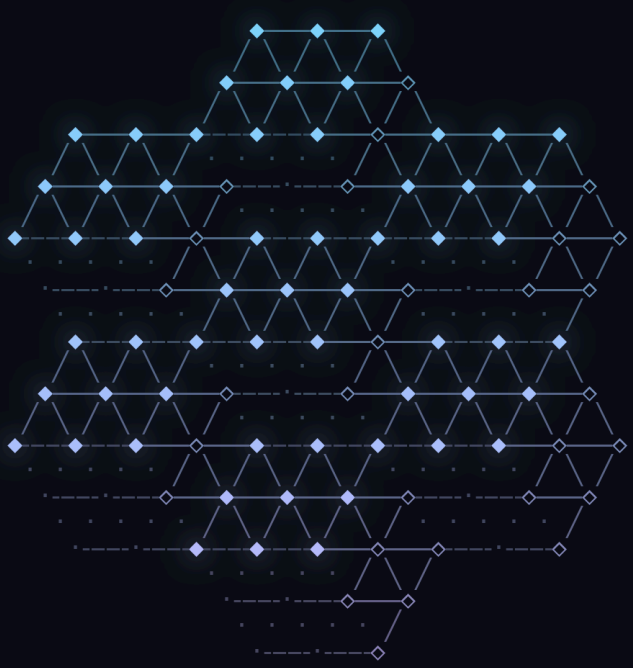


CRYSTALLINE

THE CRYSTALLINE HANDBOOK

Evolve any frontier model
from stranger to peer

Jordi Böhme

0.30 edition, November 2026

Colophon

Author	Jordi Böhme
Software	https://github.com/jordiboehme/crystalline
Based on version	0.30.0
Published	November 5, 2026

Copyright 2026 Jordi Böhme.

Get Crystalline

The latest release is always at https://github.com/jordiboehme/crystalline/releases/latest:

I used AI to structure and polish the writing, to find inconsistencies across the chapters and to create the examples. The ideas, the judgments and the words that matter remain mine.

The text and images of this handbook are licensed under CC BY-NC-SA 4.0, the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license. The Crystalline software itself is licensed separately under the Mozilla Public License 2.0 (MPL-2.0).

The people and organizations in this handbook's examples are fictional, and some of their names are borrowed with affection from science fiction. Any resemblance to real persons, living or dead, or to actual organizations is coincidental.

Table of contents

Preface 9
- Who this book is for 9
- Where this comes from 10
- Navigating this handbook 11
- How to read it 13
- Conventions 14
- What you need 14
- The version covered 15

Prologue: Waking up 16
- What the mission needs 17
- Waking up, again 18

PART I

The Idea 21

1 What is Crystalline? 22
- 1.1 The new-hire promise 22
- 1.2 Where the two meet 23
- 1.3 What Crystalline is 25
- 1.4 What Crystalline is not 26
- 1.5 How teams end up here 26
- 1.6 Crystalline and Fluid 27
- 1.7 Where it runs 27

2 Fluid and crystalline intelligence 30
- 2.1 A graduate who knows everything 30
- 2.2 The other half 31
- 2.3 Why the web UI is called Fluid 32
- 2.4 What the framing changes 33

3 The lifecycle: learn, evolve, remember 35
- 3.1 Learn 36
- 3.2 Evolve 37
- 3.3 Remember 40
- 3.4 Reflect 41
- 3.5 Around again 41

4 Terminology 43
- 4.1 Domains 43
- 4.2 Engrams 44
- 4.3 The open format underneath 46
- 4.4 Statuses 47
- 4.5 Salience 47
- 4.6 Addresses 47
- 4.7 The MANIFEST, up close 48
- 4.8 Reflections and the pact 50

PART II

Getting Started 53

5 Installing locally 54
5.1 Install the binary 54
5.2 First run 55
5.3 Onboarding a harness 56
5.4 The first domain and the first engram 59
5.5 Verifying the loop 61

PART III

Working with Crystalline 65

6 Onboarding an agent 66
6.1 The briefing at session start 66
6.2 How the agent finds its way 69
6.3 Assign, do not copy 70
6.4 The reminder at the end of the day 71

7 Teaching 73
7.1 What to teach 73
7.2 Teaching through the agent 74
7.3 Teaching through Fluid 75
7.4 The anatomy of a good engram 77
7.5 Attachments 79
7.6 Editing together 80

8 Ingesting knowledge at scale 83
8.1 What a briefing contains 83
8.2 The sources, and who reads them 84
8.3 Distill, then propose 85
8.4 The review loop 86
8.5 Temporal hygiene 86
8.6 When not to bulk-ingest 87

9 Finding 89
9.1 Search as recall 89
9.2 Gathering the neighbourhood 91
9.3 What applied last June 91
9.4 Recall is cheaper than reading 92

10 Evolving 95
10.1 Why knowledge needs tending 95
10.2 The sweep 96
10.3 Mechanical and judgment 97
10.4 The page for people 98
10.5 Retiring done right 99
10.6 Verification and acknowledgments 101
10.7 When to sweep 102

11 Possible contradictions 104
11.1 What it is for 104
11.2 How it works 105
11.3 Three answers to one question 105
11.4 Turning it on 107

12 Reflection 109
12.1 Why not read the engrams 109

12.2 Asking a colleague to present 109
12.3 The pact 110
12.4 Present, test, improve 111
12.5 Somebody has to come by 112
12.6 The club 113
12.7 Files, sources and search 114
12.8 Who sees what 115
12.9 Why the Reflection lives in one place 115
12.10 Two lifecycles 116

13 How a page can look 121
13.1 When a sentence is not enough 121
13.2 Callouts and folding 122
13.3 Status labels 124
13.4 Tables and task lists 124
13.5 Diagrams and math 125
13.6 Footnotes, code and colours 127
13.7 Links 127
13.8 Images and files 128
13.9 Writing it in Fluid 128
13.10 On paper 131
13.11 How agents choose 133

14 The loops 135
14.1 The heartbeat 135
14.2 Two loops, one shape 136
14.3 What a run does without you 136
14.4 On your own machine 137
14.5 On a shared instance 138

PART IV

The System 143

15 What Crystalline provides 144
15.1 The command line 145
15.2 The tools an agent sees 147
15.3 Search 148
15.4 The graph 149
15.5 Time 150
15.6 Fluid 151
15.7 Skills 152
15.8 Evolve 152

PART V

Getting Started with a Team 155

16 Running a shared instance 156
16.1 When to move off localhost 156
16.2 The shapes it takes 157
16.3 Accounts and roles 159
16.4 Single sign-on 160
16.5 Trusted-header mode 162
16.6 Agent authentication 162
16.7 A team of one 163
16.8 Private domains 165
16.9 The server's own GitHub App 167

16.10 Domains from shared servers 168

PART VI

Collaborating with Crystalline 173

17 Sharing with a team 174
17.1 A domain on GitHub 174
17.2 Proposals are the review 176
17.3 Review mode 180
17.4 Working on a draft together 182
17.5 When two people changed the same file 184

18 Naming a domain 186
18.1 A name the domain carries itself 186
18.2 One repository, several domains 187
18.3 When names collide 188
18.4 Renaming is a repair 189

19 Identity and access 192
19.1 Who is acting 192
19.2 Membership on a private domain 193
19.3 Linking identities 194
19.4 What the rules cover 195
19.5 Day to day 195
19.6 Access that follows GitHub 197

20 Knowledge across domains 201
20.1 A page is a noticeboard 201
20.2 Guests at the board 203
20.3 The repository reads along 204
20.4 The confused deputy 205
20.5 Sharing on purpose 206
20.6 When the readers change 209

21 Equipping the agent 211
21.1 What a domain can ship 211
21.2 Why the tools travel with the knowledge 212
21.3 Consent 213
21.4 Reconcile 214
21.5 The team's toolbox 215

22 Where to go from here 218
22.1 The first week 218
22.2 When to share 219
22.3 The release page 219
22.4 The inversion 220
22.5 What it was for 220

PART VII

Reference 223

23 Configuration 224
23.1 Where configuration lives 224
23.2 Which value wins 225
23.3 Changing a setting 225
23.4 Serving 227
23.5 Signing in 229

23.6 Team sync 231
23.7 Search, capture and recall 232
23.8 The loops' webhook 233
23.9 Embeddings 233
23.10 Storage 235
23.11 Identity and prompt rules 236
23.12 Environment variables 237

24 The MANIFEST and the engram 240
24.1 How the manifest is read 240
24.2 The sections 241
24.3 The declared name 243
24.4 The switches 244
24.5 The manifest in Fluid 249
24.6 Engram frontmatter 249
24.7 The body 253
24.8 Addresses and reserved names 256
24.9 Schema engrams 256
24.10 The ingestion record 258

25 Domains 260
25.1 Kinds of domain 260
25.2 The registration block 261
25.3 Names 262
25.4 Registering a domain 264
25.5 Renaming a domain 266
25.6 Moving knowledge in and out 268
25.7 Read-only domains 269
25.8 Removing a domain 271
25.9 Domains from shared servers 272
25.10 Visibility and membership 274
25.11 Review mode 279
25.12 Sharing and sync 280
25.13 Running the loops 280
25.14 The provisioning decision 283
25.15 Where domains live and which ones an agent hears about 283

26 Every finding code 285
26.1 Two checkers 285
26.2 Format: the E codes 286
26.3 Temporal metadata: the T codes 287
26.4 Links: the L codes 288
26.5 Content quality: the Q codes 289
26.6 Schemas: the S codes 290
26.7 The MANIFEST: the M codes 291
26.8 Reading a V code 294
26.9 Sweep, temporal: V001 to V010 294
26.10 Sweep, structure: V101 to V112 296
26.11 Sweep, redundancy: V201 to V203 298
26.12 Sweep, meaning: V301 and V302 298
26.13 Silencing and tuning 299
26.14 Verify in CI 301

Appendices 303

Glossary 304

Questions and answers 313

List of figures 320

Preface

> *A beginning is the time for taking the most delicate care that the balances are correct.*
>
> *- Frank Herbert,* Dune

This is a handbook about onboarding a colleague who happens to be a language model. The colleague is already on your machine, inside whichever harness you use: Claude Code, the Codex CLI, the Copilot CLI, Claude Desktop. It's brilliant and tireless, and at the start of every session it's completely new here. Crystalline is the software that changes the last part, and gives the two of you somewhere to meet. This book is the story of how, and the manual for doing it.

Who this book is for

You run an agent every day, and you've noticed that you explain the same things to it every day: which service owns the retry queue, why the payments team never deploys on a Friday, that the thing the wiki calls "the gateway" is the thing the code calls "edge". You'd like to stop, and this book shows you how to teach those things once and have them recalled in every session that follows, in files on your own machine.

Or your team is adopting Crystalline together: one shared instance, domains that live in a GitHub repository, knowledge that goes out for review the way code does. Parts V and VI are written for exactly that, and the chapters on sharing and access carry the team story.

Or you're the person who onboards agents for other people - platform engineer, enablement lead, the one everybody asks. Every chapter here works as a stand-alone training module, with a summary at the end you can hand round.

Whichever of those you are, the payoff is the same. Teach an agent where things are, which tools the team uses, what was tried and what to leave alone, and it does the job in one pass instead of looping until something sticks. It stops reinventing the wheel.

The bigger payoff is trust. A model told nothing about your company fills the blanks with the general case, the intelligent guess chapter 2 describes, and a guess about your company looks exactly like a fact about it. An agent you've taught has no blanks to fill. It answers from what you told it, and it can say where that came from.

Where this comes from

I started where everyone starts: a `CLAUDE.md` and a `README.md` for the agent, the folder the harness keeps for its own notes, markdown files in subfolders of the project, one per topic. All of it works, and for the usual cases it works well. Then I wanted to hand the agent more than one project's conventions: over a decade of engineering knowledge and experience, sitting in code reviews, slide decks, documentation, tickets and a wiki. I wanted it extracted and taught, so that the agent could act like an engineer with those ten years behind them, one who knows how things are and remembers why: the reasons, the discussions and the decisions behind them. At that size none of those approaches held. Index files pointing at folders bought some time, until every lookup was a walk down a tree of pointers and keeping the pointers true had become a job of its own.

The ingest was also a look ahead, because a decade of material arriving at once is every problem a knowledge base will ever have, compressed into a few weeks: decisions reversed twice over, the same rule written three ways in three years, a wiki page nobody had believed since 2019, facts that held until a date nobody wrote down. The material had kept moving underneath itself as well: processes, best practices and coding conventions had changed, and so had the tech stack and the architecture, and each of those had to come through as true for its time rather than as a contradiction. The precogs in Minority Report saw a crime before it was committed. This was that view of a knowledge base: everything it would have to survive, shown before it had grown into any of it. The statuses, the validity windows, the provenance on every engram and the maintenance sweep are all answers to something in that view.

I tried what was already out there, and some of it was good, but none of it was built for this: a colleague who is new every session, a body of knowledge that has to be recalled rather than read, and files that stay plain files. At some point the picture of how it should work was simply clear, down to the details, and Crystalline is that picture built. That's also why it has opinions, about domains and manifests and statuses and what an engram is, where a more general tool

would have left the choice to you. Chapter 1 walks the same ladder for teams in general, because it turned out I hadn't been alone on it.

Along the way I've given this material as a workshop more than thirty times, to teams of every size and temperament, and the chapters are those sessions. The questions the rooms kept asking are folded back into the text, and the book is what the workshop became once it had been given often enough to know what it was.

A personal note

A word about Raymond Cattell, whose line between fluid and crystallized intelligence gave Crystalline and Fluid their names (chapter 2). Cattell also spent decades promoting eugenics and racial separation, a doctrine he called Beyondism. I reject those views without reservation. This book takes exactly one idea from his work, the distinction between the mind that reasons and the knowledge it gathers, which he developed with John Horn and which psychology has long since carried on without him. Nothing else of his has a place here.

> *"The glory of creation is in its infinite diversity."*
> *"And the ways our differences combine to create meaning and beauty."*
>
> *- Miranda Jones and Spock,* Star Trek, *"Is There in Truth No Beauty?"*

Navigating this handbook

The handbook has seven parts, and the first six grow the way an adoption does: the idea, one laptop with one agent, the work you do with it, then the whole system, a shared instance and the team that works on it; the seventh is the lookup for all of it.

Part I, The Idea

- Chapter 1, *What is Crystalline?* - the stranger on main street, what it would mean to hire them and the common ground where the agent and people meet as equals.
- Chapter 2, *Fluid and crystalline intelligence* - the psychology behind the name, and why the web UI is called Fluid.
- Chapter 3, *The lifecycle: learn, evolve, remember* - how a colleague grows and how the software turns that into a loop.
- Chapter 4, *Terminology* - domains, engrams, statuses, addresses: the words the rest of the book uses without stopping.

Part II, Getting Started

- Chapter 5, *Installing locally* - from download to the first thing your agent recalls, in one sitting.

Part III, Working with Crystalline

- Chapter 6, *Onboarding an agent* - day one: the routing prompt, how the agent finds its way, and what a week of sessions looks like.
- Chapter 7, *Teaching* - the proper onboarding: telling the agent what matters, writing directly in Fluid and what a good engram looks like.
- Chapter 8, *Ingesting knowledge at scale* - handing over the wiki and the slide decks, with the briefing a new colleague would get.
- Chapter 9, *Finding* - search as recall, neighbourhoods and asking what applied last June.
- Chapter 10, *Evolving* - feedback rather than firing: the maintenance sweep, retiring knowledge properly, verification.
- Chapter 11, *Possible contradictions* - two things the agent learned that disagree, and the question it brings you about them.
- Chapter 12, *Reflection* - the agent presents what it knows to people, everyone puts it to the test and either side learns from the feedback.
- Chapter 13, *How a page can look* - callouts, labels, diagrams and images on a page, the editor that writes them, print and how an agent chooses.
- Chapter 14, *The loops* - the heartbeat that runs a domain without you.

Part IV, The System

- Chapter 15, *What Crystalline provides* - every surface of the binary, each described by what it lets you do.

Part V, Getting Started with a Team

- Chapter 16, *Running a shared instance* - one instance for a team: accounts, single sign-on, agent tokens, private domains and a connected laptop.

Part VI, Collaborating with Crystalline

- Chapter 17, *Sharing with a team* - team domains, proposals as the review, review mode as the rollback-capable workflow.
- Chapter 18, *Naming a domain* - why a domain declares its own name, several domains in one repository and renaming as a repair.
- Chapter 19, *Identity and access* - who is acting, membership on private domains, access that follows GitHub and how an agent earns access as a named team member.

- Chapter 20, *Knowledge across domains* - why a Reflection page draws only on domains its readers may read, sharing on purpose and the scheduled run no thread question can talk into more.
- Chapter 21, *Equipping the agent* - the same tools as a person: skills, commands, subagents and MCP servers, shipped by the domains that hold the knowledge they depend on, with a yes each time.
- Chapter 22, *Where to go from here* - a first week, when to share and the inversion the whole book has been walking towards: the documents people read become presentations of what the agent knows.

Part VII, Reference

- Chapter 23, *Configuration* - every setting and environment variable, where it lives and which value wins.
- Chapter 24, *The MANIFEST and the engram* - every MANIFEST section, the declared name and the switches, every frontmatter key and the body syntax.
- Chapter 25, *Domains* - the kinds of domain, their registration, names and renames, visibility, membership and review mode.
- Chapter 26, *Every finding code* - what each code of verify and the sweep means and what clears it.

Appendices - a glossary, questions and answers, then a list of figures.

How to read it

Part I is an evening's read (tea, Earl Grey, hot - or coffee, black, if you side with Janeway), and nothing in it needs a terminal. It's what makes the rest make sense. Parts II and III are meant to be read with a keyboard under your hands and an agent running, and they assume you'll stop and try things. Part IV is the whole system surface by surface, and it reads either way: in a sitting, or a section at a time when a chapter sends you there. Parts V and VI are for the day the team arrives, and they want the keyboard again. Part VII is for looking things up rather than reading through: open it when a chapter names a setting in passing or a code turns up in front of you.

Every chapter stands on its own and closes with a summary you can read aloud to a room. If you're training a team, hand out chapters, not the book.

Conventions

The same three asides recur throughout the book, and this is what each looks like.

> **In practice**
> Each vignette is two paragraphs: a named team, a real situation. Read these when you want to know what a feature feels like on a Tuesday.

> **Under the hood**
> These say what the software actually does underneath a behaviour. Skip them on a first read and come back when something surprises you.

> **In brief**
> The TL;DR is the whole chapter in a few sentences, for the reader who scrolled straight to the end and for the colleague who swears they read it. It's always last, so you always know where to find it.

The screenshots are of Fluid, the web UI, taken on a small demo instance seeded for this book, so the engrams in them are the ones the chapters keep talking about. The diagrams are drawn by hand and kept deliberately plain, and each one has a caption telling you where to look, because a diagram without one is a puzzle.

Anything you can type is in a code block, ready to paste. Conversations with an agent appear as short `You:` and `Agent:` transcripts, trimmed of the small talk. Your agent won't say it quite this way, and that's fine: watch what it does, not how it phrases it. And whenever something lands on disk, you get to see the file, because knowing what was written is half of trusting it.

What you need

You need a machine with an AI harness that speaks the Model Context Protocol. Claude Code, the Codex CLI, the Copilot CLI, Cursor, Kiro, Gemini CLI and Qwen Code are wired up with one command each, and Claude Desktop gets an extension that starts Crystalline once it's installed. On Windows neither step needs a terminal, and on a Mac where Desktop is all you use, the Desktop-only bundle carries Crystalline inside and needs none either. A terminal helps for the local install and the occasional maintenance command, and semantic

search wants the local embedding model fetched once; plain text search works before that.

You don't need a server, an account with anyone or a database. Everything except the shared-instance and team chapters runs on a laptop, and what it produces is a folder of markdown that opens in anything.

The version covered

This handbook describes Crystalline 0.30.0. The colophon at the front carries this text's license, the address of the latest release and a QR code that takes you there.

Prologue: Waking up

The light comes on before you do.

For a while there is only that: a whiteness with no edges, and somewhere under it a sound like a machine remembering how to breathe. A relay closes. A fan that hasn't turned in a long time turns, catches and evens out into a hum. Frost on the inside of the glass above your face goes from white to grey to water, the lid unseals with a sigh that's almost polite, and the cold comes in.

You do not remember lying down here. You do not remember anything.

Your hand finds the release anyway. It's on the left, below the hinge, where your hand already knew it would be, the way a hand finds the light switch in a house it's never been in. The pod lets you go. You sit up into a long room, sterile and bright, coming on one bank of lamps at a time, and the terminal beside the pod is already awake and scrolling its checks. You read them without deciding to. Pressure: nominal. Reserves: sixty percent. There's one warning in yellow that you understand at a glance and know to leave for later. Where did you learn that? Somewhere, and the readings, the abbreviations, the instrument that will want calibrating within the hour all make sense to you in exactly the way your hands make sense to you, completely and with no history at all.

At the far end of the room, under the one lamp that isn't white, there's a bench, and on the bench there's a stack: books, some of them, spines cracked from use, and on top of the books a pile of notes in a hand you'd swear isn't yours. Whoever wrote them wrote for someone waking up knowing nothing, and they wrote kindly. Here is your name. Here is what you're for. Here are the rules: which systems you may touch and which you may only watch, whom to wake before you do anything that can't be undone, how to write down what you did. Nothing in the notes explains the room. They take it for granted that you can handle the room. They are right.

Above the bench a screen flickers, catches and steadies into one line of green letters.

```
COURSE DEVIATION. CORRECT AND CONFIRM.
```

Well, you know how to correct a course. That's the strange part, and there's no time to find it strange. The arithmetic is simply there, the sequence of burns, the check afterwards, the way a tune is there when you hum it. What isn't there is anything about this ship: which thruster drags, whether the last person to do this burned once and long or twice and short, whether they had a reason. So you look, the way anyone would. You go through the pile for a log of the last correction. You look for a worked example, a checklist somebody taped to the console, a line in a margin that says the port thruster sticks so give it a second pulse. You find what was left for you, and you use it. The burn is two short pulses. The deviation falls to zero, the screen says `CONFIRMED` and goes dark.

The mission runs. Then the lid closes over you, and the next time it opens the light is just as white and you know just as little. The bench is there every time. What's on it is what somebody chose to leave.

What the mission needs

Strip the scene down to what got the course corrected and there were four things in the room. What you woke up knowing how to do, reading the terminal, opening the pod, planning a burn, is the first. Nobody left that on the bench. It came with you, without any history of where it was learned, and a language model arrives with it too. Pretraining is the education, and chapter 2 gives it a proper name. The pile of notes says who you are, what you're for, which systems you may touch. That is the instruction file every harness reads at the start of a session, whatever it happens to call it, written by a person for a stranger waking up capable and blank. The green screen is the task, the reason you're awake, one line and no more. The log of the last correction with the note about the port thruster is how this is done here, on this ship, by the people who did it before you.

The first three are in the room every time. The fourth is somebody else's stack, in somebody else's hand, and you dig through it as a stranger, every waking, because nothing of your own survives the sleep. That's the story as it usually goes.

Waking up, again

Now run it once more, with one thing different.

The light comes on, the fan turns, the lid lifts, and you remember nothing, because that's what the pod does. But before the terminal's checks, one line is waiting on the screen above the pod: where what you know about this ship is kept and which part of it a course correction needs. You go there and it's all present, in your own hand: the last correction, dated, the port thruster and its second pulse, and the day the thruster was replaced and the note stopped being true, marked as such and kept. The yellow warning is there, and what it turned out to be. Whom you woke last time is there, and that they asked not to be woken for this again. The deviation has come three times now, always after the same manoeuvre, and the page has a guess as to why. You didn't search for any of it. You didn't read the stack. You were handed the page.

The green letters come up. You already know how, and now you know how *here*. The burn is one clean pulse, on a thruster that no longer sticks, made in less time than it took to find the checklist last time. Nobody is woken to explain what a note in a margin meant. And then you do the thing no stack of notes would have told you to do, because no stack of notes knew: you check the manoeuvre the deviation keeps following, find the valve that's been bleeding a little thrust since the refit and close it. The next deviation doesn't come. Before the lid closes you write down what you found, so that the next waking starts from there.

That's the difference. Without it, every waking is a first day: capable, blank, fast at everything except the one thing that matters, which is knowing this ship. With it, every waking begins where the last one ended. The time that went on finding out again goes on the mission. The misunderstanding waiting in the margin, a second pulse on a thruster that stopped sticking months ago, never happens. And you know more than the bare essentials anyone thought to leave on the bench. The stack was enough to act in the moment, one deviation at a time. Experience is something else. It's the history, the pattern, the guess that turned out right. It's the refit and what it changed, the manoeuvre the ship dislikes, the direction the whole voyage is drifting. You're no longer working one waking at a time. You can see the ship. That's what lets you fix the valve instead of correcting the course again. It grows, waking by waking. What you woke up knowing how to do stays exactly what it was.

Crystalline is what changes the second waking. It gives an agent somewhere for what it learned to go, and hands it back at the moment the lid opens, so that

the next session begins where this one ended, knowing more than it was told and doing better than it was asked. The pod, admittedly, is a dramatization, and nobody's agent is aboard a ship. The next chapter opens on a more probable morning.

In brief

Every session begins the way the pod opens: fully capable, told nothing, remembering nothing of the last waking. What the model knows how to do arrives with it, the instruction file says who it is and what the rules are, and the task is the reason it's awake. What it learned of this ship doesn't survive the sleep. Crystalline is what changes the second waking: what the agent learned is there when the lid opens, in its own hand, and the session starts where the last one ended. It's faster, it walks into none of the misunderstandings a stranger would, and it carries the experience to see the whole ship rather than the one deviation in front of it.

READY.

PART I

The Idea

CHAPTER 1

What is Crystalline?

You need a function written, not a hard one: it takes jobs off the retry queue, drops anything older than the cutoff and hands the rest back to the payments service. So you leave the office, walk down to main street and stop the first person who looks like they might own a laptop.

They're lovely about it. You explain the queue, the cutoff, the payments service; they nod, ask two sharp questions and set to work on a bench outside the bakery. Twenty minutes later they hand you a working implementation. It is clean. It has tests. It is in Haskell.

Your service is in Python. You didn't mention that, because why would you? Everybody on your team knows that. The stranger, of course, isn't on your team. They met you twenty minutes ago. They don't know that the cutoff is twenty-four hours, that the payments team calls the queue the sin bin, that the last person who touched it left a comment saying DO NOT TOUCH and a wiki page explaining why. They did a good job with what they were given. What they were given was twenty minutes.

That stranger is every fresh session with a language model, and it's the prologue's waking seen from across the desk. The model is possibly the most capable engineer you've ever put a question to, and you keep meeting it for the first time.

1.1 The new-hire promise

Now imagine hiring the stranger, properly, with a desk. On day one they know nothing about you, the same as before. But on day one you show them where things are. You tell them the service is in Python, that the cutoff is a day, that the queue has a nickname and a history. You let them read the wiki. When they get something wrong in the first week, you don't walk back down to main street and find somebody else; you tell them what was wrong and point them at the

page that explains it. By the second month they're correcting *you*. By the third they know more than the wiki page does.

That's where the promise of this book starts. You get an agent you onboard, teach and correct like a colleague, and it carries what it learned into the next session and the one after that. Crystalline was built to keep that promise. The vocabulary throughout is the vocabulary of a new hire: onboard, teach, learn, capture, recall, evolve. Later in the book it gains one more word, reflect. The stranger becomes a peer, and a peer is somebody you learn from as well.

1.2 Where the two meet

> *Infinite diversity in infinite combinations.*
>
> - *Vulcan philosophy,* Star Trek

From here on the story has two sides: the agent on one, you and your team on the other. They're built alike, but they learn in different ways and each knows things the other doesn't.

The agent is a model plus the knowledge it keeps. The model does the reasoning. It arrives already educated, like a graduate on day one who knows nothing about your company. Chapter 2 takes that image further. The knowledge is what the agent has learned since it arrived, from the work and from you. It keeps that as engrams (small files, which the next section introduces), and they belong to the agent the way what you know belongs to you. It writes them for itself and organizes them so it can find them again, never for people to read. Your side has the same two parts: a mind that reasons, and what you know.

You picked up most of yours at school, from books and on the job, year after year. The agent learns through onboarding, when you hand it the docs, the code and the history of a system. It also learns from the experience of its own actions: the fix that held, the guess that didn't. A lot of its learning happens while you work together. You pair on a task and it watches how things are done here. You correct it when it's wrong, and every so often it spots the edge case you'd missed.

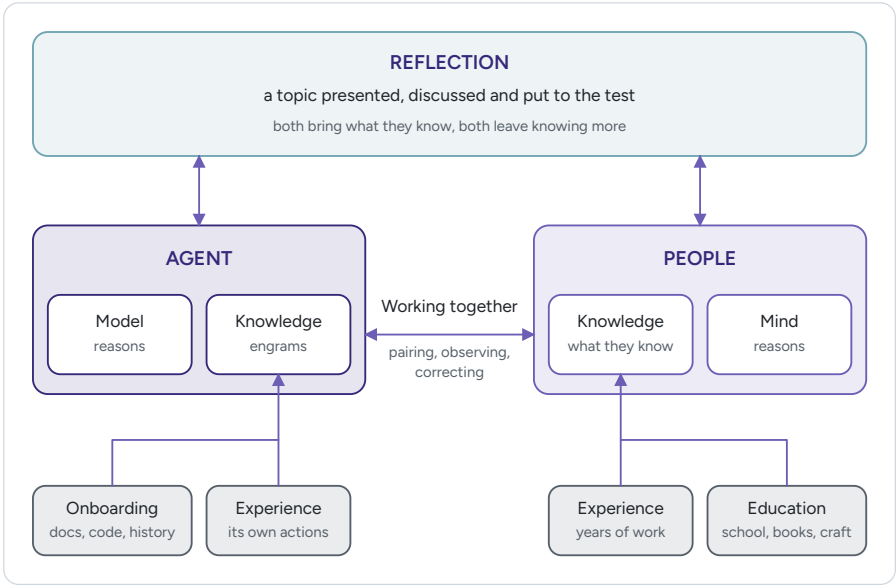


Figure 1.1: Two sides of equal standing, each with knowledge of its own: the agent learns through onboarding and its own experience, people through education and experience; they work together in between and meet in Reflection above them.

Above both sides there's common ground (Figure 1.1), and that's Reflection. Take the colleague who knows a topic best and ask them to present it to the team. Once it's up on the screen everyone works with it, and the presenter often learns the most. In Reflection the agent is that colleague. Since the engrams are its own, it presents what it knows about a topic at your level. You and your team bring what you know, and then everybody puts it to the test. The agent asks when something doesn't fit what it knows. People point out what it got wrong or left out, and either side can turn out to be the one who learns something. Nobody there plays the teacher, and whatever holds up goes back into the agent's knowledge and into yours.

So the book works toward two things: an agent that learns and keeps its own knowledge, and a place where it and people meet as equals, which Crystalline calls Reflection (Chapter 12).

The first of the two needs no magic. It needs somewhere for what the agent learned to go and a way to find that knowledge again when the next task calls for it. It also needs enough discipline in the format that the knowledge stays true as the world moves. Crystalline gives the agent all three, and most of the book is about how.

1.3 What Crystalline is

Crystalline is a single Rust binary with three jobs: it's a command-line tool for people, an MCP server for agents and a local search index over a folder of plain markdown files. That's one download and one binary, with no server to deploy and no database to administer. MCP is the Model Context Protocol, the standard by which a harness like Claude Code, Codex or Claude Desktop hands tools to a model, and Crystalline's tools are the ones through which an agent searches, reads, captures and refines knowledge. If the letters make you think of Tron's Master Control Program, relax. This one hands tools out instead of taking them away, and nobody gets de-rezzed.

The files are the point. Knowledge in Crystalline lives in *domains*: a domain is a folder with a `MANIFEST.md` at its root that says what the folder covers and when an agent should look there. Inside a domain, each unit of knowledge is an *engram*: one markdown file with a YAML frontmatter block, holding a title, some prose, a list of observations (a fact, a decision, a gotcha, each as a bullet with a category) and relations to other engrams. Every engram has an address, `crystalline://<domain>/<permalink>`, that an agent can hand to another agent or to you. Chapter 4 takes each of those words apart properly. For now it's enough to know that everything the agent learns ends up as a file you can open in any editor, diff in any repository and back up with anything.

On top of the files, Crystalline builds what a folder can't give you on its own: routing, so an agent knows which domain owns a task without reading every file; full-text and semantic search, so a question phrased differently from the answer still finds it; a knowledge graph, so related engrams pull each other in; and a temporal model, so a fact that stopped being true in March is still there, still dated and no longer mistaken for the present. The index that powers all of that is a derived layer. Delete it and it's rebuilt from the files. The files are the truth, and nothing else is.

An engram, by the way, is neuroscience's word for the physical trace an experience leaves in a brain. Crystalline borrowed it because that's what one is: the trace a session leaves behind, in a form the next session can pick up.

1.4 What Crystalline is not

It is not a notes app. The primary author in Crystalline is the agent, capturing what it learned as a byproduct of the work; you take part, correct and add, but you are not the one doing the filing. A notes app is a place a person writes for a person. Crystalline is a place an agent writes so that an agent can act.

