real prompt with at least four versions. Pretend the current version is broken:

1. Open the history and locate the version you would rollback to — the most recent one with a note that says «known good» or equivalent. If no version qualifies, this exercise has already taught you something: start writing better notes.

2. Diff the current version against your candidate target. Read it in Line mode. Switch to Word mode if the change was localized. Build the habit of using both.
3. Snapshot the current state with the note «before rollback — exercise from Chapter 26».
4. Rollback to the target version.
5. Render the prompt and confirm the output matches what you expected.
6. Bonus: snapshot the post-rollback state with a clear note («known good — restored from v3 on 2026-05-17»). Six months from now you will thank yourself for the bookkeeping.

26.5 — Pitfall and how to avoid it

The most common mistake at this level is **rolling back without first snapshotting the current state**. Even if the current state looks broken, it is rarely worthless — it contains the half-finished edit you were working on, the new section that almost worked, the parameter you renamed. Roll back without a snapshot and that work is gone for good. The rule of thumb: any rollback under stress deserves a «before rollback» version, no exceptions. The cost is one click and one short note; the cost of the missing version is sometimes hours of recovery.

The second pitfall is **diffing the wrong pair**. The history panel defaults to «selected vs current», which is what you want 90% of the time. But sometimes you want to compare two *historical* versions — for example to confirm that v3 and v5 produced equivalent output. The diff panel supports this; pick the second version explicitly from the dropdown rather than assuming the default. A diff against the wrong baseline can make a clean change look broken, or a broken change look clean.

The third pitfall is **rolling back when a forward fix is cheaper**. Rollback is the right move when you cannot understand the regression quickly. If you *can* identify the offending edit in under

five minutes, a forward fix (undo just that edit, save a new version) is cleaner and preserves more of your recent work. Treat rollback as the medium-cost option, not the default.

26.6 — Recap and next step

Your audit reflexes are now sharp. You can read the side-by-side diff in both granularities — Line mode and Word mode — and pick the right one for the change. You can rollback under stress without losing the work you had in flight. The history is no longer a museum ; it is a working tool.

The next chapter changes scope : from your own library to the team's. Tag administration — merge, rename, restructure — is where the soloist becomes a librarian.

Chapter XXVII — Tag administration

A tag taxonomy starts clean — three or four neat axes, no overlap. Then six months pass. Colleagues add tags. You add tags. Synonyms appear. `client-report` and `report-client` and `Client Report` all live in the system, each tagged on a handful of templates, and the filter chip is now useless because the same concept hides behind three labels. This chapter is the librarian's chapter — how to clean up the taxonomy without breaking the templates that depend on it.

27.1 — Goal

By the end of this chapter you will have merged two synonymous tags into one, renamed a tag everywhere it is used, restructured a tag hierarchy by moving children under a different parent, and done all three without breaking any prompt that uses those tags.

27.2 — Prerequisites

- Chapter XXVI: you can rollback if a tag operation produces an unintended result; the safety net is one click away.
- Chapter VII: you understand what tags are and how templates reference them.
- A library with at least twenty tagged templates. Tag administration is uninteresting below that threshold; you can fix synonyms by hand.
- One short Manual-style summary, here so you do not leave the book: tag administration covers three operations on the *team's* taxonomy — `merge` (two tags become one, all templates re-pointed), `rename` (one tag gets a new label everywhere it appears), and `restructure` (a tag's place in the hierarchy changes — parent, short name, icon, colour, acronym). All three are reversible through versioning if you snapshot before.

27.3 — Walkthrough

Open the tag administration panel. From the sidebar, click the *Tags* entry; the panel lists every tag in your scope (personal or group), with a usage count next to each one. The relevant endpoint is `GET /api/tags/list` in `src/api/tags.py`. Sort by usage count descending — the most-used tags are at the top, the long tail at the bottom. Synonyms almost always live in the long tail.

Spot the synonyms. Scroll the long tail. Look for pairs and trios that obviously mean the same thing — `client-report` and `report-client`, `wip` and `work-in-progress`, `marketing` and `Marketing` (yes, case matters until you merge them). Note three candidates.

Merge two synonymous tags. Pick a pair. Decide which name survives — typically the shorter, lowercase, hyphenated form. Right-click the *surviving* form is wrong here; the merge is *from* source *to* target. Right-click the synonym you want to retire and pick *Merge into...*; choose the survivor in the dialog. The dashboard calls `POST /api/tags/v2/{source_id}/merge-into/{target_id}` which moves every prompt linked to the source over to the target, then deletes the source. The response gives you the `moved` count; verify it matches what you expected.

Rename a tag everywhere. Sometimes the surviving form is itself wrong — say `clients` should really be `customer`. Open the tag detail panel; click *Rename*. The endpoint `PUT /api/tags/{tag_name}` calls `manager.rename_tag(old, new)` which walks every prompt that uses the old name and updates the reference in place. The response message tells you how many templates were updated. Note the special case in the source: if the new name already exists as a tag, the rename becomes an *implicit merge* — the response message changes from «renamed to» to «merged into» and the affected count reflects both sources combined.

Restructure a hierarchy. Tag hierarchies in IziBuddy are expressed by naming conventions and parent links visible in the tag detail panel. Open a tag that lives at the wrong level — say `urgent`

sitting at the root when it should be a child of `priority`. Edit the parent. The endpoint `PUT /api/tags/icon/{tag_name}`, the colour endpoint, the acronym endpoint, the short-name endpoint, and the show-icon endpoint all follow the same pattern: a single field update on the tag, no re-link work needed on templates (they reference tags by id, not by path). Update icon, colour, acronym, short name, and `show_icon` the same way; the templates that wear the tag inherit the new presentation immediately.

Verify across the corpus. After any tag operation, open three templates that used the affected tag, render each, and confirm the output is unchanged. Tags do not appear in the rendered prompt text, but they do drive filtering — open the filter chip for the new tag and confirm the right templates are listed.

Snapshot if anything looks off. If the result is not what you expected, the dashboard cannot undo a merge with a single click — the source tag has been deleted. The recovery path is to recreate the tag manually and re-link the affected templates, which is tedious. The cheaper habit: before any bulk tag operation, write down the source and target names and the expected `moved` count. If the actual count differs, stop and investigate before doing the next operation.

27.4 — Hands-on exercise

In your real library:

1. Sort your tags by usage count. Identify three tags in the long tail that overlap with a tag in the head.
2. For each of the three: decide which name survives (almost always the head tag — it has more templates already), and merge the long-tail tag into it. Write down the `moved` count.
3. Find one tag whose name is clumsy — capitalized, spaced, or plural where the rest are singular. Rename it to match the house style. Write down the affected count.

4. Pick one tag that lives at the root but conceptually belongs under another tag. Update its parent. Open the tag tree and confirm the new placement.
5. Render two templates from each affected tag and confirm the output is byte-for-byte unchanged.

You have just done your first day as the team's tag librarian.

27.5 — Pitfall and how to avoid it

The most common mistake is **merging a tag while colleagues are mid-edit**. Tags are a shared resource within a group ; a merge that runs at 3pm changes the filter chip for everyone connected to the group at that moment. The colleague who had `report-client` open in a filter chip suddenly sees the chip rename itself, and depending on the in-flight cache, may see an empty list. The fix is coordination : do bulk tag operations after hours, or in a quiet slot announced on whatever channel your team uses. The dashboard does not enforce a maintenance window — that habit is on you.

The second pitfall is **renaming when you should merge**. If the « new name » already exists as a tag, the rename endpoint silently becomes a merge (this is intentional — same shape, same effect). That is fine when you intend it ; it is a surprise when you do not. Read the response message ; « merged into » tells you a merge happened, « renamed to » tells you a true rename happened. If you intended the former, fine. If you intended the latter and got the former, you have just absorbed an unrelated tag into your templates. Rollback the affected templates using Chapter XXVI or rebuild the lost tag manually.

The third pitfall is **over-merging**. Two tags that sound synonymous to you may not be synonymous to a teammate. `important` and `priority` look interchangeable at 9am ; at 4pm they are two distinct concepts and your team has lost the distinction. The rule of thumb : if you cannot articulate the *difference* between the two tags before merging them, the merge is safe ; if you have to ask a colleague and they hesitate, leave them separate.

27.6 — Recap and next step

Tags scale with the team. You can now merge synonyms, rename the clumsy ones, and restructure the hierarchy without breaking the templates that depend on them. Use the operations conservatively ; each one moves the team's mental model, not just the data.

The next chapter takes the team dimension further. Tags are a *soft* sharing — everyone in the group sees the same tag, but templates can still be private. Groups and shared templates are the real teamwork primitive.

Chapter XXVIII — Groups and shared templates

Until this point in the book you have been a soloist. You wrote templates, organized them, refactored them with POO²L, and maintained them — all in your own personal scope. The dashboard’s *group* primitive is what turns the soloist into an ensemble. A group is a named scope where multiple users see the same templates, the same tags, and the same collections; what one member changes, the other members see. This chapter is the move from « my library » to « our library ».

28.1 — Goal

By the end of this chapter you will have created or joined a group, understood the role model, switched between personal and group scope in the dashboard, and shared one of your templates with a group.

28.2 — Prerequisites

- Chapter XXVII : tag administration is the warm-up for group ownership; both require you to think beyond your own files.
- A second IziBuddy user — a real teammate or a test account — to share with. Sharing with yourself is technically possible but defeats the purpose of the exercise.
- A short Manual-style summary, here so you do not leave the book: a *group* is a multi-tenant scope identified by `group_id`, with two distinct member categories — *group admin* (can manage members, share templates at the group level, edit any group prompt) and *member* (can read group templates and use them; can edit the ones they author, depending on the group’s policy). System admins are a third tier orthogonal to groups: they can manage any group. A user can belong to several

groups simultaneously; switching scope in the dashboard changes which library you see.

28.3 — Walkthrough

Create or join a group. If you are a system admin, the *Groups* panel in the sidebar shows a *New group* button; clicking it calls `POST /api/groups` and creates an empty group with you as the initial group admin. If you are not a system admin, ask one — the endpoint requires `get_admin_user` per `src/api/groups.py`. Once the group exists, the system admin or any existing group admin adds you via `POST /api/groups/{group_id}/members/{username}`.

See your groups. Once you are a member, the *My groups* panel at `GET /api/groups/me` lists every group you belong to. `GET /api/groups/me/admin` returns the subset where you are also a group admin — that distinction matters for sharing, because only group admins can *share* a prompt with the group (regular members can use the group’s shared templates but cannot publish new ones).

Understand the three tiers. The endpoint `require_group_admin_or_system_admin` in `src/api/groups.py` makes the rule explicit: management operations (add member, remove member, promote to admin, demote from admin, delete group) require either system-admin or group-admin. There is also a self-protection rule — a group admin cannot remove themselves; only a system admin can do that, to prevent the orphaned-group case where a group exists with no admins.

