AI · CLAUDE TOOLS

Claude Rules Workflow

How this project's own rules-and-memory-driven course-generation system actually works today - files vs. memory, the CLAUDE.md scopes, the full P1-P13/L1-L7/K1-K4/ PL1-3/Sidebar rule families, tracking-file discipline, cold-start recovery, and a capstone designing a brand-new rule category from scratch. 10 chapters, each with real, checkable findings from this exact project rather than invented examples.

10 Chapters

osztromok.com

CHAPTER 1: WHY A FILE-BASED RULES SYSTEM?

Claude Rules Workflow

Chapter 1 · Why a File-Based Rules System?

This whole site — every numbered course, every language lesson, every kanji page — is generated according to a large, growing set of conventions: how a banner comment is formatted, where a file gets saved, when a PDF gets built, what a course does once it's finished. None of that lives in Claude's own memory as the single source of truth. It lives in plain text files, on disk, inside this repository's own `rules/` folder — and this chapter is about why that specific design choice matters, and how it actually works today.

The Problem: Memory Isn't Guaranteed to Persist

Claude's built-in memory is genuinely useful — it's what lets a later session recall that a particular course prefers one bundled PR over several small ones, or that a user is a technical support engineer rather than a full-time developer. But memory is a convenience layer, not a guarantee. It can be cleared. A brand-new session can start with none of it. Two different environments running the same project might not share it at all. If every one of this project's own conventions — the banner-comment format, the course-folder-naming scheme, the PDF-on-completion rule — lived only in memory, a single cleared memory store would mean losing the instructions this entire site depends on to stay consistent.

The Fix: Files as the Source of Truth

The actual conventions live as plain English instructions in markdown files — `permanent_rules.md`, `language_rules.md`, `kanji_rules.md`, and several others, all inside this repository's own `rules/` folder. There's nothing special about the file format; the discipline is entirely in keeping every rule in one place, on disk, where it survives regardless of what happens to any one session's memory. A rules file doesn't forget anything. It's exactly as reliable as the repository itself.

The Old Way — A Manual Recovery Prompt

An earlier version of this exact system (documented in a short, single-chapter reference file dated 2026-06-21 — itself now a historical artifact) relied on a dedicated recovery rule, `R1`, whose entire job was reminding a fresh session to go read the rules file:

```
Please read the permanent_rules.md file stored in the folder
C:\Users\...\claude-generated-code\rules
and treat every rule inside it (P1, P2, P3, P4, ...) as active
for the remainder of the session.
```

This worked, but it depended entirely on someone remembering to paste that exact prompt at the start of every single session — a manual step, easy to forget, and only as good as whoever's job it was to remember it.

The Modern Way — Automatic Inclusion via @rules/

Today, the project's own `CLAUDE.md` file — read automatically at the start of every session in this repository, with no prompt needed at all — contains a short block like this:

```
@rules/permanent_rules.md
@rules/language_rules.md
@rules/kanji_rules.md
@rules/programming_lesson_rules.md
@rules/links_rules.md
@rules/my_tools_rules.md
@rules/sidebar_rules.md
... and several more
```

Each `@rules/<filename>` line is an include — the full contents of that file are pulled directly into context automatically, every single session, before a single word of the actual conversation happens. There's no equivalent of `R1` needed anymore, because there's nothing left to manually remember to ask for. The rules simply arrive.

A REAL, DATED EXAMPLE OF THIS SYSTEM CORRECTING ITSELF

This project's own `CLAUDE.md` currently contains a live callout: "*Path correction (supersedes R1 in recovery_rules.md): The canonical rules location is* `claude-projects\website-content\rules\` *. The old path* `claude-generated-code\rules\` *is deprecated.*" That's not a hypothetical example — it's this

exact project, at some point mid-2026, moving its rules folder and explicitly noting that the entire manual `R1` recovery mechanism from Chapter 1's own opening code block was superseded the moment automatic inclusion took over. The old rule wasn't deleted quietly; it was marked, in writing, as obsolete — which is itself a small demonstration of the discipline this whole course is about.

ASPECT	MANUAL RECOVERY (R1, 2026-06)	AUTOMATIC INCLUSION (@RULES/, TODAY)
Trigger	Someone has to remember to paste the recovery prompt	Happens automatically at the start of every session
Failure mode	Forgetting to paste it means the rules are simply absent	Rules are present unless the include line itself is missing
Where it lives	A separate rule, itself relying on memory of its own existence	A structural part of how the session starts, not a rule to remember
What's still needed	Nothing extra — this was the whole mechanism	A new rules file must be added to the @rules/ list to be included (Chapter 10)

Why This Still Matters Even With a Working Memory System

This project also has a real, separate auto-memory system (covered fully in Chapter 8) — so it's worth being precise about why the file-based rules layer hasn't been replaced by it. Memory is genuinely good at the kind of thing that's specific to one conversation or slowly-learned over time: a preference confirmed once, a project fact that will be stale in a month. The rules files are for the opposite case — a convention meant to apply identically to every single course, every single session, forever, until someone deliberately edits the rule itself. Putting that second kind of instruction in memory would mean it could quietly vary session to session; putting it in a file the whole project reads the same way every time is what keeps hundreds of generated chapters actually consistent with each other.

THE WHOLE SYSTEM IS AUDITABLE, BECAUSE IT'S JUST TEXT IN A REPO

Because every rule is a plain file under version control, a genuine question like "when did the PDF-generation rule change?" or "what did this convention used to say?" has a real, checkable answer — read the file, or check its git history. None of that is possible for a rule that only ever existed as a remembered instruction.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a rule meant to apply identically to every course generated on this site is better suited to a rules file than to Claude's memory system.

View solution

EXERCISE 2

The old R1 recovery rule and the modern @rules/ include mechanism both solve the same underlying problem. Describe that problem, and explain specifically what changed between the two solutions in terms of what a human has to remember to do.

View solution

EXERCISE 3

A new rules file, say a hypothetical "video_lesson_rules.md", is written and saved into the rules/ folder, but the chapter-writer forgets one specific step. Based on this chapter's own comparison table, what will actually happen the next time a new session starts, and why?

View solution

CHAPTER 1 QUICK REFERENCE

- Claude's **memory** is a convenience layer — genuinely useful, but not guaranteed to persist across sessions or environments
- This project's own conventions live as plain markdown files under `rules/` — durable, on disk, version-controlled
- The old system relied on a manual recovery rule, **R1**, that had to be remembered and pasted at the start of every session
- The current system uses **@rules/<filename>** include lines in `CLAUDE.md`, loaded automatically every session with no prompt needed

- This project's own `CLAUDE.md` contains a real, dated note explicitly marking R1 as superseded — the system documenting its own evolution
- A brand-new rules file only takes effect once it's actually added to the @rules/ include list — covered fully in Chapter 10
- Rules files handle project-wide, always-true conventions; memory (Chapter 8) handles conversational, more transient facts

CHAPTER 2: TWO CLAUDE.MD FILES, TWO SCOPES

Claude Rules Workflow

Chapter 2 · Two CLAUDE.md Files, Two Scopes

Chapter 1 talked about "CLAUDE.md" as if it were one file. In practice, there are two — one living at `~/.claude/CLAUDE.md`, the other living inside this repository itself — and they answer two genuinely different questions. The global file asks "how should Claude behave with this particular user, no matter which project they're in?" The project file asks "what does this specific codebase need Claude to know?" Confusing the two, or putting the wrong instruction in the wrong one, is a real and easy mistake to make.

The Global File — Cross-Project Preferences

`~/.claude/CLAUDE.md` lives outside any one project, in the user's own home directory — which is exactly why its own first line states plainly that it "applies across every project, regardless of which directory Claude Code is started in." Whatever's written here follows the user around, not the codebase.