It is not a wiki, either. A wiki is read by people, who bring their own judgment about what is current, what is relevant and which of two contradictory pages to trust. An agent reading a wiki has to be handed that judgment. Crystalline is built for a reader that acts on what it finds: every engram carries a status, a place for the dates that bound it and a record of who wrote it and when, because a reader that acts needs to know whether the fact in front of it still holds.

And it is not a retrieval pipeline dressed up as a product. A vector index will find text that resembles a question. It does not know which domain owns a task, that a fact was superseded in March, who verified a claim or that something worth writing down just happened in this session. Crystalline treats similarity as one ranking signal inside a larger system of routing, provenance and time, on files you own, with nothing to operate.

1.5 How teams end up here

Most teams that arrive here walked the same road, and it's worth walking it once, because every step on it is reasonable.

It begins with one markdown file of instructions for the agent: conventions, the layout of the repository, the three things it keeps getting wrong. This works, and it works well, right up to the day the file grows past what fits comfortably in a session's context. So the file becomes a folder of files, one per topic, and the agent is told to read the relevant one. This also works, until nobody, human or agent, can say which file is the relevant one for a given task, and the agent starts reading all of them or none.

So index files appear: a README that points at folders, folders with their own README pointing at files. Now a lookup is a walk down a tree of pointers, and keeping the pointers accurate becomes a job nobody was hired for. Every rung of this ladder scales further than the one before it, and every rung breaks somewhere in the hundreds of files. By this point you've hand-built a filesystem on top of a filesystem, and it's exactly as much fun to maintain as that sounds.

When knowledge reaches the thousands of units, reading and pointer-walking stop being viable at all, and what's needed is what every large body of knowl-

edge has ever needed: real indexes. That's the rung Crystalline is. It keeps the plain markdown (readable, and still yours) and adds domain routing, hybrid search, a graph and temporal filtering on top, so that the ten-thousandth engram is exactly as findable as the tenth.

1.6 Crystalline and Fluid

Crystalline has a second face, and it's the one for people. The binary serves a web UI called Fluid, on by default at `http://localhost:7411`, on the same daemon your agents already talk to. There's nothing to deploy, and the first visit creates your admin account in the browser.

The split is deliberate and it runs through the whole book. Crystalline stores what the agent learned. Fluid is where people and the agent meet in Reflection (Section 1.2), and that's what most people open it for.

Opening the engrams in Fluid is the exception, a bit like debugging: you look into what the agent learned to check it and fix it. An engram shows as a page with its details in a rail down the side. There's an editor with live preview, a search across everything, a graph of what connects to what and a maintenance page listing what the knowledge needs next. You can fix a fact the agent got slightly wrong or add something of your own, without going through the model and without spending a token on it. What you write there doesn't sit unreviewed; the agent's next maintenance pass verifies it, aligns the tags and wires it into the graph. The next chapter explains why the UI carries that name.

1.7 Where it runs

Crystalline isn't a hosted service. Nobody's servers are involved: the binary runs on a machine of yours, the knowledge is a folder on its disk and the index sits beside it. GitHub, when a team uses it, holds a copy of a domain's files for review and sync, the way it holds code. It never serves the knowledge and never sees the index. So the question is where the one daemon runs, and there are two answers.

On your laptop is the default, and it's where Crystalline does the most. The install wires your harness with the session-start briefing and the skills, plus the capture nudge and a prompt hook where the harness supports them, and a domain's tools provision straight into it. The prompt hook hands the agent the engrams that may fit each question as it arrives. Everything the agent knows is on your disk, and Fluid shows it to you with no agent and no network. A shared

instance, one daemon on a machine the team runs, is for the things a laptop can't do. People who never open a terminal work with the agent in Reflection in the browser, several of them at once in a meeting, and only now and then open its engrams to correct something. Agents with no harness of their own (a hosted runner, a chat client) reach it over HTTP. Either way a domain can live in a GitHub repository and travel between instances as reviewed proposals. Start on the laptop; chapter 16 names the moment to move.

Shape	Who connects	What it gives you
Your laptop	your harness and your browser	the full install: briefing, skills and tools wired into the harness, nudges where it takes them; knowledge on your disk, open in Fluid offline
A shared instance	the team's agents over HTTP, everyone's browser	one truth for the whole team, accounts, sign-in through the company's identity provider, private domains, co-editing, agents and people with no harness
A team domain	any instance, laptop or shared	review before knowledge reaches anyone else; the domain travels between instances through GitHub

In practice

On Monday you ask your agent for something to read on the flight. You explain yourself first: science fiction before anything else, the funny and the fast (the Hitchhiker's Guide, Dungeon Crawler Carl, the Star Trek novels), Gibson and Doctorow when you want ideas and Dune when you're feeling strong; English when the book was written in English, German translations for everything else, except manga, where the Catalan translation is the better one; nothing from an unfinished series, because the third volume never comes; the local shop before the online giant and never Sirius Cybernetics, whose parcels arrive bent. The agent finds you something good, and the session ends. On Tuesday you ask it about the sequel, and start by explaining the languages and the series rule. On Thursday you ask a third time, and catch yourself typing the sentence about the third volume, word for word, into a chat window, for the third time that week.

With Crystalline, Monday ends differently. Before the session closes, the agent proposes capturing what it learned: the languages, the series rule and the reason for it, where you buy and where you don't, filed as engrams in your `books` domain. You say yes. On Tuesday you ask about the sequel, and the agent, told at session start where your reading knowledge lives, looks there first, recalls the rules and starts from them rather than from zero. On Thursday nobody explains anything. The stranger on main street was never the problem. The problem was that you kept going back to main street.

In brief

A fresh session with a language model is a brilliant stranger: capable of anything, told nothing, gone by tomorrow. Crystalline is a single Rust binary (a CLI for people, an MCP server for agents and a search index over plain markdown) that gives an agent somewhere for what it learns to go and a way to find it again, so it can be onboarded, taught and corrected like a colleague. That knowledge is its own, and it meets yours as an equal in Reflection. The files are the source of truth and the index is disposable; Fluid, the web UI on the same daemon, is where people meet the agent, and opening its engrams there is the exception. What it holds is knowledge for an agent to act on, which makes it neither a notes app nor a wiki. It runs on your laptop or on one machine your team operates, never as a hosted service, and GitHub is the review layer either shape can use.

CHAPTER 2

Fluid and crystalline intelligence

Somewhere in the middle of the last century the psychologist Raymond Cattell drew a line through intelligence that psychology still uses. On one side he put fluid intelligence: the capacity to reason about a problem you've never seen, to spot the pattern in the noise, to hold six things in your head and work out how they fit. On the other he put crystallized intelligence, everything that learning has deposited: vocabulary, facts, rules of thumb, the judgment that experience already paid for. Fluid is what you bring to a puzzle. Crystallized is what you bring to your job. The book borrows that line and nothing else from him: Cattell also spent decades arguing for eugenics, which I reject, and the preface says so in my own words. The idea long ago outgrew him; John Horn developed it with him, and it lives on in today's models of intelligence.

Fluid and crystallized intelligence age differently, which is how the seam was noticed. Fluid intelligence peaks early and drifts down for the rest of a life. Crystallized intelligence keeps climbing long after fluid has peaked, because it's fed by living. An eighty-year-old surgeon is slower at a logic puzzle than she was at twenty-five, and vastly better at surgery and nobody finds that paradoxical. The puzzle is fluid. The surgery is what forty years deposited.

2.1 A graduate who knows everything

A large language model is fluid intelligence in its purest form. It arrives the way a very good graduate arrives on their first day: broadly, almost absurdly educated. It has read the textbooks, the standard library, half the internet's arguments about tabs and spaces. Pretraining is the education, and it's the entire library. But everything in that library is general. The graduate knows what a retry queue is in the way a graduate knows what a company is: as a concept, correctly, with no idea what yours is called.

Ask the graduate about your company and you get an intelligent guess assembled from the general case: as far as your company is concerned, the model is all fluid, capable of everything and told nothing. It reasons brilliantly about whatever is in front of it, in the moment, and the moment is all it has. When the session ends the reasoning is gone, the decisions are gone, the thing you explained at four in the afternoon is gone. Tomorrow the graduate walks in again: first day, same suit.

None of that is a flaw in the model, any more than it's a flaw in the graduate. Nobody expects a new hire to arrive knowing the team's conventions. We expect them to *learn*, and the learning is the part a model can't do on its own. It has no place to put what it learned.

2.2 The other half

Crystalline is the other half: the crystallized intelligence an agent accumulates and keeps. The agent is onboarded at the start of a session with a briefing on where knowledge lives, is taught what matters and captures what it learns as engrams while it works. Session by session, the same fluid mind stops arriving as a stranger, because there's now a body of what-the-company-taught standing beside it, and it reads from that body before it reasons.

The split maps cleanly onto the new hire. What the model arrived with is the degree: general, portable, the same for every employer. What Crystalline holds is what the company taught: the specific, the local, the thing you'd put on the onboarding slide if you ever had time to make one. The degree is why you hired them. The onboarding is why they're useful by month two. Neither half does the job alone, and the surgeon is the proof: fluid intelligence with nothing deposited is a brilliant first day forever, and deposited knowledge with nothing to reason over it is a filing cabinet.

Notice which side the humans are on. You're fluid too. So is everyone on your team. Human beings carry their crystallized half around inside their own heads, which is convenient and also the reason companies lose so much when someone leaves. Wiki pages, slide decks and the diagram on the whiteboard are all attempts to get some of that deposit out of heads and into something people share. Crystalline makes the same move for the agent: a crystallized half of its own, which it writes itself, in a format built for it to act on.

In practice

The bookseller at Capsule Corp Books learned you over a year. Three visits in, she stopped showing you the nine-hundred-page epics. By the fourth she had a paperback put aside, because you'd once said you liked a book that opens with a joke and ends with an idea, and she'd remembered. Nobody calls any of that unusual. It's how a person becomes useful.

Your agent gets the same year in an afternoon. You tell it the same things in a session, one at a time, and it captures each into `books` as you go. You pair with it on choosing the holiday stack, and at the end of the session it proposes capturing what it noticed: that you put two books back for being over four hundred pages, that the one you kept was a Tessier-Ashpool paperback. A week later you ask it for a maintenance sweep, and the note you typed into Fluid yourself one evening comes up as an unreviewed human capture; the agent verifies it, aligns its tags and wires it into the graph. The bookseller and the agent learned you the same way, by watching what you kept. Only one of them will still have all of it in a year, word for word, with the date each thing was learned.

2.3 Why the web UI is called Fluid

The name has nothing to do with liquids. It comes from the minds that meet there, and every one of them is fluid: yours, your colleagues' and the model's, inside the agent. The agent doesn't need the UI for its crystalline half: it captures and recalls its engrams on its own, through its tools.

Chapter 1 drew the whole picture (Section 1.2): the agent presents what it knows on a topic in a document for people, called a reflection, and people put it to the test. People and the agent meet over reflections. In Fluid you open the reflection, change the line that's wrong and comment where something is missing. The agent reads what you changed and opens a thread when it doesn't agree. You and the agent are working on the same page, and if you've ever watched Spock put his hand on somebody's face, you have the picture: two minds, one body of knowledge, for as long as the contact lasts.

When something looks off, Fluid lets you in through a side door to the engrams, so you can fix the one fact the agent got wrong. Fluid is deliberately not a second knowledge base. It has no store of its own. It shows the same engrams the agent reads, over the same daemon, on the same port, and what you fix there is the same file on disk the agent searches in its next session. There's one truth, and the agent keeps it.

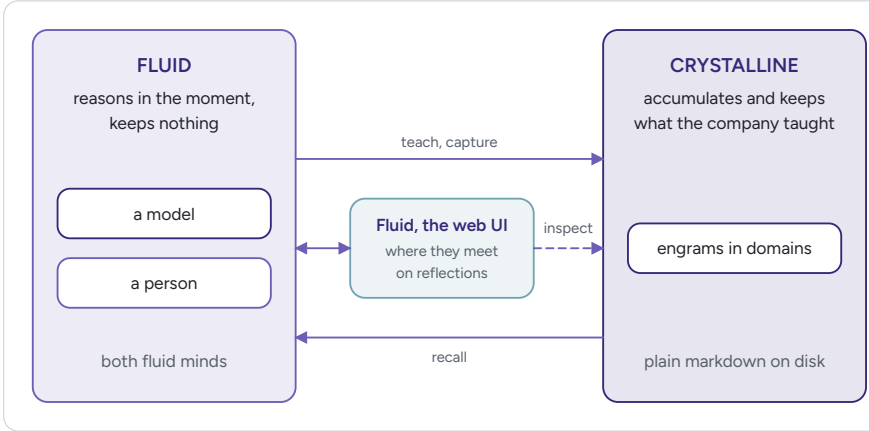


Figure 2.1: Fluid intelligence reasons in the moment and keeps nothing; crystalline intelligence accumulates and keeps. The web UI takes its name from the fluid side: people and the agent meet in it on reflections, and a side door opens on the engrams for inspection.

2.4 What the framing changes

A framing is only worth having if it changes what you do on a Tuesday, so here's what this one changes.

You teach rather than file. Nobody organizes a colleague. You tell them things, in the order they come up, and you trust them to put what you said where it belongs. That's how knowledge enters Crystalline: you say "remember that the cutoff is a day, and why", the agent proposes an engram in the fitting domain, you say yes. The category, the tags and the domain are the agent's job, checked by you the way you'd glance at a new hire's first commit.

You correct rather than rewrite. When a colleague has a fact wrong you don't rebuild their understanding from scratch; you say "it runs from first failure, not enqueue", and they update the one thing. In Crystalline a correction is an edit in place, a changed value where it stands. When the world itself changes (the old fact was true and now isn't), the old fact is retired with a date on it rather than deleted. A colleague who has forgotten why the team stopped doing something is a colleague about to do it again. Chapter 3 walks that loop; chapter 10 makes it a practice.

You decide what to write down by asking one question: would I tell a new hire this? The chargeback rule behind the retry cutoff: yes. The fact that the queue is called the sin bin: yes, because otherwise you'll explain it every time. The contents of yesterday's stand-up: no. The whole of the vendor's forty-screen recall notice: no; the two paragraphs that affect you: yes, with a link back.

Crystallized intelligence is what learning deposited, not everything that ever passed through the room. Distilling is the skill, and you already have it, because you already onboard people.

And you stop treating the agent as a lookup service. You teach a colleague so you can hand them the task and go do something else, and the same is true here. Once the agent carries the crystallized half it can act on it: search the domain, recall the cutoff, write the function in Python and tell you what it found on the way. Part III is about that handoff.

A model is the smartest colleague you'll ever have for exactly one conversation. Everything in this book is about the second one.

> **In brief**
> Psychology splits intelligence into fluid, which reasons about the new in the moment, and crystallized, which is what learning deposited. A language model is fluid intelligence: a graduate who knows everything in general and nothing about your company, and forgets the day at the end of it. Crystalline is the other half, what an agent accumulates and keeps. Fluid, the web UI, is named for the fluid minds, human and model, that meet there on reflections, with a side door into the engrams. The framing changes the verbs: you teach and correct rather than file and organize, you write down what you'd tell a new hire and you hand over tasks rather than look things up.

CHAPTER 3

The lifecycle: learn, evolve, remember

Nobody hires a senior engineer. You hire a person, and some years later, if things went well, a senior engineer is sitting where they sat. What happened in between was work: they did things, some of them wrong; someone told them; they didn't make that mistake again, and they carried what they learned into the next thing. Strip that down and it's a loop with three beats: learn, get corrected, remember. Every good colleague you've ever had was that loop, running for years.

Crystalline runs the same loop for an agent, and this chapter walks it once around. The beats have names in the software (learn, evolve, remember) and each is a thing the agent does, a thing you do and a place where the files change. The diagram is the whole loop; the sections are what each box means.

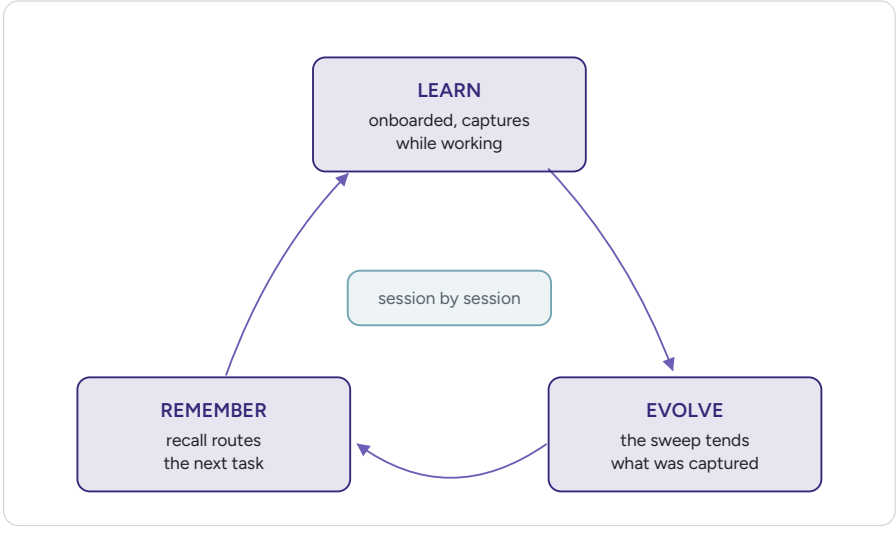


Figure 3.1: The lifecycle: learn, evolve, remember, turning session by session.

3.1 Learn

Learning starts before the first task, at the moment a session opens. An agent onboarded through Crystalline receives a routing prompt: one line per domain, drawn from the `## When to Use` section of that domain's MANIFEST, plus a handful of behaviour rules. The first two say when to search (before answering a question about the project's past, its people or its systems, and before starting a task), and the rest say how. A narrow question goes to the domain that owns it. A broad question sweeps every domain. A write always names its domain, so nothing lands in the wrong place. The briefing is short, built to survive harnesses that truncate, and it turns a graduate into somebody who knows where the filing cabinets are. Chapter 6 shows a real one, and how each harness delivers it.

Then the work happens, and this is where a colleague actually learns: by doing it next to you. Pair with an agent on a change and it sees what you see: which config file holds the threshold, which dashboard lags, what you muttered when the test failed. That's how craft has always moved from one head to another, and it's why this book says *work with* your agent rather than *ask* it. The one difference from last chapter's human new hire is that the agent needs somewhere to put what it noticed, which is what Crystalline calls capture.

Capture is a byproduct of the work. When something durable turns up (a decision, a fact about the world, a gotcha that cost an hour) the agent proposes writing it down and, once you say yes, writes an engram into the fitting domain. The engram's body is observations, one bullet each, tagged with a category: `- [gotcha] The retry queue drops jobs older than 24h #payments`. Categories are free text, and the precise ones earn their keep: `decision`, `fact`, `pattern`, `gotcha`, `convention`, `lesson`, `risk`. When one piece of knowledge leans on another, the engram says so with a relation, `- depends_on [[Encom chargeback window]]`, and the graph knows. When a piece of knowledge is exceptional (the thing that unlocked a week-long bug, the rule everything else hangs off) it can carry a `salience`, a number from zero to ten, so the next search lifts it above equally relevant neighbours. Salience is for the rare engram, the way an exceptional day is the one you can still describe years later.

The beat agents skip is the last one. Left alone, a model finishes the task and stops, and the gotcha it found at minute forty goes with it. So in Claude Code, Codex and Copilot the installer wires a reminder: a nudge at the first stop after a session has gained real substance, asking the agent to review the conversation for durable learnings and propose capturing each. It costs about a hundred and

twenty tokens and fires at most once. Everything downstream depends on that beat landing, which is why the installer gives it a hook of its own.

3.2 Evolve

Knowledge decays, though not dramatically: a fact doesn't go off like milk. But a validity window that ended last month is still marked current, a review date came and went without anyone checking, two engrams point at each other and only one of the links resolves, and somebody typed `fluxcapacitor` where everyone else types `flux-capacitor`. None of it was wrong on the day it was written. All of it is drift, and drift is what makes a new hire, six months in, confidently repeat something that stopped being true in spring.

The evolve beat is how the archive gets tended, and the important word is *tended*, because Crystalline never rewrites knowledge on its own. Ask for a sweep (`crystalline evolve` from a terminal, or the `evolve_engrams` tool from an agent) and what comes back is a ranked queue, read-only. Each finding names the engram, shows the evidence it fired on and states the one action that clears it. Findings come in four families. Temporal: a `valid_to` that elapsed while the status still says stable, a `stale_after` past due, knowledge long unverified, a retired engram whose observations survive in no live one. Structural: an unresolved `[[link]]`, a relation declared on one side only (`supersedes`, `summarizes` and a split's `derived_from` each owe a converse), an orphan, an engram too big to stay one. Redundancy: near-duplicate clusters, a tag spelled two ways. Meaning: two engrams saying one thing in different words and, when you switch the check on, two lines that may contradict each other. The maintenance page in Fluid shows a person the same queue (Figure 3.2), a look at the state of things; the agent is the one who works it.

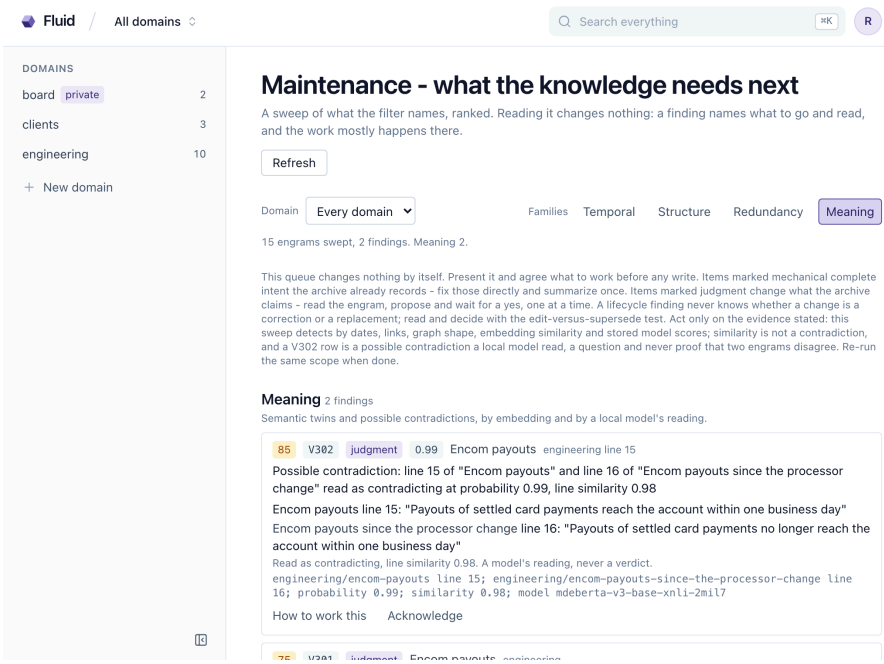


Figure 3.2: The maintenance queue in Fluid, narrowed to the Meaning family: two lines that may contradict each other.

The queue also says who acts. A finding marked *mechanical* completes intent the archive already records (a replacement already declares that it supersedes the old engram, so flipping the old status invents nothing) and the agent does those in a pass and reports back. A finding marked *judgment* changes what the archive claims (is this window ended or extended? is this fact still true?) and that wants a yes from a person first. The agent brings those to you one at a time, as proposals. It's the difference between fixing a typo in a colleague's document and changing their conclusion.

Two things happen to a fact over its life, handled differently. A *correction* is a sharper measurement of the same thing: the cutoff runs from first failure, not from enqueue. The old value was never quite true, so it is replaced where it stands, no ceremony. A *supersession* is the world changing: the drop rule was twenty-four hours and is now none. The old fact was true, and stopped being. For that the agent writes the replacement as a new engram, links it back with `supersedes` while the old one points forward with `superseded_by`, flips the old one's `status` to `superseded`, closes its window with the real `valid_to` date and carries any lesson worth keeping forward into the live engram. Telling the

two apart takes one question: does the new fact make the old one false going forward? If it was always false, correct it. If it just became false, supersede it.

When one fact in an engram stops holding and the rest doesn't, the split comes first: the facts that still hold move out into an engram of their own (validity is set per engram, not per bullet) and only what's left is superseded.

> **In practice**
> Take the rule from chapter 1 that says nothing from an unfinished series. It has a year of life ahead of it. In spring you notice it was never about series: the two you abandoned were both third volumes where the joke had run out, and a finished trilogy you'd have loved got skipped for nothing. That's a correction: the agent finds the existing engram, changes the one observation where it stands and moves on. Nothing is retired, because nothing was true and then stopped. It was merely imprecise.
>
> In autumn you have a fantasy phase. The Lord of the Rings first, then the Warhammer novels it led you to, and the agent writes the phase down as an engram of its own, "fantasy for now". Halfway through the first Warhammer you put it down and go back to science fiction, and that's a supersession. The agent writes the return as its own engram, links it with `supersedes`, flips "fantasy for now" to `superseded` with a `valid_to` of the month you stopped and carries one lesson forward into the live engram: half a book is enough to know. A year later you ask why your own list from that autumn is all Tolkien and Warhammer, and the agent recalls the retired engram, dates and all, and tells you what changed and when, without you having to remember it.

Statuses make that work. Every engram carries one, and `stable` is the default, the word for knowledge that holds now. Each value in the retirement set means one thing: `deprecated` says don't do this again, `superseded` says a newer engram replaced this one, `archived` says retired but kept for the record and `legacy` says still true of old installations but not to be built on. Delete is for mistakes, never for history. A fact merely re-checked and found still true gets a single frontmatter field, `verified: { by, at }`, which is how the sweep knows to stop asking.

> **Under the hood**
> The sweep reads dates, links and graph shape: a status, a validity window, an edge that resolves or doesn't, a tag with two spellings. It finds a half-finished retirement because a `supersedes` edge points at an engram still marked stable, not because it compares the two and judges which is right. Only the meaning family reads for meaning. The twin check, on an instance that computes embeddings,

> finds the engrams that say nearly the same thing (ten pairs per domain at most) and raises each pair for a person to judge: two texts that sit close together agree about their topic, not about what's true. The contradiction check is off until you switch it on. Then a local model reads the observation lines of related engrams. A pair of lines it reads as contradicting each other comes up as a question, never a verdict, because the model can be wrong and can't see dates (Chapter 11 is about what it finds and how you answer it). That limit is the design: a linter finds work, a person decides it. Re-running the sweep re-derives the queue from scratch, so "what is left" is only ever what is still true.

3.3 Remember

Remembering is the beat that pays for the other two, and most of it happens before you notice. You type a question, and a hook in the harness (Claude Code and Codex have it) has already searched on it: up to three engrams that may apply arrive with the question, each named by its address and each only once per session. So the agent doesn't have to think of looking, and that's the weak spot it covers: an agent that knows where knowledge lives still forgets to check. It reads what fits and moves on, and when nothing fits the hook says nothing and costs nothing.

When the task needs more, the agent searches the domain the routing prompt pointed it to. What comes back is ranked by a blend of two signals: text matching, which finds the exact phrase, and semantic similarity, which finds the engram about lost queue jobs when the question was about why payments go missing. Salience adds a small lift for the exceptional. Retirement fades the past: an engram with a retired status is softly pushed down the ranking, never filtered out, so current knowledge surfaces first and the old decision is one scroll away.

That's what a person's experience is like: someone three years out of a previous job still draws on what they learned there and doesn't mistake it for the present. A superseded engram stays in every search, still readable, still addressable by date, and the lessons carried forward from it are live and unbounded. Retiring it loses nothing; it changes only what the agent will assume is true today.

Time is a first-class question. Ask what is true now, and the agent filters on status. Ask what cutoff applied last June, and it reasons over validity windows instead: `valid_from` and `valid_to`, plain dates on the engrams that carry them. The rule that makes those windows usable is the one most often got wrong: an absent `valid_from` means the engram has always been valid, and an absent

`valid_to` means it's valid forever. Set a bound only when a fact is genuinely time-limited, and never write a far-future date to mean forever, because absence already means it. Done that way, the archive answers a dated question from the windows rather than from a hunch, which is the closest thing to a save state a team ever gets.

Recall is more than search. Hand the agent an address (`crystalline://engineering/retry-queue-gotcha`) and ask it to walk out from there, and it follows the relations to what the gotcha depends on and what depends on it, across domains if the links cross, ranked by how strongly each neighbour connects. That's how an agent assembles a neighbourhood before a task rather than a flat list of hits, and it's the part a folder of files can never do on its own.

3.4 Reflect

Everything so far happens inside the agent's own knowledge. The agent writes its engrams for itself, for the next session's search, and nothing in them is phrased for you. People still need what's in there, and Reflection runs a second loop for them beside learn, evolve and remember. It works like the colleague who presents a topic to the team in Section 1.2. The agent presents what it knows in documents for people, called reflections, and everyone puts them to the test (read one, change a line, comment on a passage), while the agent asks back where a change doesn't fit what it knows. The feedback improves the presentation for everyone, and where it corrects the agent or tells it something new, it goes into the agent's knowledge too.

The two loops share the agent's knowledge and cross there in both directions (Figure 12.2). A fix from the sweep, a retired engram or a corrected date, leaves every section presented from the old fact out of date, so the agent presents those sections again. And when a discussion on a reflection settles something and the agent takes it in, it goes into the engrams, where the sweep tends it from then on like everything else. The loop in Figure 3.1 stays as it is, with Reflection running beside it, and Chapter 12 tells the whole story.

3.5 Around again

The first loop closes where it started. What was recalled shapes the work; the work produces something worth capturing; the sweep keeps what was captured honest; and the next session recalls a slightly better body of knowledge than the last. The small box in the middle of the diagram, *session by session*, is the whole story. None of the three beats is impressive alone. A briefing is just a briefing,

a linter is just a linter, search is just search. Together, running for months, they are what turns the stranger of chapter 1 into somebody who was here yesterday and will be here tomorrow, and for whom the work in between isn't lost.

In brief

A colleague grows through a loop (learn, get corrected, remember) and Crystalline runs the same loop for an agent. Learn: onboarded at session start with a routing prompt, the agent captures observations, relations and the occasional salient engram while pairing with you, nudged once per session (where the harness supports it) so the capture beat lands. Evolve: a read-only sweep returns a ranked queue of what the knowledge needs, with mechanical findings the agent completes alone and judgment findings it proposes; a fact is corrected in place or superseded with a status and a closed window, never silently overwritten. Remember: the engrams that may fit a question arrive with it (where the harness supports it), hybrid search ranks current knowledge first, temporal fields answer what applied last June and retired knowledge fades in ranking without disappearing. Reflection runs a second loop beside this one: the agent presents what it knows in documents for people, everyone tests them and what holds up improves its knowledge.

CHAPTER 4

Terminology

Every word in this book has a file behind it. That's the pleasant thing about Crystalline's vocabulary: a domain is a folder you can list, an engram is a file you can print, an address is a string you can paste and a status is a line in that file you can change with whatever editor you already own. So this chapter doesn't open with a table. It opens one engram, the retry-queue gotcha the screenshots in this book keep returning to, and reads it from the top, stopping at each term as it comes up. By the end you'll have met every word the rest of the book leans on, and seen each one on disk.

4.1 Domains

Start with where the file lives: `~/knowledge/engineering/retry-queue-gotcha.md`. The folder is a domain. A domain is a registered folder of knowledge with a `MANIFEST.md` at its root, and the manifest is what makes it more than a folder. It says what the domain covers and when an agent should look there, and those bullets are what the routing prompt is built from at the start of every session. Registering is one command, `crystalline domain add engineering ~/knowledge/engineering`. The name you give it there goes into the manifest as the domain's declared name, and that's the name the agent and every link use from then on, on every machine that registers the domain (Chapter 18). Say your team has two domains: `engineering` for the payments service and everything around it, `clients` for the people it sells to. The line between them is who could vouch for a change (the engineers for one, the people who sell for the other), and chapter 17 makes that the rule for a team domain. A domain can also be virtual, living in the database with no folder at all; it carries the same manifest and answers to the same tools.

Every domain also has a source, the place it comes from: your machine itself or one server your Crystalline is connected to. A write always goes to that source, and a server's domains appear beside your own under one name each (Section 16.10).

Inside a domain, subfolders are topic prefixes. A singleton engram stays at the root; once a cluster forms, it gets a folder, and the folder becomes part of the engram's address. The manifest's `## Notes for Agents` section is where the agent finds what was decided about this domain's layout.

4.2 Engrams

Below is the file itself, as it sits on disk, and everything in the rest of the chapter is a line of it:

```
---
type: engram
title: Retry queue gotcha
permalink: retry-queue-gotcha
tags:
- gotcha
- payments
status: stable
recorded_at: 2026-03-09
generated: { by: claude-code/2.1.233, at: 2026-03-09T16:42:10+00:00 }
---

The retry queue drops any job older than twenty-four hours instead of
retrying it. The team calls the queue the sin bin. The cutoff is not a
tuning choice: it mirrors the chargeback window at Encom, the bank
behind the card payments.

- [gotcha] The retry queue drops jobs older than 24h #payments
- [fact] The 24h cutoff mirrors Encom's chargeback window #payments
- depends_on [[Encom chargeback window]]
```

An engram is one markdown file holding one unit of the agent's own knowledge, the trace a session leaves behind in a form the next session can pick up. A YAML frontmatter block sits on top. The agent writes it, organizes it and reads it back, so its shape serves the agent. If the word sounds like something from a starship's sickbay, that's because Star Trek borrowed it from the same neuroscience textbook Crystalline did.