Switch scope. In the dashboard’s sidebar, the scope selector lets you toggle between *Personal* and each of your groups. The prompt list filters to whatever scope is active. The split is real, not cosmetic: a personal prompt and a group prompt with the same id are two distinct entities; the storage paths differ; the tag namespace differs. The combined manager in `src/api/templates.py` is what stitches them together for read but keeps writes scoped.

Share a prompt at the group level. If you are a group admin, pick one of your personal templates and click *Share with...* in the prompt's context menu. The endpoint `POST /api/templates/share/{template_id}` copies the prompt into the group's storage and sets its `shared` flag. From that moment on, every member of the group sees the prompt in their *group* scope ; none of them sees the copy in your personal scope.

Unshare if needed. The reverse endpoint `POST /api/templates/unshare/{template_id}` exists for the case where you shared something prematurely. It removes the prompt from the group's scope ; what happens to in-flight inheritance chains depends on whether other group templates inherit from the unshared one — be careful, and prefer deprecation (Chapter XXV) when in doubt.

Render a group prompt. Open any prompt you can see in the group scope. The editor opens, the *Save* button is grey if you do not have edit rights, the render preview works just the same as for a personal prompt. If you are a group admin or the original author, you can edit ; otherwise the editor is read-only.

Notice the Locked marker convention. Group templates that the team has agreed are *finals* should wear the Locked marker from Chapter V. The Lock is the team's signal that the prompt is in production and changes should go through review. The dashboard does not enforce the convention ; the team does.

28.4 — Hands-on exercise

Set up a small group with one real teammate (or a test account) and share three templates :

1. Have a system admin create a group called `sandbox-<your-name>` with you as the initial group admin.
2. Add your teammate as a member.
3. Switch to the group scope. The list is empty — that is correct ; your personal templates are not in this group.

4. Switch back to personal scope, pick three templates that would genuinely benefit your teammate, and share each one with the group.
5. Have your teammate switch to the group scope and confirm they see the three templates.
6. Apply the Locked marker to one of the three — the most finished one. Discuss with your teammate what « Locked » means in your group convention.

28.5 — Pitfall and how to avoid it

The most common mistake at this level is **editing a group-owned prompt without coordinating**. Group templates are by definition visible to other people ; an edit you save at 11 :32 is in front of your teammate’s filter chip at 11 :33. If the edit is mid-thought, the teammate may inject a half-baked version into a chatbot before you finish. The fix is the Locked marker convention : agree as a team that anything Locked is production and any edit goes through a versioning landmark (Chapter X) plus a brief heads-up on whatever channel the team uses.

The second pitfall is **sharing too early**. A prompt becomes group-owned the moment you call `share` ; from that point on, the team sees every edit. Shipping a draft as a group prompt creates the impression that the team should be using it, which it should not be. The rule of thumb : share when the prompt has earned at least two deliberate versions and you would be comfortable letting a stranger inject it tomorrow morning.

The third pitfall is **conflating sharing with copying**. A shared prompt is not a copy that lives in two places — it is a single prompt that has been moved into the group’s storage. If you want your personal scope to keep an independent version, duplicate first, then share the duplicate. Otherwise the prompt you used to have personally is now group-owned and any group admin can edit it.

28.6 — Recap and next step

Groups unlock teamwork. You now have a shared scope, three shared templates, one Locked convention, and a sense of who can do what. Your library is no longer just yours.

The next chapter teaches how to grow the team itself — how to invite a teammate from the outside, how the onboarding token works, and which role to grant the new arrival on day one.

Chapter XXIX — Invitations and onboarding teammates

You have a group, you have shared three templates in it, and now a colleague needs an account. They cannot just sign up — IziBuddy is invitation-based, by design. This chapter teaches the three sides of the onboarding flow: how to send the invitation, how the new user accepts it, and what role they should land with on day one.

29.1 — Goal

By the end of this chapter you will have sent an invitation linked to a group, walked through the acceptance flow from the invitee's side, observed the role propagation when the new user lands in the group, and applied the start-as-member-promote-later policy deliberately.

29.2 — Prerequisites

- Chapter XXVIII: you have a sandbox group ready to receive a new teammate, and you understand the distinction between group admin and member.
- System admin access on the dashboard, or a system admin who can send the invitation on your behalf. The invitation endpoints in `src/api/invitations.py` are gated by `get_admin_user`.
- One short Manual-style summary, here so you do not leave the book: an *invitation* is a single-use, time-limited token tied to an email address and an optional list of groups. The system admin creates it via `POST /api/invitations`; the invitee uses the token to register an account via `POST /api/invitations/register`; on registration, the new user is automatically added to every group the invitation listed, with the role *member* (not group admin). Tokens expire after a configurable number of days (default 7).

29.3 — Walkthrough

Open the invitations panel. As a system admin, navigate to the *Invitations* panel from the admin sidebar. The page lists pending invitations from `GET /api/invitations/pending` and, optionally, the full history including expired ones from `GET /api/invitations`.

Create an invitation. Click *New invitation*. The form asks for three things: the invitee's email (validated as a real `EmailStr` by the Pydantic model), an optional list of groups the new user should join on registration, and an expiry in days (default 7). Submit. The endpoint `POST /api/invitations` creates the token and returns it; in production deployments, the dashboard also emails the invitee a link containing the token.

Inspect the invitation. The response includes the token and a public dict — the same one `GET /api/invitations/validate/{token}` returns to the public when the invitee opens the registration link. Note what is *not* in the public dict: no creator username, no internal audit fields. The invitee sees only what they need: the email it was issued for, the groups they will join, and the expiry timestamp.

Send the link. The registration URL is `/register?token=<token>` on your dashboard host. Send it through whatever channel your team uses — Slack, email, in-person. The link is the only secret; anyone who has it can register an account under the invited email.

Switch to the invitee's side. Open the registration URL in a private window. The frontend calls `GET /api/invitations/validate/{token}` and, if valid, pre-fills the email and shows the list of groups the new user will join. This is the invitee's only chance to confirm they were invited to the right team — if the email or groups look wrong, they should stop here and ask.

Register. The invitee picks a username and a password and submits. The endpoint `POST /api/invitations/register` validates the token one last time, creates the user account, marks the token as used, and adds the new user to every group listed on the invitation. The account is now active ; the invitee can log in.

Observe role propagation. Switch back to your admin account and open the group's *Members* panel. The new user appears immediately ; their role is *member*, not group admin. This is the intentional default — `POST /api/invitations/register` never elevates anyone to group admin. Promotion to group admin is an explicit second step via `POST /api/groups/{group_id}/admins/{username}` , which only existing group admins or system admins can call.

Walk the new user through their first ten minutes. Pair with them. Open the dashboard. Switch to the group scope. Show the three shared templates. Inject one into a chatbot. The new teammate's first experience of IziBuddy should be a working loop, not a tour of features. The features panel from Chapter XI can come later ; on day one, keep them on the Simple preset.

Revoke if needed. If the invitation was sent in error, or if the invitee never registered and the email is now obsolete, call `DELETE /api/invitations/{token}` to revoke. The same endpoint exists for cleanup of expired invitations in bulk via `POST /api/invitations/cleanup` . Revoked tokens cannot be reused.

29.4 — Hands-on exercise

Invite a real teammate to your sandbox group from Chapter XXVIII :

1. Have a system admin create an invitation for your teammate's real email, listing your sandbox group, with an expiry of 7 days.
2. Send the registration link.
3. Walk through the registration with the teammate over a video call or a shared screen. Watch them confirm the pre-filled email and group list.

4. Once they register, switch to the group's Members panel and confirm the new user appears as a member (not as admin).
5. Together, switch their scope to the group. Inject one of the shared templates into a chatbot.
6. Stop. Resist the urge to give a tour of the dashboard. Schedule a follow-up later in the week to introduce folders, tags, and markers — those concepts are part of week one, not day one.

29.5 — Pitfall and how to avoid it

The most common mistake at this level is **granting admin too liberally**. The invitation flow defaults the new user to *member* for a good reason: admin powers (add or remove members, share templates, edit any group prompt) are easy to misuse before someone has internalized the team's conventions. The rule of thumb: start every new user as member; promote to group admin only when they have demonstrably understood the Locked-marker convention, the deprecation lifecycle, and the tag administration etiquette from Chapter XXVII. Two weeks as member is a reasonable minimum.

The second pitfall is **reusing tokens or treating them as shareable**. An invitation token is a single-use credential for a single email address. Forwarding the link to a different person does not work cleanly; the email pre-fill will look wrong, and the groups attached may be inappropriate for the new recipient. If two people need invitations, create two invitations.

The third pitfall is **letting expired tokens linger**. Once an invitation has expired, it is no longer usable but it remains visible in `GET /api/invitations` (with `include_expired=true`). The bulk cleanup endpoint exists to keep the list readable; run it periodically. An expired token is not a security risk by itself, but a long expired list hides genuine pending ones in the noise.

29.6 — Recap and next step

You can now grow the team safely. The takeaway: invitations are the team's onboarding ramp; they are tied to an email, scoped to groups, time-limited, and they default the new user to the safest role. Promote later, deliberately.

The next chapter introduces the last team-scale primitive — Git sync. Once your team's library matters enough that losing it would hurt, you want it backed up to a Git remote and you want the ability to restore a single prompt from a previous commit.

Chapter XXX — Git sync

Your library now matters. Personal templates, group templates, three collections with inheritance chains, a tag taxonomy you maintain on behalf of the team, an invited teammate using all of it daily. Losing this would hurt. Git sync is IziBuddy's answer to that fear: a per-user (or per-group) Git repository that mirrors the dashboard state, commits on every meaningful change, and pushes to a remote of your choice. This chapter teaches you how to enable it, how to audit what got committed, and — most importantly — how to restore one prompt from a previous commit.

If your install uses a database backend, this chapter is informational only. *Git sync is wired to the file-based storage backend; on installs configured with [database] set to PostgreSQL, MySQL, MongoDB, etc., the database itself handles cross-backend consistency and the Git panel is hidden from the dashboard. The concepts below — auto-commit on save, audit through commit history, restore from a past commit — still describe how things work on file installs, which is the reference deployment for the learning path.*

30.1 — Goal

By the end of this chapter you will have enabled Git sync, configured a remote, observed the first push, walked through the commit history, and restored a single prompt from a past commit without disturbing the rest of the library.

30.2 — Prerequisites

- Chapter XXIX: your team is set up; backing up a one-user library before the team exists is premature optimization.

- Access to a Git host you can push to (GitHub, GitLab, a self-hosted Gitea, etc.) and an SSH key or token configured in the environment IziBuddy runs in. The dashboard does not embed credential management; it uses whatever the OS provides.
- A short Manual-style summary, here so you do not leave the book: Git sync is a per-user (or per-group) repository whose working tree is the storage path of the corresponding library. The `GitManager` class in `src/core/git_manager.py` runs `git init`, `git add`, `git commit`, `git push`, and `git pull` against that path; every save in the dashboard triggers an `auto_commit()` whose message describes the change. The dashboard owns state; Git is a mirror — never the other way around.