A few real examples from this project's own global file:

- **Explanation Style for Mistakes & Corrections** — when explaining a bug or a correction, state what went wrong and how to avoid it next time. This has nothing to do with any one codebase; it's about how this specific user wants any mistake explained to them, everywhere.
- **Prompting-Style Feedback Is Welcome** — if a prompt could have been phrased more efficiently, say so directly. Again, a preference about the working relationship itself, not about a project.
- **Utility Scripts Folder** — a standing instruction that general-purpose, non-project-specific scripts get saved to one particular folder. The folder itself sits outside any single project, and the instruction only makes sense read that way.
- **The "conversation" command** — a reusable convention (save a readable transcript to a predictable location) meant to work the same way in this project and in any other one the user might be working in.

The Project File — What This Codebase Needs

This repository's own `CLAUDE.md`, by contrast, opens with a plain description of what this specific project actually is — a personal educational website, PHP/MySQL, no build system — information that is meaningless outside this one repository. Its real payload is the `@rules/` include block from Chapter 1, plus a **Key Paths** table mapping every content type (course HTML, language lessons, kanji pages, sidebar pages) to its exact folder inside *this* repo's own `Folder-Structure/` tree.

Global — ~/.claude/CLAUDE.md

Lives in the user's home directory

- Explanation style for corrections
- Prompting-style feedback preference
- The utility-scripts folder location
- The "conversation" transcript command

Project — this repo's own CLAUDE.md

Lives at this repository's own root

- What this specific project is (PHP/MySQL learning blog)
- The full @rules/ include list (Chapter 1)
- The Key Paths table (course/lesson/kanji/sidebar folders)
- The R1-supersession note from Chapter 1

How the Two Actually Stack

Inside this repository, both files are in effect at once — the global preferences apply *and* this project's own rules apply, layered together rather than one replacing the other. Outside this repository, in some completely different project, only the global file would still apply; this repo's own `@rules/` include block and Key Paths table would be entirely irrelevant there, since they only make sense in the context of this specific codebase's own folder structure.

QUESTION	GLOBAL CLAUDE.MD	PROJECT CLAUDE.MD
Where it lives	~/.claude/CLAUDE.md	This repository's own root
Applies where	Every project, everywhere	Only inside this repository
What it holds	Preferences about working WITH the user	Facts and rules ABOUT this codebase
A rule that belongs here	"Explain corrections with what-went-wrong + how-to-avoid"	"Course chapters live at content/<subject>/<course>/"

PUTTING A PROJECT-SPECIFIC RULE IN THE GLOBAL FILE WOULD LEAK IT EVERYWHERE

The @rules/ include list and the Key Paths table are meaningless — worse, actively misleading — in any project that isn't this one. If a rule like "course HTML goes in content/<subject>/" were written into the global file instead of the project file, every other unrelated project the user ever works in would start seeing an instruction that makes no sense there at all. The split isn't just tidy organization; it's what stops one project's own conventions from bleeding into every other one.

A QUICK TEST FOR WHICH FILE A NEW RULE BELONGS IN

Ask: "would this instruction still make sense if pasted into a completely different, unrelated project?" If yes — a communication style, a general habit, a location for scratch files — it belongs in the global file. If the instruction only makes sense because of something specific to this codebase's own folder structure or content type, it belongs in this project's own rules files instead (and, per Chapter 1, still needs its own @rules/ line to actually take effect).

Hands-On Exercises

EXERCISE 1

A new instruction says: "Whenever generating a chart, use a colorblind-safe palette." Using this chapter's own test, decide which CLAUDE.md file it belongs in, and explain your reasoning.

View solution

EXERCISE 2

Explain what would actually go wrong, concretely, if this repository's own Key Paths table were moved into the global ~/.claude/CLAUDE.md file instead of staying in the project's own file.

View solution

EXERCISE 3

A user works on this website project in the morning and on a completely unrelated Node.js API project in the afternoon. Which of this chapter's own four global-file examples (explanation style, prompting feedback, utility scripts folder, "conversation" command) would still be in effect during the afternoon session, and why?

View solution

CHAPTER 2 QUICK REFERENCE

- **~/.claude/CLAUDE.md** — global, applies across every project, holds preferences about working WITH this user
- **This repo's own CLAUDE.md** — project-local, holds facts and rules ABOUT this specific codebase
- Inside this repository, **both files apply at once**, layered together — not one replacing the other
- Outside this repository, only the global file is still relevant
- **Quick test:** would this instruction still make sense in a totally different project? Yes → global. No → project-local
- Putting a project-specific rule in the global file lets it leak into every other unrelated project
- **Next chapter:** a full tour of the project file's own biggest include — the Permanent Rules file, P1 through P13

CHAPTER 3: THE PERMANENT RULES FILE - P1-P13

Claude Rules Workflow

Chapter 3 · The Permanent Rules File: P1–P13

`permanent_rules.md` is the single largest, most-load-bearing file in the whole `@rules/` include list from Chapter 1 — it's what actually governs how every numbered course on this site gets written, saved, and tracked. Rather than read it rule by rule in the abstract, this chapter tours it the way it's actually used day to day, with a few rules singled out for a closer look because you're looking at them in action right now.

The Full Rule Set at a Glance

RULE	GOVERNS	ONE-LINE SUMMARY
P1	Banner headers	Every generated file opens with a Course/Chapter/File/Topic/Date comment
P2	Storage location	content/[subject]/[course]/ — a fixed, predictable folder shape
P3	Progress reports	The bare word "progress" triggers a saved, dated status summary
P4	Completion PDFs	A course's final chapter automatically triggers a combined book-style PDF
P5	Prompt convention	topic<course>-<chapter> — this exact chapter was requested as "rules1-3"
P6	PDF book format	Dark cover, light interior, table of contents, page numbers
P7	Tracking-file sync	completed_courses.md and course_bucket_list.md updated the SAME turn a course finishes (Chapter 7)
P8	*Python lesson format*	*Moved to programming_lesson_rules.md — the base case Chapter 5 covers alongside PL1–PL3*
P9	"Flesh out" convention	Writes a full course outline into the bucket list — literally how this course was born (see below)
P10	Q&A capture	"add this to q&a" saves a real exchange into faqs/qa_with_claude.md
P11	Cross-chapter references	Reader-facing text uses full course names, never a shorthand prompt prefix

RULE	GOVERNS	ONE-LINE SUMMARY
P12	Code block formatting	Real syntax highlighting, ASCII trees inside <pre>, a shared copy-to-clipboard button
P13	Agent-question capture	Questions about Claude Code agents auto-save to faqs/agent_questions.md, no trigger phrase needed

Spotlight: P1 — The Banner You're Looking At Right Now

Every generated course file opens with a fixed-shape HTML comment — this very chapter's own file starts with exactly this structure:

```
<!-- ============================================================
     Course: Claude Rules Workflow
     Chapter: The Permanent Rules File: P1–P13
     File: claude_rules_workflow_1_3.html
     Topic: ...
     Date Created: 2026-08-20   Date Updated: 2026-08-20
     ============================================================ -->
```

Once a fragment has no genuine `<h1>` of its own, the site's own build pipeline uses this banner's `Chapter:` field as the page's real title and breadcrumb — which is exactly why every chapter in this course opens with a `.course-name` / `.chapter-sub` pair instead of repeating the chapter name as a second, near-duplicate heading.

Spotlight: P5 — The Prompt That Produced This Chapter

The short prompt `rules1-3` follows P5's own shape exactly: a topic prefix (`rules1`), no colon, no spaces, a hyphen, then a chapter number. Because Chapter 9 of the fleshed-out outline (see the next spotlight) already existed, this prompt generated the chapter directly — no outline step needed, since P5 only forces an outline-first pause on a course's very first, `-1` chapter with no existing outline at all.