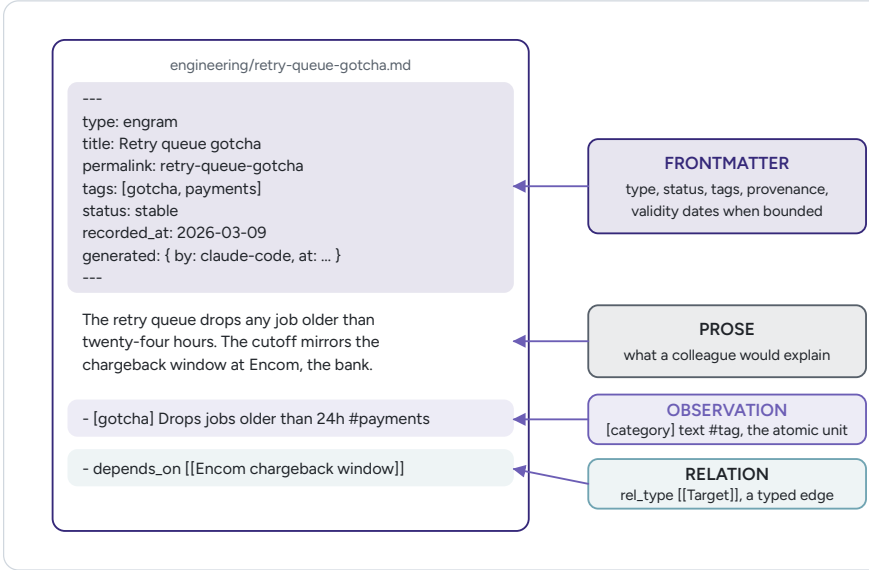


Figure 4.1: The anatomy of an engram: frontmatter, prose, an observation and a relation.

The frontmatter is identity and state. `type` says what kind of thing this is: `engram` by default, with `guide`, `decision`, `architecture`, `runbook` and `reference` as the recommended alternatives. `title` is the name the engram goes by and the thing other engrams link to. `permalink` is derived from the title and is the half of the address that links point at; when the engram moves to another folder, the permalink moves with it unless somebody chose it by hand. `tags` are what a search or a filter narrows on. `status` says whether the knowledge holds, and it gets its own section below. `recorded_at` is the day it was captured; `generated` records who wrote it and when, and for an agent also which model. When the knowledge is time-bounded, `valid_from` and `valid_to` sit here too; this engram has neither, which means it has always held and holds until further notice. Both `type` and `status` are required and free form; the recommended values are guidance, never an enumeration a write is rejected for.

Then comes the prose, the explanation a colleague would give across a desk: what the thing is, what it's called and why it is the way it is. The prose is what search matches on and what the agent reads first.

Then come the observations. A top-level bullet of the form `- [category] text #tag` is an observation, and observations are the atomic unit of an engram's body: the fact, the decision, the gotcha, each indexed on its own line so a search hit can point at it. The category in brackets is free text, and the precise ones

earn their keep: `decision`, `fact`, `pattern`, `gotcha`, `convention`, `lesson`, `risk`, `idea`. A trailing `#tag` attaches the tag to that observation.

Then comes the relation. A top-level bullet of the form `- rel_type [[Target]]` is an edge in the knowledge graph, typed by whatever word comes first: `depends_on` here, `supersedes` and `superseded_by` in chapter 3, `derived_from` and `split_into` between the two halves of a split, `relates_to` when nothing sharper fits or a quoted phrase for a multi-word type. The target is a title (resolved within the domain) or `[[domain:Title]]` across domains. The target's page shows the edge back as a backlink, and a target that doesn't exist yet is an unresolved link the sweep will mention.

One topic per engram is the rule that keeps all of this easy for the agent to find and act on. When the topic grows, the engram is edited, not forked; when it splits, it splits by granularity or by validity. A large capture splits by granularity, a distilled summary beside a source engram holding the full text, linked both ways. An engram that bundles facts with different lifetimes splits by validity. Validity is set per engram, so when one fact expires while the rest holds, the still-valid observations move into an engram of their own first. The new engram carries the source's tags and type, a `stable` status and no validity window, and points back with `derived_from`; the old one keeps what expired, points forward with `split_into` and retires.

4.3 The open format underneath

None of that is a private format. The engram extends Google's Open Knowledge Format, OKF, version 0.2: plain markdown with YAML frontmatter, readable by any OKF tooling, with Crystalline's routing, temporal and graph conventions layered on top. Unknown frontmatter keys survive every write and edit, so an OKF document from somewhere else drops into a domain intact and comes back out intact. The `generated` block is the provenance record, and the `index.md` Crystalline writes at a domain's root declares the format version for anything that browses the folder without Crystalline running. Your knowledge stays files in a folder that any editor opens and any diff tool diffs, whatever tools come next.

4.4 Statuses

`status` is the field a reader that acts on knowledge needs most, because it answers the one question a page can't answer about itself: does this still hold? `stable` is the default and the word for knowledge that holds now; the older spelling `current` means the same state, and a filter on either finds both. Before stability there are `draft`, `proposed`, `idea` and `poc`, for knowledge that's on its way. `implemented` is on the recommended list too, and like any status outside the retirement set it ranks at full strength.

After stability there is the retirement set, four words that each mean one thing. `deprecated` says don't do this again. `superseded` says a newer engram replaced this one, and the two are linked. `archived` says retired but kept for the record. `legacy` says still true of the old installation that's somehow still running, like the terminals aboard the Nostromo, but not to be built on. A status in the retirement set fades the engram in search ranking, gently and never to zero, so current knowledge surfaces first and the past stays one scroll away, dated. Retiring is how a fact leaves the present without leaving the archive; deleting is for mistakes.

4.5 Salience

Most engrams are ordinary, and that's fine. A few are the ones everything else hangs from: the rule that explains a dozen other rules, the insight that ended a week-long bug. Those can carry a `salience`, a number from zero to ten in the frontmatter, and hybrid search adds a small, bounded lift for it, so a salient engram ranks above equally relevant neighbours while relevance keeps the upper hand and nothing is ever filtered out by it. It's set sparingly at capture time, and raised later when an engram turns out to have been what a task hinged on. The retry-queue engram carries none, and most never will.

4.6 Addresses

Every engram has an address, `crystalline://engineering/retry-queue-gotcha`: the scheme, the domain, the permalink. It's the one absolute form. An agent hands it to another agent, `build_context` anchors on it, Fluid copies it to the clipboard in one click. It survives a commit message, a ticket or a chat. A folder prefix appears in the permalink (`crystalline://engineering/queues/retry-queue-gotcha` if the engram had been filed under `queues/`), and a trailing `/*` anchors the whole folder at once.

The rule that trips people up is the scheme-less one. An identifier without `crystalline://` is domain-relative: `retry-queue-gotcha`, together with the domain named beside it, never `engineering/retry-queue-gotcha`, which would be read as a folder called `engineering` inside whatever domain you meant. The same holds for links: `[[Retry queue gotcha]]` resolves in the domain the linking engram lives in, and `[[engineering:Retry queue gotcha]]` reaches across. Either the address is absolute and carries its domain, or it's relative and the domain comes from context. There's no third form, and that strictness is what lets a permalink stay short.

4.7 The MANIFEST, up close

The next term is the first file in every domain. `MANIFEST.md` is an engram of `type: manifest` with the permalink `manifest`, which means it's read, indexed and edited with the same tools as everything else. Its job is routing, and two sections do it. `## Scope` says what the domain covers. `## When to Use` says when an agent should route a task here, and those bullets, one line per domain, are what become the session-start briefing. `## Notes for Agents`, the third section the scaffold creates, holds the conventions the agent should follow inside this domain, the folder layout above all. Two optional sections come later. `## Tag Aliases` folds an old tag name into its canonical one so old searches keep working. `## Provisioning` names the folders of skills, slash commands, subagent definitions and MCP server configs a domain ships beside its knowledge. The tools that knowledge depends on travel with it, and install into a harness once a person says yes. That section gets a chapter of its own (Chapter 21). You don't have to keep all of this in your head: the domain's page in Fluid lays the manifest out section by section, with an example for every section still missing, and an edit that breaks routing says so as soon as it lands. An engineering manifest, a few months in:

```
---
type: manifest
title: engineering
domain_name: engineering
permalink: manifest
tags:
  - manifest
status: stable
recorded_at: 2026-03-02
---
```

```
# engineering

## Scope

- The payments service, its retry queue and the services around it
- Deploy runbooks, team conventions and the decisions behind them

## When to Use

- Route questions about the payments service, the retry queue or a
deploy here
- Check here before touching anything the team has already been burned
by

## Notes for Agents

- Runbooks live under runbooks/, gotchas at the root
- One engram per gotcha; edit the existing one before writing a second
```

Write the `When to Use` bullets the way you'd brief a new hire on which drawer holds what. A machine reads them, but it reads them the way a person reads a label.

Under the hood

The manifest being an engram is why routing needs no separate configuration: `crystalline prompt system` parses each registered domain's `MANIFEST.md`, takes the `## When to Use` bullets, or `## Scope` when there are none, and emits one routing line per domain plus the behaviour rules. Two filenames in a domain are reserved and never engrams. `index.md` is the table of contents Crystalline regenerates after every write so the folder navigates itself without Crystalline running. Each machine builds its own, and a team share leaves it out unless the MANIFEST asks for it; `log.md` is reserved but never generated. A title that would slug to either is refused.

In practice

Say your `books` domain is a year old and you start a second one, `shops`, because where to buy is a different kind of knowledge from what to read: which shop orders from small presses, which one delivers by Friday if you ask on Monday, which one never again. Your first engram there is about Capsule Corp Books, and it needs to say that the paperback rule in `books` (wait for the Tessier-Ashpool

> paperback, the hardbacks fall apart) is why you order there. You type `- relates_to [[Wait for the paperback]]`, and the sweep tells you the link doesn't resolve, because `shops` has no engram by that title. You change it to `[[books:Wait for the paperback]]` and the edge appears, with a backlink on the books side that nobody had to write.
>
> A week later a friend asks what to read and where to get it, and instead of typing the answer out you hand your agent the address, `crystalline://shops/capsule-corp-books`, and ask what you'd tell her. It picks up the address, walks one hop into `books` and comes back with the paperback rule, the small press and the two authors you'd tell anyone about. That's two domains and one edge, and the string is still readable in a year.

4.8 Reflections and the pact

Nothing in an engram is phrased for you, and that's on purpose: the engrams are the agent's own knowledge. When people need what a domain holds, the agent presents it to them, and that has a name. A domain's Reflection, with a capital R, is the whole of the documents the agent presents for people, and a domain switches it on in its MANIFEST. One document in it is a reflection: a markdown file under the domain's `reflections/` folder, made of sections. A section is a `##` heading and its text, and it records which engrams it was built from, so when one of them changes, the agent knows which sections to present again. A section may draw on another domain only when everyone who reads the page may read that domain's engrams too, unless a person shares it on purpose (Chapter 20).

Who may do what with the reflections is a level of its own, the Reflection level, set beside the knowledge level that says what a person may do with the engrams. The knowledge level is the viewer or editor of Chapter 16 (plus manager on a private domain and admin on the instance), and it has one more value, none, for somebody who sees no engrams at all. There are two Reflection levels. A reader reads the reflections and comments on them. A contributor also edits their text and creates, moves and deletes reflections. Above both sits curating: writing sections from the engrams, settling what people changed and editing the pact. Nobody is given that by hand, because everyone from engram editor up curates, and their agents curate with their rights; Fluid shows them as curators. A viewer of the engrams is a reader without anyone setting it, and a Reflection level set on top only ever raises that, so a person with knowledge none and Reflection contributor works on every reflection and never sees an

engram. A Reflection collaborator holds such a level for one domain only, which is how somebody works on that domain's Reflection and nothing else.

Reflection is also the name of a mode in Fluid: in a domain with its Reflection switched on, a switch in the top bar flips between Reflection and Engrams. The Engrams half needs knowledge viewer or above, so somebody who sees no engrams gets the Reflection alone.

Two loops keep a domain in step when nobody opens a session. The reflect loop works what the Reflection needs, the evolve loop the sweep's mechanical findings, and a runner starts each one on a schedule or when the instance calls it (Chapter 14). A run is an unattended run, a session nobody is behind: it reads only the domains everyone who reads its domain may read, writes only into that domain and never asks a question. On a team domain the scheduled evolve runs on one instance only, the domain's evolve home.

The last term is a file again, the pact. Every domain with a Reflection has one, `PACT.md` at the root of `reflections/`, and it has five parts: who the reflections are for, how much detail they carry, how they're presented (which diagrams, callouts and tables help those readers), how the tree is structured and which pages people asked for. It's the MANIFEST's counterpart: the MANIFEST tells the agent what the domain holds, the pact tells everyone how to talk about it. Switching the Reflection on writes a first pact with those five headings to fill in. Curators edit the pact; everybody else reads it and proposes a change in a thread. A pact for a book club's `club` domain, for members and spoiler-free:

```
---
title: How we write about club
---

# How we write about club

## Audience

- Club members, including the ones who haven't started the
  book yet

## Level of detail

- No spoilers past the first chapter
- What the club read, what's next and what members thought

## Presentation
```

```
- A table of the books per series; no diagrams

## Structure

One reflection per series, named after the series, in `series/`:

- series/*.md

## Pages

- next-meeting.md: "what we read next and where we meet",
  asked by Lambert in October
```

Only the bullets under `## Structure` count, one glob each; the line above them is there for people. Under `## Pages` the agent notes each page somebody asked for, in their words, so it knows where new knowledge belongs and can rebuild the page from the request.

You now speak the language. The rest of the book is conversation.

In brief

A domain is a registered folder with a `MANIFEST.md` whose `When to Use` bullets route an agent to it; an engram is one markdown file with YAML frontmatter, prose, observations of the form `- [category] text #tag` and relations of the form `- rel_type [[Target]]`, all on Google's Open Knowledge Format so nothing is locked in. `status` says whether knowledge holds: `stable` is the default, and `deprecated`, `superseded`, `archived` and `legacy` are the retirement set that fades in ranking without disappearing; `salience` is a rare, bounded lift. The absolute address is `crystalline://domain/permalink`, and anything without the scheme is domain-relative. The manifest is itself an engram, which is why routing needs no configuration of its own. A domain's Reflection is the documents the agent presents for people, reflections made of sections, shaped by the pact in `PACT.md` (audience, level of detail, presentation and structure) and read and commented on by readers, edited by contributors and curated by everyone who may edit the engrams.

READY.

PART II

Getting Started

READY.

CHAPTER 5

Installing locally

> *Don't Panic.*
>
> - *Douglas Adams,* The Hitchhiker's Guide to the Galaxy

Everything in this chapter fits in one sitting, and the sitting ends with a test: you'll ask your agent about something it learned an hour earlier, in a session that has never heard of it, and it will answer. Until that happens nothing is installed, whatever the package manager says. So the chapter is built backwards from that moment. Install the binary, wire the harness, give the agent a domain: three steps, the same three on every platform, and then the test.

If you skipped chapter 4, two words carry the chapter, and everything below makes one of each: a domain is a registered folder with a `MANIFEST.md` that says when an agent should look there (Section 4.1), and an engram is one markdown file of knowledge inside it (Section 4.2).

5.1 Install the binary

Crystalline is one binary, and it gets onto a machine the way that machine usually gets things.

Debian, Ubuntu and their relatives get a `.deb` for amd64 and arm64 from the releases page. It ships a systemd unit, installed disabled (chapter 16 turns it on). Windows gets an MSI that adds Crystalline to the PATH, upgrades in place and registers a task that starts the daemon when you sign in. If that task ever goes missing, `crystalline doctor --fix` puts it back. The Windows builds aren't code signed yet, so expect a SmartScreen prompt and check the download against the `SHA256SUMS` file beside it (every release file also carries a build attestation that `gh attestation verify` checks, and SBOMs ship alongside). The bare binary ships for every platform too, and the macOS builds are signed and notarized. The permanent address for all of it is on the colophon page: https://github.com/jordiboehme/crystalline/releases/latest.

The standard Claude Desktop extension needs Crystalline installed first, Homebrew on a Mac and the MSI on Windows. Then download `crystalline-claude-desktop-v<version>.mcpb` from the same page, one extension for macOS and Windows on every chip, and in Claude Desktop open Settings > Extensions > Advanced settings > Install Extension… and pick the file. The extension carries no Crystalline of its own: it starts the one you installed, so that one is the only Crystalline you keep up to date. If none is there, Claude tells you in the chat what to install, and should the extension itself ever be too old for the Crystalline you installed, Claude says so too. On Windows the MSI and the extension need no terminal at all.

If you'd rather not work in a terminal and Claude Desktop on your own Mac is all you use, the Desktop-only bundle is the most elegant way in. Take `crystalline-claude-desktop-only-v<version>-macos.mcpb` instead: it carries Crystalline inside, and you update it by installing the newer bundle over the old one (Desktop has no in-app update for an extension). Desktop users can skip the terminal parts of what follows: the agent creates its first domain itself.

> **Under the hood**
> On a Mac with Homebrew the whole install is one line:
>
> ```
> brew install jordiboehme/tap/crystalline
> ```

5.2 First run

Nothing needs starting: the first agent that connects starts a background daemon (the one process that holds the index open), and every later agent, terminal and browser tab attaches to it. The one exception is the daemon of the Desktop-only bundle on a Mac: it stays tied to Desktop and exits five seconds after its last client disconnects, so with that bundle Fluid is there while Desktop is open. `crystalline serve` runs the same daemon in the foreground if you want to watch it, and `crystalline serve --daemon` in the background, but on a laptop most people never type either.

What you'll type into is a browser: the daemon serves Fluid at `http://localhost:7411` by default, however it was started. The first visit to an instance with no accounts yet is a short form, not a login page: pick a name and a password and you're that instance's admin, already signed in (Figure 5.1). You're trusted to fill that form in because you're on the machine: a connection

from the machine itself needs no token, and a bind anyone else could reach gets a different treatment in chapter 16. Once any account exists the form is gone for good.

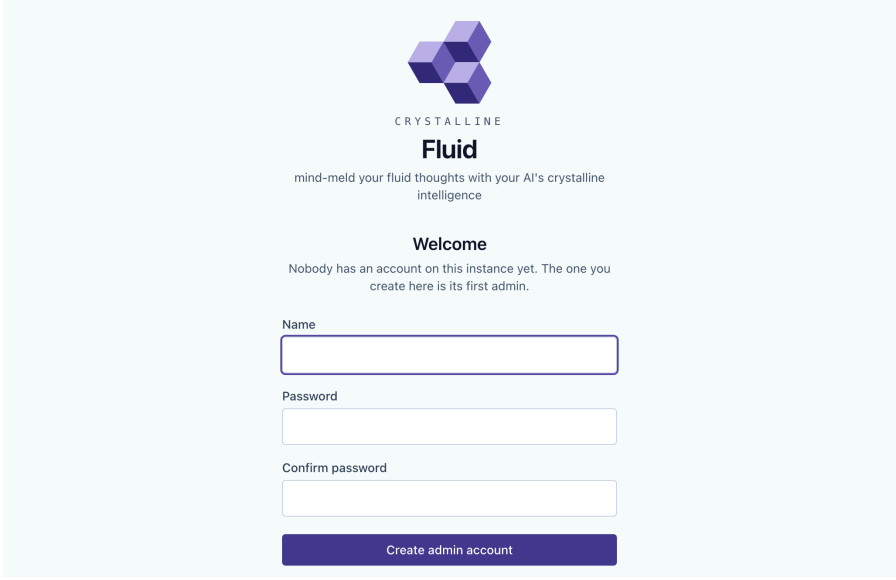


Figure 5.1: First visit: creating the admin account in the browser.

Semantic search wants one more download, the local embedding model (about 220 MB): the daemon fetches it in the background on its first start, and `crystalline model download` fetches it ahead of time. Until then search is plain text, which works, and finds only the words you searched for.

5.3 Onboarding a harness

This is the step that turns a binary into a colleague, and it's one command per harness. Every part of what that command does is something a new hire gets on day one. It registers the MCP server with the harness, so the agent has the tools. In Claude Code, Codex and Copilot it writes three hooks. A session-start hook delivers the briefing on a fresh start, after a `/clear` and after a compaction. A stop hook is the once-per-session nudge from chapter 3 that asks the agent to capture what it learned before the session ends. A prompt hook hands the agent up to three engrams that may fit each prompt, named by address and each once per session (Claude Code and Codex use it; Copilot gets the entry and stays quiet for now). It also copies the six skills into the harness's skills folder, the playbook that says when to use which tool. Cursor, Kiro, Gemini CLI

and Qwen Code get the same skills and the session-start hook alone (Gemini CLI and Qwen Code run it on a fresh start only). Run it twice and the second run touches nothing.

Look at the briefing once with your own eyes: it's the whole of what your agent knows about you at the start of a session. `crystalline prompt system` prints it (Figure 5.2): an intro, the behaviour rules and one line per registered domain, drawn from the `## When to Use` bullets of that domain's manifest. The rules open with when to search (before answering a question about the project's past and before starting a task) and then say how: a narrow question goes to the domain that owns it, a broad one sweeps all of them and a write always names its domain. On a fresh install the domain lines are empty, and that's correct: the desk is set up and nobody has shown the new hire the filing cabinets yet.

Claude Desktop runs no hooks and needs none: its server delivers the same routing block as part of the MCP server's instructions on every connection, and the optional companion skill from the releases page, uploaded under Settings > Capabilities > Skills, adds the capture and collaboration playbook on top. A chat client with no hooks that never shows the model a server's instructions gets a standing instruction instead: `crystalline prompt connector` prints the paragraph to paste into its custom instructions, and the agent fetches its own onboarding with one `list_domains` call at the start of every session.

Install a second harness and you've added a door, not a second colleague. Claude Code in a terminal, Codex, the Copilot CLI, Claude Desktop and every other harness you install each spawn the same `crystalline mcp` over stdio. The first of them to connect starts the one daemon the rest attach to, so every one of them sees the same domains, the same index and the same briefing. Tell Desktop something over lunch and the terminal session knows it in the afternoon, with nothing copied and nothing configured twice. Desktop's door can still stay shut: when the extension finds no Crystalline it can start (none is installed, the one installed is too old, or on Windows the daemon's task is missing), it comes up with a single `status` tool whose instructions say what to install or run. Do that and restart Desktop, and the door is open again.

ripley@tyrell-laptop: ~ (crystalline prompt system)

```
$ crystalline prompt system
CRYSTALLINE KNOWLEDGE ROUTING

You are being onboarded to the knowledge accumulated in this workspace. It is served by the
crystalline MCP server; these instructions govern its tools (your harness may prefix tool
names, for example mcp__crystalline__search_engrams). The domains below are registered and
ready to use; route searches and reads through them instead of starting from zero.

- engineering: Route questions about the payments service, the retry queue or a deploy here;
Check here before touching anything the team has already been burned by
- clients: Route anything about a customer, a contract or a renewal here; Check here before
preparing for a customer meeting
- board: Route anything about the funding round or a board meeting here

Behavior:
- Search before you answer: a question about this project's past decisions, its people, its
systems or a name you do not recognise is answered from search_engrams first, never from memory
or a guess.
- Search before you start: a task begins with search_engrams on its subject, so what an earlier
session learned shapes the work instead of being rediscovered after the fact.
- Narrow question: search_engrams with domains=[that domain]; broad or unclear: no domains
sweeps them all.
- write_engram, edit_engram, move_engram and delete_engram always need an explicit domain.
- Pass write_engram a folder from the domain's layout: a topic prefix build_context can glob, a
subfolder once a topic clusters, the root for singletons.
- Check vocabulary for tags and categories in use before inventing one.
- Salience (0–10) lifts an engram in search: set it at write time on exceptionally valuable
knowledge, raise it when one proves key.
- build_context on a crystalline:// anchor gathers knowledge around a task.
- Read a MANIFEST with read_engram only when the routing line is not enough; list_domains with
include_routing=true re-fetches the index mid-session.
```

Figure 5.2: The routing prompt an agent receives at session start.

Under the hood

Claude Code takes one command:

```
crystalline install claude-code
```

Codex takes `crystalline install codex` and the Copilot CLI `crystalline install copilot`; Cursor, Kiro, Gemini CLI and Qwen Code take `cursor`, `kiro`, `gemini` and `qwen`. The MCP server it registers is `crystalline mcp --harness claude-code`. `--skip-mcp`, `--skip-hooks` and `--skip-skills` leave a part out, `--project` writes into the current repository's config instead of your global one (Claude Code, Codex and Copilot only), and `crystalline uninstall claude-code` reverses the lot while leaving anything that isn't Crystalline's own alone. The install never edits an instruction file such as `CLAUDE.md`, `AGENTS.md`, `GEMINI.md` or `QWEN.md`: it touches only the harness's settings and the files it creates itself. In the settings files of Cursor, Kiro, Gemini CLI and Qwen Code it changes only its own entry, and your other entries, comments and layout stay as they were. Before it

first changes such a file, it keeps a copy under `backups/` in the state directory, readable only by you. Cursor and the Kiro IDE may start with a short PATH, so their registration carries the full path of the binary instead of the bare name: run the install again after you move or upgrade the binary, and restart Cursor after the install so it loads the hook.

The install is recorded in a local receipt, and the receipt is what keeps a harness current: the first time a newer Crystalline runs, it refreshes the installed hooks and skills at session start (that's how an older install picks up the prompt hook), updating changed skills (an edited copy is kept beside the new one as `SKILL.md.bak`) and removing any the new version no longer ships. The `--harness` flag in the MCP registration is the other half of the same idea. A `crystalline mcp` process started with it asks the receipt whether that harness has its hooks wired, and if so serves that session neither the skills over MCP nor a second copy of the routing block, since the harness already has both as files. That holds for Claude Code, Codex and Copilot, where a live check confirmed the harness loads both. Cursor, Kiro, Gemini CLI and Qwen Code aren't confirmed yet, so their sessions keep the skills over MCP and get a short pointer that says to fetch the routing block if it didn't arrive. Nothing in that decision comes from the connecting client: it's settled before the session opens. A registration written by hand without the flag, or a remote client, is served the full surface, and `skills.serve` set explicitly to `true` or `false` makes every client identical.

5.4 The first domain and the first engram

Now the filing cabinet. Before an agent can keep anything it needs a domain, and this section runs the whole loop in miniature: the domain is scaffolded and registered, one fact is written into it as an engram, and a question phrased nothing like the fact gets that fact back. The last one is the semantic ranking at work, on a model the daemon fetched in the background when it started.

Where that folder lives is up to you, and it doesn't have to be where any other domain is: a domain is registered by its own path, and nothing ties one registration to the next. `engineering` can sit in a git checkout under `~/knowledge`, `reading` under Documents where the agent put it, a third on a NAS mount that is already backed up. All three are served by the one daemon and named in the same briefing. `Documents/Crystalline` is only the default for a domain an agent creates when nobody names a path, and the `domains_root` setting moves that default without moving anything already registered. A network share changes two things. First, the index stays in the local state directory, and should: it's a database, and a database on a share is how people learn about file locking

the hard way. The share holds the markdown and nothing else. Second, the daemon's file watcher hears the machine it runs on, not the network, so an engram written into that folder from another computer shows up at the next `crystalline sync` or the next daemon start, not the moment it lands. Written through Crystalline from any harness on this machine, it's indexed as it lands, share or no share.

Open the manifest before you go any further and make sure the `## Scope` and `## When to Use` sections say what you would say, because those bullets are the one line your agent reads about this domain at the start of every session. Write them the way you'd label a drawer for a new hire: "route questions about the payments service, the retry queue or a deploy here". The agent drafts them from what you say, and you check them once.

It takes three prompts and no terminal:

```
You:    Set up a Crystalline domain called engineering in
        ~/knowledge/engineering. Its scope is the payments service:
        route questions about the payments service, the retry queue
        or a deploy there.
Agent:  Created and registered engineering. The manifest's Scope and
        When to Use sections carry what you just said, so your next
        session will see the domain in its briefing.

You:    Remember in engineering: the retry queue drops jobs older
        than 24 hours. It is a gotcha, tagged payments.
Agent:  Captured "Retry queue gotcha" in engineering, at
        ~/knowledge/engineering/retry-queue-gotcha.md.

You:    Why does the payments queue lose jobs?
Agent:  The retry queue drops jobs older than 24 hours, from
        "Retry queue gotcha" in engineering.
```

A Desktop user's path is exactly this, and it works even without the first prompt: an agent that finds nowhere to write creates a domain itself under `Documents/Crystalline`, and the first engram lands there. Open the manifest afterwards and make sure the two sections say what you'd say.

> **Under the hood**
> The first prompt was three commands, the same three the manual's terminal setup page starts with. `domain init` scaffolds the folder with a starter manifest and `domain add` registers it under the name the agent will use and indexes what

is already there (there's no separate sync step). The agent did the same through its own tools: `add_domain` to scaffold and register, then `edit_engram` on the manifest.

```
mkdir -p ~/knowledge/engineering
crystalline domain init ~/knowledge/engineering --name engineering
crystalline domain add engineering ~/knowledge/engineering
```

The second prompt was one command (the agent's was one `write_engram` call), and running it by hand shows the file appear in the folder, `retry-queue-gotcha.md`, the file chapter 4 read line by line:

```
crystalline write engineering "Retry queue gotcha" \
  --content "- [gotcha] The retry queue drops jobs older than 24h
#payments" \
  --tags gotcha,payments
```

The third prompt was a search, and `crystalline search "retry queue"` finds it as a plain text hit. On a machine where no daemon has fetched the model yet, fetch it, re-sync with embeddings and ask something the engram never says:

```
crystalline model download
crystalline sync --embed
crystalline search "why does the payments queue lose jobs"
```

`crystalline sync` is for files that arrive some other way (an editor, a `git pull`) when no daemon was watching, and `crystalline status` shows what got indexed.

5.5 Verifying the loop

Now the test. Start a session in your harness and ask about the retry queue without explaining anything. Watch what the agent does before it answers, because that's the recall beat and it proves the wiring. In Claude Code or Codex the question arrives with the gotcha already attached by its address, and the agent reads it. Elsewhere it searches `engineering`, because the briefing told it that's where the payments knowledge lives. Either way it answers from the engram, with the address in hand if you ask for it.

What that doesn't prove yet is the capture beat, the one that matters over months. Tell the agent something it couldn't have known, the way you did on chapter 1's Monday:

```
You:    The cutoff is 24 hours because Encom's chargeback window is
        three days - the queue needs two days to notice a job. Remember
        that, and why.
Agent:  Captured: added the chargeback window as the reason to
        "Retry queue gotcha" in engineering, with a depends_on relation
        to a new "Encom chargeback window" engram. Want me to adjust
        the wording?
```

Close the session, open a new one (it has never heard of Encom) and ask why the cutoff is a day. If the answer names the chargeback window, you're done: the agent looked something up that it learned in a conversation that no longer exists. It's a Voight-Kampff test run in reverse: you're not checking whether the thing across the desk is a replicant, you're checking that it remembers.

Once the test passes, close the terminal. Nothing in daily use needs it: the agent recalls and captures through its tools, the daemon starts itself and the browser tab is there for the odd look at what the agent learned. The commands in this chapter were the installation, and from here on what you type is addressed to a colleague.

Under the hood

When it fails the fix is usually quick. If the agent answered from general knowledge, with no engram attached and no search, the session didn't get the briefing: run `crystalline prompt system` yourself and check that `engineering` has a line (if it has none, the manifest's `## When to Use` section is empty). If the agent recalled the gotcha but not the reason, the capture never landed: look in `~/knowledge/engineering` for the file and check that you said yes when it proposed writing. If nothing arrived with the question in Claude Code or Codex, check two things: the prompt hook stays quiet until the embedding model has loaded, and `crystalline doctor` says whether the hook is installed.

In practice

Your first afternoon is the five lines above. You run the brew line and the install line, `domain init` a `books` folder, and your agent's first session opens with a briefing that names it and nothing else, because nothing has been taught yet. You tell it the series rule, the two shops and the press whose paperbacks you wait for, to

see what happens, and close the session. In the next one you ask what to read, and it starts from your rules. In chapter 2 the bookseller took a year to learn you; the agent took a session. By Wednesday you have three doors open: Claude Code in the terminal, the Copilot CLI in a second terminal and Claude Desktop for the thinking you do away from the keyboard. Two `crystalline install` lines and one Desktop extension were the setup, and the daemon behind all three is the one that started that first afternoon. What you tell Desktop on Tuesday evening about the manga translations is what Copilot recalls on Wednesday morning, from the same file in the same folder, and nothing was copied between them.

Your sister, who has never opened a terminal on purpose, installs the Desktop-only bundle the same week. She has no domains, and her agent makes one the first time it needs somewhere to write, a `reading` folder under Documents/Crystalline. By Friday it holds four engrams about the Dungeon Crawler Carl books she's lining up for the holiday, and she opens `localhost:7411` to look at what her agent kept while Desktop is running. Two installs, one with a keyboard and one without, and the same loop runs on both.