30.3 — Walkthrough

Check Git status. Open the *Git* panel in the dashboard sidebar. The endpoint `GET /api/git/status` returns whether Git is enabled and whether a remote is configured. On a fresh account both are false; the manager exists but its `_enabled` flag is false and every operation no-ops until you turn it on.

Initialize the repository. Click *Enable Git sync*. The dashboard calls `POST /api/git/init`, which runs `git init`, writes a `.gitignore` (excludes `*.tmp`, `*.bak`, `.DS_Store`), and creates the first commit «Initial commit - Git tracking enabled». The panel now shows «enabled, no remote».

Configure a remote. Click *Add remote*. The dialog asks for a URL (HTTPS or SSH) and a name (default `origin`). Submit; the endpoint `POST /api/git/remote` runs `git remote add origin <url>`.

First push. Click *Push*. The endpoint `POST /api/git/push` runs `git push -u origin HEAD`. If your SSH key or token is healthy, the first push succeeds and your library is now mirrored to the host. If it fails, the response includes the `stderr`; the most common cause is a missing or wrong key — fix the environment, not the dashboard.

Trigger an auto-commit by saving a prompt. Make a small edit and save. Behind the scenes, the dashboard calls `GitManager.auto_commit()`: it stages the affected files, runs `git commit -m "<message> (<timestamp>)"`, captures the SHA, and — if a remote is configured — runs `git push -u origin HEAD` with a 30-second timeout. The result dict includes `committed`, `pushed`, and `push_error`; any push error surfaces in the Git panel.

Browse the commit history. Open the *History* tab. The endpoint `GET /api/git/log?limit=20` returns the most recent commits with SHA, author, timestamp, and message. Each commit corresponds to one save.

Inspect one commit. Click any commit. The endpoint `GET /api/git/diff/{commit_hash}` returns a standard unified diff — green for added lines, red for removed lines — not the structured `DiffResponse` from Chapter XXVI.

Restore one prompt from a past commit. This is the move that makes Git sync valuable on day one. Suppose three commits ago, your `code-reviewer` prompt was in a state you now want back, but you do not want to undo the eight other commits since then. The dashboard does not embed a one-click restore for this case; the operation is a shell move on the repository path. SSH into the host where IziBuddy runs and, in the user's storage directory:

```
git show <commit_sha>:path/to/code-reviewer.txt > /tmp/restored.txt
```

Then in the dashboard, open the prompt, replace its body with the content of `/tmp/restored.txt`, and save. The save triggers an auto-commit, so the restoration itself is now in the Git history with a clear message. One prompt restored, eight other commits intact.

Pull (rare). `POST /api/git/pull` runs `git pull` against the configured remote. This is the *rare* operation — useful after a host migration or an emergency remote-side edit. In normal operation, data flows from dashboard to Git, not the reverse.

30.4 — Hands-on exercise

Restore one prompt from a previous commit :

1. Pick a prompt you have edited at least three times since enabling Git sync. Open its history in the Git panel.
2. Pick a commit at least three back. Read its diff; note the prompt's content at that point.
3. SSH into the host (or use whatever shell access you have to the IziBuddy storage). Run `git show <sha>:<path-to-the-template-file>` and save the output.
4. Back in the dashboard, snapshot the *current* state of the prompt as a version (Chapter XXVI discipline applies here — never restore without snapshotting).
5. Paste the restored content into the prompt body and save.
6. Open the Git panel. Confirm the new commit appears with a message that includes the prompt's name. The restoration is itself part of the audit trail now.

30.5 — Pitfall and how to avoid it

The most common mistake at this level is **treating Git as the source of truth and the dashboard as a viewer**. Some users come from a Git-first workflow and instinctively want to edit templates in their checkout, commit, and pull on the dashboard. This works *sometimes*, then breaks the day the dashboard auto-commits while you have uncommitted changes locally — you get a merge conflict in a file format the dashboard does not expect to see in a conflicted state. The rule is absolute: the dashboard owns state. Edit in the dashboard, push to Git, treat the Git side as a read-only mirror plus an audit log. The only exception is emergency restore (the exercise above), and you do not commit your local change — you paste it back through the dashboard.

The second pitfall is **ignoring push errors**. A commit that failed to push is still in the local repo; the next ten commits will pile up locally until you notice and fix the remote. The fix is to subscribe to

whatever alerting your team uses for the `push_error` field in the auto-commit result. The dashboard surfaces the error in the Git panel ; check it weekly.

The third pitfall is **mixing personal and group syncs in the same remote**. A user with a personal library and three group memberships has four logical libraries, each with its own GitManager. Pushing all four to the same remote branch is a recipe for confusion ; either give each library its own repo or use distinct branches per library and document the mapping. The quarterly restore drill in Chapter XXXIII is what catches mis-mapped remotes before they matter — a backup that has never been restored is not a backup, it is a hope.

30.6 — Recap and next step

Backup is solved. You have a Git repository mirroring your library, a remote you can survive a disk failure with, an audit trail tracking every save, and a working procedure for restoring a single prompt from history. The dashboard owns state ; Git is the mirror and the audit log ; you have one quarterly drill to keep on the calendar.

The next chapter opens the lab — tests, scoring, A/B experiments. With backup behind you, measurement is the next layer of maturity.

30.7 — Continuous sync between two backends

The scenario worth one extra walkthrough: same user, two machines, one git remote. Laptop and a small cloud VM both running IziBuddy against the same `git_remote` URL. Without extra config, each backend pushes its own commits but never pulls the other's —

so the laptop sees the VM's templates only when you click *Pull* manually. Two switches in the `[git]` section of `izibuddy.ini` (or the *Auto-sync* block of the *Git Settings* modal) fix that:

- `sync_on_startup = true` runs `git fetch + git pull --ff-only` once when the backend boots. Catches up on overnight commits made from the other machine.
- `sync_on_read = true` runs the same fetch + pull before every prompt open, copy or render. Throttled by `sync_throttle_seconds = 10` so opening five templates in a row triggers one fetch, not five.

Keep `pull_strategy = ff-only` and accept the occasional manual resolution rather than letting auto-merge invent content. If a non-fast-forward shows up, the read proceeds on the local view with a warning in the log — go to *Git Settings* and resolve explicitly. This is the price of running the same library on two machines without a shared database; if it becomes annoying, that is the signal to migrate to a real DB backend (Chapter XXVI of the user manual covers the trade-off).

Chapter XXXI — The lab — tests, scoring, experiments

Up to here, the book has taught you to *write*, *organize* and *share* templates. This chapter teaches you to *measure* them. The dashboard hides — by default — a small experimentation surface that engineers recognize immediately and that most users never need. It is gated behind two features (`labTests` and `viewScoring`), and turning them on changes the way you decide whether a change to a prompt is an improvement or a regression.

31.1 — Goal

By the end of this chapter you will have enabled the lab, run the built-in test-fixture suite, read the quality score on one of your templates, and designed a small A/B comparison between two parameter sets on the same prompt.

31.2 — Prerequisites

- XXX — you have a Git remote configured. Tests happily live alongside your templates and you want them in version control from day one.
- You are at least on the Advanced preset. `labTests` is an Expert-preset feature and `viewScoring` a Standard one; you can enable either individually from *Options* → *Features* without committing to Expert.
- One prompt with at least two parameter sets (built in XVII). The A/B exercise needs two bindings to compare.

31.3 — Walkthrough

Enable the lab features. Open *Options* → *Features* and turn on two toggles: `labTests` (the test runner, in the *Advanced* category) and `viewScoring` (the score badge in the list and the editor, in the *Display* category). They are independent on purpose — you can score without running fixtures and vice versa — but the daily workflow uses both.

Open the score panel for one prompt. Pick a prompt you know well. Open it in the editor. With `viewScoring` on, a small *Score* badge appears next to the name in the toolbar. Click it. The panel shows a single quality **score out of 10** and a badge, computed by the analysis engine (`src/scoring/engine.py`) for a chosen *goal*. That one number is a weighted measure of which recommended sections the prompt actually contains, combined with coherence checks across those sections. The score is *not* a verdict — it is a starting point.

Read the goal first. The score is always relative to a *goal* (`PRODUIRE`, `ANALYSER`, ...): the goal defines which sections are expected and how heavily each one weighs. The goals are listed by `GET /api/scoring/goals`; the default is a generic production goal. Switch the goal and watch the score recompute against a different section standard — the same prompt can score well for one goal and poorly for another.

Run the built-in fixtures. Click *Run tests* in the editor toolbar (the button appears only with `labTests`). The lab runs its **built-in** fixture suite and shows green/red badges. Each fixture is a self-contained scenario embedded in the app source (`src/api/test_fixtures.py`): a small POO²L render with fixed parameter and system-variable values and the exact expected output. The Lab exercises the backend render stage — you *run* the suite, you do not author it, and there is no « save this binding as a fixture » button.

Watch the score across edits. Edit the prompt, re-open the score badge, and note how the number moves. A prompt that gains a missing recommended section, or tightens an incoherent one, climbs; the honest signal is the *trend* across your edits, not the absolute number on any single reading.

31.4 — Hands-on exercise

Design a small A/B between two parameter sets on the same prompt.

1. Pick a prompt that has two parameter sets (for example *marketing_tone* and *engineering_tone* from XVII).
2. In the render preview, bind the prompt with the first parameter set, then with the second. The expected output should differ — that is the whole point.
3. Read the quality score of each *rendered output*. The engine scores text, so it scores renders, not templates.
4. Edit the prompt's wording in a way you suspect will improve one tone (for example tighten the bullet-list format). Re-render both bindings.
5. Compare the score deltas. If both tones improved, the edit is a win for the template. If only one improved, the edit favoured one tone at the expense of the other — that is a design choice, not a bug, but it should be conscious.

Bonus : run the built-in fixture suite before and after your edit to confirm you did not break a render the app already checks.

31.5 — Pitfall and how to avoid it

The most common lab mistake is **optimising the score number**. The score is a weighted sum of section presence and coherence; it can be nudged up by padding a prompt with recommended-but-empty section headers a reader will hate. Two rules of thumb :

- **Readability beats a higher score.** If a sentence reads awkwardly but the score ticked up, revert.

- **Use the trend, not the absolute number.** A prompt scoring 7.2 today is not «worse» than one scoring 8.1; an edit that drops a *given* prompt from 8.1 to 7.2 is probably a regression on *that* prompt.

The second-most-common mistake is **ignoring a red fixture**. The built-in suite is a regression check on the render engine; a fixture that turns red after you upgrade or reconfigure something is telling you a render changed. Read it before dismissing it.

31.6 — Recap and next step

You have enabled `labTests` and `viewScoring`, read a prompt's quality score, run the built-in fixture suite, and compared two parameter sets in a small A/B. The lab is now part of your workflow. Used sparingly it pays for itself; used too eagerly it turns prompt authoring into score chasing — keep an eye on that.

The next chapter introduces Expert mode itself. You have been turning on Expert-only features one at a time since this part began; the next chapter shows you the rest of the surface in one sitting and helps you decide what to keep.