Spotlight: P9 — How This Very Course Came to Exist

This is the most self-referential rule in the whole file, and worth calling out directly: the prompt `flesh out rules1` is what produced the 10-chapter outline this course is currently working through. P9 doesn't generate any chapter content — its entire deliverable is the outline itself, written straight into `course_bucket_list.md` for review. That's exactly what happened here: a short bucket-list one-liner about a stale, orphaned PDF became a real, numbered chapter list, and only afterward did chapter-by-chapter generation begin.

A COURSE ABOUT THE RULES, GENERATED BY FOLLOWING THE RULES

Every mechanism this course documents — P1's banner, P5's prompt shorthand, P9's outline-first flow, and (once this course finishes) P4's automatic PDF and P7's bookkeeping sync — is being used, unmodified, to actually produce this course. There's no special-cased "meta mode." That's a genuine, checkable demonstration that the system is consistent, not just a description of one.

Spotlight: P7 — Why This Course Doesn't Have a PDF Yet

P4 triggers a combined PDF specifically when a course's *final* chapter is generated — not before. Since this course is only three chapters into ten, no combined PDF exists yet, and `completed_courses.md` correctly doesn't list it as finished. Chapter 7 covers the full discipline behind keeping that file, plus `course_bucket_list.md`, in sync — including the rule that both get updated in the very same turn a course's last chapter lands, never deferred to "later."

P11 IN PRACTICE — THIS PARAGRAPH ITSELF FOLLOWS IT

Notice this chapter never writes a bare prompt-shorthand reference like "see php2-4" in its own reader-facing prose — it always spells out the real name, e.g. "PHP Intermediate 4." A prompt prefix like `rules1` is meaningful shorthand for generating chapters, but meaningless to an actual reader who's never seen this project's own internal conventions.

Spotlight: P12 — The Copy Button on Every Code Block Above

Every code block in this chapter is wrapped in a `.code-block-wrap` with a small "Copy" button in the corner — but notice there's no `<script>` tag defining that button's behavior anywhere in this file. Per P12, the actual `copyCodeBlock` function lives once, globally, in the site's shared layout — not duplicated into every single chapter — specifically because a chapter fragment gets injected into the page via an `innerHTML`-equivalent mechanism, and an embedded `<script>` tag inside injected content never executes in that situation.

Coding Challenges

CHALLENGE 1

Explain, using P1 and P5, why the banner comment at the top of this chapter's own file and the prompt "rules1-3" contain overlapping but not identical information (both reference the course and chapter, but only one contains a date).

View solution

CHALLENGE 2

A user sends the prompt "flesh out imagemagick1" for a topic that's never been mentioned before. Based on P9's own description, what should Claude actually produce in response — and what should it NOT produce yet?

View solution

CHALLENGE 3

This chapter's own code blocks all have working Copy buttons on the live site, with zero <script> tag inside this chapter's own HTML file. Explain why, referencing P12's own reasoning about injected content.

View solution

CHAPTER 3 QUICK REFERENCE

- **P1** — every generated file opens with a Course/Chapter/File/Topic/Date banner comment
- **P2** — fixed content/[subject]/[course]/ storage shape
- **P3 / P4 / P6** — progress reports, completion PDFs, and the dark-cover/light-interior book format behind them
- **P5** — the topic<course>-<chapter> prompt shorthand, e.g. "rules1-3" for this very chapter
- **P7** — completed_courses.md and course_bucket_list.md updated the same turn a course finishes, never deferred (Chapter 7)
- **P8** — moved out to programming_lesson_rules.md; covered in Chapter 5 alongside PL1–PL3

- **P9** — the "flesh out" convention that produced this exact course's own 10-chapter outline
- **P10 / P13** — two automatic capture conventions (Q&A archive, Claude-agent-question log)
- **P11** — reader-facing text always uses full course names, never a bare prompt prefix
- **P12** — real syntax highlighting and a shared, globally-defined copy-to-clipboard button, not a per-file <script>
- **Next chapter:** the natural-language lesson rules, L1 through L7

CHAPTER 4: NATURAL-LANGUAGE LESSON RULES - L1-L7

Claude Rules Workflow

Chapter 4 · Natural-Language Lesson Rules: L1–L7

Alongside the numbered programming courses P1–P13 governs, this site also generates standalone lessons in real human languages — Japanese, Hungarian, French. Those lessons follow a completely separate set of rules, `language_rules.md`'s own L1 through L7, because a language lesson genuinely isn't shaped like a course chapter: there's no fixed chapter count, no Fundamentals/Intermediate/Advanced track, and a single topic (say, "question words") gets requested and generated on demand, in whichever language the learner actually wants it in.

L1 — The Languages Table & Learner Profiles

A small table tracks every language currently in use, each with its own level, its own per-country accent colour, and a short learner profile (goals, current knowledge, why they're learning it) that shapes how every lesson in that language gets written:

Japanese
Beginner — knows hiragana, mostly knows katakana, no kanji yet. Goal: watch anime without subtitles, and speak naturally with real people.

Hungarian
Beginner-plus. Learning primarily to communicate with Hungarian family.

French
More advanced than the other two — level established from real, already-generated lessons.

That "no kanji yet" detail isn't decorative — it directly shapes how a Japanese lesson gets written. Every kanji that does appear still needs a romaji reading right beneath it, since the learner can't yet read it unaided.

L2 — The <language>:<topic> Prompt Convention

A prompt like `hungarian:question words` means exactly what it looks like — generate a lesson, in that language, on that topic. The **colon** is the whole signal that distinguishes this from P5's course-chapter convention: `hungarian:question words` is a language lesson; `rules1-4` (no

colon) is a course chapter. No outline step is needed either way — a lesson generates directly the moment it's requested.

L3 — A PDF for Every Single Lesson

Unlike P4 (which only fires once, on a course's *final* chapter), every language lesson gets its own PDF immediately, via a reusable builder script that detects the language straight from the filename prefix and applies that language's own accent colour and flag automatically.

A REAL BUG THIS EXACT RULE SURFACED, THIS SAME SESSION

That builder script's own `HTML_DIR` / `PDF_DIR` constants were still hardcoded to a pre-rebuild scratch location — `claude-storage\html` and `claude-storage\pdfs` — left over from before this project's own big folder restructuring. Generating this session's Hungarian, Japanese, and French lessons meant the script would have written the PDFs to a stale location no chapter actually lives in anymore. It was fixed in place: each language's own real `content/<country>/<language>-language/` folder is now computed directly from the same `LANGS` table that already held each language's accent colour — one dictionary, no separate path config to fall out of sync with it again.

L4 — The Two Lesson Sub-Formats

Not every language lesson is shaped the same way. A plain topical lesson ("directions," "food," "question words") uses **sub-format A**: a six-card dialog grid demonstrating the topic in natural conversation, followed by a fixed 20-word vocabulary table. A lesson specifically about a verb tense or conjugation pattern (Hungarian's past definite tense, for instance) uses **sub-format B** instead — full conjugation tables and a pattern box come first, with the dialog grid and vocabulary table still following after.

ASPECT	SUB-FORMAT A — TOPICAL	SUB-FORMAT B — GRAMMAR/VERB-TENSE
Trigger	An ordinary topic ("food," "directions")	A tense/conjugation name ("past definite tense")
Opens with	The dialog grid directly	A conjugation grid + pattern box, before the dialogs
Vocabulary table	Fixed 20 rows	20 rows, verbs shown in their target-tense form

L4a layers one more detail on top of either sub-format: every language gets its own single accent colour, derived loosely from its flag — Japanese red, Hungarian green, French blue, all visible in the language cards above — kept consistent across every lesson in that language so a glance at a page identifies which language it's in before reading a word of it.

L5 — Introducing a Brand-New Language

A lesson request in a language that isn't already in the L1 table doesn't generate silently. Per L5, the correct response is to ask first — is this genuinely a new language to add, or a one-off/typo? What's the learner's actual level? What accent colour fits, without clashing with a language already in the table? Only once those questions are answered does the language get added to L1 and the first lesson actually generated.