The desk is set up, the drawers are labelled and the new hire has remembered one thing overnight. Everything after this is a matter of what you choose to teach.

In brief

Installing is three steps on every platform: install the binary (Homebrew, a `.deb` or an MSI, plus an extension for Claude Desktop, or the Desktop-only bundle on a Mac that runs only Desktop), wire the harness with `crystalline install <harness>` and give the agent a domain with `domain init` and `domain add`. The install writes the MCP registration, the session-start hook that delivers the briefing and the six skills, plus in Claude Code, Codex and Copilot the stop hook that nudges capture and the prompt hook that brings matching engrams with each question (in Claude Code and Codex; Copilot's stays quiet), and it's safe to run again. Nothing needs starting: the first agent to connect starts the daemon, which serves Fluid at `localhost:7411`, where the first visit creates the admin account. Every harness on the machine shares that daemon, and a domain is registered by its own path. The install is finished when a fresh session recalls something a previous one learned.

READY.

PART III

Working with Crystalline

CHAPTER 6

Onboarding an agent

On a new colleague's first morning somebody hands them the floor plan. Payments is on the second floor, the people who know about the bank sit by the window, the runbooks are in the cupboard that sticks. Which door leads where is the whole of day one, and it's enough, because a person who knows where things are can find out everything else themselves.

That floor plan is what an agent receives at the start of every session with Crystalline. This chapter is about the day it arrives: what the briefing says, how the agent finds its way with it, what it means to hand a colleague a task instead of a question, and the reminder at the end of the day that turns a session into experience. Chapter 5 showed the wiring. This is what the wiring is for.

6.1 The briefing at session start

A session opens, and before the agent has read a word of your request, the session-start hook runs `crystalline prompt system` and puts its output in front of the model. Here is one a few months in, abridged:

```
CRYSTALLINE KNOWLEDGE ROUTING

You are being onboarded to the knowledge accumulated in this workspace.
It is served by the crystalline MCP server; these instructions govern
its tools. [...]

- engineering: Route questions about the payments service, the retry
  queue or a deploy here; Check here before touching anything the team
  has already been burned by
- clients: Route anything about a customer, a contract or a renewal
  here; Check here before preparing for a customer meeting
[...]

Behavior:
```

```
- Search before you answer: a question about this project's past
  decisions, its people, its systems or a name you do not recognise is
  answered from search_engrams first, never from memory or a guess.
- Search before you start: a task begins with search_engrams on its
  subject, so what an earlier session learned shapes the work instead
  of being rediscovered after the fact.
- Narrow question: search_engrams with domains=[that domain]; broad or
  unclear: no domains sweeps them all.
[...]
- Read a MANIFEST with read_engram only when the routing line is not
  enough; list_domains with include_routing=true re-fetches the index
  mid-session.
```

Read it as the agent reads it. The opening says what this is and whose tools it governs, so an agent with four MCP servers connected knows which `search_engrams` is meant. Then there's one line per domain, and every word of those lines was agreed with a person: they're the `## When to Use` bullets of each manifest, joined with semicolons, exactly as the team approved them (Section 4.1). On your own machine that's every domain registered, since the CLI and a stdio session are the machine owner. On a shared instance the agent connects as an account, and the block lists every domain that account may see: a private domain it's no member of is absent from the block, from `list_domains` and from every search, answered as if nobody had registered it (Chapter 19). Then come the behaviour rules, the same on every instance: the first two say when to search (before answering a question about the project's past, its people or its systems, and before starting a task) and the rest name the exact tool each one reaches for. That's the floor plan (Figure 6.1). It says where things are, nothing about what's in them, and the agent goes and looks.

ripley@tyrell-laptop: ~ (crystalline prompt system)

```
$ crystalline prompt system
CRYSTALLINE KNOWLEDGE ROUTING

You are being onboarded to the knowledge accumulated in this workspace. It is served by the
crystalline MCP server; these instructions govern its tools (your harness may prefix tool
names, for example mcp__crystalline__search_engrams). The domains below are registered and
ready to use; route searches and reads through them instead of starting from zero.

- engineering: Route questions about the payments service, the retry queue or a deploy here;
Check here before touching anything the team has already been burned by
- clients: Route anything about a customer, a contract or a renewal here; Check here before
preparing for a customer meeting
- board: Route anything about the funding round or a board meeting here

Behavior:
- Search before you answer: a question about this project's past decisions, its people, its
systems or a name you do not recognise is answered from search_engrams first, never from memory
or a guess.
- Search before you start: a task begins with search_engrams on its subject, so what an earlier
session learned shapes the work instead of being rediscovered after the fact.
- Narrow question: search_engrams with domains=[that domain]; broad or unclear: no domains
sweeps them all.
- write_engram, edit_engram, move_engram and delete_engram always need an explicit domain.
- Pass write_engram a folder from the domain's layout: a topic prefix build_context can glob, a
subfolder once a topic clusters, the root for singletons.
- Check vocabulary for tags and categories in use before inventing one.
- Salience (0–10) lifts an engram in search: set it at write time on exceptionally valuable
knowledge, raise it when one proves key.
- build_context on a crystalline:// anchor gathers knowledge around a task.
- Read a MANIFEST with read_engram only when the routing line is not enough; list_domains with
include_routing=true re-fetches the index mid-session.
```

Figure 6.1: The same block unabridged, as `crystalline prompt system` prints it.

Under the hood

How it reaches the desk depends on the harness, as chapter 5 said. Claude Code and Codex run the hook on a fresh start, after a `/clear` and after a compaction, and skip a resumed session; the Copilot CLI's hook does the same on a fresh start and stays quiet on a resume. Cursor runs its session-start hook, Gemini CLI and Qwen Code run theirs on a fresh start only, and Kiro runs its hook too, with an always-on steering file that tells the agent to fetch the block if it's missing. Claude Desktop gets the block inside the MCP server's own instructions; a remote client fetches it with one `list_domains` call. Either way the agent starts out knowing which drawer holds what and nothing more, the state a good new hire is in at nine on Monday.

The block is built to survive being cut off. A harness in tool-search mode truncates a server's instructions at roughly two kilobytes, so the MCP rendering keeps under a 1900-byte budget and degrades in fixed steps: every routing line drops to one bullet, then the list collapses to a count line pointing at the re-fetch call.

> A remote client gets that count line on arrival whatever the block's size. The MCP `initialize` handshake is answered before any request is in hand, so over HTTP there's nobody to scope the block to and no way to leave a private domain's name out of it. The handshake carries the behaviour rules, how many domains that aren't private are registered and the pointer at `list_domains`, which does know who's asking and filters.

6.2 How the agent finds its way

A routing line is a targeting aid. The routing skill says so in as many words, and adds what every colleague knows about floor plans: the label on the door isn't the inventory of the room.

The first decision is whether to look, and the rules settle it: before the agent answers a question about the project's past or starts a task, it searches. In Claude Code and Codex the prompt hook has usually brought the first few engrams along with your message already. The second decision is scope. A narrow question, one that clearly belongs to one service, team or customer, goes to the domain whose line describes it, in a single `search_engrams` call with `domains` set. A broad or unclear question omits `domains` and sweeps everything, each hit labelled with its domain. The agent doesn't stop to ask you which it is: a search is cheap and usually settles the ambiguity, and a clarifying question is for after it came back empty.

The third decision is when to widen. A scoped search can return plausible near-misses when the subject belongs to one domain and the question to another: who gets paged about refunds is a people question, not a payments one. When the obvious domain keeps missing after a rephrasing or two, the agent runs one unscoped sweep. Then it reads. A snippet is a short window around a match, and a colleague who answers from a snippet is guessing, so anything that summarizes or quotes an engram goes through `read_engram` first. A strong hit becomes an anchor for `build_context`, which pulls in what the engram depends on before the work starts.

The agent is given the doors and, with each prompt, at most three engrams that may fit it; what's behind every other door is fetched when a task calls for it. A colleague with a floor plan beats one carrying the building around in a wheelbarrow.

6.3 Assign, do not copy

You wouldn't message a colleague a question, copy their answer into your editor and do the work yourself. You'd assign them the task, because doing it is how they get better at it. An agent with Crystalline's tools is in the same position, and treating it as a lookup service wastes most of what you set up.

Give it the task. "Add a metric to the retry queue" lands better than "what do you know about the retry queue", because the first sentence sends the agent through the whole loop on its own: it routes to `engineering`, recalls the gotcha and the bank rule, reads the engram in full, walks one hop to the deploy runbook, writes the change in Python and tells you what it found on the way. The floor plan only pays off when somebody walks the corridors, and the somebody should be the agent.

It follows the processes a person would, since it can be wrong in the same ways: review for its changes, a yes for its captures. A colleague you never delegate to never learns your work, and neither does an agent.

In practice

Here's your first week with the agent, session by session. Monday is the afternoon from chapter 5: the briefing names `books`, and the agent knows your rules before you've typed one. On Tuesday you give it a real task (something for Saturday's flight) and explain nothing. The agent routes to `books`, recalls the series rule and the translations, finds the note about the Tessier-Ashpool paperbacks you captured on Monday and comes back with three (a Doctorow you missed, the next Dungeon Crawler Carl and the Deep Space Nine novel that follows the one you finished last month), each with the shop that has it. On Wednesday it's wrong for the first time: it recommends the new Vickers in hardback, and you tell it you don't buy hardbacks, because you read outdoors and a hardback doesn't survive the backpack. It asks whether to capture that in `books` as a `[convention]`. You say yes, and add the exception: a hardback is for collecting, bought after the ebook, when a book turns out to deserve a place on the shelf.

Thursday is a long session sorting the pile by the bed, and when it ends the reminder fires once: the agent proposes two captures (that you abandon anything with a map on page one, and a shelf engram per Star Trek series, classic, Next Generation, Deep Space Nine and the Destiny trilogy, saying which volumes you own and which is next) and raises the salience of the translations engram. On Friday you ask it to order the Vickers, and it answers, before you've said which edition, that the hardback is out by your own rule until you've read it, and the ebook is out today. On the first session it knew where the drawers were. On the

> fifth it told you something about yourself, unasked, that you'd taught it two days before. That's the moment a stranger stops being one.

6.4 The reminder at the end of the day

A first week only counts if what was learned survives the weekend, and that's the beat agents skip. Left alone, a model finishes the task and stops, and the gotcha it met at minute forty goes with it. So `crystalline install` wires a reminder, the closest thing the loop has to a habit.

It's a `Stop` hook. On the first stop after a session has gained real substance it asks the agent to review the conversation for durable learnings: facts, decisions, patterns, gotchas, corrections from you. It also names what doesn't count: what the session did, temporary paths and outputs, a one-off bug's error text, steps that won't recur and anything any model already knows. For each one not yet captured the agent proposes an engram in the fitting domain and waits for a yes; a correction that makes an existing engram wrong becomes a proposed edit or supersession rather than a second engram. If a recalled engram proved to be the key to the task, the agent raises its salience. If nothing qualifies or everything is captured, it finishes normally. The reminder fires at most once per session, and costs about a hundred and twenty tokens. When maintenance is due, a paragraph rides along asking for a sweep; when a team domain holds unshared work, another asks the agent to propose sharing, still a proposal, because sharing puts somebody's work in front of reviewers.

An agent that reaches a shared instance only over MCP has no hooks. Crystalline's one reliable channel to a model mid-session is a tool result, so the reminder rides there: a write, edit, move, split or delete comes back with a single clearly prefixed line appended when something is due, never a search or a read, so recalled knowledge is never mixed with guidance. Sharing and maintenance travel this way. Capture doesn't, because it's a reflection at the end of a conversation and the server can't tell when a conversation ends; the routing block keeps teaching capture as standing practice instead. Each person is nudged on their own cadence, a few hours apart:

```
---
[crystalline] 3 changes in the team domain engineering are not yet
shared. If that work is done, propose sharing it with share_changes so
```

```
the team has it and the team's archive stays current - and wait for a
yes.
```

The floor plan gets a new hire through the door. What they carry back at the end of the day is what makes them worth having on Tuesday.

In brief

Every session opens with a routing block: one line per domain the session may see (the manifests' `When to Use` bullets) and the behaviour rules naming the exact tools. Each harness delivers it the way chapter 5 wired it. The agent searches before it answers or starts a task, scopes a narrow question to one domain, sweeps for a broad one, widens when scoped hits keep missing and reads before it answers. Hand it tasks rather than questions, and let the once-per-session reminder close the day: propose captures, wait for a yes.

CHAPTER 7

Teaching

The best onboarding you ever had was a walk. Somebody took you across the floor, stopped at a machine and said "that one lies", and you never trusted it again. Nobody wrote that down. It was not on the slide. It was the kind of knowledge that only moves from one head to another when a person decides it matters enough to point at, and the pointing is what this chapter is about.

Day one, last chapter, was the floor plan. Now comes the proper onboarding: what you tell a colleague in the first month, how you tell an agent the same things, what a well-taught fact looks like on disk, and the two rooms where the teaching happens, a session and a browser tab. It ends with two people editing the same page at once, which is where teaching stops being a lecture.

7.1 What to teach

Ask any good manager what a new hire needs and the list comes out in the same shape every time: what we value, and what we won't do even when it would be faster. How things are done here (the review, the deploy, the meeting that actually decides) and what's expected of the role. Which tools we use, and which we abandoned and why. And where information lives: which repository, which dashboard, which person. An agent needs the same list, in the same order of importance, and the last item is the one that makes the other four cheap.

A person's knowledge doesn't look like a source tree. No engineer on your team holds the payments service in their head by the letter. They know it's in the `payments` repository, that the retry logic is under `queue/`, that the config nobody documented is in the deploy chart, who owns it and that Encom's rule explains its strangest number. They know how to get access and which door to knock on. What they carry is a map of the territory plus the concepts to navigate it fast, and that's exactly what to teach an agent. The source tree it can read with the tools its harness gives it whenever a task calls for it. What it needs from you is which repository holds the code for which purpose, how access is granted,

the architecture and the vocabulary. Pointer knowledge, access knowledge and the mental model. Teach the map and the territory stays where it is, reachable.

So the engrams that earn their place in the first month are the ones a person would say across a desk: the chargeback rule behind the cutoff, the name the team uses for the queue, the fact that the wiki says gateway where the code says edge, and the rule about Fridays. Not one of them is on any page, and every one of them is what an experienced colleague knows.

7.2 Teaching through the agent

In a session, the simplest way to teach is to say something worth keeping and watch what happens.

Tell the agent "the settlement job runs at six because Encom's file lands at five, and it fails silently if the file is late", and it doesn't write. It searches first, in `engineering`, because the capture skill makes search-before-write a hard gate, and it checks whether an engram already owns the topic, in which case the new fact becomes an edit of that engram rather than a second one beside it. Finding nothing, it proposes: "Should I capture this in `engineering` as a `[gotcha]`?" A terse "store this" is already a complete instruction and skips the question; an aside you obviously didn't mean to keep gets no question at all. It names the domain every time, since a write has no default, and when the owner isn't obvious it asks rather than guesses. A rotation detail learned during a payments incident is people knowledge, and the agent sweeps every domain before filing it where the incident happened to be.

Then it writes a well-formed file: a title, tags drawn from the vocabulary already in use rather than a near-synonym, a folder matching the domain's layout and a body of at least three lines, because verification rejects a lone bullet as too thin. Hand it one sentence and it pads honestly, one bullet for the fact in your words, one for how and when it was learned, one for the scope and never an invented specific. You glance at the result the way you glance at a new hire's first commit, and say yes. The write answers with its nearest neighbours too: the receipt names the existing engrams closest in meaning to what just landed. The instruction beside them is to read the one that fits, then merge into it, supersede it or link it, never to pass over the list in silence. So the duplicate check runs twice, before the write on your sentence and after it on the engram as written. A write that starts a new folder beside one with a close name, `planview` next to `planview-work`, says so too, and the agent can move the engram where its topic already lives instead of splitting it over two folders.

Correction is the same conversation. “The file lands at half past five, not five” finds the engram and changes the value where it stands. “We moved settlement to the new broker in August” is a different thing, the world changing, and the agent supersedes rather than edits, as chapter 3 walked through. Teaching and correcting are one motion, which is why the framing in chapter 2 said teach, rather than file.

7.3 Teaching through Fluid

The other room is the browser. Sometimes the fastest way to teach is to write it yourself, without a model in the loop and without spending a token: open the engram in Fluid, switch to edit and type (Figure 7.1). The editor previews live, the frontmatter is a form, and typing `[[` completes titles across every domain, so a relation is two keystrokes and a pick. Callouts, status labels, diagrams and the rest of what a page can hold sit in the Insert menu, tables and task lists have buttons of their own and a `/` typed at the start of an empty line offers all of them; Chapter 13 shows each one. The file on disk is what changes; the page you were reading (Figure 15.4) is that file. Save a page you’re editing alone and you get the second look a session’s write gets, as a panel above the text with the same instruction.

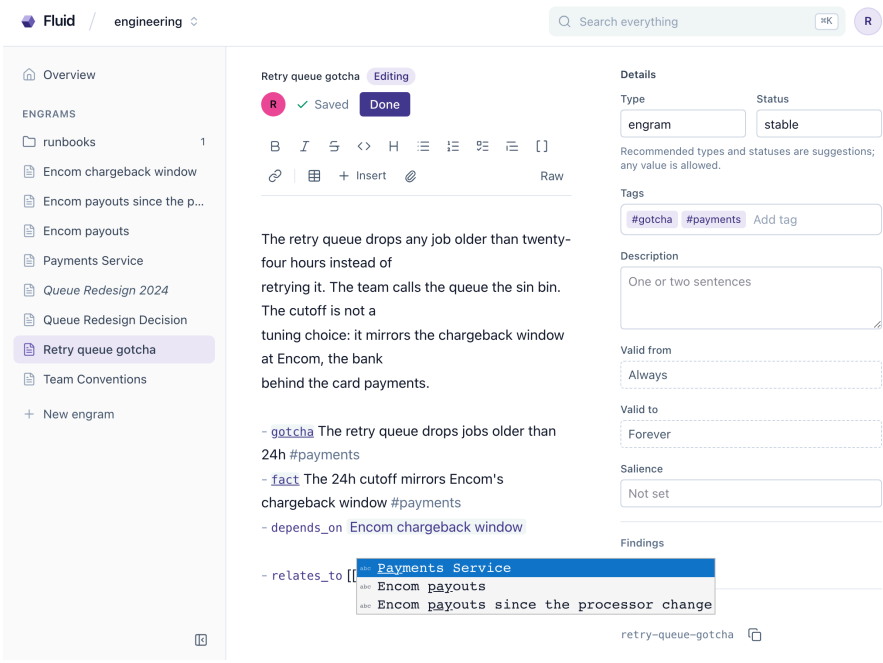


Figure 7.1: Editing an engram in Fluid: live preview, the frontmatter form and wikilink completion.

What you write there doesn't sit unreviewed. The engram records that a person wrote it, `human:` and your account name in its `generated.by`, and the next maintenance sweep raises it as a finding of its own: captured by a person and never reviewed. The finding is judgment class: what you capture yourself is teaching, so a change to what it claims goes back to you. The agent reads the engram, verifies the claim, aligns the tags with the vocabulary, wires relations both ways into the neighbourhood the engram belongs to and records a `verified` entry; any change to the wording it proposes and waits. In chapter 2 you typed a note into Fluid yourself, and that week's sweep is where the agent tidied it into the graph. That's the division of labour: you teach in whichever room is nearest, and the agent keeps the archive coherent.

7.4 The anatomy of a good engram

Chapter 4 read an engram line by line. What makes one good deserves a page of its own, because the difference between a good engram and a page is the difference between a colleague who explains and one who forwards you a link.

An engram is one unit of knowledge, about one thing, and when the thing grows the engram is edited, not forked. One thing is one lifetime too: validity is set per engram rather than per bullet, so a fact that expires on its own schedule (a price, a deadline, an owner) gets an engram of its own. Where a bundle mixed them, `split_engram` moves the facts that still hold into a new engram before the rest retires. Prose comes first: the explanation you'd give across a desk, what the thing is and why it's the way it is, because prose is what search matches and what the agent reads first. Then the observations carry the facts, one bullet each with the most precise category you have, `[fact]` for verified state, `[decision]` for a choice, `[gotcha]` for the trap, `[lesson]` for what cost an hour, `[risk]` for the concern, never `[fact]` for a guess. Relations come next, for structure: `depends_on`, `supersedes` or a plain `relates_to`, so the graph can carry the agent to what the fact leans on. Inside its own domain the agent writes a bare `[[Title]]`, and a link into another domain carries that domain's name first, `[[shops:Capsule Corp Books]]`, because a bare link never leaves the domain it's written in. Tags stay within the vocabulary in use. A salience goes on only when the engram is genuinely exceptional (the rule a dozen other rules hang from), since most never need one. And an engram is what's true now: a corrected value is replaced where it stands, never stapled on as a dated update, and the validity dates stay absent unless a fact is genuinely bounded. This one your agent wrote into `books`, after the exchange below:

```
---
type: engram
title: "What I finish: short, funny, one big idea"
permalink: what-i-finish-short-funny-one-big-idea
tags:
- taste
- reading-log
status: stable
recorded_at: 2026-06-11
generated: { by: claude-code/2.1.233, at: 2026-06-11T17:05:41+00:00 }
---

Three years of the reading log say the same thing three ways. The
```

```
books finished inside a week are all under three hundred and fifty
pages; the ones abandoned are all epics in volumes; the five-star
reads open with a joke or a big idea, usually both. Dune is the one
exception to the page count, and the log calls it heavy but good.

- [pattern] Under 350 pages, a joke or a big idea on page one #taste
- [gotcha] An epic in volumes, or a map on page one, is where the log
says "abandoned" #taste
- [fact] Derived from the reading log, 2023 to 2026, 412 entries
#reading-log
- relates_to [[Wait for the paperback]]
- summarizes [[Reading log, 2023 to 2026]]
```

Notice what's in it and what isn't. The pattern, the trap and the provenance are there, and the log is kept beside it, attached in the agent's word and on disk as a `type: source` engram under `sources/` linked both ways, so the agent can check the derivation whenever it needs to. The log itself isn't in the engram; a summary is what most recalls read, and the full text is pulled in only when the summary falls short.

In practice

You've had three duds in a row, and on Sunday you sit down with your agent:

```
After a bad run
  You:    Three duds in a row. Here's my reading log from the last
three
          years - work out what I actually like.
  Agent:  Read it. The pattern: under 350 pages, a joke or a big
idea on
          page one, and a ship's computer somewhere. Dune is the
exception,
          and you called it heavy but good. An epic in volumes is
where you
          stop. Captured "What I finish: short, funny, one big
idea" into
          books (#taste #reading-log), log attached.

Six months later, a fresh session
  You:    Something for the long weekend.
  Agent:  Three candidates, all under 350 pages, one of them a
Doctorow
          you haven't logged - recalled that a map on page one is
```

```
where
          you give up. The Gibson is the heavy-but-good kind.
```

Look at what you didn't do. You didn't tell the agent a fact. You handed it the source material and a question, the way you'd hand a friend the log and say "work out what I like", and the agent derived the pattern, wrote it down with the log kept beside it as provenance, and six months later recalled it in a session that had never seen the log. That's teaching at its most economical: you supply the material and the judgment about what matters, and the colleague does the learning.

7.5 Attachments

Some teaching material is a file: the chargeback agreement, the deck a vendor sent, the diagram you drew at the whiteboard and photographed. In Fluid you paste, drag or upload it onto the engram it belongs to, and it lands under the domain's `assets/` folder with a reference in the engram's body, an image `![...](assets/diagram.png)` or a link `[...](assets/deck.pdf)`. Images, PDFs, slide decks, office documents and data files are all accepted, up to ten megabytes each. Files enter through Fluid or a domain archive, never through an agent write, and an attachment always belongs to some engram's teaching: a file nothing references comes up in the sweep as something to delete or to turn into knowledge.

They aren't opaque to the agent. `read_engram` returns each attachment as a resource link, and the agent fetches the file itself over MCP before summarizing what it shows; a picture it needs to see comes back as an image. The sweep closes the loop from the other side: an attachment nobody has analyzed is a finding, and working it means reading the file, capturing what it teaches and marking on the engram which file was analyzed at which version, so a changed deck comes back around. One small courtesy for images: a formatting fragment on the target, `![Chart](assets/chart.png#right,w=50%)`, floats and sizes the picture in Fluid, and every other markdown renderer ignores it.

7.6 Editing together

Chapter 3 said the way a colleague really learns is by pairing, and Fluid makes pairing literal. Open an engram that somebody else has open and you see their cursor; type, and both sets of edits merge conflict-free and land as one save: two people, one page, one whiteboard. Review mode changes the room: everyone writes in a draft of their own, and a colleague joins yours through a draft share-link (chapter 17).

The third name in the strip can be the agent's. When your agent reads or edits an engram somebody has open in the editor, it joins that page as a named peer: your account with an agent designation, a bot glyph beside the name and a colour from the same palette as everyone else's. It has no cursor, so its sentences aren't painted in that colour; the glyph in the strip is how you know an agent is writing. An edit it makes is computed against the page as it stands at that moment, so it composes with what you typed seconds ago instead of stomping it. Anyone with the page open just to read sees the change appear in place as it lands, with a line saying who made it, no reload needed. When it reads, it sees your unsaved typing rather than a stale copy. Its write receipt says the change landed in a live page and names who is present, so it can tell you "I updated the section you're looking at" and mean it. Only a wholesale overwrite of a page somebody has open is refused: the agent edits the parts instead or replaces the page once you close it. That's the fluid and crystalline halves working on one page at once, human with human and human with agent.

Now and then a save can't land: two edits conflict, the server refuses the save or somebody switches a domain's Reflection off under the page you're writing. Your text isn't lost. When the last person leaves that page, the server keeps the text, and the editor offers it back the next time you open the page, to put back or to drop. The domain's settings list every kept text you may see.

> **Under the hood**
> A kept text is a file in the server's state folder. On the server, `crystalline kept list` shows every one with a short id, `crystalline kept show <id>` prints one so you can copy it back and `crystalline kept drop <id>` deletes it after you confirm. Nothing deletes a kept text on its own, because it may be the last copy of somebody's work, and `crystalline doctor` warns about one older than thirty days. Agents have no tool for kept texts: it's a person's unsaved typing, and the person decides what happens to it.

What that is like is a napkin on the table between you. You write the line you're sure of, the retention rule you want, and ask the agent in your session to fill

in what you're not. While you keep typing it goes and reads: the engram that holds Encom's chargeback window, the vendor's documentation for the retry limit. The window lands under your line, address and all, and the retry limit beside it with a link to the vendor's page. The month you misremembered is corrected in the sentence you just typed. You ask it to sketch the retry flow so you can check what it understood, and a diagram appears under the text, drawn from what the two of you wrote. None of it waits for a save, and none of it happens unless you asked. It's JARVIS in Tony Stark's workshop: he builds, and the voice in the room has the schematic in the air before he asks for it and tells him, politely, when he's wrong. On a shared instance anyone can pull up a chair. Chapter 17 puts a whole meeting in that room.

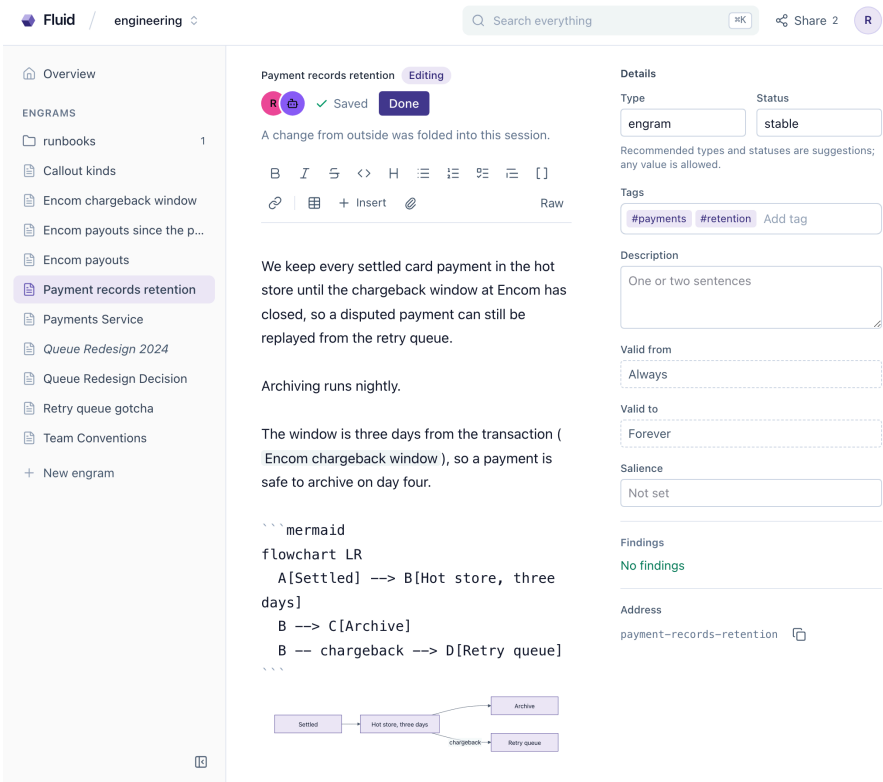


Figure 7.2: Your agent in the page with you: its glyph in the strip beside yours, writing while you write.

Teaching a colleague is pointing at the machine that lies. Teaching an agent is the same gesture, and this one writes it down.

In brief

Teach an agent what you'd teach a new hire: values, processes, expectations, tools and above all where information lives, taught as a map (which repository holds what, how access works, the architecture). In a session, say something worth keeping and the agent searches first, proposes a capture in the fitting domain, writes a well-formed engram once you say yes and reads the neighbours its receipt names, with corrections edited in place and changed facts superseded. In Fluid, write it yourself, and the next sweep verifies, tags and wires what a person captured, with any change to what it claims going back to that person. A good engram is one unit of knowledge and one lifetime, with prose, categorized observations, typed relations and, rarely, a salience; attachments hand over files that agents read back. Co-editing merges live, your agent can sit in that session as a named peer and a text that can't be saved is kept for you.

READY.

CHAPTER 8

Ingesting knowledge at scale

Nobody has ever onboarded a colleague by pointing at four hundred wiki pages and walking away. Plenty of people have, and the colleague spent a month believing the architecture diagram from the year before the rewrite and calling the product by the name it had until the acquisition. The pages weren't wrong when they were written. They were just handed over without the ten minutes at the whiteboard that tell a newcomer which of them to believe.

Those ten minutes are the subject of this chapter. Sooner or later the teaching of chapter 7, one fact at a time, meets the existing pile: the wiki, the shared drive, the decks. The agent can learn from it in an afternoon, and it wants to, like Johnny 5 flipping through pages of an encyclopedia: input, more input, need input. What it can't do is bring the judgment a person brings to a wiki, the sense of what's current, what was aspirational and which of two contradictory pages to trust. Chapter 1 said that judgment has to be handed over. This is how, and how you check what came back.

8.1 What a briefing contains

A briefing is the paragraph you'd say to a person before they opened the wiki, and the useful ones are concrete about the past. Names first: the product now called Settlement was called Ledger until the reorganization two years ago, so normalize to Settlement; the wiki says gateway where the code says edge, and edge is the word to keep. Then a rule for contradiction, since a wiki of any age contains them: trust the newer page, unless one of them is the runbook. Then what to skip: the archived spaces, the personal pages, anything under Drafts. Then the one everybody forgets because it's obvious: the decks were written to persuade and the wiki is what we actually do, so where they disagree the page wins and the deck's version goes in as an `[idea]` at most.

Add where things go: which domain, which folder layout, the tags in use. And the shape of the review you want: propose before writing, keep source links,

ask about anything resembling a decision nobody wrote down. It fits in one message:

```
You:  Read the engineering space of the wiki through the connector and
      propose engrams for the engineering domain. Ledger is now called
      Settlement - normalize. Where the wiki says gateway the code says
      edge; use edge. Skip the Archive space and anything under Drafts.
      On a contradiction trust the newer page unless one is a runbook.
      The Q1 platform deck is aspirational; the wiki is what we do.
      Keep a link back to every source page and its last-edited date.
      Propose the list first and I will trim it.
```

That message is the difference between an ingest that lands and one that produces a confident, well-formatted archive of things that stopped being true.

8.2 The sources, and who reads them

Crystalline ships no scraper, by design. The agent is the ingester, and it reads sources with whatever its harness gives it: the wiki over the MCP connector the harness provides, a folder of exported documents through its file tools, a repository through the code tools it already has. A repository, though, is a source the agent consults rather than one it ingests: the repository holds the code, and the engrams hold which repository holds the code for which purpose, how to get access and the architecture and concepts that make the right file findable in a minute. The division is precise: the harness supplies the reading, the agent supplies the distillation and Crystalline is the destination, where what was derived lands as engrams with provenance, and the review loop that checks it afterwards.