Chapter XXXII — Discovering Expert mode

You have spent the last few chapters turning on higher-tier toggles one at a time: `versioningDiff` for the diff viewer, `tagsAdmin` for the tag administration panel, `labTests` and `viewScoring` for the lab. Each was justified by a chapter. This one is different: it invites you to see the *whole* Expert surface at once, decide what you actually want, and walk away with a configuration that fits your hands — not the preset’s hands.

Tied to tip: `expert-discovery` — the dashboard fires this tip when a user on the Standard or Advanced preset crosses 200 templates. The reasoning is statistical, not prescriptive: by the time a library reaches that size, at least a handful of the Expert-only features pay for themselves. The tip simply offers the *applyExpert* shortcut. This chapter does the same thing slowly, with commentary.

32.1 — Goal

By the end of this chapter you will have switched to the Expert preset for one full session, toured the features that the previous chapters did *not* surface, and ended the session with a configuration that is either Expert, Advanced-plus-extras, or back to Advanced — all three are legitimate outcomes. The point is intent, not adoption.

32.2 — Prerequisites

- XXXI — you have run the fixture suite and read a score, so two of the Expert-only features are already familiar.
- Two hundred templates (roughly). The exercise still works with fewer, but several panels you will tour are only interesting at scale.

- Twenty minutes of uninterrupted dashboard time. Switching presets rearranges chips and panels; mid-task is the wrong moment.

32.3 — Walkthrough

Take a snapshot of where you are. Before flipping anything, open *Options* → *Features*. Note the preset selector’s current value (Standard or Advanced) and skim the feature list — the toggles that are *off* now are the ones the next move will switch on. This five-second pause is not paranoia; it is the only way you can later tell which new chips were already familiar from your previous preset and which arrived with Expert.

Apply Expert. In the preset selector, pick *Expert*. The dashboard reapplies the preset by writing every feature id from the registry into the active set — `PRESETS.expert` is literally `FEATURES.map(f => f.id)` in `src/static/index-features-registry.js`, so nothing is hidden. The page does not reload; the change takes effect through CSS class toggles on `<body>`.

Tour the filter chips. The filter bar now carries the full set: *Important*, *Locked*, *WIP*, *Note*, *Hidden*, *Disabled*, *Template*, plus the container filters *Folder*, *Collection*, *Template*, the *Clear all* chip, and crucially the *Exclude* mode (`filtersExclude`), which enables the exclude state in ordinary attribute buttons: neutral, include, exclude. This lets you select everything except WIP items. Hidden instead always cycles through normal (non-hidden only), green (hidden only), and grey (all). The grey ghost remains clickable; reset restores non-hidden only. Use `flag:hidden` for hidden items and `show:hidden` to include them without removing other criteria. Enter a hidden container to browse its contents without flattening subfolders. Container badges show the total, or X of Y templates when a criterion is active.

Tour the display options. Two new display features arrived: `viewTableDensity` (a compact/normal toggle for the list view, useful when the library passes a few hundred) and `viewScoring` (already on if you completed the lab chapter). Try the dense table density for ten seconds, then revert if your eyes prefer the normal one. This is exactly the kind of feature where personal taste wins.

Tour the container UI. Four new container features became visible: `containerKindUI` (icons that distinguish prompt/folder/ collection at a glance in lists where everything used to be a name), `containerPromoteDemote` (the *Promote folder* → *collection* and *Demote collection* → *folder* actions), `containerPathDrop` (drop a prompt onto a breadcrumb segment to move it up the container path), and `containerNewFolder` (already on under Standard). Of these, `containerKindUI` is the one that survives almost everybody's post-Expert review.

Tour the editor's POO²L surfaces. They are not features, so they are not in this list and no preset switches them on: they follow the **editing-level ceiling** (*Options* → *Editor* → *Input*). Two stand out — the **Structure** tab (a structural outline of the prompt's sections, clickable for jump-to, from L1) and the inheritance chain displayed at the top of the editor for any prompt that declares `inherits` (L4). The second matters most for libraries with deep chains. Raising the ceiling is a different gesture from adopting a preset, and deliberately so: one question, one control.

Tour the action features. `actionCopyPaste` and `actionCross-PaneF5F6` round out the keyboard-driven workflow. In the split editor, F5 copies the current selection to the other pane and F6 moves it — the clipboard-free shuttle you met in IX; Tab switches which pane has focus.

Tour the organization features. `sourcesGroups` adds a *Source* column that distinguishes personal templates from group-shared templates at a glance — a luxury at five templates, a

necessity at five hundred. `clickTagToFilter` turns every tag chip in the list into a one-click filter, which is the kind of detail you do not notice until you have it for an hour and then resent its absence.

Cherry-pick. Now open *Options* → *Features* again. For every feature you do *not* want, flip its toggle off. You are no longer on the Expert preset — the preset selector silently moves to *Custom*. This is the intended outcome: presets are starting points, not destinations.

32.4 — Hands-on exercise

Spend one hour using the dashboard with Expert mode fully on. Do real work — not a tour — so that you bump into the new chips and panels in the course of tasks you would have done anyway.

At the end of the hour, do the following in *Options* → *Features* :

1. For each new feature that proved useful, leave it on.
2. For each new feature you ignored, turn it off.
3. For each new feature you actively disliked (it added noise, distracted you, slowed a familiar gesture), turn it off and write its id on a piece of paper. Try it again in three months when your library has grown — taste shifts with scale.

End the session with either :

- Expert preset still selected (you kept everything),
- a *Custom* set (you cherry-picked),
- Advanced preset reselected (you reverted — also fine, the lab features can be turned back on individually).

Bonus: if you reverted to Advanced, list the *three* Expert features you would re-enable individually right now. That list is your personal expansion path.

32.5 — Pitfall and how to avoid it

The classic pitfall is **staying on Expert by inertia**. The preset turns on every feature in the registry — currently forty-nine — and two or three of them are noise for any given person. A user who stays on Expert because reverting feels like a demotion ends up with a busier UI than they need.

The cure is the cherry-pick step. The features panel keeps no « Expert mode achievement » anywhere. Custom is the most honest preset : it says « I made a deliberate choice. » The dashboard treats Custom and Expert identically for visibility purposes ; the only visible difference is the label in the preset selector.

The second pitfall is **discovering Expert mode on a deadline**. The preset rearranges chips ; if you flip it on five minutes before a demo and your muscle memory is searching for *Important* on the left and finds it on the right, you will lose time. Reserve the discovery to a slow afternoon.

32.6 — Recap and next step

You have toured every Expert-only feature, kept the ones that earn their place, and discarded the rest. Whatever preset label sits in your selector now — Expert, Custom, or back to Advanced — it sits there *intentionally*. That is the point of this chapter.

The next chapter is the final one : it turns everything you have learned into a small set of recurring habits — weekly, monthly, quarterly — that keep a healthy library healthy.

Chapter XXXIII — Your maintenance routine

The last chapter is not about a new feature. It is about the small set of recurring gestures that keep a real working library healthy over months and years. None of the tasks below is new — what is new is the *cadence*: weekly, monthly, quarterly. A library that nobody maintains decays the way a code repository nobody prunes decays: slowly enough to ignore, then suddenly enough to hurt.

33.1 — Goal

By the end of this chapter you will have a written routine — three recurring tasks with three different cadences — scheduled in your calendar, and you will have run each of them at least once so you know how long they actually take. Your prompt library will be in better shape than it was an hour ago, and you will have a habit that keeps it that way.

33.2 — Prerequisites

- XXXII — you have a stable configuration of features and a working library of around 200 templates.
- XXX — a Git remote is configured. The quarterly restore drill needs it.
- A real calendar app. The chapter ends with three recurring events; you need somewhere to put them.

33.3 — Walkthrough

The routine has three loops. They are independent on purpose; if you skip a quarterly because life got in the way, the weekly is still doing its job.

The weekly loop — fifteen minutes, every Friday. The goal is *recent hygiene* : catching the trash that piled up this week before it becomes permanent furniture.

1. *Clean the Trash*. For every item older than seven days that you do not recognize, purge it. Trash that sits longer than a sprint is just clutter. Purge is irreversible at the trash slot, but the backup compresses to xz and rotates on the filesystem (`src/core/backup_manager.py`), so a system backup exists for a while.
2. *Review WIPs*. Filter by the *WIP* marker. For each WIP older than two weeks, promote (remove the marker, add *Important* or *Locked*) or trash. Lingering WIPs erode the meaning of the marker.
3. *Skim Recent*. Rename anything with a placeholder name like *New prompt 3*. The « name first » discipline from Chapter I is enforced here, after the fact.

The monthly loop — forty-five minutes, first Monday of the month. The goal is *taxonomy hygiene* : keeping the tags and the deprecation list from drifting.

1. *Audit tags*. Open the tag administration panel (Chapter XXVII). Sort tags by frequency. Look at the bottom of the list — any tag with one or two uses is a candidate for removal or merge. Look at the top of the list — any tag with more than fifty templates is too broad and is a candidate for splitting along a useful axis.
2. *Archive deprecated*. Identify the deprecated templates — they carry a `@@@deprecated` field and surface a warning at render (XXV). For each prompt that has been deprecated for more than three months and whose successors are stable, archive it : move it to an *Archive* folder, remove sharing if any, and keep the deprecation message pointing at the successor. The history survives in versioning ; the active list gets shorter.

3. *Review group ownership.* For each shared group (Chapter XXVIII), remove anyone who left the team and promote or demote roles that no longer match actual usage.
4. *Glance at the lab.* Run the full fixture suite from Chapter XXXI once. A chronically red test is either a bug or a stale fixture; either way, deal with it or delete it.

The quarterly loop — two hours, once per quarter. The goal is *structural hygiene*: the things you only need to do four times a year, but that you absolutely need to do.

1. *Rotate group ownership.* For each group where you are the sole admin, name a second admin. Single-admin groups are an operational risk; the day you cannot log in is the wrong day to discover this. The role propagation rules from Chapter XXIX make this a five-minute change.
2. *Review the Git remote.* Open *Options* → *Git* and check three things: the remote URL still resolves; the last push timestamp is recent; the branch name still matches what your team agreed on. The `GitManager` push/pull machinery in `src/core/git_manager.py` writes a clean commit history; if the history shows long gaps, the sync is silently broken. Push a manual commit if needed.
3. *Run the restore drill.* This is the single most important quarterly task. Pick a random non-critical prompt. Note its current state (open it, read it). Then, from the Git history, restore the version from three months ago into a new prompt (Chapter XXX). Verify that the restored prompt matches what the Git commit shows. Delete the test restore.

The drill is short but the only test that proves a backup works is restoring from it. A team that has never run a restore drill does not have backups; it has a folder of files that nobody has tried.

33.4 — Hands-on exercise

Open your calendar. Create three recurring events :

1. *IziBuddy weekly hygiene* — Friday, 16:30, fifteen minutes, weekly. Body of the event : « Trash, WIPs, Recent. »
2. *IziBuddy monthly audit* — first Monday of the month, 10:00, forty-five minutes. Body : « Tags, deprecated, groups, lab. »
3. *IziBuddy quarterly drill* — first Monday of January, April, July, October, 14:00, two hours. Body : « Rotate admins, check Git, restore drill. »

Then run each of them once, today and now, in compressed form (you do not need the full hours ; you need to feel the shape of the task). Time each loop. If your library is small, the weekly will take five minutes — adjust the calendar event accordingly. The quarterly always takes longer than expected the first time and shorter than expected from then on.