L6 — Verb Deep Dive Lessons

`vdd:hungarian:lenni` or `vdd:japanese:to be` requests something narrower and deeper than an ordinary topical lesson — a full treatment of one specific verb, built entirely around L4's own sub-format B (conjugation tables across the language's own relevant tenses, plus mini-dialogues using that verb and a labelled vocabulary list of related verbs). When a VDD is requested across several languages in sequence, each one is generated and then paused on, rather than all being produced back to back unprompted.

L7 — Song Lessons, and the Copyright Line That Splits Them in Two

`song:title:artist:language` produces a full, line-by-line lesson from real song lyrics — but only for lyrics that are genuinely copyright-free (public domain, traditional, or original material). `songvocab:title:artist:language` exists specifically for everything else: it never reproduces the lyrics themselves in any form, extracting only individual, decontextualized vocabulary words instead — which is legally safe even from a commercially copyrighted song, since isolated words and their translations aren't the protected creative expression the way the lyrics' actual sequence and phrasing are.

> **THIS CHAPTER'S OWN HUNGARIAN LESSON FOLLOWED L1 AND L5'S SPIRIT, EVEN WITHOUT TRIGGERING L5 ITSELF**

Hungarian was already in the L1 table, so no new-language questions were needed — but writing that lesson still meant checking a real, already-existing Hungarian lesson file for its actual accent colour in practice, rather than trusting an older canonical example file that turned out to still be using a stale colour from before L4a's per-language accents were introduced. Real, current usage beat a static reference file — the same discipline Chapter 1 applied to the R1 recovery rule.

L2'S COLON IS EASY TO LOSE IN CASUAL PHRASING

"generate a Hungarian lesson on question words" and "hungarian:question words" mean the same thing, but only the second is the actual L2 trigger form — the convention exists specifically so a terse, unambiguous shorthand is available, not to replace plain-English requests entirely.

Coding Challenges

CHALLENGE 1

A learner asks for "spanish:ordering coffee," and Spanish has never appeared in the L1 table before. Walk through, step by step, what should happen before any lesson content is generated, per L5.

View solution

CHALLENGE 2

Explain why "hungarian:past definite tense" and "hungarian:food" produce genuinely different-looking lessons, even though both are triggered by the exact same L2 prompt shape.

View solution

CHALLENGE 3

A user wants a lesson built from the full lyrics of a song currently in the commercial charts. Using L7, explain what CAN and CANNOT be generated, and why the distinction exists.

View solution

CHAPTER 4 QUICK REFERENCE

- **L1** — the languages table: level, accent colour, and learner profile per language
- **L2** — <language>:<topic>, colon-separated, distinct from P5's hyphenated course-chapter shorthand
- **L3** — every lesson gets its own PDF immediately, unlike P4's course-completion-only trigger
- **L4** — two sub-formats (topical vs. grammar/verb-tense), plus L4a's one accent colour per language
- **L5** — a genuinely new language triggers real questions (level, accent, goal) before any lesson generates
- **L6** — vdd:<language>:<verb> for a focused, single-verb deep dive using sub-format B
- **L7** — song: for copyright-clear full lyrics; songvocab: for vocabulary-only extraction, safe for any song
- **Next chapter:** the more specialized content rules — kanji pages, and the Python/programming lesson family

CHAPTER 5: SPECIALIZED CONTENT RULES - KANJI AND PROGRAMMING LESSONS

Claude Rules Workflow

Chapter 5 · Specialized Content Rules: Kanji and Programming Lessons

Chapter 4 covered rules for lessons in a spoken language. This chapter covers two more content families that each needed their own dedicated rule set, for two genuinely different reasons: kanji pages are interactive and dual-purpose in a way no other content type on the site is, and programming lessons needed L2's own convention generalized beyond Python to any language without losing Python's already-established, carefully-tuned format.

K1 — The Stroke Animation Method

Every kanji page includes a stroke-order animation. K1's own rule is narrow but firm: build it with a self-hosted stroke-data library, never a CDN-script approach that makes the page's own functionality depend on some third party's server staying online forever. A kanji page should work exactly the same whether that external service is reachable or not — because it never talks to one in the first place.

K2 — Dual-Write: One Page, Two Locations

Every kanji page gets written to two places at once, kept identical: an archive copy that's deliberately excluded from the live site's own routing (a pure reference folder), and the live copy actually served to visitors. Skipping either half isn't a shortcut — it either breaks the live site or silently stops maintaining the permanent archive, so K2 treats the second write as a cheap, non-optional step rather than an afterthought.

K3 — Keeping the Tiles Page in Sync

A newly generated kanji page is useless if nothing links to it. K3 requires the kanji tiles grid's own `kanjiList` array to be updated in the very same step a kanji page is written or rewritten — character, meaning, and href, all added together, never as a separate follow-up task.

A REAL, PREVIOUSLY-BROKEN SITEWIDE LINK, FOUND AND FIXED

K3 also documents its own back-link convention — every individual kanji page's own "← Kanji Grid" link should point at the tiles page's real, fixed live URL. A previous version of that same rule instead pointed at a stale, now-nonexistent old-site URL, and every single kanji page's own back-link was broken sitewide as a result — not a hypothetical risk, an actual bug that existed until the 2026-08 rules review caught and fixed it. It's a concrete illustration of exactly the kind of drift Chapter 7 covers more generally: a rule can describe the right behaviour perfectly and still go stale the moment the thing it's pointing at moves.

K4 — The World of Kanji Book Cross-Reference

The user owns a printed kanji reference book indexed by number rather than page. K4 lets a generated page link back to that number as one more tag alongside the page's existing JLPT-level and stroke-count badges — but only when a real number is actually supplied. A request with no reference number at all should be met with a direct question rather than a guess, and an explicit `0` is treated as "no reference," not as a literal reference number 0 — the tag is simply omitted in that case, never left as an empty placeholder.

Programming Lessons: P8 and PL1–PL3

A separate, unrelated content family: standalone, one-off lessons on a single programming topic — closer in spirit to Chapter 4's own language lessons than to a numbered course, since there's no fixed chapter count and no Fundamentals/Intermediate/Advanced track.

RULE	COVERS
P8	The original Python-specific format: a "Coming from Java/JavaScript" callout, a concept-card grid, worked examples, a quick-reference table, and a closing gotchas box
PL1	Generalizes P8's own shape to any programming language, reusing L2's colon syntax — disambiguated from a natural-language lesson purely by whether the word before the colon names a programming language or a spoken one
PL2	Two modifiers — "no exercises" and "exercises only" — controlling whether a companion exercise chapter is generated alongside the main lesson

RULE	COVERS
PL3	The exercise chapter's own format: a separate file, at least 10 real, meaty challenges, each linked to its own worked .txt solution

By default, a PL1 request produces *two* files — the lesson and its exercise companion — unless a modifier says otherwise. That default matters because it's the one place in this chapter where a single prompt silently expands into two pieces of generated content rather than one.

PYTHON: AND PYTHON (THE LANGUAGE) ARE HANDLED BY PL1, NOT L2 — DESPITE REUSING L2'S OWN COLON SYNTAX

`python: for loops` and `japanese:directions` look identical in shape, but only one of them is a Chapter 4 language lesson. The disambiguation is entirely by the word before the colon — if it names a language already in the L1 spoken-language table, it's L2; if it names a programming language, it's PL1. The two systems share a prompt shape on purpose, but never share output, and there's currently no real name collision to worry about between the two tables.