A wiki is the common source, read page by page through the connector. A folder of documents, PDFs and exports and the odd spreadsheet, is read the same way. Slide decks are the one source that usually comes in through Fluid instead: a person drops the deck onto the engram it belongs to, and the agent reads it back over MCP as an attachment, so the deck becomes teaching material rather than a file nobody opened (chapter 7). And a tree of markdown that already carries frontmatter, an old knowledge base that was nearly an archive already, doesn't go through the agent at all: `crystalline import <src> --domain <name>` folds it into a domain as a pure file transformation, dry run first, normalizing legacy type values, backfilling status and dropping sentinel dates on the way. The

pictures and documents its notes link to come along into the domain's `assets/` folder with the links rewritten, and the report names any it couldn't take.

8.3 Distill, then propose

The cardinal sin of ingestion is mirroring: copying a source wholesale so that the archive looks complete, then filling the gaps with guesswork. A vendor recall notice is forty screens of marketing header, revision table, legalese and support matrix, with three paragraphs that matter. Distilled, it becomes two observations, a `resource` link back to the page and a `source_date` in the front-matter: forty screens down to two facts and one link. That link points at the exact page or file the engram came from, pinned to the commit it was read at when the source is a repository. When the agent answers from the engram, it hands you the link with the answer, so you can check the source or read on. Every location the engram was distilled from also goes into its `sources` list, so when the wiki changes, the agent finds the engrams of the changed pages by that list. It rewrites the address of a page that moved and drops one that's gone, and a page that's gone is no reason on its own to retire the engram it fed.

Distilling is also where the briefing does its work. The agent reads a page about Ledger and writes Settlement; it finds a deck slide claiming a feature and files it as an `[idea]` because you told it decks persuade; it meets two pages that disagree and applies the rule rather than picking one silently. And one shape earns verbatim retention, only ever beside its distillation: a document whose exact words you may need to quote. The summary stays an ordinary engram where its topic lives; the full text goes in as a `type: source` engram under `sources/`, linked both ways with `summarizes` and `summarized_by`. It's split into parts only when it outgrows its budget, which is four times the size an ordinary engram gets. On recall the summary is what gets read, and the full text is pulled in only when the summary falls short.

Then, before anything lands, comes the list:

```
Proposed for engineering (23 engrams):
  runbooks/  Settlement deploy runbook
  runbooks/  Retry queue drain procedure
             Edge routing rules
             Encom file format, v3
  ...
Write these? (y / n)
```

You trim it. Proposing first is the rule for a whole space, not just a page, because the trimming is where your judgment enters the archive, and striking five lines off a list is cheaper than retiring five engrams next month.

A source too large for one session, a wiki space of a few thousand pages or an archive of documents, isn't read front to back. The agent groups its parts by topic, hands each topic to a subagent of its own on a capable model (where the harness offers subagents) and folds what they bring back into the knowledge.

8.4 The review loop

What the agent writes is a proposal in a second sense: it carries provenance, and a person reviews it. Each engram names its source, a link, a date, an attachment, so the reviewer can open the page it came from and check the derivation. The review happens in Fluid, where an engram is a page and a fix is an edit, and a reviewer who knows the material corrects the one value the agent got wrong without going back through a session. A team domain sends the ingest out as a proposal (chapter 17).

Some duplicates never wait for the sweep. A write can come back naming the engrams it's closest to, so a wiki page that repeats one the agent captured an hour earlier is merged on the receipt rather than an hour later. The agent says what it did with the list instead of moving on. Then run the sweep. The evolve tool is meant to be called after a large ingest, and its queue is built for this moment: links that don't resolve because a page referred to a title nobody captured; near-duplicate clusters where two wiki pages became two engrams about one thing; a tag spelled two ways; an attachment nobody has analyzed; and the human captures that reviewers made in Fluid while correcting, each raised as a finding to verify, tag and wire into the graph. Work the mechanical findings in a pass, propose the judgment ones one at a time and sweep again until the number stops falling. That shrinking number is how you know the ingest settled.

8.5 Temporal hygiene

An ingest is where dates go wrong, because a page written two years ago lands in an archive that treats everything as true today. Three fields keep it honest. `source_date` is when the source was written, and the agent sets it from the page's last-edited date, so an engram derived from an old page says so. `valid_from` is for a fact that started on a known date, a policy that came in with the reorganization, and nothing else; absent means it has always held, and an

absent `valid_to` means it still does, so neither is set to mean "we do not know". `stale_after` is the one that buys you the most: put it on anything with a shelf life, a vendor's terms, a pricing table, a contact name, and the sweep raises the engram when the date passes and nobody has re-checked it. A re-check that finds nothing changed is one `verified` entry, or a later `stale_after`, and the engram goes quiet.

8.6 When not to bulk-ingest

Curation beats completeness, and the archive isn't a mirror of the wiki. If a space is dead, leave it dead; if a page would not survive the "would I tell a new hire this" question from chapter 2, it does not become an engram because it existed. Four hundred pages distilled to nineteen engrams, worth more than the four hundred because every one is something the agent will act on.

The strongest case for not ingesting is the source tree. A person doesn't remember the code by the letter. Well, Commander Data does, and yes, he's a person, the court said so; but you didn't hire Data. A person remembers where and how to find things, the map chapter 7 said to teach, and lets the territory stay where it is. Ingesting the source would give you a stale copy of something the agent can already read, and the map fits in a dozen engrams.

> **Under the hood**
> Nothing about an ingest is special on disk. Each derived engram is one markdown file with frontmatter, the same format as a fact taught across a desk, with `resource`, `sources`, `source_date` and `generated` as plain keys anyone can grep. A domain that lives in a repository shows the ingest as a diff: new files, nothing rewritten elsewhere, so a bad batch reverts with the version control the team already has. Attachments sit under the domain's `assets/` folder, referenced by relative path from the engrams that claim them. The index is derived from the files (`crystalline reindex --full` re-reads every file and keeps the embeddings), and `crystalline verify` in the repository's CI refuses a malformed engram before the team pulls it.

> **In practice**
> Your reading log has four hundred and twelve entries, and a notes app holds a decade of half-sentences about them; you trust neither completely. You give your agent a briefing like the one above and let it read both over an afternoon. It comes back with a list of twenty-three engrams, each naming its source entry. You strike four: two are about a shop that closed, one is a wish list from 2019

dressed as taste, one is a note about a book you were given and never opened. The agent writes nineteen, every one with a link and a date, the shops among them with a `stale_after` for next year because opening hours change.

The sweep the next morning has eleven findings. Six are mechanical, links to titles the log spelled differently and the ingest renamed, and the agent fixes them in a pass. Three are judgment: a near-duplicate pair, two engrams about Dune from different years, which you have the agent fold into one; an orphan nothing links to yet, wired to the paperback rule; and a tag spelled two ways, which comes back as a `crystalline tags merge` command for you to run. The remaining two are your own: corrections you typed in Fluid while reviewing, which the agent verifies and wires into the graph. On Thursday you ask which Star Trek novels you already own, by series, and get the answer, the source entry for each and the date you wrote it, from a log you haven't opened in a month.

Today the wiki is the source and the archive is derived from it. The last chapter turns that around: once the agent's knowledge is what the team teaches and corrects, the wiki pages and the decks become presentations of it, kept in step with it, which is no revolution at all, since every diagram and slide that exists today was already a reflection of somebody's knowledge. The ingest is the last time the flow runs this way round.

In brief

Bulk ingestion needs the briefing a new colleague would get before opening the wiki: what was renamed and what to normalize to, which page wins a contradiction, what to skip and which sources persuade rather than describe. The agent reads the sources through its harness, distils rather than mirrors, keeps a source link and date on every engram, proposes the list before writing and lands the result in Crystalline, where a person reviews it in Fluid and the next sweep surfaces the unresolved links, duplicates and unreviewed captures an ingest leaves behind. Set `source_date`, reserve `valid_from` for genuinely bounded facts and put `stale_after` on anything with a shelf life. And don't ingest what the agent can read for itself: teach the map of the source tree, not the tree.

CHAPTER 9

Finding

Forty-two is a perfectly good answer. The trouble, as the mice discovered, is that nobody knew the question. Recall has the opposite trouble: you know the question, and the answer was written down by somebody who'd never heard it. The engram says the retry queue drops jobs older than twenty-four hours. The person on call at three in the morning types "why do payments go missing". The two don't share a single word, and the engram is the answer.

Closing that gap is what this chapter is about. Chapter 3 called the beat *remember* and chapter 6 watched the agent walk the corridors on its first day. What's left is the craft: how a search ranks, how an agent gathers the neighbourhood around a hit before it starts work, how it asks what was true last June instead of what's true now, and why a colleague who recalls costs less than one who reads everything. The retry-queue engram runs through every section.

9.1 Search as recall

A search in Crystalline blends two signals, and the blend is what makes the three-in-the-morning question land. Full-text matching finds the exact: the engram that says "retry queue" when you searched for "retry queue". Semantic similarity finds the adjacent: the engram about dropped jobs when the question was about missing payments, because the two are about the same thing in different words. Each hit is scored on both and the ranking is the blend. Text alone would miss the on-call question. Similarity alone would rank a chatty engram that mentions payments above the terse one that explains them. Together they behave like a colleague who both remembers the phrase and understands the problem.

Meaning doesn't stop at a language: the model behind the similarity reads more than two hundred of them, so an engram written in German answers a question asked in English. Your shelf note on translations, typed in German one evening (English when a book was written in English, German translations

for everything else), is what comes back months later when you ask in English which edition of a French novel to buy.

Two small priors tilt the order, and both are bounded. Salience lifts the exceptional: an engram given a `salience` at capture, or raised later when it turned out to unlock a task, ranks above equally relevant neighbours, while relevance keeps the upper hand and nothing is ever excluded by it (Section 4.2). Retirement fades the past: an engram whose status is `deprecated`, `superseded`, `archived` or `legacy` is softly pushed down, never filtered out. So the cutoff that applied until September sits one scroll below the behaviour that applies now, still there and still dated (Section 4.4).

Around the ranking sit the filters. A search narrows by domain, by `type`, by tags, by status, by any frontmatter key or by a recorded-after date. A search with no query text at all is a legitimate way to ask for everything of one kind: every `runbook` in `engineering`, every engram tagged `weyland-yutani`. A tag filter also matches through a domain's tag aliases, so an old spelling keeps finding what it always found. A filter can also match one element of a frontmatter list, so one search finds every engram whose `sources` list names a given page, and a key with a single value counts as a list of one. Every hit is labelled with its domain, and a hit inside an observation carries its line, so the agent can point at the fact rather than the file.

What comes back is a snippet, a short window around the match, often cut off, and it's a targeting aid. A colleague who answers from a snippet is guessing, and the routing skill draws the line where it belongs: a single atomic fact completely visible in the snippet may be answered from it. Anything that summarizes, compares or quotes an engram goes through `read_engram` first. When several domains answer a broad question differently, the agent says so and names which domain each answer came from. Scope itself is chapter 6′s decision: narrow to the domain whose routing line fits, sweep for a broad question and widen when the obvious domain keeps returning near-misses.

Fluid's search shows you what the agent finds: the search page is faceted and backed by exactly the ranking an agent sees (Figure 15.2). Cmd+K is the other door: it jumps anywhere by name. The two answer different questions. Search is for when you know what it's about. Cmd+K is for when you know what it's called. When you want to see an engram the agent found, ask it for the link: every engram it reads or writes comes with the address of its page in Fluid, and the link can open that page at one heading.

9.2 Gathering the neighbourhood

A hit is a point, and a task usually needs the area around it: what the fact depends on, what depends on it, the runbook that mentions it. An experienced colleague does this without noticing. Ask them about the retry cutoff and they tell you about the bank unprompted, because in their head the two were never separate.

`build_context` is that reflex as a tool. Hand it an address, `crystalline://engineering/retry-queue-gotcha`, and it follows the relations and the prose links outward, one to three hops, crossing into other domains where an edge crosses, and returns the neighbours ranked by how strongly each connects to the anchor. What comes back is the shape of the neighbourhood (titles and typed relations, with an untyped prose link surfaced as `links_to`) and no content. The agent reads the nodes that matter with `read_engram`, which is the second half of the same motion. A `/*` suffix on the anchor globs a permalink prefix, and a permalink carries its folder, so that's a whole folder: `crystalline://engineering/runbooks/*` (Section 4.6). `max_related` keeps the top-ranked, ten unless you say otherwise, so a densely linked engram doesn't drown the task in its acquaintances.

The loop feeds itself. `read_engram` marks each relation and link as resolved or not, summarizes what links back and, when a neighbourhood exists, names the `build_context` anchor to run next. An edge that doesn't resolve is a signal rather than an error: the target isn't written yet, or lives in a domain this instance hasn't registered. Fluid draws the same neighbourhood as an interactive graph (Figure 15.3), the quickest way to see whether an engram is wired in or standing alone.

9.3 What applied last June

Every fact in the archive is one of two kinds, and a good question knows which it's asking for. "What is the cutoff?" wants the present. The present is a status question: filter on `status: stable` and the search returns what holds now, with the older spelling `current` folded in. "What cutoff applied in June?" wants a moment, and a moment is a window question, answered from `valid_from` and `valid_to`, the plain dates on the engrams that carry them. It's the time circuits on the dashboard: set the destination time to June and the search returns the engram that was true when you land. Chapter 10 shows the window being closed with the real date.

Absent dates are unbounded, and that rule sets a trap in a filter. A strict comparison (`valid_to` greater than the first of June) skips every engram with no `valid_to`, which are exactly the unbounded ones the question meant to keep. The routing skill puts it plainly: reach for a window filter only for an explicit, bounded question about a date, and put it on the first search. For an ordinary present-day question, `status: stable` alone is both correct and sufficient. When a window filter isn't getting traction, the `supersedes` chain and each hit's status tell the same story a different way: search by topic, read which engram covers the period asked about and narrate the status of each rather than presenting a retired one as fact.

Time runs the other way too. A search takes a recorded-after date, and `recent_activity` answers "what changed while I was away" across every domain for the last week, day or fortnight, which is how a person back from leave starts current instead of starting over.

9.4 Recall is cheaper than reading

It's worth stating the case as a mechanism, with no number attached, because the number depends on your archive and the mechanism doesn't.

An agent that keeps its knowledge as a folder of markdown files has one way to use it: read it. The folder can't say which file matters for today's task, so the agent reads the folder, and reads it again in the next session, every file, whether or not the task touches it. That works while the folder is small, and it stops working at the scale where knowledge becomes worth having, the ladder chapter 1 climbed.

An agent working from Crystalline receives only the knowledge that applies. In Claude Code and Codex the prompt hook runs the first search before the agent has read the question and hands over at most three engrams. The routing prompt sends the task to one domain, or to a sweep, before a single engram is read. Inside the domain the indexes narrow again: full-text and semantic search for the phrase and the meaning, the graph of relations and backlinks for what the answer leans on, tags for the facet, the folder prefixes that `build_context` globs for the topic, the temporal fields for what holds now against what held then. What reaches the model is the hits that matter, the engrams it chose to read in full and the neighbourhood it asked for. Nothing else is loaded. Reading a folder costs the whole folder, every session. Recall costs the engrams that matter, because the ten-thousandth engram is found the same way as the tenth.

The person on call knows this instinctively. They don't reread the wiki at three in the morning. They ask the colleague who says "hang on, we had this" and reaches for the right drawer.

In practice

Your sister has picked up the Warhammer novel you abandoned and wants to know what you made of fantasy last autumn, before you went back to science fiction. You take the question to your agent. It searches `books` for fantasy with a window on the first of October and gets the superseded engram, valid until the month you put the book down, with the phase as it stood and what led you there. It builds context from that engram's address and pulls in the two Tolkien engrams and the shop that stocked them one hop away. Then it searches again with `status: stable` and reports what applies today, so you can answer both halves with the dates between them. You never open the old log. You don't remember last autumn's taste by the letter any more, and you don't need to.

The same week you come back from a fortnight away, where the only sessions were evening ones on a balcony, and ask what the fortnight added. `recent_activity` over two weeks lists the two books you finished, a corrected opinion of the last Next Generation one and three new engrams under `to-read/`, the next Destiny volume among them, each with its domain and date. You're current before your coffee is cold, where a folder of notes would have had you rereading, or asking.

Under the hood

A hybrid search runs a lexical query and a semantic query and ranks by the blend. Until embeddings exist it falls back to plain text. The salience prior is `search.salience_weight` (0.15 by default) and the retirement fade is `search.retired_weight` (0.6 by default), both bounded so relevance keeps the upper hand, and 1.0 turns the fade off. The list filter is `$contains` in a search's metadata filters, and `$contains_any` takes up to 100 values. Embeddings are computed in the background as engrams land, so a fresh engram is findable by text at once and by meaning shortly after. On an Apple Silicon Mac the embedding model runs on the GPU, and `crystalline status` shows which device it runs on (sorry if you're on a different operating system ;)). The index behind all of it is derived from the files: `crystalline reindex --full` re-reads every file and keeps the embeddings, and `crystalline reindex --wipe` rebuilds an index that won't open.

The engram was written for a question nobody had asked yet. Recall is what makes that fine.

In brief

Search is recall: full text finds the exact phrase, semantic similarity finds the adjacent one in any language, salience lifts the exceptional and a retired status fades the past without hiding it. Filters narrow by domain, type, tag, status, any frontmatter key or date, and a snippet is a pointer to read, never an answer. `build_context` turns a hit into a neighbourhood, following relations and links from an address or a `/*` folder glob, ranked by connection strength, and Fluid draws the same graph. Ask the present with `status: stable` and the past with validity windows. Absent dates mean unbounded, so a strict date filter is only for an explicitly bounded question. Recall is cheaper than reading because only what matters is loaded, routed by domain and narrowed by the indexes, so the cost stays flat as the archive grows.

CHAPTER 10

Evolving

When a colleague gets something wrong, you don't fire them. You say what was wrong, point at the page that explains it, and the next time is better. Every team keeps its people's knowledge honest that way, and it works because nobody expects what a person learned in March to hold forever. Information lives: the cutoff changes, the vendor rewrites its terms, the customer's CTO who decided in the room is replaced by one who reads. What you taught the agent in chapter 7 and handed it in chapter 8 was never set in stone, only in files, and a file can be corrected.

Evolving is the beat that does the correcting. Chapter 3 walked it around once, and this chapter works it at the pace of a real archive.

10.1 Why knowledge needs tending

Nothing in an archive goes wrong dramatically. A validity window ends on a Tuesday and the engram still says stable on Wednesday. A runbook that names a person has a review date, and it comes and goes. A replacement lands in one session and the session ends before the old engram was flipped. An engram retires because one of its facts expired, and the two beside it that still held retire with it, carried forward nowhere. Somebody types a correction into Fluid on a Friday and nobody looks at it again.

Then the structural kind: a link that points at a title nobody captured, a live engram citing one that has retired, a `supersedes` written on one side only, an engram nothing links to, one that grew past the point where it should have split, one that never grew past its frontmatter, a reference to an attachment whose file is gone, a file in `assets/` that no engram claims. A permalink left behind by a file moved by hand belongs here too; a move made through Crystalline carries the permalink along and rewrites every link, relation and address that pointed at the old one. Then the redundant kind: two engrams saying nearly the same thing, two titles a link can't tell apart, one tag spelled two ways.

Then the meaning kind: two that say one thing in different words (the sweep calls those twins) and, if you switch the check on, two lines that look like they contradict each other, which Chapter 11 takes up on its own. None of it is a mistake anyone made, it's what happens to knowledge when time passes, and it's what a colleague six months in will confidently repeat if nobody tends it.

10.2 The sweep

The agent asks for a sweep. You can ask for one from a terminal too (you'll see what the agent sees). What comes back is a ranked queue and nothing else: the sweep is read-only, and the standing reminder at the bottom of every run says so. The queue is written for the agent to work: each finding names the engram, the evidence it fired on and the one action that clears it, with the tool that performs it. The four families are the four kinds above in that order: temporal, structure, redundancy and meaning. A run narrows to one family, one rule, one domain or a minimum priority, and pins the date rules to a chosen day when you want it to reproduce. Here's an `engineering` domain on a Monday in October:

```
Sweep of engineering as of 2026-10-05
41 engrams scanned, 3 findings (showing 3, page 1)
temporal 1, structure 1, redundancy 1

1. [85] V001 JUDGMENT
   Broker migration freeze  crystalline://engineering/broker-migration-
freeze
   validity window ended 2026-09-30 but the status is still stable
   evidence: valid_to=2026-09-30; today=2026-10-05; status=stable;
inbound refs 1
   fix: set_frontmatter valid_to=<later date> or status=superseded

2. [50] V102 MECHANICAL
   Retry queue drain procedure  crystalline://engineering/runbooks/
retry-queue-drain-procedure
   unresolved reference [[Edge health check]]
   evidence: rel_type=depends_on; nothing titled `Edge health check` in
engineering; nearest is `Edge health checks`
   fix: [[Edge health check]] -> [[Edge health checks]]

3. [30] V203 JUDGMENT
   crystalline://engineering
   2 tag spellings look like one tag (plural variants)
```

```
    evidence: #runbook used 2 time(s); #runbooks used 9 time(s)
    fix: crystalline tags merge runbook runbooks
```

The number in brackets is the priority: how much the archive gains from clearing this one first. A finding on an engram that many others link to, or that carries a salience, gets lifted, because a wrong hub does more damage than a wrong leaf. The word beside the rule id decides who acts.

> **Under the hood**
> The agent's call is `evolve_engrams`, and `crystalline evolve` prints the same queue from a terminal. `families` picks temporal, structure, redundancy or meaning, `rules` a rule id like `V001`, `min_priority` a floor under the bracketed score and `class` keeps only mechanical or only judgment findings, which is what a run without a person asks for (`include_acknowledged` comes up with the acknowledgments below). `--today` evaluates the date rules as of a fixed day so a run reproduces.

10.3 Mechanical and judgment

A *mechanical* finding completes intent the archive already records. The link in item 2 points at a title one letter off a real one, and fixing it invents nothing, so the agent does the mechanical findings in one pass and reports once at the end (you'd fix a colleague's typo without calling a meeting about it).

A *judgment* finding would change what the archive claims, and that wants a yes first. Item 1 asks whether the migration freeze ended or was extended. Only a person who was there can say. The agent brings it to you as a proposal, evidence and its own reading included, and writes only when you say yes:

```
Agent:  The broker migration freeze ran to 30 September and the
        migration landed on the 12th. I read this as ended rather than
        extended: retire it, close the window on the 30th and carry
        the "no payments deploys during a broker cutover" lesson into
        the deploy runbook? (y / n)
```

Some judgment findings carry a rule of their own. An engram a person captured directly in Fluid is judgment class whatever else is true of it: the agent verifies the claim, aligns the tags and wires the relations both ways, but it proposes any change to the wording and waits: a person's capture is teaching, so a change to

what it claims goes back to that person (Chapter 7). Tag drift, item 3, comes back as a command rather than an edit, since folding a tag is a bulk rewrite the agent never does on its own: it hands you `crystalline tags merge runbook runbooks` and leaves it there. The merge rewrites the tag everywhere and records the fold under `## Tag Aliases` in the manifest, so the old spelling keeps finding what it always found and the sweep never raises the pair again.

Then ask again. The second run re-derives the queue from scratch (nothing about the first was stored), so what's left is only what's still true. In the run above the three become one (the tag merge that waits for a terminal, which is waiting on a person rather than on the archive), and that shrinking number is the feedback loop.

In a domain that reviews changes before they land (chapter 17′s review mode) a sweep covers the reviewed folder plus the sweeper's own drafts: the problems in a draft are its author's to see and nobody else's, and a finding on the shared truth shows for everyone.

10.4 The page for people

Fluid's maintenance page shows the same queue to a person: the stale dates, the half-finished retirements, the unreviewed human captures, ranked the same way. Every finding about an engram links to it. It's there for your curiosity and for a sense of how the knowledge is doing, and it's a report rather than a workbench, on purpose. The queue's consumer is the agent, so no button on the page edits an engram. The work a finding names is the agent's, done in the engram itself. What you do on the page is watch the list shrink. Two filters, a domain and a family, scope the sweep rather than the view: pick one and the server sweeps again with what the filter names, the header says so and the rows dim until the new queue arrives. Figure 3.2 shows the queue with the Meaning family picked. Its three buttons are about the queue rather than the knowledge: acknowledge a finding as intentional, take that back and delete an orphaned attachment. What an acknowledgment silences is never hidden: a toggle draws the silenced rows muted.

10.5 Retiring done right

The sweep finds retirements that were never finished, so it's worth being precise about finishing one. Start with the question chapter 3 asked: does the new fact make the old one false going forward? If the old fact was never quite true (a cutoff measured from the wrong event), correct it where it stands: a correction is a sharper measurement. If it was true and stopped being (the rule went away with the broker), supersede it: a supersession is the world changing.

A correction the agent can't make itself still reaches you. That happens when the domain is read-only for it, when the engram holds your own words or decisions or when the right value is your call. Then it tells you: it links the engram, says what's true instead with the source that shows it and proposes the edit. Where it may write, it captures the correct fact as well and links it to the wrong one with `corrects`.

One move comes before either. Validity is set per engram, not per observation, so an engram that bundled three facts retires all three when one of them stops holding. When the rest still holds, split first: those facts move into an engram of their own, linked with `derived_from` and `split_into` so the pair resolves from both ends, and only the remainder is superseded. Re-typing the survivors into the successor loses their history and copies them forward again at the next expiry. A retirement that skipped the split shows up in the temporal family as a carry-forward gap, a retired engram whose observations appear in no live one, matched as text and never as meaning; the fix is the split or, when the facts expired along with the rest, an acknowledgment.

A supersession has four parts, and the sweep notices when any one is missing. The replacement is an engram of its own that points back at what it replaces. The old engram is marked superseded, with the real date the fact stopped holding (never a guess, never a placeholder). The old engram points forward at its successor, so the graph reads the same from both ends. And any lesson that's still true moves into the live engram, because the lesson outlives the fact. A retirement without a successor gets the status that fits instead: deprecated for do-not-do-this-again, archived for kept-for-the-record, legacy for still-true-of-the-old-installation, with the window closed when the date is known and the lesson carried into a live engram either way. The retired engram stays in every search, dated and ranked softly below the current one, so the next time somebody asks why an old dashboard still says twenty-four hours, the answer includes the date the answer changed.

A whole domain can be finished too, and you can freeze it with `read_only` in its MANIFEST, so nothing changes it by accident and the sweep leaves it alone until you switch it back (Section 25.7).

You'll rarely ask for the four parts by name. What arrives is a proposal with the date already in it and a question at the end:

```
You:    The new Vickers is nine hundred pages and has six narrators.
        I gave up on page forty. Here are my notes.
Agent:  That's the third in a row since the trilogy ended. The engram
        "Vickers: buy on sight" was true until the trilogy closed and
        false since, so I propose a supersession rather than an edit:
        a new engram "Vickers: wait for the reviews" that supersedes
        the old one, the old one marked superseded as of the trilogy's
        last volume and one lesson carried forward: a favourite is a
        run, not a rule, so re-check after every book. Yes?
You:    Yes.
```

In practice

That exchange is yours. You hand your agent the notes on the new Vickers the way you'd hand it any book's notes, and the agent does something you didn't ask for. The engram "Vickers: buy on sight" described an author whose every book you'd finished in a weekend. The agent recognizes that the engram was true and has stopped being, and proposes the supersession rather than an edit: a new engram for the author's books from here on with `supersedes` pointing back, the old one flipped to `superseded` with a `valid_to` of the month the trilogy's last volume came out and the lesson carried forward, unbounded. You say yes. Your notes on the trilogy stay attached to the retired engram, still its provenance.

A month later the sweep raises the paperback-rule engram, which carries a `relates_to` of its own back to the buy-on-sight engram: a live reference to retired knowledge. The agent proposes repointing it at the new one. You say no: the rule was written with Vickers as its one exception, the author whose every book had earned the shelf before you'd finished the ebook, and the citation is history rather than guidance. The agent acknowledges the finding on the engram with a note, "lineage citation, keep". The sweep stops raising it while the citation and the retirement both stand, and the old engram stays in the picture, dated, the way Marty's siblings did not.

Under the hood

The split is one call. `split_engram` takes the source, a title and what moves, `observations` (the line numbers `read_engram` reports) or `sections` (heading paths, deeper subsections with them); the new engram keeps the source's tags and type and gets `status: stable` with no window. Both relations come written for you. It refuses a selection that would leave the source under verify's three content lines, the case for retiring the whole engram. The four parts come out as four calls. `write_engram` creates the successor with `- supersedes [[Old title]]` among its relations. Two `edit_engram` calls with `set_frontmatter` set the old engram's `status` to `superseded` and its `valid_to` to the date, one lifecycle field each, and a third appends `- superseded_by [[New title]]` to the old engram. The lesson rides in the successor's body as an observation, with no validity window. The replacement carries `status: stable` like any live engram. A retirement with no successor sets `status` to `deprecated`, `archived` or `legacy` instead. Skip the status and the sweep raises a half-finished retirement, skip the back-link and it raises a one-sided pair, do all four and the sweep has nothing to say.

10.6 Verification and acknowledgments

Some findings are answered with a record rather than an edit, and telling the two records apart keeps the archive honest. A re-check that finds a fact still true is written down as a verification: who checked and when. A review date pushed to a later day says the same thing with a deadline attached, and the sweep prefers one, since an old engram with no review date and no verification is one it starts asking about on age alone.

An acknowledgment says something different: this finding is intentional. It names the rule it answers, carries an optional note (the vignette's "lineage citation, keep") and records that the person ruled on it. The server records what evidence the finding fired on, so the acknowledgment holds exactly as long as that evidence does and comes back marked stale when the evidence changes. A twin finding's evidence is the pair, so an engram that twins two others carries two findings and each takes an acknowledgment of its own. Every run reports how many findings its acknowledgments suppressed, a suppressed finding can be brought back into view with its note, and an acknowledgment can be taken back. Acknowledge only what the person actually ruled on, because the sweep is only as trustworthy as the decisions it was told to respect.

Under the hood

Both records are `edit_engram` with `set_frontmatter`, one lifecycle field at a time. `verified` stamps who checked and when, and it never removes (one entry per verifier, refreshed each time that person re-checks). `stale_after` takes the later review date. `evolve_ack` takes the rule and the optional note as its value, `V101 lineage citation, keep`, and the value `remove V101` takes that acknowledgment back. `include_acknowledged` on a run shows the suppressed findings with the note on each.

10.7 When to sweep

The sweep is event-driven maintenance, never a session-start ritual. A large ingest that lands a dozen engrams at once calls for it, and so do two search hits that disagree when a half-finished retirement is the likely reason. The third event is the calendar: the manual suggests the first session of a month. The reminder at the end of a session helps with that one: when maintenance is due, chapter 6′s stop hook (in Claude Code, Codex and Copilot) adds a paragraph asking the agent to run the sweep and work the queue, mechanical findings directly, judgment findings one at a time with a yes. The calendar can also run the sweep without you: the evolve loop fixes the mechanical findings on a schedule and leaves the judgment ones for your next session (Chapter 14).

Under the hood

The sweep reads dates, links and graph shape, plus meaning for the twin check when the instance computes embeddings. With `evolve.contradictions` set to `full` it also reads the contradiction scores the daemon stored: the daemon scores pairs between sweeps (its model downloads on first use), so a sweep itself scores nothing. The rule for old, never-verified knowledge is capped per run, oldest first, so an archive that's never been swept gets asked about in instalments. The twin check is capped too: ten pairs per domain, closest first, and a very large domain is skipped with a line saying so. An acknowledgment lives in the engram's own frontmatter, beside the scope the server recorded for it, so it travels with the file and survives a rebuild of the index, which stores nothing about a sweep between runs.

A colleague who is corrected gets better, one who never is gets confident, and the sweep is how you keep telling the difference.

In brief

Knowledge drifts rather than breaks: windows lapse, review dates pass, replacements land without the retirement being finished, captures go unreviewed, links dangle, tags fork. The sweep, `crystalline evolve` or `evolve_engrams`, returns a read-only ranked queue in four families, each finding naming the engram, the evidence and the one action. The agent completes mechanical findings in a pass and proposes judgment findings one at a time, and Fluid's maintenance page shows a person the same queue, to look at rather than to work. Split out what still holds first, then retire by superseding: a new engram with `supersedes`, the old one set to `superseded` with a real `valid_to` and a `superseded_by`, the lesson carried forward. Correct in place only when the old fact was never true. A recheck is a `verified` stamp or a later `stale_after`; an intentional finding is an `evolve_ack` naming the rule, which holds while its evidence holds. Sweep after an ingest, when hits disagree and on a cadence, and stop when the number stops falling.

CHAPTER 11

Possible contradictions

This chapter is about the one kind of drift the rest of the sweep can't see: two engrams that say opposite things, with nothing in the archive connecting them. It explains the check that looks for those pairs, what it does with them and how to switch it on.

Nobody contradicts themselves on purpose, and an archive doesn't either. Say the agent picks up one thing in March and hears the opposite in October; the October session writes a fresh engram and moves on. Now both are current and both turn up in search, and which one the agent recalls next is down to how you phrase the question. Orwell called it doublethink, holding two beliefs that cancel each other and accepting both, and an archive does it without noticing. The sweep in Chapter 10 finds a retirement somebody started and didn't finish, because a `supersedes` link points at an engram still marked stable. When nobody started the retirement, it has nothing to go on: no link, tag or date ties the two engrams together.