Bonus : if you maintain a shared group, schedule the monthly and the quarterly on a day when at least one teammate is around.

33.5 — Pitfall and how to avoid it

The classic pitfall is **the unrun quarterly drill**. The weekly stays alive because the cost of skipping it is immediate. The monthly mostly survives. The quarterly is the one that gets skipped — quietly, then loudly, then in retrospect.

Two countermeasures : schedule it in a calendar with a notification (not in your memory), and pair it with something you cannot skip — for example the morning before your team's quarterly planning meeting.

The second pitfall is **expanding the routine until it becomes a chore**. If your monthly grows to two hours, you will skip it. Cut ruthlessly : anything you add must displace something.

33.6 — Recap and next step

You have a weekly loop (fifteen minutes), a monthly loop (forty-five minutes), and a quarterly drill (two hours). The calendar has them. You have run each once. Your library is in better shape than it was, and the routine that keeps it that way is now scheduled, not aspirational.

This is the last chapter. The appendices that follow are reference material: exercise solutions (Appendix A), a thirty-day plan (Appendix B), the reverse feature-to-chapter index (Appendix C), and the tips crosswalk (Appendix D).

Welcome to maintenance. The library is yours now.

Appendices

Appendix A — Appendix A — Exercise solutions

A short reference solution for each hands-on exercise. The solutions are *one* viable answer ; the exercises were designed to admit many. If yours differs from the one below and works, yours is correct.

A.1 — Chapter 1 — Your first prompt

A working 30-line *Code reviewer*: opening line declares the role (You are an expert code reviewer.), one paragraph requests *at most three* issues, one paragraph requests *a concrete fix* per issue, one closing line requests *a one-line overall verdict*. Saved under the exact name `Code reviewer`. Inject with a ten-line Python snippet underneath. If the response comes back with four issues, the prompt is too soft about the *at most* limit — strengthen the wording.

A.2 — Chapter 2 — Undo and recover

Three gestures, in this order: delete *Code reviewer*; confirm the Trash counter incremented ; *Restore from Trash* ; verify the prompt reappears in the active list with no edits lost. Then duplicate the restored *Code reviewer* (it becomes *Code reviewer (copy)*) and trash the *copy*. The original keeps its name, its versions, and its position.

A.3 — Chapter 3 — The Stash

Three templates marked as stashed, then the chat tab reloaded and the stash tray inspected at the top of the *Templates* tab of the extension. Stashing tags each prompt `STASH` on the server, so the tray persists across reloads and browsers rather than living only in the extension.

A.4 — Chapter 4 — The extension in ten minutes

Panel repositioned (top-right or bottom-left both work), theme switched to a non-default value, the *ChatGPT pink* profile selected from the Profiles tab. Verify by reloading the chat tab — the position, theme and profile survive the reload.

A.5 — Chapter 5 — Markers

Ten templates marked. A reasonable distribution: roughly half *Important*, two or three *Locked* (the templates you have committed to and do not want anyone editing), one or two *WIP* (drafts you are still working on). If you marked every prompt *Important*, the marker has lost its meaning — keep *Important* for the top quarter at most.

A.6 — Chapter 6 — Folders

A four-folder taxonomy along *one* axis. Example: `code/`, `writing/`, `meetings/`, `personal/` (axis: domain). Or `drafts/`, `production/`, `archive/`, `templates/` (axis: lifecycle). Mixing the axes — `code/`, `production/`, `personal/`, `drafts/` — works for one week, then collapses.

A.7 — Chapter 7 — Tagging

A tag taxonomy orthogonal to the folder taxonomy. If folders are domain (`code/`, `writing/` ...), tags can be audience (`#team-internal`, `#external-client`, `#self`). Two axes, two filters, no duplication.

A.8 — Chapter 8 — Filtering and searching

Three composite filter recipes recorded (the dashboard keeps active filters in local storage, not in the URL, so note the chip combination rather than bookmarking a link). Example recipes: *Important* + *code/*, *WIP* + *this week*, *Locked* + *production/*. Each takes under five seconds to reapply.

A.9 — Chapter 9 — Split-pane editing

Source in the left pane, destination in the right pane. Select the source paragraph and press F5 to copy it into the other pane (F6 moves it). No system clipboard involvement. The accent border marks the focused pane; Tab switches focus between the two editors. F5/F6 need the `actionCrossPaneF5F6` feature (Expert preset, or enabled individually).

A.10 — Chapter 10 — Versioning

Three versions, each with a meaningful note: `baseline 2026-01-15`, `tightened`, `three-step restructure`. Diff v2 vs v1 shows deletions only. Diff v3 vs v2 shows reorganized structure. Rollback to v2 — v3 stays in the history, ready to be recovered.

A.11 — Chapter 11 — Milestone (Simple → Standard)

Preset selector moved from *Simple* to *Standard*. New chips visible: *Important*, *Locked*, *WIP*, *Note*, the container filters. Ten minutes spent clicking each chip and reading its tooltip. Stay on Standard at least a week before any further preset change.

A.12 — Chapter 12 — Sections (L1)

A five-section conversion of a previously monolithic prompt. Recommended sections for a « code reviewer » template: `role`, `context`, `task`, `format`, `examples`. Identifiers without spaces (`code_format`, not `code format`).

A.13 — Chapter 13 — Parameters (L3)

Three parameters absorb the variation that used to require duplicates. Example for a «summary writer»: `audience`, `length`, `tone`. The duplicates are now deleted; the single parameterized prompt covers all three originals.

A.14 — Chapter 14 — Constants

Three parameters moved to constants because their values never varied across calls. Example: `language: text = "English"`, `format: select("markdown", "plain") = "markdown"`, `max_examples: number = 3`. The injection dialog no longer asks for them.

A.15 — Chapter 15 — Includes

A *glossary* prompt extracted as its own file, included from two consumer templates via `@@@{include glossary}`. The rendered output of each consumer carries the glossary at the include site. Editing the glossary updates both consumers on next render — the whole point.

A.16 — Chapter 16 — Attachments

A *describe this picture* prompt with one `to-include` attachment of type `png`. At injection time the dashboard asks for the picture; at render time the picture is embedded inline in the message sent to the chatbot.

A.17 — Chapter 17 — Parameter sets

A prompt with two named parameter sets: *Marketing tone* (audience=consumer, tone=playful, length=short) and *Engineering tone* (audience=engineer, tone=precise, length=medium). At injection the picker offers both; choosing one fills the dialog.

A.18 — Chapter 18 — Rendering and injection

`@@@=DATE` and `@@@=USER` substituted at render time. Inject twice in the same day: same date, same user; inject tomorrow: new date. The render preview panel shows the final output before any chatbot receives it.

A.19 — Chapter 19 — Inheritance (L4)

A three-level chain: `base_reviewer` (generic), `code_reviewer` (inherits from `base_reviewer`, overrides the *role* section to mention code), `python_reviewer` (inherits from `code_reviewer`, overrides *examples* with Python idioms). Render the leaf to verify all three levels collapse correctly.

A.20 — Chapter 20 — `parent` and `self`

Three variants of the same override on `python_reviewer`'s *examples* section: pure override (parent content gone), extend via `@@@{include parent.examples}` followed by Python-specific text (parent content kept), and merge (parent then child interleaved). Compare the three renders side by side.

A.21 — Chapter 21 — Folder to collection

A folder of four related templates promoted to a collection. The collection gains its own `@@@collection` block (inherits parent, public member listing, collection-level parameter sets), and the four members are now methods. Promoting too early (one or two members) is the pitfall the chapter warned about.

A.22 — Chapter 22 — Virtual templates

A `prose_reviewer` collection derived from a `reviewer` base whose `summarize`, `score`, and `suggest` members each carry the `virtual` modifier on their `@@@template` header. The derived collection implements all three. The linter is silent — that is the success criterion.

A.23 — Chapter 23 — Collection inheritance

A third collection (`legal_reviewer`) derived from the same `reviewer` base. Its `summarize` member implements the virtual contract and reuses a concrete shared member of the base via `@@@include ^parent.preamble` before adding legal-specific content. The render shows the base preamble followed by the legal-specific additions.

A.24 — Chapter 24 — Aliases

One alias-protected rename of a heavily-referenced prompt. The old name becomes an alias; every `@@@{include old_name}` in your library still resolves; the editor warns about callers that could be updated to the new name but does not break them.

A.25 — Chapter 25 — Deprecation

A `prompt` deprecated with the message `Use code_reviewer_v2 instead – same parameters, tighter format.` Two callers migrated by hand. The render warning fires at instantiation and at include-time so nothing slips through.

A.26 — Chapter 26 — Diff and rollback

A regression diagnosed by diffing the current version against last week's. The diff shows the change that introduced the regression. Rollback applied with a «before rollback» snapshot saved first — the audit trail remains complete.

A.27 — Chapter 27 — Tag administration

Three overlapping tags merged into one. Example: `#draft`, `#in-progress`, `#wip` merged into `#wip`. The merge propagates to every tagged prompt in one transaction; the version history records the renames.

A.28 — Chapter 28 — Groups and shared templates

A small group created (yourself plus two teammates), three templates shared at the group level. Both teammates, as members, can see and use the three shared templates; a group-admin can moderate the group and its sharing. There are no editor/viewer roles — the model is members plus group-admins (with a single system-admin over the instance).

A.29 — Chapter 29 — Invitations and onboarding

A teammate invited via email, the invitation accepted, the teammate now visible in the group as a member (grant group-admin later if appropriate).

A.30 — Chapter 30 — Git sync

A single prompt recovered from a previous commit by reading its content out of the Git history and pasting it into a new version. There is no one-click *Restore from Git* button (XXX is explicit about this): the dashboard owns state and Git is a mirror, so recovery is a manual copy. Snapshot the current state first — the rollback safety rule applies here too.

A.31 — Chapter 31 — The lab

A small A/B: the same prompt rendered with each of its two parameter sets, and the single quality score read for each rendered output (the built-in fixture suite is run separately as a regression check

— you do not author fixtures). An edit that moved both scores up is a template win ; an edit that moved one up and the other down is a deliberate tone trade-off.

A.32 — Chapter 32 — Discovering Expert mode

One hour with Expert mode on, ended with a deliberate decision : some features kept, some turned off. The preset selector now reads *Custom* (or *Advanced* if you reverted, or *Expert* if you kept everything). Three Expert features re-enabled individually if you reverted — that list is your expansion path.

A.33 — Chapter 33 — Maintenance routine

Three recurring calendar events created — weekly, monthly, quarterly — and each run at least once today in compressed form. The quarterly restore drill performed on a non-critical prompt ; the restored content matches the Git commit. The library is now in better shape than an hour ago and the routine that keeps it there is scheduled.