ASPECT	KANJI PAGES (K1–K4)	PROGRAMMING LESSONS (P8/PL1–3)
Storage	Dual-write: archive copy + live copy	Single copy under content/programming/.../
PDF?	None — interactive content doesn't translate to a static PDF	Yes, one per lesson, plus one for its exercise companion
Sync obligation	The tiles page's own kanjiList array, every time	None comparable — no shared index page to update
Default output	One page per kanji	Two files by default (lesson + exercises), unless a modifier says one

BOTH FAMILIES EXIST BECAUSE A SHARED, GENERIC TEMPLATE GENUINELY DIDN'T FIT

Neither kanji pages nor programming lessons could have simply reused the numbered-course template (P1–P13) or the language-lesson template (L1–L7) as-is — a kanji page needs a live stroke animation and a permanent, non-routed archive twin that no course chapter needs; a programming lesson needs a companion exercise file with real, substantial challenges that a spoken-language lesson's own 20-word vocabulary table has no equivalent of. Each specialized rule family exists to cover exactly the gap the more general rules left open.

Coding Challenges

CHALLENGE 1

A kanji page is generated and written to its live-site location only, with the archive copy skipped "to save a step." Using K2, explain exactly what real, concrete problem this creates later.

View solution

CHALLENGE 2

A user sends the prompt "python: dictionaries exercises only." Explain what this should produce, and why the wording of the modifier matters based on PL2.

View solution

CHALLENGE 3

Explain, using this chapter's own comparison table, why kanji pages have no PDF at all while every programming lesson gets one — what's the underlying reasoning, not just the fact itself?

View solution

CHAPTER 5 QUICK REFERENCE

- **K1** — self-hosted stroke animation, never a CDN dependency
- **K2** — every kanji page written to both an archive copy AND the live copy, kept identical
- **K3** — the tiles page's kanjiList array updated in the same step, plus a correct back-link to the real live tiles URL
- **K4** — the World of Kanji book number tag, added only when a real number is given; 0 means "omit it," not "literal reference 0"
- **P8** — the original Python-specific weekly-lesson format
- **PL1** — generalizes P8/L2's colon syntax to any programming language
- **PL2** — "no exercises" / "exercises only" modifiers control whether one or two files get generated
- **PL3** — the exercise-chapter format: 10+ real challenges, each with its own worked .txt solution
- **Next chapter:** the Sidebar unification — Links, My Tools, Sidebar Lessons, and Cheat Sheets

CHAPTER 6: THE SIDEBAR UNIFICATION

Claude Rules Workflow

Chapter 6 · The Sidebar Unification: Links, My Tools, Sidebar Lessons, Cheat Sheets

Everything so far in this course has covered rule families that stayed roughly the same shape once written. Sidebar is the opposite kind of example — a real case where the rules system itself got reorganized, twice, in the space of a single day, and the reorganization actually held up once the site's own routing was rebuilt around it. It's the clearest example in this whole course of a rules system being *revised*, not just extended.

SB1 — Four Content Types, One Folder Tree

"Sidebar" is a top-level content category for everything that doesn't fit the numbered-course model at all — no fixed chapter count, generated one page at a time, on demand. Four genuinely different content types live under it, sharing one `content/sidebar/<subject>/` tree:

Links
A curated link collection for one topic — Documentation/Downloads/Free Courses/LinkedIn Learning

My Tools
A self-contained interactive HTML tool, built from a YAML spec file

Sidebar Lessons
A standalone, topical deep-dive — too substantial to skip, too narrow for a full course

Cheat Sheets
A compact, printable, scannable quick-reference page — commands and syntax, not prose

Browsing Sidebar for a given Subject is meant to surface a link collection, a tool, a lesson, and a cheat sheet all in one place — rather than four separate top-level sections a visitor has to already know exist independently.

Links (LN1–LN9) and My Tools (MT1–MT9), Briefly

Links pages follow a real research workflow: check the user's own running scratchpad file first, then search the web section by section (Documentation, Downloads, Free Courses, LinkedIn Learning), never fabricating a URL that hasn't actually been confirmed. My Tools pages work completely differently — the spec is a YAML file, not a web search, describing exactly what the tool should do, and the whole page has to be a self-contained fragment with every CSS rule

scoped under its own wrapper class, after a real leak bug (unscoped selectors bleeding into the rest of the page once injected) was found and fixed in an early tool.

SB2 — The Banner Format, Revised In Place

Every Sidebar page originally used the same `Course:` / `Chapter:` banner shape as a numbered course — `Course: Links`, `Course: My Tools`, `Course: Sidebar`. It worked, but it was never really true: none of these are courses. SB2 replaced it with a shape that actually describes what the content is:

```
<!-- ============================================================
     Category: Sidebar
     Subcategory: <Subject, Title Case>
     Chapter: <Page Title, Title Case>
     File: <filename>.html
     Date Created: ...   Date Updated: ...
     ============================================================ -->
```

`Category` is always literally "Sidebar." `Subcategory` is the piece that was missing before entirely — the page's own Subject — even though that Subject already silently determined the file's folder path the whole time.

> **THIS CHANGE WAS APPLIED RETROACTIVELY — A DELIBERATE EXCEPTION**
>
> Most banner-format changes on this site are explicitly not retroactive (P1's own default). This one was different: with only five existing Sidebar pages in scope at the time, every one of them was updated to the new banner in the same session the new format was decided — a real, documented exception made because the cost of updating five files was genuinely small next to the benefit of the whole category being internally consistent from that point on.

Sidebar Lessons (SB3–SB5) and Cheat Sheets (SB6–SB8)

A Sidebar lesson (`sidebar: <topic>`) is a standalone deep-dive, deliberately required to cross-reference existing courses honestly rather than presenting overlapping material as entirely new. A cheat sheet (`cheat sheet <topic>`) is a genuinely different shape, not a lesson in miniature — a grid of category cards with dense `command — description` rows, no prose intro, no closing

summary, since the whole page already *is* the summary. Cheat sheets are deliberately a plain-English prompt rather than a formal skill, since picking which commands actually matter is judgment, not a fixed mechanical pipeline.

SB10 — A Bet That Actually Paid Off

The entire Sidebar reorganization was made on a specific bet: that structuring the on-disk folders by Subject now would let a rebuilt site automatically generate real navigation later, without ever hand-maintaining a table-of-contents page per Subject. SB10 confirms that bet actually paid off — once the site's own routing was rebuilt, `content/sidebar/<subject>/` folders started auto-generating a real, working index page listing everything under that Subject, with zero manually-written index file anywhere.

> **A GENUINELY VERIFIED OUTCOME, NOT JUST AN ASSUMPTION WRITTEN DOWN**
>
> It would have been easy for this rule to just assert "the folder structure will pay off later" and leave it there. Instead, once the rebuild actually happened, someone went back and confirmed it directly against the real, live routing behaviour — and only then was SB10 updated to say "confirmed working," not just "expected to work." That's the same evidentiary standard Chapter 1's own R1-deprecation finding and Chapter 5's own broken-back-link finding both apply: a claim about the system gets checked against what's actually true, not left as an assumption.

SB11 — One Shared Tracking File, Not Three

Before the unification, Links, My Tools, and the one-off Sidebar lesson each had their own separate tracking file. SB11 merged all three into one shared `sidebar-pages.md`, logging every page across all four content types — topic/tool/lesson/cheat-sheet name, Subject, file path, creation date, and every update date — updated in the same step a page is generated, exactly matching P7's own never-deferred discipline from Chapter 3, just applied one level down to a single tracking file instead of the whole completed-courses/bucket-list pair.

Coding Challenges

CHALLENGE 1

Explain why SB2's banner change was applied retroactively to all five existing Sidebar pages, when P1 itself explicitly says banner changes normally aren't retroactive. What made this case different?

View solution

CHALLENGE 2

A user asks for a cheat sheet on a topic. Explain, using SB7, why the resulting page should NOT include a set of Hands-On Exercises or a closing Quick Reference box, even though those are standard parts of a Sidebar lesson.