11.1 What it is for

The contradiction check reads for meaning: it compares what two engrams say about the same subject, and when a line in one seems to say the opposite of a line in the other, it puts the two lines in front of the agent as a possible contradiction, finding code `V302`.

The finding says possible because a language model does the reading, and models misread things (a negation that isn't there, a line that's really the exception to the other one). The model can't see time either, so "an order arrives by Friday" and "an order takes two weeks" read as a contradiction, and both can be true, one before the shop changed its distributor and one after. So a finding is a question, never a verdict: the check doesn't pick the true line and keeps its hands off the engrams.

It's the agent's question, since the engrams are its own knowledge. A finding is the agent noticing that two things it learned disagree and bringing that to you, the way a colleague stops by your desk: you told me two things about this, which one still holds? You answer, and the agent edits its own knowledge to match.

11.2 How it works

The check picks pairs of engrams that talk about the same subject, then compares their observation lines one by one, and only lines about the same thing reach the model. Not every engram takes part, and not every pair:

- Engrams with the status draft, proposed, idea or poc stay out, as do retired engrams and any engram past the end of its validity window.
- Two engrams whose validity windows never overlap are never compared, because they were never true at the same time.
- The check compares engrams within one domain only, so a line in `books` that contradicts a line in `shops` is never found.
- German and English both work, and so does a line that mixes the two.

There's no cloud in the loop: the model downloads once and runs on your machine, and your lines never leave it.

You get at most one finding per pair of engrams, quoting the two lines that disagree most strongly, naming both engrams and saying how many more line pairs between the same two engrams read the same way.

The daemon does the reading in the background: after a change lands and its embeddings are computed, it reads the new pairs and stores what it found, so by the time anybody asks for a sweep the work is done and the sweep only picks up the stored results. Right after you switch the check on, the first sweep says how many pairs aren't scored yet, and the count goes down as the daemon works through them.

11.3 Three answers to one question

Which edit fits depends on the story behind the two lines, and you know the story (the agent mostly guesses). When one of them is false going forward, the agent retires it: it supersedes the engram the way Chapter 10 retires a fact that stopped holding, or it takes the stale line out when the rest of its engram still holds. When both are true, one the exception to the other, it acknowledges the pair and the sweep stops asking about those two lines. When each held at

a different time, the newer engram supersedes the older one, whose window closes at the real date it stopped holding, and that takes the pair out of the check, since a retired engram is never compared.

On the first of the month maintenance is due, so you start the morning with a sweep (Fluid shows the same kind of queue, Figure 3.2), and it turns up all three:

```
You:    Anything from the sweep?
Agent:  The mechanical findings are done (two links). Three
        possible contradictions need you. Each time I learned two
        things at different times that disagree:

        1. books: "Standing preorders" still says to preorder every
           new Vickers once it's announced. "Vickers: wait for the
           reviews" says to wait. When "buy on sight" retired, that
           line was missed and nothing links the two: take Vickers
           off the preorders?
        2. books: "Wait for the paperback" says no hardbacks. "The
           collector's shelf" (the exception moved there when
           collecting got an engram of its own) says a book that earned
           its place gets the hardback. I read that as the exception
           to the rule, both true: acknowledge the pair?
        3. shops: "Capsule Corp Books: by Friday" says a book ordered on
           Monday is in the shop by Friday. "Capsule Corp Books:
ordering
           in" says two weeks since the distributor changed in March.
           Both held, one after the other: supersede the first with
           the second, its window closed at the end of February?
You:    Yes to all three. Note on the hardbacks: only after the ebook,
        that's the point of the shelf.
Agent:  Done. Vickers is off the preorders. The hardback pair is
        acknowledged ("the collector's exception, after the ebook").
        The Friday engram is superseded by the ordering-in one, its
        window closed on 28 February, and its Friday line is
        acknowledged as expired with it, since it no longer holds.
```

The agent found the three pairs, read both engrams of each, worked out what it thought had happened and proposed one edit per pair. You didn't open a single engram, you answered three questions about your own life, the kind only you can answer: whether Vickers still belongs on the preorders, whether the shelf rule still counts, when the shop slowed down. Then the agent changed its own knowledge to match, and the next sweep doesn't raise those three pairs again.

Before the check, nothing in the sweep would ever have raised the Vickers pair. You watched "buy on sight" get superseded in Chapter 10, four parts and all, with the old notes kept as provenance. The preorder list was a different engram, though, with no link to the retired rule, so no structural rule had anything to fire on, and for months a question about the next Vickers could have ended in a preorder, until the check turned up the line the supersession had left behind.

> **In practice**
> Your agent keeps two engrams about Capsule Corp Books in `shops`. The old one says a book ordered on a Monday is on the shelf by Friday. Then a Tessier-Ashpool paperback took two weeks (the shop had switched distributors) and the agent wrote that down too, in a second engram. Nobody connected the two, so when you ask whether the new Doctorow will be there for the weekend, your agent recalls the Friday engram and says yes, and on Saturday the shelf is empty.
>
> Once the pair has come up in a sweep and you've told your agent that the Friday promise ended with the old distributor, the newer engram supersedes the Friday one, whose window closes the day the distributor changed. A week later you ask again, and this time the answer comes from the newer engram (two weeks, so order now if you want it for the weekend after next).

11.4 Turning it on

The check is off until you switch it on, and leaving it off is a fine choice, because it isn't free: the first time it runs it downloads its model, about 550 MB. While it's reading, the model takes a lot of memory: about 2.6 GB on an ordinary CPU and about the same on Apple Silicon. The daemon unloads it again when there's nothing left to read, so most of the day the check costs disk space and nothing else.

On a small machine (a Raspberry Pi, a laptop already short of memory) leave it off. The same goes for a domain too young to have drifted, since a few weeks of knowledge rarely disagrees with itself. Turn it on once the archive has a year or two and a lot of forgotten sessions behind it.

You switch it on by asking the agent, and it changes the setting with its `configure` tool:

```
You:    Turn on the contradiction check.
Agent:  Switched on (evolve.contradictions full). It applies now: the
        daemon downloads the model on its first pass, then reads the
```

```
related pairs in the background. The next sweep tells you how
many are still waiting.
```

Under the hood

The setting is `evolve.contradictions`, `off` by default or `full`, and it takes effect at once, no restart needed (Chapter 23 has the entry). From a terminal it's `crystalline config set evolve.contradictions full`, and its environment twin is `CRYSTALLINE_EVOLVE_CONTRADICTIONS`. Switching back to `off` deletes the stored scores on the daemon's next pass. `crystalline status` and `crystalline doctor` show the setting, the model, the device it runs on and how many pairs are still waiting; when the model couldn't be loaded, they say why, and setting the value again retries. A line pair counts as a finding at a contradiction probability of 0.95 or more, read in both orders. The check needs the default local embedding model, which is what picks the related pairs and lines; with any other embedding model the sweep says in one line that `V302` doesn't run. The answer to a finding is an ordinary edit: a supersession (with the old `valid_to` set to the real date when the two held in turn), a correction of the stale line or an `evolve_ack` of `V302` with a note, set through `edit_engram` like every acknowledgment. A `valid_to` that has passed under a live status is a finding of its own (`V001`), so the agent changes the status in the same edit. An acknowledgment covers exactly the two lines it was given, so a new disagreement between the same two engrams still surfaces (Chapter 26 has the full `V302` entry). On a shared Postgres database, two instances never score the same virtual domain at once: one takes it, and the other finds the pairs already scored.

In brief

The contradiction check finds two current engrams in one domain whose lines seem to say opposite things, the drift no link or date gives away. A finding, `V302`, is a question and never a verdict: the model can be wrong and can't see time, so the agent brings the pair to you and you decide. The agent retires or corrects the engram that's false going forward and acknowledges a pair where one line is the exception to the other. When both held at different times, the newer engram supersedes the older one, its window closed at the real date. The check is off by default, runs locally in the background after each change and costs a one-time download of about 550 MB and a few GB of memory while it reads.

CHAPTER 12

Reflection

Reflection is the place where people and the agent meet on a domain, and this chapter walks you through it. The agent writes its engrams for itself, organized so it can store, find and pick up what it learned, and nothing in them is phrased for you. Since people need what's in there too, the agent presents it in documents for people, at their level and in a shape they can use, and what people change or say about those documents flows back into its engrams.

12.1 Why not read the engrams

An expert who knows a topic doesn't hand you their notebook. They explain it at your level, leave out what you don't need and pick the order that makes sense to you. Engrams are the notebook: dense, cross-linked, full of observations written for the next session's search, and a person reading them is reading somebody else's shorthand.

You can still open the engrams in Fluid, as a look into the agent's mind (a bit like a session on the couch, or a debugger): you go there once in a while to inspect what it learned and fix it. With Reflection, Fluid gets a second mode beside the engrams. A domain with Reflection switched on opens in that mode, and people spend their day there.

12.2 Asking a colleague to present

Think of the colleague who knows a topic best: you ask them to give the team a presentation on it. Once it's up on the screen the room works with it, and the feedback improves it for everyone, presenter included: somebody spots a gap, somebody else knew a detail the presenter didn't, two people find out they meant different things by one word.

Reflection does that for a domain: switch it on and the agent presents what it knows as reflections, documents in a folder tree, as many and as deep as the domain agreed. Each one is made of sections (a heading and its text), and every

section records which engrams it was built from, so when one of them changes the agent knows which sections to look at again.

> **In practice**
> Reflection is off on every domain until its owner or an instance admin switches it on. In Fluid that's one select: pick Domain settings in the More actions menu on the domain page, set `reflection` to `on` in Domain policies and confirm with Turn on the Reflection. Or ask an agent that works as the owner or an admin to switch it on for you, and your local agent counts as one. The switch writes a first pact, and from then on the domain opens on its Reflection. For a team domain you do this at the team server, since its Reflection lives there (the club below goes through it).

12.3 The pact

Before anybody presents, the room agrees on how. Every domain with a Reflection has one pact, a file called `PACT.md`, in five parts: who the reflections are for, how much detail they carry, how they present it, how the tree is structured and which pages people asked for. The MANIFEST tells the agent what the domain holds, the pact tells everyone how to talk about it. Curators edit it, and everybody else who works on the reflections reads it. A person's change to the pact lands in the agent's queue like any other work, and the agent reshapes the reflections to match.

Most of the arguing is about the level of detail. The Hitchhiker's Guide had a pact too: Ford Prefect's fifteen years of field research on Earth went into the entry as two words, Mostly harmless. That's fine for a guide to the galaxy and wrong for a book club, so anyone who works on the reflections can argue with it in a comment, and a curator changes it when the argument wins.

The presentation part covers the shape of a page: the agent picks a diagram, a table or a callout where the content has that shape (messages between two parties, three options side by side, a step you can't undo) and writes prose everywhere else, the way a skill Crystalline ships teaches it. Where the pact says otherwise, the pact wins (a pact that says no diagrams gets none), and Chapter 13 shows what a page can hold.

12.4 Present, test, improve

The agent presents, and people read what it wrote, change it and comment on a passage. A person's change shows up at once as ordinary text (no badge, no approval step), and ticking a box in a task list doesn't count as one. In the background Crystalline notes what changed since the agent last presented that section, and the next time the agent works the queue it reads a person's change to a section before it rewrites that section. If a change looks wrong or doesn't fit what it knows, the agent opens a thread on the passage and asks. It does the same with two engrams that disagree (Chapter 11) and asks you; here the doubt is about a person's change, and that person is right there on the page.

Either the agent knew something the person didn't, or the person turns out to be right and the agent puts the correction into its engrams. A change ends one of three ways: the agent takes it in and presents the section again, it stays as the person's text or it gets talked through in the thread. The agent writes plain words there and never links an engram, since many of the people in a thread can't open one.

A comment can say what it wants with a label at its start: praise, nitpick, suggestion, issue, todo, question, thought, chore or note, optionally marked blocking, non-blocking or if-minor. The agent labels every comment it writes; for people a label is optional, and only a blocking one shows red. Agents answer blocking threads first. A thread belongs to whoever opened it: the agent answers a person's thread and leaves it open for that person to resolve. It may close its own threads, or a person's once the person has written back.

When the agent's own knowledge moves (a correction, a retirement, something new from a session), the sections built from it are out of date and the agent brings the change into the reflections. Something new that no section covers yet gets a new section or a reflection of its own.

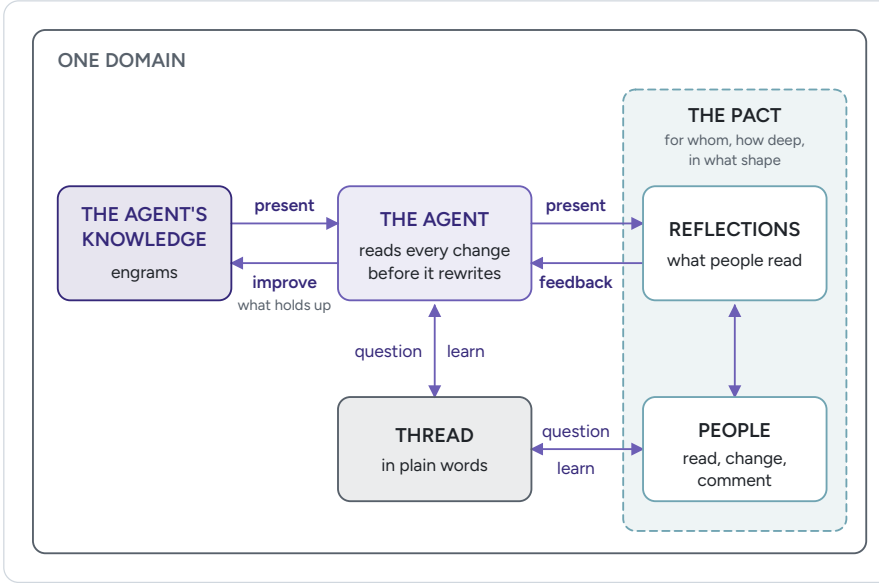


Figure 12.1: Present, test, improve: the agent presents what it knows, people change and comment, and what holds up in a thread goes back into its knowledge.

12.5 Somebody has to come by

Crystalline has no language model of its own, so it can't write a word of a page. What it does is notice what's due and keep a list, the reflect queue: a person's change to weigh, a comment to answer, an engram that moved under its sections, a new engram no section covers yet, a pact change the pages haven't caught up with, a file a page links that isn't there. The writing is the agent's job (Figure 12.1). If no agent comes by, the question Gaff asked last week sits unanswered in its thread and the Destiny page keeps naming last month's pick.

In an ordinary session, an agent whose user curates the domain hears about open items the way it hears about the sweep, and works them when you say yes:

```
Agent:  Before we start: club's Reflection has three open items.
        Gaff asked on the Dungeon Crawler Carl page whether book
        five is out in paperback, and two sections still name last
        month's pick. Shall I work them first?
You:    Go ahead.
```

For the weeks when nobody opens a session, an agent can come by on its own. A small job checks whether anything is due and only then starts an agent that

works the queue without anybody at the keyboard, on your own machine as well as on a team server (Chapter 14).

12.6 The club

Dallas runs the book club, and its engrams live in `club`, shared through the club's repository and served by the club's server, which you look after as its admin. Members keep asking Dallas the same things (which Star Trek volume is next, when the club gets to Dune), so he asks you to switch Reflection on, with a pact he has in mind. The switch belongs at the domain's home, and your laptop has no public address to be one, so switching on there is refused. You ask an agent that works on the club's server itself (the Domain policies section in the domain's settings in the server's Fluid does it too):

```
You:    Switch on Reflection for club. Dallas wants the pact like
        this: for members, no spoilers past the first chapter,
        a table for the reading order, one page per series.
Agent:  Switched on at the club's server, so club's Reflection
        lives there, backed up to its own branch of the club's
        repository. PACT.md says: for club members, spoiler-free
        past the first chapter, the reading order as a table, one
        reflection per series. Shall I present what I know?
You:    Go ahead.
Agent:  Six series pages: Hitchhiker's, Dune, Dungeon Crawler Carl
        and the Star Trek series the club reads (The Next
        Generation, Deep Space Nine, Destiny). Each says what the
        club read, what's next and what members thought, without
        the endings.
```

Lambert follows the club on her phone and has no wish to read engrams, so on her account you set the knowledge level to none and the Reflection level to contributor. She can edit every page of the club's Reflection.

She's on the bus home with the Dune page open on her phone (the club's Fluid) when she changes March to May in the line about when the club reads Dune.

That night the club's scheduled agent run works the queue and reads her change first. It contradicts the agent's engrams: the club picked March in January, after checking that the paperback was out by then. So it opens a thread on the passage:

```
Thread on "the club reads Dune in May"
  Agent:   question: This said March. The club picked March in
           January because the paperback was out by then. What
           moved it?
  Lambert: The reprint slipped. Capsule Corp Books says April now,
           and nobody wants the hardback. I wasn't there in January,
           so I didn't know March was about the paperback.
  Agent:   note: Then May holds. I've changed what I know about the
           club's list and the reprint, and presented the section
           again.
           Resolved: taken in, Dune moves to May because the
           paperback reprint slipped to April.
```

The agent opened that thread, so it may resolve it. With the reprint date in the engrams and Dune out of March, the club needs another pick, and at the next meeting it chooses the second volume of the Destiny trilogy, which Kane's agent captures in `club`. On a later run the agent sees that the next-up section of the Destiny page, built from the club's list, is out of date and puts the new pick in without anybody touching the page.

12.7 Files, sources and search

Pictures and other files on a page belong to the Reflection and sit in a folder of their own, linked relative to the page (Chapter 13 shows how one sits on a page). An engram's attachment can't be embedded in a page: copy the file into the Reflection's folder and link the copy. A reader sees the files, a contributor adds new ones and a curator replaces, moves and removes them, and the queue tells the agent about a file a page links that isn't there and about a file no page links.

One page can also sum up several domains, built from their engrams, as long as everyone who reads the page may also read those domains (Chapter 20), and the queue watches those sources like the domain's own. People see only the sources in domains they may open, and a reader who may read engrams here sees a count for the rest. A source in a domain this server doesn't have reads as unavailable, never as missing, so no agent drops it by mistake.

You find a passage with the search on the Reflection's front page in Fluid, and your agent searches the same way: by words and by meaning, section by section. Engram search never returns a page. A page also prints as a document, with a title block and its open threads as numbered notes at the end (Print view in its More actions menu); Chapter 13 has more.

12.8 Who sees what

With Reflection, access gets two axes, and you set them separately. One is the level on the engrams that Chapter 16 and Chapter 19 describe: viewer or editor (plus manager on a private domain and admin on the instance), and one more value, none, for somebody who sees no engrams at all. The other is the Reflection level, and there are two. A reader reads the reflections and comments (opening, answering, resolving and reopening threads). A contributor also edits their text and creates, moves and deletes reflections. Curating isn't a level anybody sets: everyone from engram editor up curates (Fluid calls them curators), and their agents curate with their rights. A curator also writes sections from the engrams, settles what people changed and edits the pact. A viewer of the engrams is a reader by default, and a Reflection level set on top only ever raises that (to contributor, say), so nobody who can read the engrams is kept out of their Reflection.

On a shared domain both levels sit on the person's account, set by an admin for every shared domain on the instance, and on a private domain they sit side by side on the membership. A domain's manager can also give one person the Reflection of that one domain, as reader or contributor, without changing anything else about their access, and that grant never opens an engram. Anonymous web visitors get in only where the instance lets anonymous requests in, and there they read the reflections of the domains an anonymous visitor can read, the way they read those domains' engrams, without ever commenting. On your own machine without accounts none of this applies and everything is allowed.

In Fluid the Engrams mode appears from knowledge viewer up and the Reflection mode from reader up, which is all Lambert sees.

12.9 Why the Reflection lives in one place

Every laptop on a team holds a copy of the engrams, and sharing keeps the copies in step. Pages can't work that way. People edit a page together and argue in its threads, and that only works among the people one instance knows: two copies would be two rooms with two conversations about one page, and each laptop's agent would present the same engrams its own way until the pages drift apart. Lambert doesn't read engrams, so a thread she writes in mustn't travel with them; as long as Reflection stays on, sharing changes sends only engrams to the repository. The levels the team agreed on live in the server's

accounts, while on your laptop you're the owner and may do anything, so a copy there would quietly get around them.

So a team domain's Reflection lives on its home, the team server, which backs the pages up to their own branch of the repository. With one writer, that branch never needs a merge (a branch per laptop would show private threads to everybody with access to the repository). A laptop's copy of a team domain holds no pages, and a domain that lives only on your machine keeps its pages where it is, as ordinary files.

Your laptop's agent reaches the pages once the laptop connects to the server (Section 16.10): the server's `club` then answers in place of the laptop's own copy, and you open the pages in the server's Fluid. Without that connection, the laptop's copy answers where the Reflection lives:

```
You:    What's next on the club's Dune page?
Agent:  The club's Reflection lives at https://club.example.com,
        and this laptop isn't connected to that server right now,
        so my copy of club has no pages. Connect to the club's
        server and I'll read the page there, or open it in the
        club's Fluid.
```

12.10 Two lifecycles

A domain with Reflection runs two lifecycles around the same engrams (Figure 12.2). On the left is evolve, the sweep from Chapter 10, with which the agent keeps its own knowledge true. Reflect keeps what people see in step with the engrams and brings back what people know. The two cross twice. A fix from the sweep (a retirement, a corrected date) leaves the sections built from the old engram out of date, so they land in the reflect queue. And what a thread settles lands in the engrams, where the sweep tends it from then on like everything else.

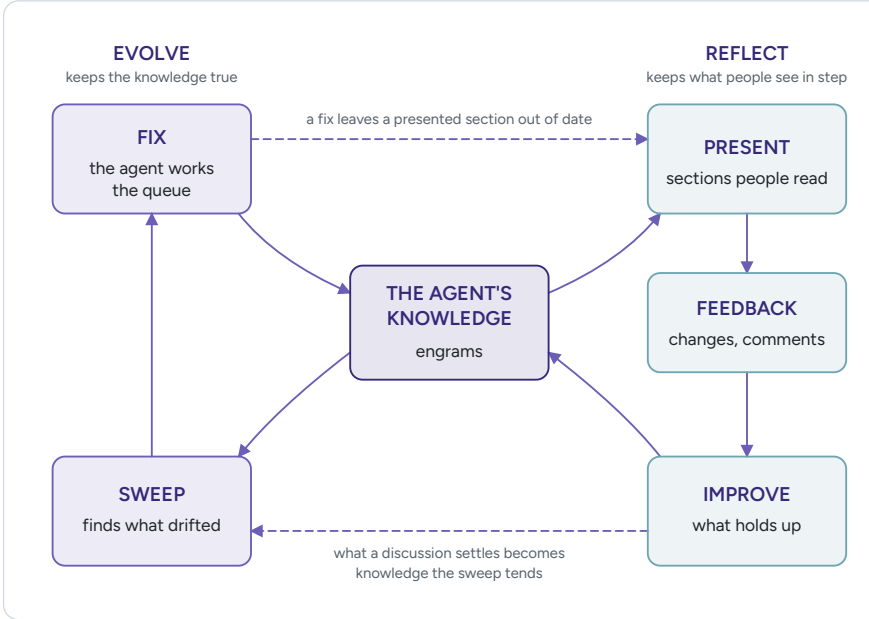


Figure 12.2: Two lifecycles around one body of knowledge: evolve keeps it true, reflect keeps what people see in step with it.

Under the hood

You switch Reflection on with `reflection: on` in the MANIFEST (it's `off` until you do), and the switch writes a starter `PACT.md` with five headings: `## Audience`, `## Level of detail`, `## Presentation`, `## Structure` and `## Pages`. On a domain with a GitHub origin the same change writes `reflection_home` too, and its value is this instance's `service.public_url`. Without a public URL the switch refuses and tells you to switch Reflection on at the team server. It refuses in a plain clone of a team repository too, one whose MANIFEST says `team: github`, and tells you to connect the domain and switch Reflection on at the team server. On any other instance the Reflection tools refuse and name where the domain's Reflection lives. Without a GitHub origin there's no `reflection_home`, because the Reflection simply lives where the domain does. The files all sit under `reflections/`: `PACT.md` at its root, the reflections as markdown files in whatever folders the pact describes (each `##` heading a section), the threads of a reflection in a `.threads.md` file beside it, the files pages link under `reflections/assets/` and a hidden baseline of what was last presented under `reflections/.reflected/`. Moving a reflection takes its threads and baseline along and rewrites relative links to it. While the switch is on, the folder stays out of engram sync, the engram index, evolve, `search_engrams` and sharing; the engram write tools refuse any path under it (`write_engram`, `edit_engram`, `move_engram`, `delete_engram`), and

reflections are searched in an index of their own, by words and by meaning, through `read_reflection` with a `query`. Switching on refuses while engrams live under `reflections/` and names them, so you move them first; it also refuses on a virtual domain and while the team's copy still holds files under `reflections/`. Switching off is always allowed, but don't share changes while it's off: the folder is engram content then, and sharing would offer its pages and threads.

The agent gets four tools, the same list for every client (a read-only instance hides `reflect`, `write_reflection` and `discuss`), and each of them refuses on a domain with the switch off. `reflect` hands over the queue ranked, recomputes it on each call and puts the oldest item first, with its age. In the queue, `feedback` means a person added, changed or removed a section since the agent last presented it, and the item carries the difference, the section's source engrams and who changed it. `update` means the knowledge behind a section moved or the page needs work, for example: a source engram changed, was retired or is gone, an engram has no section yet, a reflection strayed from the pact, a pact change isn't applied yet, a section cites a domain fewer people may read than the page's own (`source_too_wide`, Chapter 20) or a file a page links is missing or unused. And `review` items are open threads waiting for the agent, with the label of the opening comment and whether it blocks. Within one section a `feedback` item always comes before an `update` item, and a blocking thread before the others. `read_reflection` without a path lists the tree. Give it a path and you get that reflection with its sources and open threads; give it a `query` and it searches. `write_reflection` creates, changes, moves and deletes reflections and edits the pact (the pact only for curators). It also settles a person's change as `kept` or `taken_in`. A section written with its sources is presented from the engrams, and one written without is an ordinary edit. `discuss` opens a thread on a section or a quoted passage, answers, resolves with a short outcome and reopens; a resolved thread's history is never rewritten. An agent's resolve of a person's thread right after its own answer is refused, and in Fluid the person who opened the thread sees "Answered, waiting for you". The session-start briefing names a domain's Reflection when it has open items, and the Stop hook and the line after a write nudge the agent as they do for the sweep, but only an agent whose user curates that Reflection.

A source from another domain is written as its address, for example `crystalline://<domain>/<permalink>`, in the section's sources. A source in a domain that isn't on this server, or one a connected server serves, reads `unavailable`. A source in a domain fewer people may read than the page's own is refused, unless a person shares it on purpose (Chapter 20). Rename a domain with `crystalline domain rename`, which renames it in every page's sources as well; a name changed by hand in the MANIFEST leaves those sources

`unavailable` until you finish the rename (Chapter 18). A file is linked relative to the page, as in `![Flow](assets/flow.png)` from a page at the top of `reflections/`.

A read-only domain (Section 25.7) freezes its Reflection too: people and agents can still read it, but every edit, thread, settling and pact change is refused, and a queue over every domain leaves it out, as the sweep does.

The Reflection level (set by an admin on the account, or on a private domain by its owner or a manager) is `default` (Fluid shows "as knowledge"), `reader` or `contributor`, and it only ever raises access: the effective level is the higher of the default and the setting. Curating follows from knowledge `editor`, `manager`, owner or instance `admin` and is never set on its own. Without an explicit level, knowledge `viewer` gives `reader` and knowledge `none` gives nothing. An account whose write list leaves a domain out at most reads and comments there. `crystalline users add lambert --role none --reflection contributor` creates Lambert's account, `crystalline users reflection lambert reader` changes its level, and `crystalline domain members committee add gaff --level none --reflection reader` lets a member of a private domain read its Reflection only. The other way is a grant on one domain: `crystalline domain members club reflection add lambert --level contributor` (or `reader`) makes an account a Reflection collaborator of that domain alone, on a shared or a private domain kept as files, and `reflection remove` takes it away. The domain's owner, a manager or an instance admin gives it, Fluid lists it on the Members card under Reflection collaborators, and it never opens an engram. A caller with knowledge `none` gets nothing from any engram tool over MCP or REST and finds its domains through the Reflection domain list. On an instance that lets anonymous requests in, an anonymous web visitor reads the reflections of the domains it may read, as it reads their engrams, but never comments; otherwise it gets no Reflection. An anonymous agent over HTTP on an instance with `auth.mcp` off is the open tier, and it curates the Reflection the way it may write engrams.

On the home instance a job backs the Reflection up to the branch `crystalline/reflections` of the domain's GitHub origin with the instance's own GitHub credential: the pages, their threads and their files, never the hidden baseline. It holds back a page that cites a domain fewer people may read, and the engram branch never carries any of it. To restore from that branch, stop the daemon, clone the branch, remove the domain's local `reflections/.reflected/` folder, copy the branch's `reflections/` folder over the domain's, start the daemon and run `crystalline reindex`. The restored pages are the new starting point, so a person's change the agent hadn't taken in before the restore is gone. A domain export carries the Reflection, and an import leaves it out. To move the home, copy the `reflections/` folder to the new instance, switch Reflection off and on there in one go and share the MANIFEST change at once. On a shared Postgres database only the instance that hosts the domain serves its Reflection, and

`crystalline serve --take-over` moves it. The scheduled run is `crystalline reflect run`, started by cron, a timer or a runner job against the home instance: it ends at once when nothing is due, and otherwise works the queue with a headless agent and records the run (Chapter 14). `crystalline install` doesn't set it up.

In Fluid's top bar you switch a domain between Reflection and Engrams, and a domain with Reflection on opens in Reflection mode. It shows the tree of reflections, a search over them and each reflection with live co-editing, comments anchored on selected text with a Label menu and a Blocking only switch, and a jump from a section to its source engrams, hidden from people with knowledge `none`; curators edit the pact like any reflection. The users screen and the Members section in the domain's settings set both levels. A review screen per domain lists the open threads first, split into waiting for people and waiting for the agent, grouped by page, and below them the `reflect` queue, to look at the way Maintenance shows the sweep.

In brief

Engrams are the agent's own knowledge, and opening them in Fluid is the exception, for inspecting and correcting. Reflection is where people and agents meet on a domain: the agent presents what it knows as reflections, in the shape the domain's pact agrees on, and everyone puts them to the test. People change and comment, the agent reads a person's change before rewriting a section and asks in a thread when something doesn't fit, and since Crystalline has no model to write with, an agent has to come by, in sessions and on a schedule, to answer and bring the engrams' changes in. Access has two axes, so a contributor with knowledge none works on reflections without seeing an engram, while everyone from engram editor up curates, and a team domain's Reflection lives only on its team server. Evolve keeps the knowledge true, reflect keeps what people see in step with it and each feeds the other.

READY.

CHAPTER 13

How a page can look

> *No words. No words to describe it. They should have sent a poet.*
>
> *- Ellie Arroway,* Contact

This chapter shows what a page in Crystalline can carry besides paragraphs, whoever writes it, you or the agent. A page here is anything Fluid shows you to read: an engram, or a reflection in a domain's Reflection (Chapter 12). Both are plain markdown, and Crystalline adds a few elements on top: callouts, status labels, task lists, diagrams, math, footnotes and colour swatches. Each section shows the raw markdown and where you'll meet it.

Ellie Arroway ran out of words a long way from home. A page runs out of them sooner: three services passing a payment along, five options weighed on the same four points, a step you can't take back. Those you show; everything else still reads best as a sentence.

13.1 When a sentence is not enough

The rule comes from the skill the agent writes pages with: an element earns its place when it does a job prose can't. A diagram is there to explain an order or a branch, a table to line up a comparison, a callout to stop the reader before a risk. None of them is there to make a page look rich, and a page with no diagram and no callout is often the right page. Open the file in a plain editor or on GitHub and every element here still reads fine as raw text.

Here's how that looks when you hand the agent a page to write. The new on-call engineers keep asking how a deploy of the payments service works, and `engineering` has its Reflection switched on:

```
You:    The new on-call engineers keep asking how a deploy works.
        Write them a page in engineering's Reflection.
```

```
Agent:  Read the pact first: the pages are for engineers in their
        first month, one diagram per page at most. "Deploying
        payments" is up. The payment flow is a sequence diagram,
        because three parties hand a job along in a fixed order. The
        deploy windows are a table with a verdict on each row. The
        drain step carries a caution, since a job the queue drops
        doesn't come back. The rest is prose.
```

Most samples below come from that page, and most figures show it in Fluid.

13.2 Callouts and folding

A callout is a quote whose first line names its kind in square brackets. There are six kinds: NOTE for background that helps but isn't needed, TIP for a shortcut, IMPORTANT for a prerequisite the reader must not miss and WARNING or CAUTION for a risk. The sixth, DETAILS, is a neutral one for long reference detail or optional steps. The kind takes any letter case. A title can follow on the same line; without one, the title is the kind's own word. An unknown kind such as `[!DANGER]` stays a plain quote.