Appendix B — Appendix B — A 30-day plan

This appendix maps the thirty-three chapters of the book to a thirty-day reading schedule. The pace is deliberate : a chapter takes thirty to sixty minutes including the hands-on exercise, and the plan front-loads the early parts where the gestures are new and back-loads the later parts where you are mostly practicing what you have already seen.

The plan assumes one work-week of reading per part, with the weekends used as catch-up or rest. The *check-in* days are breath-points where you confirm the previous part stuck before moving on — they are also the moments the book’s parts naturally transition.

B.1 — The plan

Day	Chapter	Task
1	Chapter I	Write your first prompt ; inject it into a chat tab.
2	Chapter II	Delete, restore, learn the trash lifecycle.
3	Chapter III	Stash three templates in the extension.
4	Chapter IV	Tour the floating panel ; change theme and position.
Check-in	—	Part I done. You can author a prompt and inject it. Take the rest of the day off the book ; use what you learned on real work.

Day	Chapter	Task
5	Chapter V	Switch to Standard ; mark ten templates.
6	Chapter VI	Design a four-folder taxonomy.
7	Chapter VII	Add an orthogonal tag taxonomy.
8	Chapter VIII	Bookmark three composite filters.
9	Chapter IX	Edit two templates side by side without Ctrl-C.
10	Chapter X	Produce three meaningful versions and a roll-back.
11	Chapter XI	Confirm Standard preset ; tour the new chips.
Check-in	—	Part II done. You have a working organizational system. Stay on Standard for at least one week.
12	Chapter XII	Convert one monolithic prompt to five sections.
13	Chapter XIII	Collapse a duplicated prompt into a parameterized one.
14	Chapter XIV	Move three would-be parameters to constants.
15	Chapter XV	Extract a glossary ; include it from two templates.
16	Chapter XVI	Build a <i>describe this picture</i> prompt.

Day	Chapter	Task
17	Chapter XVII	Bundle two parameter sets on the same prompt.
18	Chapter XVIII	Use <code>@@@=DATE</code> and <code>@@@=USER</code> ; inject twice.
Check-in	—	Part III done. You can engineer reusable templates. Spend an evening converting two of your favorites to the L1-L3 idioms.
19	Chapter XIX	Build a three-level inheritance chain.
20	Chapter XX	Compare override / extend / merge on one section.
21	Chapter XXI	Promote a real folder to a collection.
22	Chapter XXII	Declare three virtuals ; derive a collection.
23	Chapter XXIII	Derive a second collection from the same base.
24	Chapter XXIV	Perform one alias-protected rename.
25	Chapter XXV	Deprecate a prompt ; migrate two callers.
Check-in	—	Part IV done. OO ² is in your hands. Resist the urge to refactor everything — only convert what you would naturally re-edit anyway.
26	Chapter XXVI	Diagnose a regression with diff ; rollback safely.

Day	Chapter	Task
27	Chapter XX-VII	Merge three overlapping tags.
28	Chapter XX-VIII	Create a sandbox group ; share three templates.
29	Chapter XXIX	Invite a teammate.
30	Chapter XXX	Enable Git sync ; first push.
Check-in	—	Part V (early) done. The remaining chapters do not have to land on consecutive days — they finish the book at a slightly slower pace.
31	Chapter XXXI	Enable <code>labTests</code> and <code>viewScoring</code> ; run a fixture.
32	Chapter XXXII	One hour on Expert ; cherry-pick.
33	Chapter XXXIII	Schedule the three loops ; run each once.
Check-in	—	Book done. Open Appendix A and compare your exercise outputs with the reference solutions. Open Appendix C and confirm every feature you actually use was learned.

B.2 — How to adapt the plan

The plan above assumes you can spare one chapter per day. Real calendars rarely cooperate. Three legitimate adjustments :

- **Stretch.** Double every day. The plan now spans sixty days, which is still fast — the book has been read in six months with no loss of comprehension.
- **Compress Part III.** Chapters 12-18 are the most consequential in the book and they reward slow reading. Do not compress them. Compress Part V instead — the last three chapters can fit in one weekend.
- **Pause at the check-ins.** The check-in days are *not* optional — they are where the previous part settles into muscle memory. Skipping them is the most common cause of «I read the book but I do not remember any of it.» A check-in day with no new chapter is still a productive day.

B.3 — If you have less than 30 days

Pick one chapter per part — typically Chapter I, Chapter XI, Chapter XIII, Chapter XXI, and Chapter XXXIII. Read them in order over a weekend. You will not be a power user on Monday, but you will know enough to grow into one.

Appendix C — Appendix C — Reverse index — feature → chapter

The dashboard ships with forty-four toggleable UI features grouped into nine categories, as declared in `src/static/index-features-registry.js`. The four presets — Simple, Standard, Advanced, Expert — are starting points: each one enables a subset of the forty-four, and *Expert* is declared as the whole registry rather than as a list, so it can never fall behind it.

The editor’s POO²L surfaces are NOT in this index, because they are not features: they follow the editing-level ceiling (`pool_max_level`, Options → Editor → Input) — one question, one control.

This appendix lists every feature id, the category it belongs to, the first preset that enables it, and the chapter in this book that first invites you to use it. *First invited* is not the same as *only mentioned*: a feature may appear in several chapters; this table points to the chapter that teaches the gesture.

A handful of features ride the preset switch without their own dedicated chapter — `filterClearAll` is the most obvious example. For those, the chapter column points to the chapter that introduces the preset where the feature first surfaces, or to the Expert-mode tour in Chapter XXXII for features that appear only in Expert.

C.1 — Filters

Feature id	First preset	First invited in
<code>filterImportant</code>	Standard	Chapter V
<code>filterLocked</code>	Standard	Chapter V
<code>filterWip</code>	Standard	Chapter V

Feature id	First preset	First invited in
<code>filterNoScoring</code>	Standard	Chapter XI
<code>filterHidden</code>	Advanced	Chapter XXXII
<code>filterDisabled</code>	Advanced	Chapter XXV
<code>filterExample</code>	Advanced	Chapter XXXII
<code>filterFolder</code>	Standard	Chapter VIII
<code>filterCollection</code>	Standard	Chapter XXI
<code>filterTemplate</code>	Standard	Chapter VIII
<code>filterClearAll</code>	Simple	Chapter VIII
<code>filtersExclude</code>	Advanced	Chapter XXXII
<code>filterLevel</code>	Advanced	Chapter XXXII

C.2 — Display

Feature id	First preset	First invited in
<code>viewDetails</code>	Standard	Chapter XI
<code>viewTileSizes</code>	Standard	Chapter XI
<code>viewTableDensity</code>	Advanced	Chapter XXXII
<code>viewSplitPane</code>	Expert	Chapter IX
<code>viewScoring</code>	Standard	Chapter XXXI

C.3 — Containers

Feature id	First preset	First invited in
containerKindUI	Advanced	Chapter XXXII
containerPromoteDemote	Advanced	Chapter XXI
containerNewFolder	Standard	Chapter VI
containerNewCollection	Simple	Chapter XXI
containerPathDrop	Advanced	Chapter XXXII
collection002	Standard	Chapter XXIII
containerColorStripe	Standard	Chapter VI

C.4 — The editor's POO²L surfaces (no feature ids)

The parameters and constants dialogs, the parent selector and chain viewer, section management: none of them is a feature. They answer to the **editing-level ceiling** alone — the Structure and Verify tabs appear from L1, inheritance is L4, collections L5. Four pool21* toggles did exist; two drove nothing from the day they were added, and two contradicted the ceiling. They were removed rather than reconciled. Chapters XII, XIII, XIV and XIX teach these surfaces.

C.5 — Input (editor)

Feature id	First preset	First invited in
editorNotes	Standard	Chapter X
editorAttachments	Standard	Chapter XVI
editorGlyphReplace	Standard	Chapter XI

Feature id	First preset	First invited in
editorSmartQuotes	Expert	Chapter XXXII

C.6 — Actions

Feature id	First preset	First invited in
actionDragDrop	Simple	Chapter VI
actionCutPaste	Standard	Chapter IX
actionCopyPaste	Advanced	Chapter XXXII
actionUndoRedo	Simple	Chapter II
actionCrossPaneF5F6	Expert	Chapter IX
actionRubberBand	Standard	Chapter VIII

C.7 — Organization

Feature id	First preset	First invited in
tagsHierarchy	Standard	Chapter VII
tagsAdmin	Advanced	Chapter XXVII
sourcesGroups	Advanced	Chapter XXVIII
clickTagToFilter	Advanced	Chapter VIII

C.8 — Advanced

Feature id	First preset	First invited in
versioningDiff	Advanced	Chapter X
gitSettings	Expert	Chapter XXX

Feature id	First preset	First invited in
labTests	Expert	Chapter XXXI
chromeExtensionLink	Simple	Chapter IV
tipsEnabled	Simple	Chapter XI

C.9 — Reading the table

Four legitimate uses of this index :

- **You see an unfamiliar chip or toggle in the dashboard.** Find the feature id in the *Options* → *Features* panel (the toggle label maps to the id via the i18n keys), then look it up here to find the chapter that teaches it.
- **You picked the Expert preset and the UI feels noisy.** Open the table, find every Expert-first feature, and look up the chapter that introduces it. If the chapter has not been read, the feature is probably not earning its place yet.
- **You followed the 30-day plan in Appendix B.** This table is the post-plan audit: by day 33, every row should map to a chapter you have read.
- **You are auditing a teammate’s setup.** Open their *Options* → *Features* with their consent. Any feature on that is *off* in the Expert preset is impossible — Expert turns on all forty-four. Any feature off in their actual configuration is a cherry-pick worth a short conversation.

C.10 — On the *First preset* column

The *First preset* column reads from `src/static/index-features-registry.js` — the PRESETS map is the canonical source. Counts: Simple enables 6, Standard 25, Advanced 42, Expert all 49. The

progression is monotonic — moving up the preset ladder never removes a feature. Custom configurations are the only place the monotonicity breaks, and that is the intended freedom.

Appendix D — Appendix D — Crosswalk — tips.json triggers → chapter

The dashboard fires contextual tips at moments where most users benefit from a small, targeted hint. The eleven tips are declared in `src/static/data/tips.json` and surfaced by the engine in `src/static/index-tips.js`. The engine throttles tips to at most three per day and at least five minutes apart, so the worst case is a small interruption — never a flood.

This book teaches every tip’s topic *before* the tip fires. The table below is the audit: for each tip, the trigger condition, the chapter that pre-emptively teaches the same gesture, and a one-line description of what the tip suggests.

A reader who follows the 30-day plan in Appendix B will never see a tip surprise them — by the time the trigger fires, the topic is familiar.

D.1 — The eleven tips

Tip id	Trigger	Chapter	What the tip suggests
<code>poolevel-max-remin-der</code>	Every app boot.	Chapter XI	Reminds the reader that the editable POO ² L complexity ceiling is adjustable ; opens Settings.
<code>first-deletion-undo</code>	The first prompt deletion ever.	Chapter II	Reassures the reader that Trash is reversible ; no action button.