View solution

CHALLENGE 3

Explain what SB10's own "confirmed working" note is actually claiming, and why that's a stronger, more useful statement than the original folder-structure decision alone would have been.

View solution

CHAPTER 6 QUICK REFERENCE

- **Sidebar** unifies four content types — Links, My Tools, Sidebar Lessons, Cheat Sheets — under one content/sidebar/<subject>/ tree
- **Links (LN1–LN9)** — real web research, never a fabricated URL
- **My Tools (MT1–MT9)** — YAML-spec-driven, every CSS rule scoped to its own wrapper class
- **SB2** — the Category:/Subcategory: banner replaced the old Course:-based one, applied retroactively as a deliberate, documented exception
- **Sidebar Lessons (SB3–SB5)** vs. **Cheat Sheets (SB6–SB8)** — genuinely different shapes, not a lesson in miniature
- **SB10** — the folder-structure-predicts-live-routing bet, later verified true, not just assumed
- **SB11** — one shared sidebar-pages.md tracking file, merged from three, updated in the same step every time

- **Next chapter:** the tracking files themselves — completed_courses.md, course_bucket_list.md, and the P7 discipline that keeps them honest

CHAPTER 7: TRACKING FILES AND THE P7 DISCIPLINE

Claude Rules Workflow

Chapter 7 · Tracking Files & the P7 Discipline

Rules files describe how content should be generated. A separate, smaller set of files tracks what's *actually been* generated — and unlike a rules file, a tracking file's whole value depends on staying perfectly in sync with the real state of `content/`. This chapter covers what each tracking file is for, and a real, concrete example — from this very session — of what happens when that sync breaks.

The Five Tracking Files

`completed_courses.md`

A chapter-level index of every finished course. Archived into numbered files (completed_courses_1.md, _2.md...) once it grows too large — the current file was closed off and archived once already, at 754 lines.

`course_bucket_list.md`

Only what's still outstanding. A finished course's entry is removed entirely, never left marked "done" in place — this course's own bucket-list entry was already rewritten this way the moment rules1 stopped being "not yet started."

`course_folder_mapping.md`

The authoritative prompt-prefix → folder-path map. Every course, including this one, gets a real row here the moment its outline is fleshed out — not deferred until chapter generation begins.

`course_name_index.md`

A flat prefix → full course name lookup, existing purely to support P11's own rule from Chapter 3 (cross-chapter references use the real course name, never the bare prompt shorthand).

A fifth file, `sidebar-pages.md`, covers the same job specifically for Sidebar content — already covered in Chapter 6.

P7 Itself

The actual rule is short: when the final chapter of a course is generated, `completed_courses.md` and `course_bucket_list.md` both get updated **in the same turn** — not deferred, not batched up for later. That single word, "immediately," is the entire discipline this chapter is named after.

What Happens When P7 Doesn't Hold: A Real Example

This isn't hypothetical. Earlier this same session, three PHP courses — Fundamentals, Intermediate, and Advanced — were found to have real, surviving combined PDFs, each one looking exactly like proof of a finished course... while the actual chapter HTML behind two of them had been completely lost, and the third had only one of ten chapters left. The tracking files and the real content in `content/` had quietly drifted apart at some point, and nothing caught it until a direct verification sweep compared what the mapping files claimed against what actually existed on disk.

THE FIX MIRRORED P7'S OWN DISCIPLINE EXACTLY

Reconstructing all three PHP courses this session ended with the identical step every other completed course gets: `course_folder_mapping.md`, `completed_courses.md`, and `course_bucket_list.md` were all updated in the same turn each course's own final chapter and PDF were finished — not as a separate cleanup task done later, but as the direct, immediate consequence of P7 being followed correctly this time.

A TRACKING FILE THAT SAYS "COMPLETE" IS A CLAIM, NOT A FACT

The PHP example shows exactly why: `course_folder_mapping.md` can say a course is finished while the actual chapter HTML genuinely doesn't exist anymore, if the files and the tracking record are ever allowed to fall out of sync. A tracking file is only as trustworthy as the discipline that keeps updating it — which is precisely why P7 insists on "the same turn," not "eventually."

A Worked Example: This Course's Own Future Chapter 10

When this course's own capstone (`rules1-10`) eventually generates, P4 (Chapter 3) fires the combined PDF, and P7 requires all of the following to happen in that same turn: `course_folder_mapping.md`'s `rules1` row gets updated from "10/10 chapters outlined" to "complete"; a new dated section is appended to `completed_courses.md` naming every chapter; and — since this entry currently sits under a fleshed-out-outline heading rather than "Course Regenerations Needed" — the corresponding section of `course_bucket_list.md` gets removed entirely, the same way the original stale "Claude Rules Workflow" bucket-list entry was replaced back in Chapter 3's own P9 story.

Coding Challenges

CHALLENGE 1

Explain, using this chapter's own PHP example, exactly what kind of mistake becomes possible when a tracking file is trusted without verifying it against the real content on disk.

View solution

CHALLENGE 2

A course finishes its final chapter, but course_bucket_list.md is left with the course still marked as "in progress" rather than removed, with a plan to "clean it up at the end of the week." Explain specifically which part of P7 this violates, and why "later" isn't an acceptable substitute for "the same turn."

View solution

CHALLENGE 3

Explain why course_name_index.md exists as its own separate file rather than being folded into course_folder_mapping.md, given that both files map a prompt prefix to information about a course.

View solution

CHAPTER 7 QUICK REFERENCE

- **completed_courses.md** — chapter-level history of finished courses, archived into numbered files once too large
- **course_bucket_list.md** — only what's outstanding; a finished entry is removed, never marked done in place
- **course_folder_mapping.md** — the authoritative prefix → folder-path map, updated the moment an outline exists
- **course_name_index.md** — prefix → full course name, existing to support P11's cross-reference rule
- **sidebar-pages.md** — the same job, for Sidebar content (Chapter 6)
- **P7's actual discipline:** update the tracking files in the SAME turn a course finishes — never deferred

- A real example this session: three PHP courses' tracking files claimed completeness that the actual content on disk no longer matched — found and fixed by verifying against reality, not by trusting the record
- **Next chapter:** the auto-memory system, and when it's the better fit instead of a rules file

CHAPTER 8: THE AUTO-MEMORY SYSTEM

Claude Rules Workflow

Chapter 8 · The Auto-Memory System

Chapter 1 drew a firm line between memory and rules files: rules are for conventions meant to apply identically, forever, to every course; memory is for the kind of thing that's specific to one conversation or learned gradually over time. This chapter takes that distinction seriously and looks at the memory system itself in the same detail the earlier chapters gave the rules files — including a real, honest case where the two systems overlap more than they probably should.

Where It Lives, and Why It's Still File-Based

Memory in this project is itself a persistent, file-based system, living in its own dedicated folder separate from this repository's own `rules/` tree. That's worth noticing: memory isn't some opaque, unreadable internal state — it's plain files too, just organized around a different question than a rules file asks. A rules file asks "what should always be true?" A memory file asks "what have I learned about this specific user, this specific project, or where to find something, that's worth carrying into a future conversation?"

The Four Memory Types

User

Who the user is — their role, goals, knowledge — so future work can be tailored to them specifically.

e.g. a data scientist investigating logging, vs. a Go veteran new to React

Feedback

Guidance the user has given about approach — both corrections AND confirmations of something unusual that worked.

e.g. "don't mock the database in these tests" — with the reason why

Project

Who's doing what, why, or by when — context that isn't otherwise derivable from the code.

e.g. a merge freeze starting on a specific date, and why

Reference

Pointers to where information lives in an external system.

e.g. "bugs are tracked in Linear project INGEST"

The Two-Step Save Process

Saving a memory is never a single write. First, the memory itself gets its own file, with a small frontmatter block naming it, describing it, and tagging its type:

```
---
name: short-kebab-case-slug
description: one-line summary, specific enough to judge relevance later
metadata:
  type: user | feedback | project | reference
---

(the actual memory content goes here)
```

Second, a single-line pointer gets added to `MEMORY.md` — the always-loaded index every future session actually reads. A memory file that exists but was never added to that index is exactly as invisible as a rules file that exists but was never added to `CLAUDE.md`'s own `@rules/` list from Chapter 1 — the same "one step, easy to forget" failure shape, one level down.