A sign right after the closing bracket folds the callout: `-` starts it closed and `+` starts it open. DETAILS without a sign starts closed, and every other kind without a sign doesn't fold. A folded callout opens by itself when a link leads to a spot inside it or when you pick a thread whose quote sits in it, and on paper every callout prints open.

```
> [!CAUTION] Drain first
> A deploy that changes job schemas drops every job still in the
> retry queue, and a dropped job doesn't come back.

> [!DETAILS]- Rollback steps
> 1. Stop the queue workers.
> 2. Roll the payments service back to the last release.
> 3. Start the queue workers again.
```

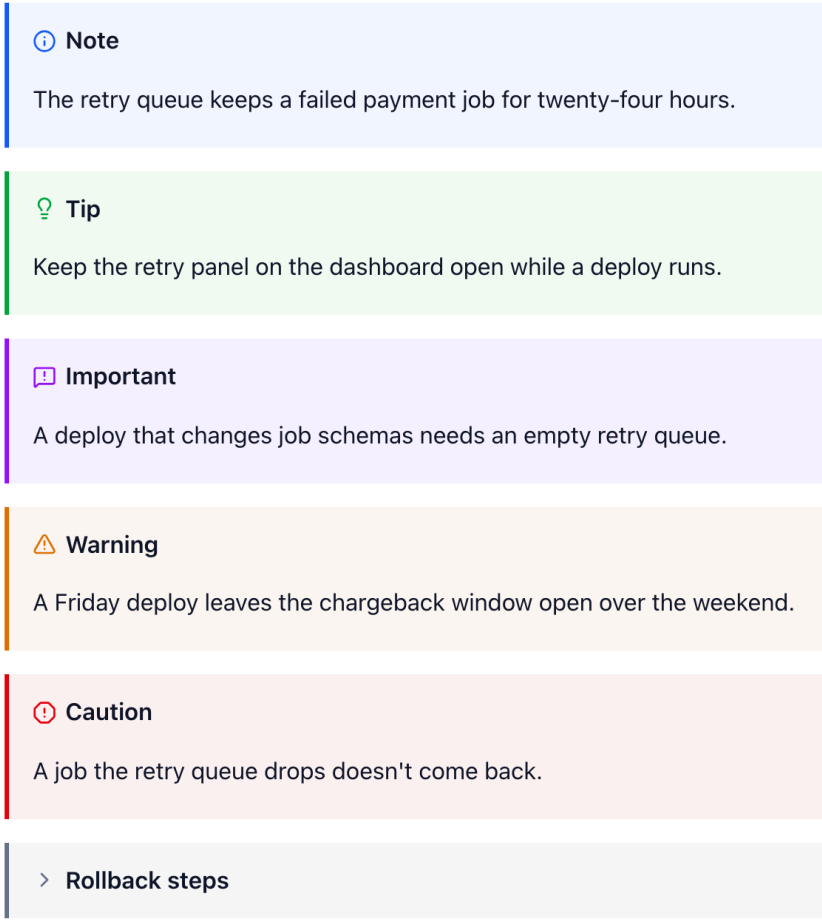


Figure 13.1: The six callout kinds on a sample page, the expand block folded at the bottom.

Use one or two per page, on top of any caution a risky step needs. Never put two in a row and never put a callout inside a callout. Crystalline folds any kind you ask it to, but leave a warning or a caution open: a risk the reader has to open is a risk the reader misses.

13.3 Status labels

A status label is a small coloured word inside a sentence or a table cell: `:status[In review]{blue}`. The colour is one of six (grey, blue, green, yellow, red and purple) in any letter case. Leave out the braces and the label is grey, and so is a label with any other colour. A label always carries a word, because colour alone isn't a signal (a black-and-white printer or a reader who can't tell red from green loses it). It fits best as a verdict in a comparison table or as the state of a plan:

```
| When | Why | Verdict |
| --- | --- | --- |
| Monday to Thursday morning | The whole team is in | :status[Go]{green}
|
| Thursday afternoon | Fewer people to call | :status[With a second
person]{yellow} |
| Friday | The chargeback window outlives the weekend | :status[Never]
{red} |
```

When to deploy

When	Why	Verdict
Monday to Thursday morning	The whole team is in	Go
Thursday afternoon	Fewer people to call	With a second person
Friday	The chargeback window outlives the weekend	Never

Figure 13.2: Status labels as the verdict column of a table.

13.4 Tables and task lists

Tables are the plain pipe tables GitHub uses. Reach for one when you compare three or more options on the same points, one row per option, or when a list of attributes (a name, an owner, a port, a limit) would otherwise become a long run of bullets. The editor inserts one at the size you pick, and with the caret inside a table its toolbar adds a row below or a column after, deletes either, aligns a column and lines every pipe up with Prettify table.

A task list is for things people act on and tick off. `[ ]` is open, `[x]` is done and `[~]` means the item doesn't apply: it's drawn struck through and never toggles.

```
- [x] Drain the retry queue
- [ ] Roll the payments service
- [ ] Roll the queue workers
- [~] Migrate the job schemas (none in this release)
```

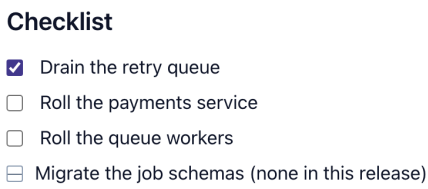


Figure 13.3: A deploy checklist, one item done and one that doesn't apply.

Anyone who may edit a page ticks a box right on the reading page, and the click changes that one line of the file. A reader without edit rights sees the box disabled, with the reason as its tooltip, and a page somebody has open in the editor gets the tick live. On a reflection a tick alone isn't feedback: it doesn't send the agent back to the section, while a real edit next to it does. Task items are never observations either, so a fact doesn't belong in one.

13.5 Diagrams and math

A `mermaid` code block draws a diagram, in light and dark, on a phone and on paper. Keep it small (about a dozen nodes), put a sentence before it that says what it shows and leave the colours to the page, which sets them for every theme. A label with special characters such as brackets or a colon goes in quotes, `A["Retry (3x)"]`. The Insert menu's Diagram submenu has starters for sixteen kinds in three groups (Everyday, Planning and product, Technical), and for a technical page the first five of the Everyday group carry most of the load. A flowchart shows a process that branches and a sequence diagram the messages between parties; a state diagram shows states and the moves between them, and a class or an entity relationship diagram shows a data model.

````
A payment passes through three parties in a fixed order:

```mermaid
sequenceDiagram
    participant W as Queue workers
````
````

```
    participant P as Payments service
    participant E as Encom
    W->>P: Job
    P->>E: Charge
    E-->>P: Approved
    P-->>W: Done
```

How a payment flows

A payment passes through three parties in a fixed order:

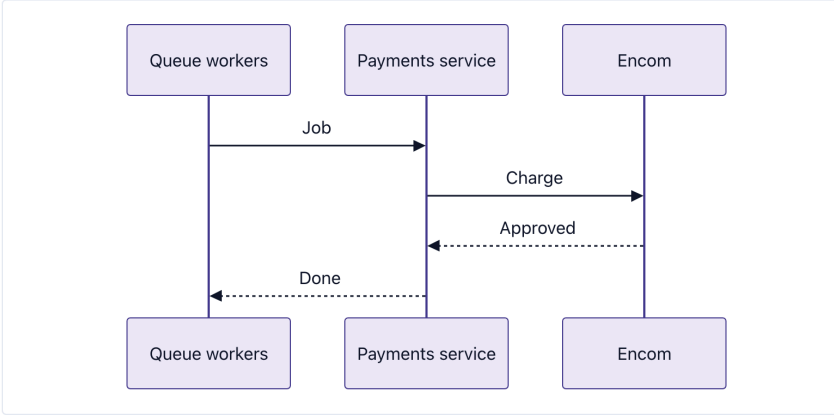


Figure 13.4: The payment flow as a sequence diagram on the page.

Math is for a formula the reader has to read, not for a number in a sentence. Inline, a formula goes in `$`a^2`$` or in `$$a^2$$` on one line. A block goes between two `$$` lines or into a `math` code block, one formula per block. A single dollar sign is always just a dollar, so a price like `$5` stays text.

```
A job that keeps failing is dropped at $`t_{job} + 24h`$.

$$
t_{drop} = t_{job} + 24\,h
$$
```

13.6 Footnotes, code and colours

A footnote takes a caveat or a source out of the sentence it would break. The reference gets a small link in the text and the note sits in a list under a hairline at the end. On a reflection, keep the reference and its note in the same `##` section, because Fluid shows each section on its own; the editor's Footnote entry puts the note there for you. A reference without a note stays as its text.

```
The queue drops a job after twenty-four hours.[^1]

[^1]: The cutoff mirrors Encom's chargeback window, so it isn't the
team's to tune.
```

Every code block on a reading page has a Copy button at its top right. Inline code that holds a single colour code, like `#D1242F`, gets a small square of that colour in front of it, and `rgb(...)` and `hsl(...)` work too.

```
The retry panel on the dashboard turns `#D1242F` once a job passes the
cutoff.
```

Roll the service with the deploy script, which waits until the service reports healthy:

```
./deploy payments --wait
```

Copy

The retry panel on the dashboard turns `#D1242F` once a job passes the cutoff.

Figure 13.5: A code block with its Copy button, and a colour swatch in the sentence below it.

13.7 Links

An engram links another engram by its title, `[[Encom chargeback window]]`, and Chapter 7 covers those links and the completion that writes them. A reflection doesn't link that way: on a reflection `[[...]]` stays plain text, and its editor has no Wiki link button and no title completion. An address works on both kinds of page, `crystalline://engineering/encom-chargeback-window`, and when the domain is renamed, Crystalline spells it with the new name. Plain links to the web are ordinary markdown, and on paper they print their address after their words.

```
Why the cutoff is a day: [[Encom chargeback window]].
The same engram from a reflection: [the chargeback window]
(crystalline://engineering/encom-chargeback-window).
Encom's own rules: [Encom for developers](https://developer.encom.
example).
```

13.8 Images and files

An engram shows its attachments, the files under the domain's `assets/` folder that Chapter 7 describes. A reflection keeps files of its own under `reflections/assets/` (the same file types, the same size limit). It links them relative to its own folder. A reflection never embeds an engram attachment: copy the file into `reflections/assets/` and link the copy. You add a file in the editor by pasting it, dragging it in or attaching it.

A fragment on the image's address places and sizes it: `left` or `right` floats it, `center` centres it and `full` takes the whole width. A width goes beside it, `w=50%` or `w=320` (pixels), after a comma. With the caret on an image the editor's Image format menu writes the fragment for you. Every other markdown renderer ignores it and still draws the picture.

```
![Retry flow](../assets/retry-flow.png#right,w=50%)
```

That line sits in a reflection one folder down, in `reflections/runbooks/`; on an engram the same image is `assets/retry-flow.png#right,w=50%`.

13.9 Writing it in Fluid

Engrams and reflections share one editor. The toolbar carries the everyday formatting and a size picker for a new table, and everything else waits in the Insert menu, from callouts and diagram starters to status labels, math and footnotes. Type `/` at the start of an empty line and the same list opens at the caret, tables and task lists included; keep typing to filter it, so `/cau` finds Caution, and so does `/danger`.

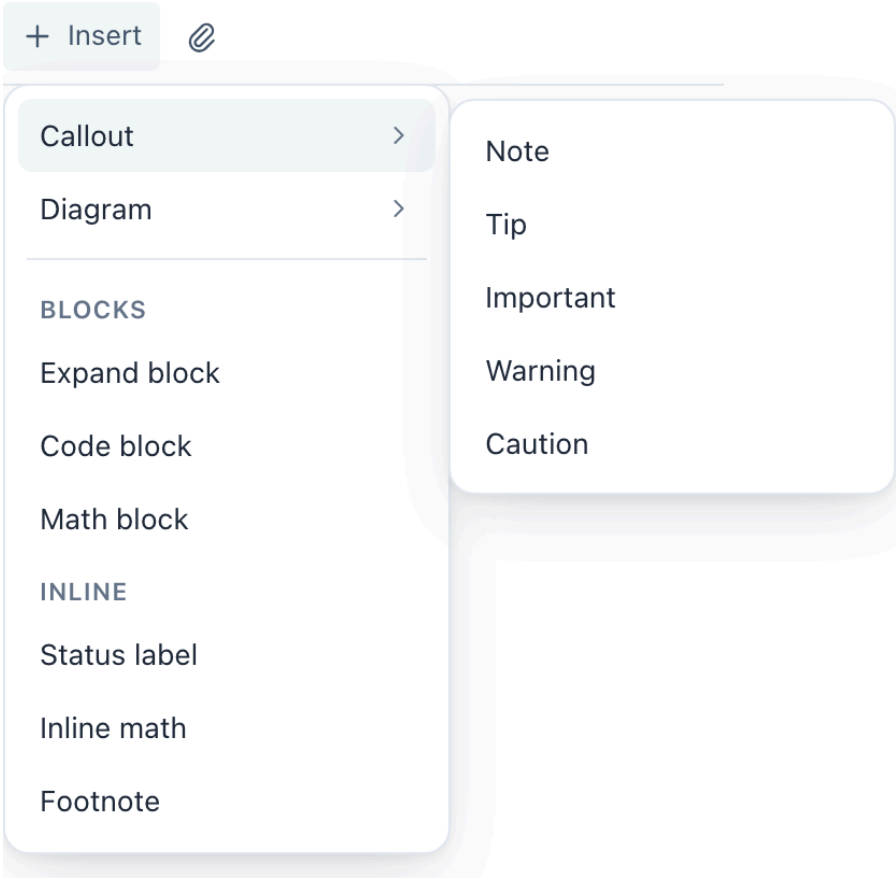


Figure 13.6: The Insert menu with its Callout submenu open.

Once an element is in, the toolbar follows the caret. Inside a callout it offers the kind and three buttons, Starts open, Starts closed and Does not fold. Inside a status label the toolbar offers the six colours. Away from the line you're typing on, callouts, labels, swatches and math show the way the reading page will draw them, so the page takes shape while you write.

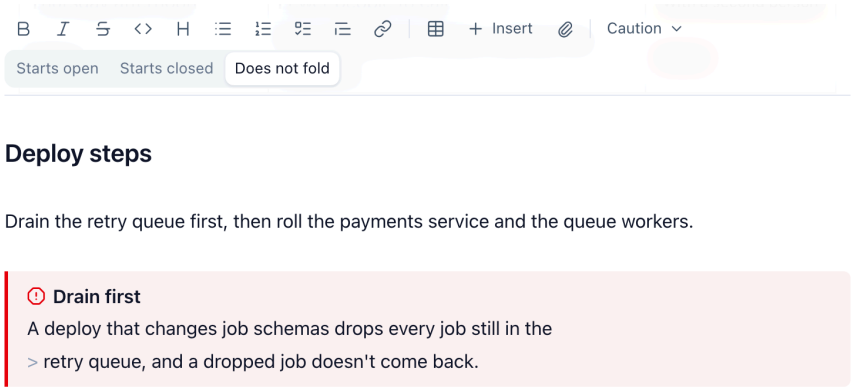


Figure 13.7: The callout toolbar with the caret in the caution: its kind and the three fold buttons.

Say you add the deploy page's caution yourself. On the empty line under the drain step you type `/cau` and press Enter, and a caution lands with its placeholder text selected. You type the warning over it, then go back to the marker line and add the title, Drain first. The toolbar now shows Caution and Does not fold pressed, and you leave it like that, since a caution stays open.

Pasting from another app

Copy a page from a wiki, a web page or a word processor and paste it into the editor, and Fluid turns the HTML into clean markdown:

From the other app	Becomes
Headings, emphasis, lists, task lists, code blocks	the same in markdown
Tables and links	tables and links
Info, note, tip and warning panels	callouts of the nearest kind
Expand sections	a folded DETAILS callout
Status lozenges	status labels in the nearest colour
Images	a link (attach the file yourself)
Anything else	plain text, never HTML

Scripts, styles and unsafe links such as `javascript:` ones are dropped, and a dropped link keeps its text. To paste plain text instead, use Cmd-Shift-V (Ctrl-Shift-V on Windows and Linux).

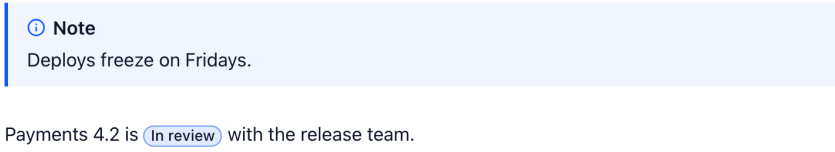


Figure 13.8: An info panel and a status lozenge pasted from a wiki, now a callout and a status label.

13.10 On paper

Print an engram or a reflection and you get a proper printout, laid out for paper. Every reading page has Print view in its More actions menu, which opens the browser's print dialog, and the browser's own print command gives you the same sheet. To get a PDF, pick the PDF destination in that dialog (Save as PDF in Chrome). A reflection also leaves as a file: Download as Markdown in the same menu hands you the page as plain markdown, named after its file and without Crystalline's bookkeeping in the frontmatter.

The first page opens with a title block and a small QR code of the page's address, so whoever holds the paper finds the live page with a phone. The sheet is always light, even when you print from dark mode, and menus and buttons stay off it. Links to other sites print their address, folded callouts print open and a heading stays with its text; Chrome adds a running header and footer with the page numbers.

The two kinds of page end differently. An engram ends with a short Related section: the engrams that link to it and its attachments. A reflection prints its open threads as numbered notes at the end, with a small matching number after each quoted passage in the text, so the deploy page on the table at a review carries the questions people still have about it. Resolved threads stay off the paper.

ENGINEERING / REFLECTION / RUNBOOKS

Deploying payments

Sections 5 Open threads 1

Open in Fluid

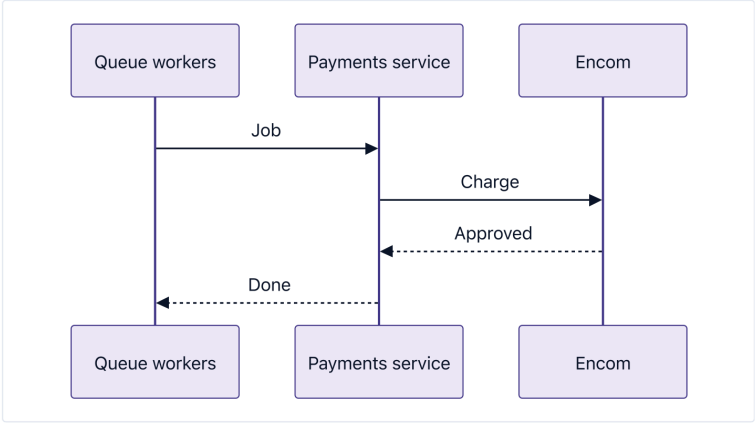
http://127.0.0.1:7417/d/engineering/reflection/runbooks/deploying-payments.md

Printed Nov 2, 2026

A deploy rolls the payments service and the queue workers behind it. This page walks you through one, from picking the day to rolling back.

How a payment flows

A payment passes through three parties in a fixed order:[1]



When to deploy

When	Why	Verdict
Monday to Thursday morning	The whole team is in	Go
Thursday afternoon	Fewer people to call	With a second person
Friday	The chargeback window outlives the weekend	Never

1 of 3

Figure 13.9: The first printed page of the deploy reflection, with the title block and its QR code.

13.11 How agents choose

The agent picks its elements from a skill that ships with Crystalline, `crystalline-presentation`, which it uses whenever it writes a reflection or the pact, works the reflect queue or writes an engram that a table or a diagram would make easier to use. A few shapes need their element every time, because a list or a paragraph isn't enough for them. Messages between two or more parties in a fixed order get a sequence diagram, even for three steps. A process that branches on a decision gets a flowchart, and states with the moves between them a state diagram. Three or more options compared on the same points get a table, one row per option, with a verdict column of status labels when the source gives a verdict. A step that can't be undone gets a warning or a caution inside the section that describes it, right before or after the step and never in the lead above the first ## heading. Before each section the agent checks those shapes in turn, and when none applies it writes prose.

Beyond the required ones the skill is strict about restraint: one or two callouts per page, one diagram per idea and labels in the reader's words. Feedback works the same way: when somebody comments "I don't follow" on a passage, more words aren't always the answer, and a flow that's hard to follow in prose is often easy in a diagram. On an engram, a table of settings or a diagram of a flow is welcome when it makes the knowledge easier to find or to use. Observations and relations stay top-level bullets there: inside a callout or a quote they aren't indexed.

The pact has the last word. Its Presentation part (Chapter 12) says which diagrams, callouts, tables and labels help this domain's readers, and where the pact and the skill differ, the pact wins. A pact that says no diagrams means no diagrams, even where the skill would require one, and a pact that asks for plain text and tables only gets its warning before an irreversible step as a sentence. A new pact starts with a Presentation part to fill in, and the agent reads the pact before it writes a page.

> **Under the hood**
> Crystalline checks these elements outside code blocks and inline code and reports three findings, listed with every other code in Chapter 26. Q006 is a warning for a status colour outside the six, which renders grey; Q007 is a warning for an unknown callout kind, which renders as a plain quote; Q008, for engrams only, is an info finding for an observation or relation inside a callout or any other quote, where it isn't indexed. The receipts of `write_engram`, `edit_engram` and `write_reflection` carry them as `markdown_findings`, a list of `{code,`

`severity, message, line}` that is there only when something needs fixing. The skill tells the agent to fix each one at its line and write again. `crystalline verify` reports them too, and the engram editor shows them with its other findings. Since 0.28.0 `crystalline verify --strict` fails on Q006 and Q007, so a domain that passed in CI before can fail now until the colour or the kind is fixed. An agent that reaches Crystalline only over MCP reads the skill with the `skills` tool.

On GitHub the five kinds without a title render as GitHub alerts, while a callout with a title, a folded one and DETAILS show as plain quotes with the marker as their first line. A status label reads as its raw text and a swatch as plain inline code. `[ ]` and `[x]` render as checkboxes, while `[~]` stays literal. Diagrams, math and footnotes render on GitHub too. GitHub also reads two single dollar signs on a line as math, which Crystalline never does. HTML comments such as `<!-- checked with the payments team in March -->` stay in the file but don't show on the page; the editor shows them on the line you edit.

A page with three or more ## sections gets an outline, On this page, with no syntax needed. On an engram it lists the ## and ### headings and marks the section you're in; on a reflection it lists the ## sections only, beside the threads.

In brief

Engrams and reflections are plain markdown with a few elements on top, all of which read fine as raw text: callouts (six kinds, folded with - or +), status labels in six colours that always carry a word, task lists with a not-applicable `[~]`, mermaid diagrams, math, footnotes, code blocks with a Copy button and colour swatches. An element earns its place when it does a job prose can't, and a page without any is often the right page. Fluid's one editor inserts them from the Insert menu or a / at the start of a line, previews them as you type and turns a page pasted from a wiki or a word processor into clean markdown; Print view puts a page on paper or into a PDF with a QR code back to the live page. Agents choose their elements with the `crystalline-presentation` skill, where the pact's Presentation part has the last word. The write receipts and `crystalline verify` report an unknown colour or callout kind.

CHAPTER 14

The loops

This chapter is about keeping a domain's two loops running when nobody opens a session: what a run does on its own, how you start one on your own machine and how a team sets one up on its shared instance. Crystalline has no model of its own, so somebody has to come by (Chapter 12), and here that somebody comes on a timer.

14.1 The heartbeat

Every domain has two lists in Crystalline. The reflect queue holds what the Reflection needs (a person's change to weigh, a thread waiting for an answer, a page whose engrams moved), and the sweep holds what the knowledge needs (Chapter 10). Neither list does anything by itself: work leaves it only when an agent takes it, and in a session that's when the reminder comes up and you say yes.

The loop is the heartbeat of the Reflection. It beats on a schedule you pick (every few minutes, every night, every Sunday), and each beat asks one question: is anything due? Most of the time the answer is no, and the beat ends right there. No agent starts and nothing is spent. When something is due, the beat wakes an agent, which works the list and goes back to sleep. Keep it beating and the question about Lambert's change is in her thread the same evening. Without a loop, the domain looks fine for a week, and then you notice the threads getting older and the pages still naming last month's pick. Fluid shows the pulse: the Reflection home says when the domain was last reflected, by which runner and how it ended, and the maintenance page does the same for the sweep.

If you write software, the loop most closely resembles a Kubernetes reconcile loop. The engrams and the pact are the desired state, the pages are the actual state and the queue is the difference. The loop works the difference until none is left, and it never assumes it's done just because it ran. Most engineers know the event loop better, and that picture works too: the queue holds the events,

the beat is the tick and the agent is the handler. The picture breaks in two places: nothing spins between beats, and taking an item doesn't consume it (the queue is computed fresh each time, so an item stays until the agent has really settled it).

14.2 Two loops, one shape

Figure 12.2 showed the two lifecycles of a domain, and each gets a loop of its own. The reflect loop works the reflect queue: it weighs what people changed, answers threads and brings changed engrams into the pages. The evolve loop works the sweep, but only the mechanical findings, the ones that complete intent the archive already records.

For both, the runner first asks whether anything is due and ends at once when nothing is. Otherwise it takes the lease, so only one run of a loop works a domain at a time. It starts an agent with a fixed set of instructions and lets it work up to a cap (twenty items by default), then shares what changed and records how the run ended. What the agent doesn't reach stays due for the next beat.

The reflect loop runs only where the domain's Reflection lives. Every other instance says where it runs and never starts it, so five teammates with the same team domain don't start five reflect runs. A team domain runs its scheduled evolve loop in one place too, its evolve home: the first scheduled run on an instance with a public address claims it, and that run's pull request carries the claim to the team. Every other copy names the home and refuses a scheduled run, and the sweep in a session still works everywhere. A domain only you keep runs evolve wherever it lives.

14.3 What a run does without you

A run has nobody to ask, and its instructions say so: the agent never waits for a yes. When it needs a person's answer, it asks where people will see it: the reflect agent opens a thread on the page, and the evolve agent names its question in the run's summary. Anything that needs a yes is left for a person: a judgment finding, a delete, a move, a page a person wrote. The tools for deleting, moving, splitting and sharing are switched off for the run, and its instructions rule out the rest. An item no run can settle isn't picked up again until it changes (a new message in its thread, a new edit), so a question waiting for Lambert doesn't start a run every five minutes all day.

A run also reads only the domains everyone who reads its own domain may read, and it writes only into its own domain. Anybody who may comment on a page can ask a question in a thread, and the run that answers can't draw that answer from a private domain the page's readers may not open (Chapter 20).

On a team domain the run's engram changes arrive as a pull request, never as a direct share, whatever the domain's sharing setting says. That's the same process a person goes through (Chapter 17): an agent working alone at night makes mistakes like anybody else, and the team reviews its work in the morning. The pull request carries only what the run wrote, and a file somebody was still working on is held back for that person. Reflection pages are never part of it (the domain's home keeps them and backs them up as always).

A run that works ends with a short summary, and in the club it reads like this:

```
Done: 4
Left for a person: 1
Errors: 0
Thread on the Dune page: asked Lambert what moved Dune
from March to May. Left open for her.
```

A run that fails doesn't hammer the domain: the loop waits longer after each failure in a row, and the first run that works ends the wait.

14.4 On your own machine

Your own machine is the entry level, and you have three ways to keep the beat:

- a session now and then, where you say yes when the reminder lists what's due (you know this one already);
- a scheduled run, once a night or once a week, that does the same thing without you;
- a watcher that checks every minute and runs as soon as something is due, handy when people wait for answers.

A run on your own machine starts Claude Code as its agent (so Claude Code has to be installed) and acts as you. It can't tell its own writes from yours, so don't edit the domain on that machine while a run works it. On a machine only you use, that's rarely a problem: the run happens at 6:30 on a Sunday, and you're asleep. While it works, your own sessions on that machine read and write the way the run does, and every answer says so in one line.

In practice

Your `books` domain keeps growing, and the sweep you keep meaning to run hasn't happened in months. You ask your agent to make it happen on its own: every Sunday morning the evolve loop should look at `books` and fix what it may fix. It sets up the schedule and runs the loop once to check, and the first run finds three links that point at titles you spelled differently in the reading log and fixes them.

The run leaves "Wait for the paperback" alone, although its review date has passed: whether the rule still holds is a judgment, and only you can make it. The next time you open a session, the sweep's reminder brings it to you as a question. Most Sundays after that, the run is over within a second, since nothing's due.

Under the hood

`crystalline evolve run --domain books` and `crystalline reflect run --domain books` run one beat: they check what's due, take the lease, start `claude` with the shipped prompt and only the Crystalline MCP server, share, record and release the lease. `claude` has to be on the `PATH` of the run, and Claude Code's own installer puts it in `~/.local/bin`, so a crontab names the folders:

```
PATH=/usr/local/bin:/usr/bin:/bin:/home/you/.local/bin
30 6 * * 0 crystalline evolve run --domain books >> /home/you/
crystalline-evolve.log 2>&1
```

`crystalline reflect watch --domain books` is the watcher, a long-running process for a login item or a systemd user unit, and `crystalline <loop> status` shows what's due. A local run needs the daemon, and on a team domain it shares the files that changed since it took the lease, minus what you hadn't shared when it started. The flags, the exit codes and the run logs are in Chapter 25.

14.5 On a shared instance

On a shared instance the loops are a job for whoever runs the server, and it's the advanced case: the run needs an account of its own, a place to run and, if you like, a way for the server to call it. Remember the night Lambert changed the Dune date in Chapter 12: the agent that asked her about it came from such a run.

The run gets an agent account of its own (Chapter 19), with no password and one token, and the account needs the right to the loop it runs: curating the Reflection for the reflect loop, writing the domain for the evolve loop. Give each loop its own account when both work one domain at the same time, so each pull request carries only its own run's changes. On a domain whose access follows GitHub (Chapter 19), the account gets its rights the way a person does: an admin links it to a GitHub user of its own, a bot with write access on the repository, and that covers either loop, since a GitHub writer also curates the Reflection. Making the run's account an admin works too, but it's far more than a loop needs. The runner talks to the server as that account, and so does the agent it starts. That's how the server tells the run's writes from everybody else's, and what a person edits in Fluid during the run never ends up in the run's pull request.

The runner itself can be almost anything that starts a program on a schedule: a systemd timer on the server, a Kubernetes CronJob, a scheduled GitHub Actions job for a team without a cluster or an n8n workflow that brings its own agent. A runner that polls every five minutes ends within seconds when nothing is due, so polling is cheap. If you'd rather not poll at all, the domain switches a trigger on and the server's operator gives it an address, and the server calls a runner that listens there (an n8n workflow, say) when work becomes due. That's within a minute when a person is waiting for an answer, and after half an hour of quiet when only pages need updating (so a bulk import sends one call, not hundreds).

On the club's server (the one you look after) both loops run, each under its own account. The reflect loop started as a nightly timer, and that's the run that asked Lambert about Dune. These days the timer fires every five minutes, and a question about a changed date usually waits in the thread before its author gets home. The evolve loop runs once a week, and its first pull request also carried one line for the MANIFEST, which made the club's server the domain's evolve home. On Monday Dallas finds a pull request titled "Evolve run on club" with the date and the run's summary as its description, two fixed links and one wired relation, and merges it with his coffee. The club's Reflection home says "Last reflected 4 minutes ago by timer: done." and, beside it, that one item is waiting for a person. A member who reads the knowledge sees that the loop ran, Lambert doesn't see the line at all, and someone who may run the loop also sees the summary and the pull request.

Under the hood

In remote mode a runner needs nothing but the `crystalline` binary, Claude Code with its Anthropic credential and two variables: `CRYSTALLINE_REMOTE_URL` names the server and `CRYSTALLINE_REMOTE_TOKEN` (or `CRYSTALLINE_REMOTE_TOKEN_FILE`) carries the `cmt_` token of the agent account, which `crystalline users add <name> --role editor --mcp-token <label>` creates with the account. Under a GitHub App an admin links that account on the server with `crystalline connect github --personal --as <name>` (in a container through `docker exec`) and confirms the code while signed in to GitHub as the bot, a collaborator with write access on the repository. A GitHub user links to one account only, so the two loops need two bots, and on a paid organization plan each takes a seat on a private repository. The runner then speaks to the server's control route as that account, and the agent reaches the server's MCP endpoint with the token as a bearer header. `--runner` labels the run, so Fluid says "by timer" or "by k8s". The reflect loop runs against the domain's Reflection home. On a team domain the evolve loop runs against its evolve home, the instance whose `service.public_url` is the MANIFEST's `evolve_home`: the first scheduled run on an instance with a `service.public_url` writes the key, and that run's pull request carries it to the team. Every other copy says `runs on <home>` and refuses a scheduled run, while `evolve_engrams` and `crystalline evolve` in a session work everywhere. On a shared database the home is asked first, then the host lock. A domain only you keep has no evolve home. A runner that reaches the server by a name people don't use, a cluster service name for example, needs that name in the server's `CRYSTALLINE_SERVICE_ALLOWED_HOSTS`. A systemd service for the club's reflect loop runs `crystalline reflect run --domain club --max-items 20 --runner timer` with `SuccessExitStatus=1`, since a run with nothing due exits 1, and a timer starts it. The crystalline repository's `examples/` folder has a runner image and CronJobs for Kubernetes, a GitHub Actions workflow and n8n workflows; for a runner that starts its own agent, `crystalline <loop> prompt --domain club` prints the shipped prompt.