Tip id	Trigger	Chapter	What the tip suggests
promote-folders-50	Library reaches 50 templates while filter-Folder is off.	Chapter VI	Offers a one-click apply of the Standard preset to unlock folders.
simple-preset-many-templates	80+ templates while still on the Simple preset.	Chapter XI	Offers a one-click apply of the Standard preset ; the milestone moment.
duplicate-burst-pool21	Three duplicates within five minutes, on Simple or Standard.	Chapter XIII	Offers a one-click apply of Advanced — duplicates are a parameter smell.
promote-collections-3-folders	Three folders exist and zero collections.	Chapter XXI	Enables <code>containerPromoteDemote</code> so the user can promote a folder to a collection.
cut-paste-burst-split	Five cut-paste operations in ten minutes, split off.	Chapter IX	Enables <code>viewSplitPane</code> so the user stops bouncing between two open templates.
first-version-saved	A prompt accumulates two versions, <code>versioningDiff</code> off.	Chapter X	Enables <code>versioningDiff</code> so the history panel shows the diff.
no-tags-30-templates	30+ templates and zero tag usage, <code>tagsHierarchy</code> off.	Chapter VII	Enables <code>tagsHierarchy</code> and invites the reader to start tagging.

Tip id	Trigger	Chapter	What the tip suggests
standard-using-collections	Standard preset + at least two collections.	Chapter XXIII	Offers Advanced preset and enables <code>viewSplitPane</code> ; the OO ² invitation.
expert-discovery	200+ templates on Standard or Advanced.	Chapter XXXII	Offers a one-click apply of the Expert preset for users who have outgrown Advanced.

D.2 — How the engine decides

A tip fires when its trigger matches *and* every condition holds. Reading one row in detail — `duplicate-burst-pool21` :

- *Trigger*: the `duplicateBurst` event with `count: 3` and `windowMs: 300000` (five minutes). The engine counts duplicates in a rolling window and emits the event when the threshold is crossed.
- *Condition*: `presetIn: ["simple", "standard"]`. If the user already has POO²L parameters enabled, the tip is silent — they do not need the suggestion.
- *Action*: a single CTA button labelled *Apply Advanced preset* which calls `IziFeatures.applyPreset('advanced')` on click.

The same template applies to every row.

D.3 — Disabling the tips entirely

If you find any contextual hint distracting — even the polite, throttled, dismissable kind — the `tipsEnabled` feature in the *Advanced* category turns the system off entirely. It is enabled by every preset by default, including Simple, because the tips are designed to help newcomers most. Turning it off is a defensible choice for an expert who has read this book.

The tips engine also tracks a *dismissed* list locally ; tips you have closed do not return in the same session. The throttling quotas — five minutes minimum gap, three tips per day, never twice in one session — are hard-coded in `index-tips.js` and not user-configurable, by design.

D.4 — A reader's contract

The promise this book makes about the tips system is simple : **every tip the reader of this book might encounter has been taught in advance**. The table above is the verification. If a new tip ships in a future release of IziBuddy and is not in this table, the next edition of the book will add a chapter or a section that pre-empts it.

Resources

Download

- PDF format (03-learning-path-outline_v1.29_en_20261008.pdf)
- EPUB format (03-learning-path-outline_v1.29_en_20261008.epub)
- Markdown format (03-learning-path-outline_v1.29_en_20261008.md)

Lexique

Active (lifecycle state) — A prompt in normal use. The default state right after a prompt is created or restored from the trash. → Ch. II

Alias — An alternative identifier for a prompt that lets you rename the canonical `id` without breaking includes or inheritance that still use the old name. Declared as `@@@alias <old_id>`. → Ch. XXIV

Attachment — A file or binary blob bound to a prompt, described by two orthogonal axes. WHEN it is bound: `fixed` (bound at definition time) or `to-include` (file provided at injection time, required by default, optional with `?` suffix). SOURCE (only for `fixed`; where the content lives): `file` (a path), `blob` (an inline data URI), or `record` (an external store ref). A `to-include` attachment carries no source. → Ch. XVI

Bulk operations — Multi-selection actions on templates: multi-select with Ctrl/Shift click, drag-and-drop, cut/copy/paste across panes — all collapsed into a single `PUT /api/templates/update/bulk` request. → Ch. IX

Collection — A container of templates with POO²L Level 5 semantics: inheritance between collections, virtual templates, and `^parent.<template_id>` resolution on the collection axis. An entity is a collection because a `@@@collection <name>` block declares it in-language; a folder has no such block. Collections **nest** by containment (unbounded depth); a member's qualified id is the dotted containment path followed by its prompt id (`<collection>{.<sub-collection>}.<template_id>`). → Ch. XXI, Ch. XXIII

Collection inheritance — Declaring on a collection that it `inherits` from another. The derived collection includes every member of the parent and may override or extend them. The horizontal OO² axis. → Ch. XXIII

Constant — Named immutable value bound at definition time. Declared `@@@const <NAME> : <type> = <value>`. Same property shape as a parameter, but the value is fixed and does not appear in the injection dialog. → Ch. XIV

Continuous sync — Two file backends sharing one git remote, with `sync_on_startup` and `sync_on_read` enabled so each side automatically picks up the other's commits without a manual *Pull*. → Ch. XXX

Deprecation — Marker that announces a prompt is scheduled for removal. Declared `@@@deprecated "<migration message>"`. Does not prevent instantiation; surfaces a warning at injection, include and inherit time. → Ch. XXV

Diff viewer — The side-by-side comparison view between versions of a prompt, with a Line/Word granularity toggle (there is no unified mode). Reached from the version list. Supports rollback to a chosen version. → Ch. X, Ch. XXVI

Expert preset — The most feature-rich of the four UI presets. Enables every feature in the registry. Adopt it intentionally once your library is large enough to justify the noise. → Ch. XXXII

Feature — A granular UI element that can be turned on or off independently. The registry `src/static/index-features-registry.js` enumerates 44 features in 9 categories. → Ch. XI, Ch. XXXII

Filtering — Narrowing the visible prompt list with one or more cumulative criteria — text search, marker filters, tag filters, container restrictions. The `filtersExclude` toggle unlocks an *exclude* state on the filter chips (hide matching templates); `filter-ClearAll` resets them. Active filters live in the browser's local storage, not in the URL. → Ch. VIII

Final prompt — Template whose collection-qualified `@@@template` header carries the `final` modifier — *sealed* against override by any derived collection. Symmetric to virtual templates: a derived collection that declares a member with the same id raises OVER-

RIDE_OF_FINAL_TEMPLATE. Mutually exclusive with `virtual` on the same header (a header bearing both is `INVALID_SYNTAX`). → Ch. XXII

Final section — Section declared `@@@section final <name> :` sealed. No descendant may redeclare it (`OVER-RIDE_OF_FINAL_SECTION` fires otherwise). Symmetric to the `final` modifier on a `@@@template` header. → Ch. XIX

Folder — Pure organizational container, file-system-directory equivalent. Available at every POO²L level. A plain storage location with no `@@@collection` block declaring it, hence no compositional semantics. → Ch. VI, Ch. XXI

Git sync — Per-user or per-group Git repository that mirrors the dashboard state. Used for backup, audit, and disaster recovery — the dashboard remains the source of truth. → Ch. XXX

Group — Multi-tenant scope owning its own templates and tags. Two membership levels — members and group-admins (who moderate and invite) — plus a single system-admin over the instance. There are no editor/viewer roles. → Ch. XXVIII

Hands-on exercise — The concrete, self-contained task at the end of each walkthrough — designed to be done in your own dashboard with templates you actually care about, not toy examples. → Ch. I

Include — Reference mechanism that pulls another prompt's content (or a section of it) inline. Declared `@@@{include <template_id>}`. Supports parameter overrides on the call site. → Ch. XV

Inheritance (prompt) — A child prompt declares an inline `inherits <parent_id>` clause on its `@@@template` header and receives the parent's sections, parameters and constants. The child may override or extend any of them. POO²L Level 4. → Ch. XIX, Ch. XX

Injection — The act of sending a rendered prompt from the Izi-Buddy floating panel to the chatbot’s input area. The render pipeline runs **locally** — the extension embeds the same POO²L engine (compiled to WebAssembly) that the dashboard uses — and the result is typed into the chatbot. → Ch. XVIII

Invitation — Token sent by e-mail that opens access to a group or the platform. Has an expiry and may be revoked. Generated and tracked by `src/api/invitations.py`. → Ch. XXIX

Lab — The optional, opt-in experimentation surface. Holds saved tests, score-tracking, and small A/B harnesses on parameter sets. Enabled by the `labTests` feature. → Ch. XXXI

Maintenance routine — The weekly, monthly and quarterly habits that keep a library of templates healthy: trash cleanup, WIP review, tag audit, deprecation sweeps, Git restore drill. → Ch. XXXIII

Marker — Special label carried by a prompt and surfaced through a matching filter chip. Six markers: Important , Locked , WIP, Disabled, Template, Hidden. → Ch. V

Parameter — Named variable in a prompt that gets a value at instantiation. Required by default; optional with the `?` suffix on the keyword. Six standard types — text, textarea, number, date, boolean, select. → Ch. XIII

Parameter set — Named bundle of parameter values that can be applied to a prompt in one click. Declared inline as `@@@parameter-set <name> ... @@@end-parameter-set`; forbidden inside collection-scoped templates (where collection-level sets apply instead). → Ch. XVII

parent (POO²L reference) — Inheritance-aware include reference on the **prompt** axis (OO #1) that returns content from the *parent*, skipping the current prompt’s override. Forms: `parent`, `parent.<section>`, `parent.<prompt>.<section>`, used through `@@@include`. The collection axis (OO #2) is written with

the `^` prefix — `^parent.<template_id>`, `^parent.<template_id>.<section>` — reaching the parent *collection*'s member; the two axes are disjoint. → Ch. XX

Pitfall (in this book) — The single most common mistake at the chapter's stage of the learning curve, with the recovery move. One of the six recurring sections of every chapter. → Ch. I

POO²L — Template Object-Oriented² Language — the declarative language for the templates IziBuddy manages. Six maturity levels (L0 plain text, L1 sections, L2 includes, L3 structured data and validation, L4 inheritance, L5 collections). → Ch. XII, Ch. XIII, Ch. XIX

Preset — Ready-made feature profile. Four presets — Simple, Standard, Advanced, Expert — each enabling a curated subset of the 44 features. Adopting a preset is the canonical way to progress on the learning curve. → Ch. XI, Ch. XXXII

Pull strategy — How the dashboard resolves divergence when pulling from the git remote. `ff-only` refuses merges (safest, recommended), `rebase` replays local commits over the remote, `merge` creates a merge commit on divergence. → Ch. XXX

Render pipeline — The deterministic eleven-phase pipeline that turns a POO²L source into the text injected into the chatbot: parse, collection context, inheritance, section selection, parameter binding, constants, attachments, includes, substitution, assertions, output/strip. → Ch. XVIII