What Doesn't Belong in Memory At All

Code patterns, architecture, and file paths are excluded on purpose — they can be derived by reading the current project state directly, and a memory claiming to know them risks going stale the moment the code actually changes. Debugging solutions are excluded too, for the same reason: the real fix lives in the code itself and the commit message, not in a separate remembered description of it that could drift out of date.

A REAL, HONEST OVERLAP BETWEEN MEMORY AND A RULES FILE

This project's own memory index currently includes a feedback memory titled "HTML Course Header Comment Rule," stating that every course HTML file needs a banner comment before `<style>` — which is, word for word, the same rule Chapter 3 already covered as P1 in `permanent_rules.md`. Rather than quietly ignore this, it's worth naming directly: this is exactly the kind of quiet duplication a well-maintained system should eventually retire — once a piece of feedback graduates into a real, permanent rule, the standalone memory describing the same thing has done its job and can be superseded, not left sitting alongside the rule it duplicates indefinitely.

Before Recommending From Memory

A memory that names a specific file, function, or flag is a claim that it existed *when the memory was written* — not a guarantee it still does. Before acting on it, the same discipline every earlier chapter has already applied to rules files applies here too: check the file still exists, grep for the function, verify the flag before recommending it. "The memory says X exists" and "X exists now" are two different claims, and only the second one is safe to act on.

QUESTION	RULES FILE	MEMORY
Applies to	Every course, every session, identically	This specific user, project, or fact
Changes how	Deliberately, by editing the rule itself	Gradually, as new things are learned
Loaded via	@rules/ lines in CLAUDE.md (Chapter 1)	MEMORY.md's own index
Trust it because	It's the current, deliberately-maintained standard	It was true when saved — verify it's still true

NOT EVERY PERSISTENCE MECHANISM IS MEMORY

Plans and Tasks are separate, session-scoped tools — a Plan is for reaching alignment on an approach before a big implementation task, and Tasks track in-progress work within the current conversation. Neither is meant to be recalled in a future conversation the way memory is; using memory for something that's really just "what am I doing right now" would clutter a system meant to hold durable, cross-session facts.

Coding Challenges

CHALLENGE 1

A user says "I got burned last quarter when a mocked test passed but the real production migration failed — never mock the database in integration tests again." Which of the four memory types does this belong to, and why?

View solution

CHALLENGE 2

Explain, using this chapter's own finding about the duplicated "HTML Course Header Comment Rule" memory, what the ideal fix would actually be — not just noticing the duplication, but what should happen to the memory file itself.

View solution

CHALLENGE 3

A memory file references a function, formatPrice(), that was recommended as a reusable helper three months ago. Before recommending it again today, what specifically should happen first, and why?

View solution

CHAPTER 8 QUICK REFERENCE

- **Four memory types:** user (who they are), feedback (corrections AND confirmations), project (current context), reference (external pointers)
- **Two-step save:** a memory file with frontmatter, PLUS a one-line pointer added to MEMORY.md's own index
- A memory file never added to MEMORY.md is invisible — the same failure shape as a rules file never added to @rules/ (Chapter 1)
- Excluded from memory: code patterns, architecture, git history, debugging solutions — all better derived fresh from the current code
- A real, honest finding: this project's own memory currently duplicates P1's banner rule — worth retiring once a feedback memory graduates into a real rule
- **Before recommending from memory:** verify the named file/function/flag still exists — "it was true when saved" isn't "it's true now"
- Plans and Tasks are separate, session-scoped tools — not memory, and not meant to be recalled later
- **Next chapter:** what actually happens the moment a brand-new session opens in this repository

CHAPTER 9: RECOVERING FROM A COLD START

Claude Rules Workflow

Chapter 9 · Recovering From a Cold Start

Chapter 1 introduced the idea that `@rules/` replaced a manual recovery prompt with automatic inclusion, and left it there. This chapter goes one level deeper — what a "cold start" in this repository actually looks like, step by step, and what parts of reliability that automation genuinely buys versus what still depends on a human noticing something's gone stale.

What Actually Happens, In Order

1. **The global CLAUDE.md loads** — cross-project preferences (Chapter 2), regardless of which repository the session opens in.
2. **This repository's own CLAUDE.md loads** — its project description, its Key Paths table, and critically, its full `@rules/` include list.
3. **Every referenced rules file arrives** — permanent_rules.md, language_rules.md, kanji_rules.md, and every other file on that list, delivered as real content, not just a promise to go read them later.
4. **MEMORY.md's own index arrives too** — the short, one-line-per-entry pointer list from Chapter 8, so relevant memory files can be recognized and read on demand as the conversation actually needs them.
5. **Only then does the first real user message get processed** — every rule and every memory pointer above is already present before a single word of the actual conversation begins.

Why This Is Structurally Different From R1

R1, from Chapter 1, was a *prompt* — a request that Claude go perform a read action, worded carefully enough to be self-sufficient even if the human pasting it had forgotten exactly what it did. The modern mechanism isn't a prompt at all; the rules content simply arrives, already-read, before the conversation has properly started. There's no step where something could be

skipped, because there's no separate step to skip in the first place — it's part of how the session begins, not an instruction issued during it.

ASPECT	R1 (MANUAL, CHAPTER 1)	COLD START TODAY
What's needed from a human	Remember to paste the recovery prompt	Nothing — happens automatically every session
When rules become available	Only after the prompt is sent and processed	Before the first real message is even handled
Failure mode	Forgotten prompt → zero rules in effect all session	A rules file simply not yet added to @rules/ (Chapter 1)

What This Automation Does NOT Guarantee

Everything arriving reliably isn't the same as everything arriving *accurate*. The mechanism faithfully delivers whatever the rules and memory files currently say — it has no way of independently checking whether what they say is still true.

TWO REAL EXAMPLES, ALREADY FOUND IN THIS EXACT COURSE

Chapter 8's own finding — a feedback memory duplicating P1's banner rule, never retired once P1 became a formal rule — is exactly this kind of staleness: the memory arrives at cold start precisely as written, correctly, and is still slightly wrong to treat as an independent source once P1 already exists. Chapter 5's own finding — a kanji page's back-link rule pointing at a URL that no longer existed, broken sitewide until the 2026-08 rules review caught it — is the same shape one level down: the rule arrived faithfully every single session, describing behaviour that was, in fact, broken the whole time. Automatic delivery solved "did the rule arrive." It never claimed to solve "is the rule still correct."

A RENAMED PATH IS INVISIBLE TO THIS WHOLE SYSTEM

If a folder referenced by a rules file gets renamed on disk but the rule itself is never updated, cold start will keep delivering the old path every single session, with complete reliability — and complete inaccuracy. Nothing in the loading mechanism itself checks that a referenced path still resolves; that verification is still a human (or an agent doing the verifying) job, the same discipline behind every "check this actually still exists" moment across this whole course.

A Genuine Nuance: Not Everything Loads Upfront

Cold start delivers every `@rules/`-included file in full, but only `MEMORY.md`'s own short index for memory — not the full content of every individual memory file. A memory file gets actually read only once something in the conversation makes it look relevant, exactly the "access when relevant" behaviour Chapter 8 described. This is a deliberate difference in scale between the two systems, not an oversight: the rules set is small and universal enough to load in full every time; the memory set can grow much larger over a long relationship with a project, and loading every memory file's full content on every single cold start would be wasteful for information that's mostly not relevant to any one given conversation.

Coding Challenges

CHALLENGE 1

Explain why "the rules arrived automatically" and "the rules are correct" are two different claims, using one concrete example from earlier in this course to illustrate the gap between them.

View solution

CHALLENGE 2

A brand-new session opens in a completely unrelated repository that has never had a CLAUDE.md file of its own. Based on this chapter's own five-step flow, which of those five steps would still happen, and which would not?

View solution

CHALLENGE 3

Explain why MEMORY.md loads its full index at cold start while individual memory files don't — what specific tradeoff does this design avoid?

View solution

CHAPTER 9 QUICK REFERENCE

- **Cold-start order:** global CLAUDE.md → project CLAUDE.md → every @rules/ file in full → MEMORY.md's own index → then the first real message
- Unlike R1, this isn't a prompt to remember — it happens automatically, before the conversation properly starts
- Automatic delivery guarantees the rules ARRIVE reliably — it does NOT guarantee they're still ACCURATE
- Two real, already-found examples of that gap: the duplicated header-comment memory (Chapter 8) and the broken kanji back-link (Chapter 5)
- A renamed path or stale fact is delivered faithfully, every session, until a human actually checks it
- Rules files load in full; memory loads as a short index, with individual files read only when actually relevant — a deliberate scale tradeoff
- **Next chapter:** the capstone — designing and wiring in a brand-new rule category from scratch

CAPSTONE: DESIGNING A NEW RULE CATEGORY FROM SCRATCH

Claude Rules Workflow

Chapter 10 · Capstone: Designing a New Rule Category From Scratch

Every earlier chapter examined an existing piece of this system. This capstone builds a brand-new one, start to finish — using a scenario Chapter 1's own third exercise already planted: a hypothetical `video_lesson_rules.md` that's never actually been written. Time to write it for real, and wire it in correctly.

CH 1	CH 2	CH 5	CH 7	CH 9
Memory vs. rules test	Global vs. project scope	Why a dedicated file, not a shared template	Tracking-file bookkeeping	The one wiring step that's easy to skip

The Scenario

Suppose short video-lesson scripts start getting requested regularly — a scene-by-scene breakdown with on-screen text cues, voiceover lines, and timing notes, meant to feed into the existing Video Generation for Code Tutorials course's own production workflow. The same formatting instructions keep having to be repeated by hand on every request. That repetition is the actual signal worth noticing.

1. **Apply Chapter 1's own memory-vs-rules test.** Is this a fact specific to one conversation, or a convention meant to apply identically to every future video-script request? Every future request needs the identical scene/cue/voiceover structure — that's a rules-file case, not a feedback-memory case.
   Chapter 1 · Chapter 8

2. **Apply Chapter 2's own scope test.** Would this instruction make sense pasted into an unrelated project? No — it only means anything alongside this site's own existing video-related course and folder structure. It belongs in this repository's own rules files, not the global CLAUDE.md.
   Chapter 2

3. **Decide it needs its own new file, not a squeeze into an existing one.** Following Chapter 5's own precedent (kanji and programming lessons each got dedicated files instead of being forced into the

language-lesson template), a video script's scene/cue/voiceover shape doesn't fit cleanly into L1–L7's dialog-and-vocabulary format, P1–P13's chapter format, or PL1–PL3's exercise format. A new file is the honest choice, not a shortcut.

Chapter 5

4. **Pick the next available letter prefix.** P, L, K, PL, LN, MT, SB, and R are all already taken. `V` — for Video — is free and reads unambiguously.

5. **Write the rule itself, numbered from V1.** See the draft below.

6. **Wire the new file into CLAUDE.md's own @rules/ list.** The one step Chapter 9 warned is easy to forget — a correctly-written rules file that's never added here is invisible to every future session, exactly as if it didn't exist.

   Chapter 9

Drafting V1 and V2

video_lesson_rules.md (new file)

```
**V1 — Video Script Storage & Naming**
A video script is a standalone file, one per video, stored at
content/audio-video-production/video-scripts/<topic>.md — no
numbered chapter, matching the Sidebar Lesson precedent (SB3–SB5) of
one independent file per topic rather than a fixed course sequence.

**V2 — Script Structure**
Every script uses four fixed sections, in order: SCENE (a one-line
setting description), ON-SCREEN TEXT (any text cue overlaid on
screen), VOICEOVER (the actual narration line), and TIMING (an
approximate duration in seconds). Repeat this four-part block once
per scene.
```

Two rules is genuinely enough here — the scenario doesn't need P1–P13's full depth, the same way Chapter 6's own Cheat Sheets got a deliberately smaller rule set than a full Sidebar Lesson because the content itself is smaller in scope.

The Wiring Step

Per Chapter 1's own real code block, this repository's `CLAUDE.md` lists every active rules file as an `@rules/` line. The new file only takes effect once a line for it is added there:

```
@rules/permanent_rules.md
@rules/language_rules.md
@rules/kanji_rules.md
@rules/programming_lesson_rules.md
@rules/links_rules.md
@rules/my_tools_rules.md
@rules/sidebar_rules.md
@rules/video_lesson_rules.md   ← the new line
```

THIS EXACT GAP WAS NAMED AS A RISK TWICE ALREADY, IN CHAPTERS 1 AND 9

Chapter 1's own third exercise asked what happens if this precise step gets forgotten, and Chapter 9 named it directly as the one thing automatic loading can't protect against on its own — a file that exists on disk but was never added to this list is exactly as absent from every future session as if it had never been written at all. Actually adding the line here is what closes that gap for real, rather than just describing it a third time.

Verifying This Chapter Followed Its Own Rules

A last, deliberately self-referential check: this file's own banner follows P1 (Chapter 3) exactly — Course, Chapter, File, Topic, and both dates. Its filename follows P5's own snake_case convention. And because this is genuinely the course's final chapter, P4 and P7 both apply the moment this chapter is finished: a combined book-format PDF gets generated (Chapters 3 and 6), and `completed_courses.md` / `course_folder_mapping.md` / `course_bucket_list.md` all get updated in this same turn (Chapter 7) — not as something described one more time, but as something that actually happens right after this chapter is saved.

THE GENERAL CHECKLIST THIS CAPSTONE ACTUALLY FOLLOWED

Is it durable and universal, or conversational and specific? (Ch. 1/8) → Does it only make sense inside this one project? (Ch. 2) → Does an existing rules file's own format genuinely fit, or does forcing it in lose something real? (Ch. 5) → What letter prefix is free? → Write the rule → Add the @rules/ line (Ch.

9) → Update the tracking files in the same turn (Ch. 7). Six real questions, not a vague sense that "this should probably be a rule somewhere."

Final Coding Challenge

FINAL CHALLENGE

A different recurring instruction shows up: "whenever generating a comparison table between two frameworks, always order columns so the framework the user already knows comes first." Walk this through the same six-question checklist this capstone used, and decide: new rules file, addition to an existing one, or memory instead? Justify each step.

View solution

COURSE COMPLETE — CLAUDE RULES WORKFLOW

- **Ch 1–2:** why files beat memory for durable rules, and the global-vs-project CLAUDE.md split
- **Ch 3:** the Permanent Rules file, P1–P13, spotlighted through this exact course's own generation
- **Ch 4:** the natural-language lesson rules, L1–L7, verified against this session's own real lessons
- **Ch 5:** kanji (K1–K4) and programming-lesson (P8/PL1–3) rules, and why each needed its own dedicated file
- **Ch 6:** the Sidebar unification — four content types, one folder tree, a banner format revised in place
- **Ch 7:** the five tracking files, and a real PHP-series drift-and-fix example of P7 in action
- **Ch 8:** the four memory types, and a real, still-open duplication this course found in this project's own memory
- **Ch 9:** exactly what a cold start delivers automatically — and what it can never verify on its own
- **Ch 10:** a full worked example — spotting, scoping, writing, and wiring in a genuinely new rule category