Rollback — Reverting a prompt to a prior version. The current state should be saved as a new version *before* rollback, so the rollback itself can be undone. → Ch. XXVI

Scoring — Optional rating attached to a prompt. Enabled by the `viewScoring` feature. The `filterNoScoring` filter surfaces templates without a score for cleanup sweeps. → Ch. XXXI

Section — Named, delimited unit of a prompt body. Two orthogonal axes: presence toggle (`? opt-in / ! opt-out / none`, glued to the keyword) and inheritance modifier (`virtual / final / none`, after the keyword). General shape: `@@@section[?!] [virtual|final] <name> [: <description>]`, where `<name>` may be a dotted `<parent>.<child>` sub-section. The unit of override under inheritance. POO²L Level 1. → Ch. XII

XII

self (POO²L reference) — Inheritance-aware include reference on the prompt axis that returns the *most specialized* version available in the inheritance chain. Forms: `self.<section>`, `self.<prompt>.<section>`, used through `@@@include`. The collection-axis counterpart is `^self.<template_id>`. → Ch. XX

Sharing — Grant another user or group read-only access to a prompt. Distinct from *duplication* (an independent copy that diverges over time). → Ch. XXVIII

Simple preset — The minimal of the four presets. Six features enabled. The starting territory of Part I in this book. → Ch. XI

Split pane — Two-panel layout — edit a prompt on the right while browsing the list on the left, or edit two templates in parallel. F5 copies the selection to the other pane, F6 moves it; Tab switches focus. Enabled by the `viewSplitPane` feature (Expert preset). → Ch. IX

Standard preset — The second of the four presets. Adds organizational features (folders, tags, markers, filtering) on top of Simple. The territory of Part II. → Ch. XI

Stash — Server-side working set: stashing tags a prompt `STASH`, and the extension surfaces that set at the top of its **Templates** panel. It follows you across chat tabs and browsers. Treat it as a *today* tray, not durable organization. → Ch. III

System variable — Read-only runtime value substituted automatically at rendering: `@@@=DATE`, `@@@=TIME`, `@@@=DATETIME`, `@@@=YEAR`, `@@@=MONTH`, `@@@=DAY`, `@@@=WEEKDAY`, `@@@=USER`, `@@@=LANG`, `@@@=GROUPS`, `@@@=GROUPS_ADMIN`. There is no `{{...}}` form. → Ch. XVIII

Tag — Multi-value, optionally hierarchical label attached to a prompt. Click a tag to filter the list. Admins can merge, rename and restructure tags globally. → Ch. VII, Ch. XXVII

Template — The unit of content in IziBuddy. Has a name, an id, a description, a body (POO²L source), tags, a container, markers, versions, scores. The body is rendered at injection time. → Ch. I

Tip (contextual) — Non-blocking toast in the bottom-right corner that suggests an optimization based on actual usage. Each chapter of Parts I-V is tied to the tip it would otherwise teach via surprise — see Appendix D for the full crosswalk. → Ch. II, Ch. VI, Ch. VII

Trash — Soft-delete state. The prompt is hidden from the default list but restorable until purged. Endpoint `/trash/list`, `/restore/{id}`, `/purge/{id}`. → Ch. II

UI level (legacy) — The pre-preset 3-level setting (Simple, Standard, Advanced). Migrated automatically to the new preset on first load. Mentioned only for completeness. → Ch. XI

Verbatim — **Block delimited** by `@@@verbatim ... @@@end-verbatim` whose content is preserved literally — POO²L syntax inside is not interpreted. Used for meta-templates and escaping. → Ch. XII

Version — Frozen snapshot of a prompt, created on save and deduplicated by checksum (an unchanged save adds no version). Visible in the version list, diffable, restorable. → Ch. X, Ch. XXVI

Virtual prompt — Abstract prompt member of a collection. Declared with the `virtual` modifier on its collection-qualified `@@@template` header. The collection's derived versions must override the `virtual` to be instantiable. → Ch. XXII

Virtual section — Section declared `@@@section virtual <name>` : abstract, no body. A concrete descendant must implement it (`UNIMPLEMENTED_VIRTUAL_SECTION` fires otherwise). Symmetric to the `virtual` modifier on a `@@@template` header. → Ch. XII, Ch. XIX

Walkthrough — The step-by-step procedure of each chapter — one of the six recurring sections. Designed to be done while reading, not after. → Ch. I

WIP (marker) — Marker tagging a prompt as Work In Progress — not ready for general use. Surfaced through the `filterWip` filter. → Ch. V

Index thématique

COLLABORATION

Git sync → Ch. XXX

Groups → Ch. XXVIII

Invitations → Ch. XXIX

Roles (members / group-admins / system-admin) → Ch. XXVIII, Ch. XXIX

Sharing templates → Ch. XXVIII

Tag administration → Ch. XXVII

CONTEXTUAL TIPS

cut-paste-burst-split → Ch. IX

duplicate-burst-pool2l → Ch. XIII

expert-discovery → Ch. XXXII

first-deletion-undo → Ch. II

first-version-saved → Ch. X

no-tags-30-templates → Ch. VII

poolevel-max-reminder → Ch. XI

promote-collections-3-folders → Ch. XXI

promote-folders-50 → Ch. VI

simple-preset-many-templates → Ch. XI

standard-using-collections → Ch. XXIII

HABITS

Maintenance — monthly → Ch. XXXIII

Maintenance — quarterly → Ch. XXXIII

Maintenance — weekly → Ch. XXXIII

Restore drill → Ch. XXXIII

MARKERS

Disabled marker → Ch. V

Hidden marker → Ch. V

Important marker (★) → Ch. V

Locked marker (🔒) → Ch. V

Template marker → Ch. V

WIP marker → Ch. V

PACING

Day one (Part I) → Ch. I, Ch. II, Ch. III, Ch. IV

Month one (Part III) → Ch. XII, Ch. XIII, Ch. XIV, Ch. XV, Ch. XVI, Ch. XVII, Ch. XVIII

Quarter one (Part IV) → Ch. XIX, Ch. XX, Ch. XXI, Ch. XXII, Ch. XXIII, Ch. XXIV, Ch. XXV

Week one (Part II) → Ch. V, Ch. VI, Ch. VII, Ch. VIII, Ch. IX, Ch. X, Ch. XI

Year one (Part V) → Ch. XXVI, Ch. XXVII, Ch. XXVIII, Ch. XXIX, Ch. XXX, Ch. XXXI, Ch. XXXII, Ch. XXXIII

POO²L CONSTRUCTS

Aliases → Ch. XXIV
Attachments → Ch. XVI
Collection inheritance → Ch. XXIII
Collections → Ch. XXI, Ch. XXII, Ch. XXIII
Constants → Ch. XIV
Deprecation → Ch. XXV
Includes → Ch. XV
Inheritance (template-level) → Ch. XIX, Ch. XX
Parameter sets → Ch. XVII
Parameters → Ch. XIII
parent / self references → Ch. XX
Sections → Ch. XII
System variables → Ch. XVIII
Virtual templates → Ch. XXII

PRESETS

Advanced preset → Ch. XIX, Ch. XXXII
Expert preset → Ch. XXXII
Simple preset → Ch. XI
Standard preset → Ch. XI

UI SURFACES

Containers (folders, collections) → Ch. VI, Ch. XXI
Diff viewer → Ch. X, Ch. XXVI
Features panel → Ch. XXXII
Filters → Ch. VIII
Floating panel (extension) → Ch. IV
Lab → Ch. XXXI
Markers → Ch. V
Profiles tab (extension themes) → Ch. IV
Search → Ch. VIII
Smart editor → Ch. XII, Ch. XIII

Split pane → Ch. IX

Stash → Ch. III

Tags → Ch. VII, Ch. XXVII

Versioning → Ch. X, Ch. XXVI

Index alphabétique

A

Advanced preset → Ch. XIX, Ch. XXXII

Aliases → Ch. XXIV

Attachments → Ch. XVI

C

Collection inheritance → Ch. XXIII

Collections → Ch. XXI, Ch. XXII, Ch. XXIII

Constants → Ch. XIV

Containers (folders, collections) → Ch. VI, Ch. XXI

cut-paste-burst-split → Ch. IX

D

Day one (Part I) → Ch. I, Ch. II, Ch. III, Ch. IV

Deprecation → Ch. XXV

Diff viewer → Ch. X, Ch. XXVI

Disabled marker → Ch. V

duplicate-burst-pool2l → Ch. XIII

E

Expert preset → Ch. XXXII

expert-discovery → Ch. XXXII

F

Features panel → Ch. XXXII

Filters → Ch. VIII

first-deletion-undo → Ch. II

first-version-saved → Ch. X

Floating panel (extension) → Ch. IV

G

Git sync → Ch. XXX

Groups → Ch. XXVIII

H

Hidden marker → Ch. V

I

Important marker (★) → Ch. V

Includes → Ch. XV

Inheritance (template-level) → Ch. XIX, Ch. XX

Invitations → Ch. XXIX

L

Lab → Ch. XXXI

Locked marker (🔒) → Ch. V

M

Maintenance — monthly → Ch. XXXIII

Maintenance — quarterly → Ch. XXXIII

Maintenance — weekly → Ch. XXXIII

Markers → Ch. V

Month one (Part III) → Ch. XII, Ch. XIII, Ch. XIV, Ch. XV, Ch. XVI, Ch. XVII, Ch. XVIII

N

no-tags-30-templates → Ch. VII

P

Parameter sets → Ch. XVII
Parameters → Ch. XIII
parent / self references → Ch. XX
poolevel-max-reminder → Ch. XI
Profiles tab (extension themes) → Ch. IV
promote-collections-3-folders → Ch. XXI
promote-folders-50 → Ch. VI

Q

Quarter one (Part IV) → Ch. XIX, Ch. XX, Ch. XXI, Ch. XXII, Ch. XXIII, Ch. XXIV, Ch. XXV

R

Restore drill → Ch. XXXIII
Roles (members / group-admins / system-admin) → Ch. XXVIII, Ch. XXIX

S

Search → Ch. VIII
Sections → Ch. XII
Sharing templates → Ch. XXVIII
Simple preset → Ch. XI
simple-preset-many-templates → Ch. XI
Smart editor → Ch. XII, Ch. XIII
Split pane → Ch. IX
Standard preset → Ch. XI
standard-using-collections → Ch. XXIII
Stash → Ch. III
System variables → Ch. XVIII

T

Tag administration → Ch. XXVII
Tags → Ch. VII, Ch. XXVII
Template marker → Ch. V

V

Versioning → Ch. X, Ch. XXVI

Virtual templates → Ch. XXII

W

Week one (Part II) → Ch. V, Ch. VI, Ch. VII, Ch. VIII, Ch. IX, Ch. X, Ch. XI

WIP marker → Ch. V

Y

Year one (Part V) → Ch. XXVI, Ch. XXVII, Ch. XXVIII, Ch. XXIX, Ch. XXX, Ch. XXXI, Ch. XXXII, Ch. XXXIII