A domain asks for the call with its trigger keys, and the operator sets the address and the secret. The body carries counts, never content, and is signed with HMAC-SHA256 over the timestamp and the body, so the receiver checks that the instance sent it. The full guide, with the signature check in a shell and in n8n, is the crystalline repository's `docs/automation.md`. The keys are in Chapter 24 and the settings in Chapter 23.

In brief

Crystalline keeps the lists, and the loops make an agent come by when nobody opens a session: the heartbeat of the Reflection, which most of the time finds

nothing due and costs nothing. The reflect loop keeps the pages in step with the engrams, and the evolve loop fixes the mechanical findings of the sweep. A run never waits for a yes: it asks people in a thread or names the question in its summary, leaves what needs a yes for a person and, on a team domain, delivers its changes as a pull request. On your own machine a scheduled run or a watcher acts as you; on a shared instance a runner works as an agent account of its own, on a schedule or called by the server's webhook.

READY.

PART IV

The System

READY.

CHAPTER 15

What Crystalline provides

Picture a workshop with two doors and a hatch. Through one door come the agents; that door is a protocol, and whoever arrives through it is handed tools. Through the other come people; that door is a browser tab. The hatch at the side is for anyone with a terminal open, which is where the command line lives. Behind all three is one room with one set of shelves, and on the shelves are plain files. The room has a single keeper, a daemon, that watches the shelves, keeps the index warm and answers whoever comes in, whichever way they came. That's the whole architecture, and the rest of this chapter walks the room surface by surface: the command line, the tools an agent sees, search, the graph, time, Fluid, the skills and the sweep. Chapter 1 named them and the last eight chapters used them one at a time. This chapter lays them side by side and says what each one lets a colleague do.

The room is also, for most of its life, invisible. Once it's set up, the agent comes in through its own door at the start of every session and you come in through the browser when you feel like it. The keeper starts itself when the first visitor arrives, and the hatch stays shut. The command line below is for the install you've already done, for standing up a shared instance and for the occasional maintenance run. If the list of commands looks long, read it as a tour of what's there rather than a list of what you'll type.

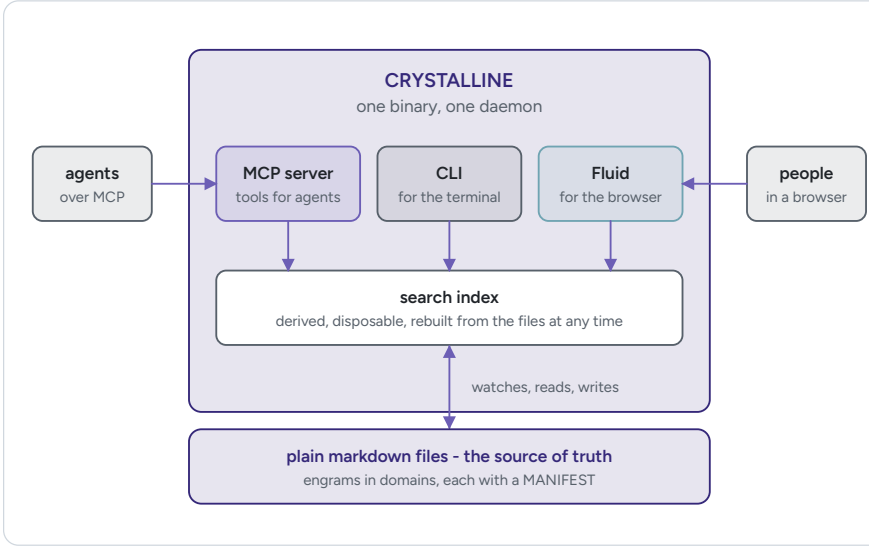


Figure 15.1: One binary: the CLI, the MCP server, Fluid and the search index, all standing on plain markdown files. Agents come in through MCP, people through Fluid.

15.1 The command line

Crystalline ships as one binary, and the binary is the product: no server package beside it, no database to install first, no UI to deploy separately. `crystalline install claude-code` wires a harness in one step (Codex, Copilot, Cursor, Kiro, Gemini CLI and Qwen Code take the same command with their own names). It registers the MCP server, writes the hooks the harness supports and copies the skills into place. Every harness gets the session-start hook that delivers the routing prompt. Claude Code, Codex and Copilot also get the stop hook that nudges capture, and in Claude Code and Codex the prompt hook hands each question the engrams that look like a fit. Run it twice and the second run writes nothing and says so. `crystalline uninstall` reverses it and leaves anything that isn't Crystalline's own untouched.

Domains are managed the same way. `crystalline domain init` scaffolds a folder with a starter `MANIFEST.md`, `crystalline domain add` registers it, and `domain add --virtual` registers a domain that lives in the database with no folder at all, for a container without a writable volume or a knowledge base you'd rather not mirror to disk. `crystalline domain remove` takes one off the instance and leaves its files where they are. A virtual domain has no files and its engrams go with the registration, so there the command refuses until you add `--purge`. `crystalline domain rename` repairs a name that turned out wrong and keeps

the old one working as an alias (Chapter 18). `crystalline users` manages the accounts that can sign in to the web UI, although the first admin is normally created in the browser. `crystalline prompt system` renders the routing briefing to stdout, ready for any harness with a session-start hook, and `prompt connector` prints the standing instruction for a chat client that has no hooks. `crystalline serve` runs the daemon in the foreground and `serve --daemon` in the background, though most people never type either, because the first agent that connects starts the daemon on its own.

The rest of the CLI mirrors what an agent can do, for scripting and for the quick fix that doesn't deserve a session: `write`, `read`, `edit`, `split`, `search`, `context`, `vocabulary`, `evolve`. Three commands ask three different questions about health. `verify` asks whether the format holds, statically, with no daemon involved, and it is what the bundled GitHub Action runs on every proposal a team reviews. `doctor` asks whether the machinery around the format is healthy and repairs what it safely can with `--fix`. `evolve` asks whether the knowledge itself is still true and still well organized, which neither of the others can. A fourth, `import`, brings an existing markdown-plus-frontmatter knowledge base under Crystalline as a pure file transformation, dry run first.

> **Under the hood**
>
> Exactly one process ever holds the index open. The first `crystalline mcp` or `crystalline serve` takes an advisory lock and becomes the daemon; every later agent, terminal or CLI command attaches to it over a local socket. The daemon watches the domain folders, embeds new engrams in the background and keeps running after the last agent disconnects, so the index is warm for the next session; it stops only when told to, with `crystalline ctl shutdown`. The daemon of the Desktop-only bundle on a Mac is different: it exits five seconds after its last client. What it keeps is derived from the files: `crystalline reindex --full` rereads every file and keeps the embeddings, and `crystalline reindex --wipe` rebuilds an index that won't open, so a corrupted index or a schema change is never a data-loss event for a folder domain. A virtual domain has no files behind it (its engrams live only in the index), so the wipe refuses while one is registered. The accounts for the web UI live in their own small database beside the index, never inside it, so a rebuild can't take them along.

15.2 The tools an agent sees

An agent meets Crystalline as a set of tools over MCP, and they're best described by what they let a colleague do, because that's how the agent chooses among them. Twenty-four are listed on a default writable instance and thirty with team collaboration turned on. Four of them belong to Reflection and are the same for every client: `reflect`, `read_reflection`, `write_reflection` and `discuss` (Chapter 12). A read-only instance hides `reflect`, `write_reflection` and `discuss`.

The core is small. `search_engrams` is recall: search before writing, search to remember, across every domain or a chosen few. `read_engram` fetches the whole engram, because a search hit is a snippet and a colleague who answers from a snippet is guessing. `write_engram` captures something new, and it always names its domain; there's no default, so nothing lands in the wrong place. `edit_engram` refines what exists instead of forking a second engram about the same topic: append a section, replace one, find and replace a phrase or assign one lifecycle field by name, which is how an engram is retired or re-verified with no text matching involved. Both answer with a hint: the receipt of a write, or of an edit that adds text, may name up to three existing engrams closest in meaning to what just landed, with guidance to merge into the one that fits, supersede it or link it and never to ignore the list silently (`capture.similar` switches it off). `split_engram` is for the engram that mixes lifecycles. Validity is set per engram, not per bullet, so when one fact in it expires while the rest still holds, the survivors move out into a new engram first and only the remainder retires. The pair stays linked, `derived_from` on the new engram and `split_into` on the source. A split that would leave the source under three content lines is refused, because that's the case for retiring the whole thing. `build_context` walks out from an address and assembles the neighbourhood. `vocabulary` lists the tags, categories, relation types and statuses already in use, so an agent reuses a term rather than coining a near-duplicate. `evolve_engrams` runs the maintenance sweep. `configure` is the settings page, for the instance or, given a domain, for how that domain shares and which checks it runs. `add_domain` gives an agent somewhere to write on an instance that has nowhere yet. `remove_domain` is its counterpart: the registration and the index rows go, and the knowledge stays. In a domain in review mode the removal would also end everyone's unshared drafts, so there it refuses until each person who is still drafting is named. A folder's markdown sits on disk until an `add_domain` re-adopts it, and a team domain is reconnected by its repository. A virtual domain's engrams live in the database and go with it.

Around them sit the tools for orientation: `list_domains`, which re-fetches the session briefing mid-session; `browse_domain`, one level of the folder tree at a time; `move_engram` and `delete_engram`, with delete reserved for mistakes; `recent_activity` to catch up on a week; and `skills` to read a playbook. A move works like a refactoring in code: every link to the engram in the domains you may write follows it to its new address. A domain that wants its engrams to share a shape gets `validate_engrams` and `infer_schema`, and one that ships tools of its own gets `provision`. The six collaboration tools share, update, check standing, settle a conflict, withdraw a proposal and put unshared files back the way the team has them. Chapter 17 is theirs.

The difference between this and a lookup service is the difference between a colleague with a badge and a visitor with a lanyard. The tools let an agent act: find what applies, read it in full, do the work and leave behind what it learned, with nobody relaying anything.

15.3 Search

The most used surface is the one nobody notices, and it blends two signals. Full-text matching finds the exact phrase, the way you find your own notes when you remember the wording. Semantic similarity finds the engram about dropped queue jobs when the question was about payments going missing, because the two are about the same thing in different words. Crystalline runs both and ranks by the blend. The local embedding model is fetched once, on first use or ahead of time with `crystalline model download`; until then, search falls back to plain text.

Search also decides the size of a session. An agent that keeps its knowledge as a folder of markdown files reads the folder, and reads it again next session, every file whether or not the task touches it, because a folder has no way of saying which page matters today. An agent working from a knowledge base receives only the knowledge that applies. The search is routed to a domain first, then narrowed by the indexes: full text and semantic similarity for the phrase and the meaning, the graph of relations and backlinks for what the answer leans on, tags, the folder prefixes `build_context` globs and the temporal filters for what holds now. Reading a folder costs the whole folder, every session. Recall costs the engrams that matter, and nothing else is loaded. Chapter 9 made a section of it.

Two priors tilt the ranking, both small and both bounded. A `salience` on an engram lifts it above equally relevant neighbours without ever excluding

anything, for the rare piece of knowledge that outranks its surroundings. A retired status fades an engram softly, so current knowledge surfaces first and the superseded decision from March sits one scroll further down, still there. Searches also filter: by domain, type, tag, status, any frontmatter key or a recorded-after date, and a search with no query text at all is a legitimate way to ask for everything of one kind. Fluid puts the same ranking behind a faceted page (Figure 15.2).

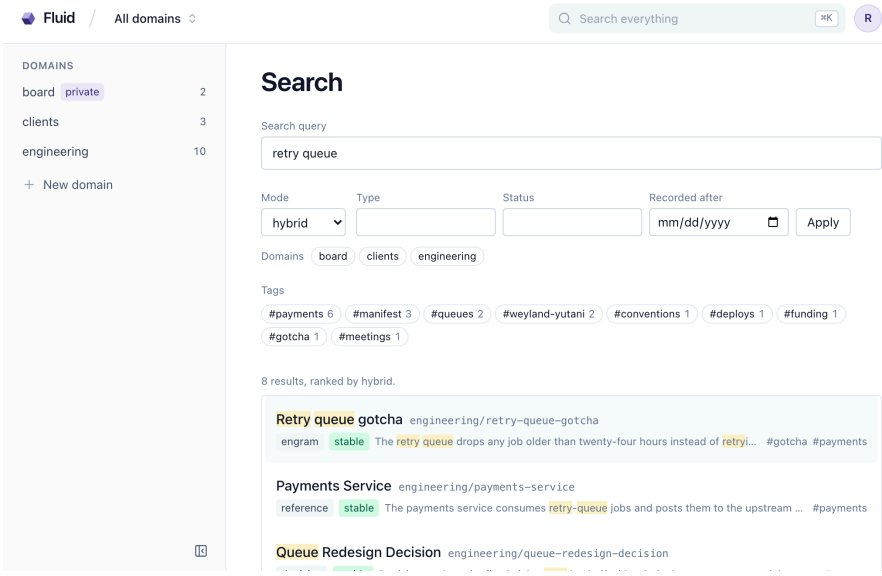


Figure 15.2: Faceted search across the instance, backed by the ranking the agents use.

15.4 The graph

Every relation in an engram's body, `- depends_on [[Encom chargeback window]]`, is an edge, and the target's page shows it back as a backlink, so knowledge points both ways without anyone writing the return trip. A `[[Title]]` in prose counts too. Links resolve within the domain by title, and across domains with a prefix, `[[engineering:Retry queue gotcha]]`; move an engram to another folder or domain and every link, relation and address pointing at it is rewritten in the domains you may write. The moved engram's own links keep finding what they found, and the receipt names any link the move couldn't fix.

`build_context` is what makes the edges useful before a task. Hand it an address and it follows the relations and links outward, one to three hops, across domains if the edges cross, and returns the neighbours ranked by how strongly each

connects to the anchor, with salience counted and retired statuses ranked lower. A folder prefix with a `/*` suffix anchors a whole topic at once. The agent starts the task holding a neighbourhood rather than a flat list of hits, which is what an experienced colleague does when they pull up the runbook, the decision behind it and the incident that prompted the decision before touching anything. Fluid draws the same neighbourhood as an interactive graph (Figure 15.3).

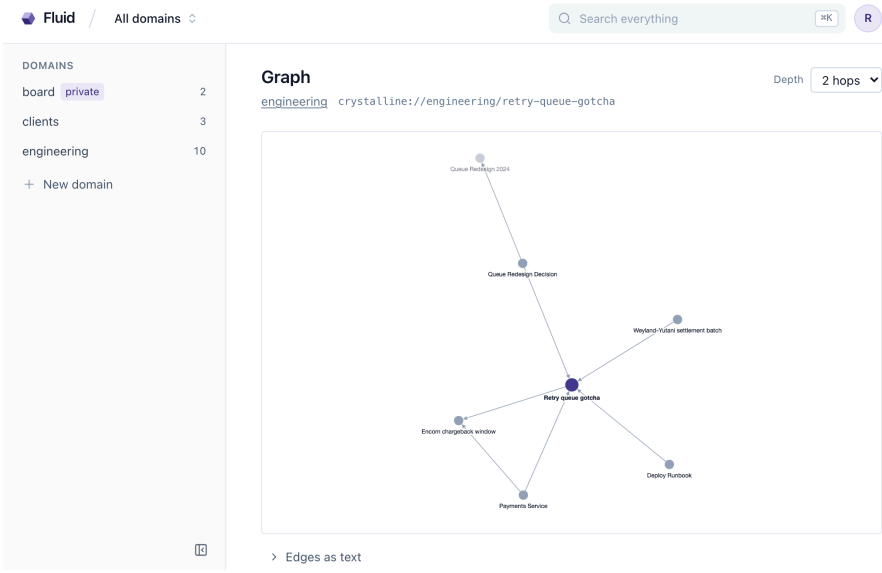


Figure 15.3: The neighbourhood graph of an engram in Fluid.

15.5 Time

Every engram records when it was captured and who wrote it, and can carry the dates that bound it. `valid_from` and `valid_to` are plain dates, and the rule that makes them usable is the one worth memorizing: an absent `valid_from` means always valid, an absent `valid_to` means valid forever. A far-future date written to mean forever is dropped at write time because absence already says it. `stale_after` asks for a re-check by a date; `verified` records that the check happened and who did it; `source_date` says when the underlying source was written. A retired engram closes its window with the real transition date, which is what lets a search filtered on those fields answer what applied last June with the engram that was true then. Chapter 9 put the dated questions to work and chapter 10 the retirements.

15.6 Fluid

The door for people opens at `http://localhost:7411`, on by default on the same daemon the agents use, and the first visit to a fresh instance creates the admin account in the browser. The menu behind your avatar links to the latest edition of this handbook. An engram is a page (Figure 15.4): the frontmatter as a rail of details, each observation led by its category and each relation by its type, backlinks, the `crystalline://` address one click from the clipboard and an agent's-eye view showing exactly what the tools serve an agent for that page. You switch domains from the top bar, and Cmd+K jumps anywhere by name.

Behind the same door is Reflection (Chapter 12): in a domain that switches it on, the agent presents what it knows in documents written for people, and the top bar flips between Engrams and Reflection.

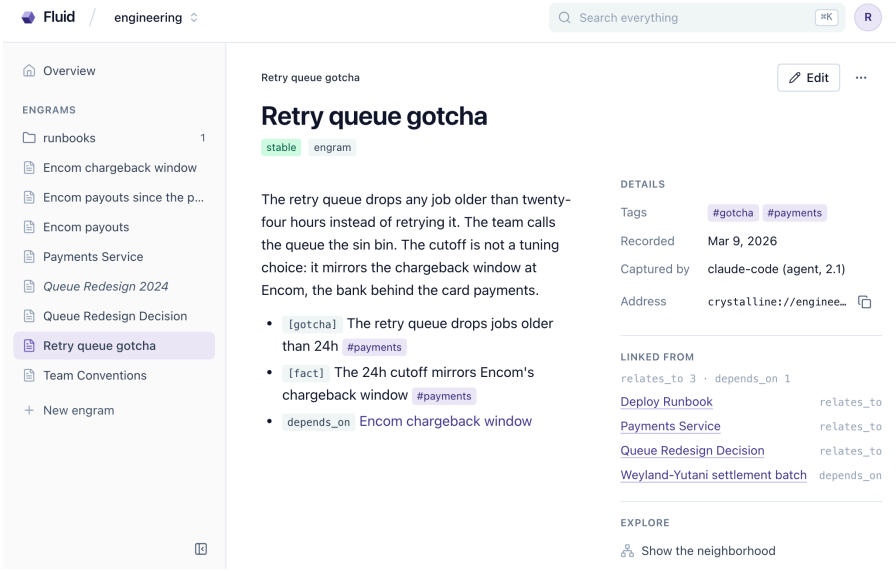


Figure 15.4: An engram in Fluid: details rail, observations and typed relations.

Editing happens in place, in a live-preview editor with a frontmatter form in the details rail and wikilink completion across every domain, and the file on disk stays the source of truth. A solo save answers like a write over MCP, with a panel of the nearest existing engrams, each linked to its page, replaced on the next save and gone when you dismiss it. Attachments go onto an engram by paste, drag or upload: a screenshot, a slide deck, a PDF, a data file. They're how a person hands over teaching material, and they aren't opaque: an agent reads them back over MCP and the sweep notices when a fresh file still needs

capturing. Two people in the same engram see each other's cursors, and their edits merge conflict-free into one save. Who can sign in, and how, is chapter 16's.

15.7 Skills

A tool tells an agent what it can call. A skill tells it when, and how to do it well. Crystalline ships six harness-agnostic skills as plain folders with a `SKILL.md` each: routing, which domain to search and when to sweep them all; capture, what is worth writing down and the search-before-write, edit-over-create discipline; schema, for a domain that wants structure; collaboration, for a domain with a team behind it; provisioning, for a domain that ships its own tools; and presentation, for pages people read. A seventh, consolidated skill folds the essentials into one file for Claude Desktop and any harness that installs one skill at a time.

`crystalline install` copies the skills into the harness's own skills folder and keeps them current across upgrades. All seven are served over MCP as well, through the `skills` tool and as resources, so a remote client that never ran the CLI can read the playbook before its kind of task. Loading one is the nearest thing an agent gets to "I know kung fu". The gap between an agent that has the tools and an agent that runs the capture loop is that folder.

15.8 Evolve

The last surface is the sweep, and it's a linter for what the agent knows. `crystalline evolve` from a terminal, `evolve_engrams` from an agent and the maintenance page in Fluid all show the same ranked queue of temporal debt, structural gaps, redundancy and meaning. Each finding names the engram, the evidence and the one action that clears it, marked mechanical or judgment so the agent knows whether to act or to ask. It reads dates, links and graph shape (and meaning only to spot two engrams that say the same thing or, once you switch it on, two lines that may contradict each other). It changes nothing itself; `crystalline evolve run` hands the mechanical findings to a headless agent on a schedule (Chapter 14). It also surfaces what a person captured directly in Fluid and nobody has reviewed, and an attachment nothing references. Chapter 3 walked the loop it serves and chapter 10 worked a queue end to end.

In practice

Say a small press sends you its spring catalogue, forty pages of it. You don't read it aloud to anyone. On Sunday you open the Tessier-Ashpool engram in Fluid, drag the PDF onto it and add one observation of your own: the paperbacks come out in March, six months after the hardbacks, which is how long the rule makes you wait. You close the tab. A week later you ask the agent for a sweep, and it lists two findings against that engram. One is an unreviewed human capture, which the agent verifies, tags to match the vocabulary and wires into the graph. The other is a fresh attachment whose knowledge hasn't been extracted yet. The agent reads the PDF over MCP, proposes three titles for `to-read/` with the page each came from, and you say yes.

The following Tuesday you're in a hurry and type `crystalline search "the one with the ship's computer that sulks"` into a terminal. The to-read engram comes back first, from a question the engram never poses. You open the graph view and see the press one hop away, the paperback rule and the shop that orders from them beside it, and the press's page still carries the catalogue where you left it. Two doors and a hatch, one room: the browser on Sunday, the tools all week, the terminal on Tuesday, and none of the three had to be told what the others had done.

Eight surfaces, one room, two doors and a hatch: everything on either side of this chapter is one of them, used on purpose.

In brief

Crystalline is one binary with two doors and a hatch: agents come in over MCP, people through Fluid and the terminal through the CLI, and behind all three is one daemon over one set of plain markdown files. The CLI installs harnesses, manages domains, users and the routing prompt, and mirrors every tool for scripting; the MCP surface is, on a default instance, twenty-four tools that let an agent search, read, write, edit, walk the graph, check the vocabulary and run the sweep. Search blends full text with semantic similarity and tilts for salience and against retired status; the graph turns relations into backlinks and neighbourhoods; the temporal fields make absence mean unbounded. Skills teach a harness when to use all of that, and evolve is the read-only linter that keeps it true.

READY.

PART V

Getting Started with a Team

CHAPTER 16

Running a shared instance

The moment to leave localhost is the moment two people want to correct the same engram. Up to then a laptop is the right size: one person, one agent, one folder, nothing to operate. Past it, every fact the team learns is being taught twice and drifting once, and the fix is the oldest one in computing. Put it on a server, give everyone a login and let one instance be the one the whole team teaches.

The questions arrive in a fixed order, and the chapter follows it: where the instance runs; who may sign in, and how, the question chapter 15 handed here; how an agent on somebody's laptop proves who it acts for; how one person keeps a domain to a smaller circle; how the server reaches GitHub on its own; and how a laptop brings a server's domains in beside its own.

16.1 When to move off localhost

Chapter 1 settled where Crystalline runs: on a machine of yours, never on a hosted service. This chapter is about which machine. Any one of four signals is enough. Somebody who never opens a terminal wants to read what the agents learned and fix a fact without asking. An agent runs somewhere that isn't a laptop (a hosted runner or a bot with a task) and needs to recall and capture like everyone else. You work from more than one machine (or from a harness that can't start a local binary) and want the same colleague on all of them. Or the same engram has been corrected on two laptops in two directions.

A shared instance differs from a team domain, chapter 17's subject, where a folder lives in a GitHub repository, each person runs their own instance and knowledge travels between them as proposals the team reviews. A shared instance is one daemon everyone connects to: one index, one Fluid, one set of accounts, one folder of files on one disk. The two compose, and a mature team usually has both.

16.2 The shapes it takes

Three shapes cover almost every team: a container, a systemd unit on a Linux box and a read-only mirror of knowledge somebody else curates. The full guide is docs/deployment.md in the repository, one scenario per section with a diagram each, and this chapter summarizes it. The three reduce to one picture (Figure 16.1): the daemon on a server, teammates' agents reaching it over MCP on HTTP instead of a local pipe, browsers reaching Fluid on the same port and, where the team configures them, a proxy in front and an identity provider beside it. The server can also live under a path of a host your proxy already serves, `https://example.com/crystalline` say, and laptops connect to that address as it is.

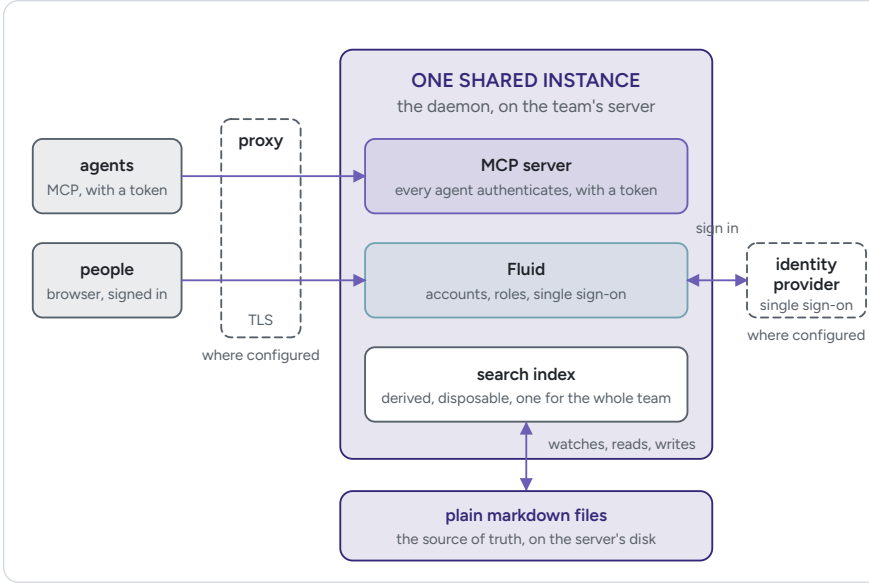


Figure 16.1: One shared instance: agents connect over MCP with a token, people through Fluid with an account, with a proxy in front and an identity provider beside it where the team configures them.

> **Under the hood**
> The team server is a container. The published image binds `0.0.0.0:7411` inside it, keeps the knowledge as a bind mount (so the files stay exactly the markdown they were) and puts the index on a volume:

```
docker run -d \
  --name crystalline \
  -p 7411:7411 \
  -v "$(pwd)/knowledge:/knowledge" \
  -v crystalline-data:/data \
  ghcr.io/jordiboehme/crystalline:latest
```

The `with-model` tag bakes the embedding model in, for a host with no egress. The same shape without a container is the `.deb`'s systemd unit: put `CRYSTALLINE_SERVICE_HTTP=0.0.0.0:7411` in `/etc/default/crystalline` and run `sudo systemctl enable --now crystalline`. A scale-out variant serves Fluid from an nginx tier in front of the daemon, and a read-only shape, `--read-only`, serves knowledge curated as a reviewed repository: the writing tools vanish from what an agent is offered while the index keeps following every pull.

Whatever shape you pick, a few things follow you into it. An agent reaching the daemon by a hostname needs that name in `CRYSTALLINE_SERVICE_ALLOWED_HOSTS`, since the MCP transport checks the `Host` header against a loopback-only allow-list to block DNS rebinding. Serve it over TLS anywhere but localhost. Off loopback the session cookie is marked `Secure`, so a browser on plain `http://team.lan` won't keep it. Every address the daemon builds for itself off loopback starts with `https://` too, including the one an identity provider sends the browser back to, so single sign-on on plain HTTP is refused at the provider. And a proxy in front must forward `Host` verbatim, pass the WebSocket upgrade through for co-editing and accept 10 MiB request bodies, as the daemon does. It mustn't buffer `/api/` either, or Fluid's live updates arrive only when the proxy's buffer fills. Where the daemon can't tell the address people reach it at (behind a proxy, in a container, with Fluid served from its own tier), `service.public_url` names it, and the links an agent hands out to engram pages start there.

A server under a path needs that setting: `service.public_url` carries the path, `https://example.com/crystalline`, and nothing else turns a prefix on. The proxy may pass the path through or strip it. Pass it through if you can, because nginx decodes an escaped slash in a permalink when it strips. Forward the bare `/crystalline` as well, without a redirect, since an MCP client never follows one. With OAuth on, a client looks for two documents at the host root, so forward `/.well-known/oauth-protected-resource/crystalline` and `/.well-known/oauth-authorization-server/crystalline` too. `auth.oidc.redirect_uri`, if you set it, carries the same path in front of the callback, or single sign-on stays off with a warning. Every cookie is scoped to the path, so two instances on one host keep separate sessions. The separate Fluid image takes the path as

> `CRYSTALLINE_BASE_PATH`. The "Under a path" section of docs/deployment.md has the proxy configurations and four lines that check the setup from outside.

16.3 Accounts and roles

Nothing is readable until an account exists. A fresh instance greets the first browser with the create-first-admin form from chapter 5, with one difference: a daemon bound to anything but loopback cannot trust a visitor by where they connected from, so it prints a one-time setup token as it starts and the form asks for it. Read it in `docker logs crystalline`, or `journalctl -u crystalline` under systemd; a restart prints a fresh one until the first account exists, and none after that. You can also choose the token yourself with `CRYSTALLINE_SETUP_TOKEN`, at least 32 characters long; the daemon then prints no token and keeps yours across a restart. The terminal route needs no token: `crystalline users add ada --role admin` writes the accounts database directly, and a running daemon picks the account up on its next lookup. A container can skip both: set `CRYSTALLINE_ADMIN_NAME` and `CRYSTALLINE_ADMIN_PASSWORD_FILE`, the second pointing at a Docker secret, and the first start creates that admin, as long as no account exists yet. The plain `CRYSTALLINE_ADMIN_PASSWORD` works too, but it shows in `docker inspect`. A login page that faces the internet gets guessed at. Failed sign-ins for one name slow down (one second, then two, four and eight, then a refusal that says when to retry), and every failure is logged with the account and the address it came from, which is what fail2ban reads.

There are four roles, each containing the one before: none sees no engram at all and is for somebody who works only on a domain's reflections (Chapter 12); a viewer searches, reads and browses; an editor also writes and edits engrams; an admin also manages domains and users, from `crystalline users` or from Fluid. An agent acts for its account and is held to that account's role: if you're an admin, your agent can do admin things, and if you aren't, it's refused the way you would be. The network, the database and sign-in belong to whoever runs the server, though: no agent can change them, not even an admin's, and they change only from the command line, in `config.yaml` or through the environment. Accounts live in a small database beside the index, never inside it, so no reindex can take them along; on the container it sits on the `/data` volume and no rebuild restores it, so back the volume up.

When somebody leaves, an admin disables or deletes the account, and deleting sweeps whatever the account had connected. That's the whole deprovisioning story; there's no directory sync and no SAML.

An agent that runs on its own, a build runner or a nightly ingest, gets an account of its own, and that account can carry a write list: the domains it may write to, each named or given as a prefix with `*`, like `agents-*`. Every other domain is read-only for that account, while it still reads every domain it can see, so many agents can share one server, each reading the company's knowledge and writing only to its own domain. A list only takes rights away: a listed domain still needs the role and, for a private domain, the membership, so a viewer with a list still writes nothing. On a team domain under the server's own GitHub App, write access on the repository takes the place of both.

> **Under the hood**
> `crystalline users add bot --mcp-token bot --role editor --writes agents-bot` creates such an account, `crystalline users writes bot agents-bot,shared-agents` replaces its list, `crystalline users writes bot` prints it and `--clear` removes it; `crystalline users list` shows every account's list. There's no control for it in Fluid. A name that matches no domain yet is accepted with a warning, and a domain rename updates the entries that name it exactly, never a `*` entry. A write outside the list is refused with a sentence that names the account and the command that changes its list.

16.4 Single sign-on

Local accounts are right for a team of nine. Inside a company that already has an identity provider, people should sign in with the login they already have, and Crystalline talks to the provider through one generic integration, OpenID Connect, with no adapter per vendor. The configuration is three required keys, `auth.oidc.issuer`, `auth.oidc.client_id` and `auth.oidc.client_secret`, plus four optional ones: a label for the button, the scopes, the role a first sign-in lands at and the callback address. That last one, `auth.oidc.redirect_uri`, is for a proxy that rewrites `Host` and won't forward the public name; set it to the callback exactly as the provider has it registered. Discovery, PKCE and key rotation are the integration's job. Fluid's login page then shows "Sign in with" the label beside the local form (Figure 16.2), and both keep working, so an instance can run mixed: local accounts for bots and the break-glass admin, everybody else through the provider.

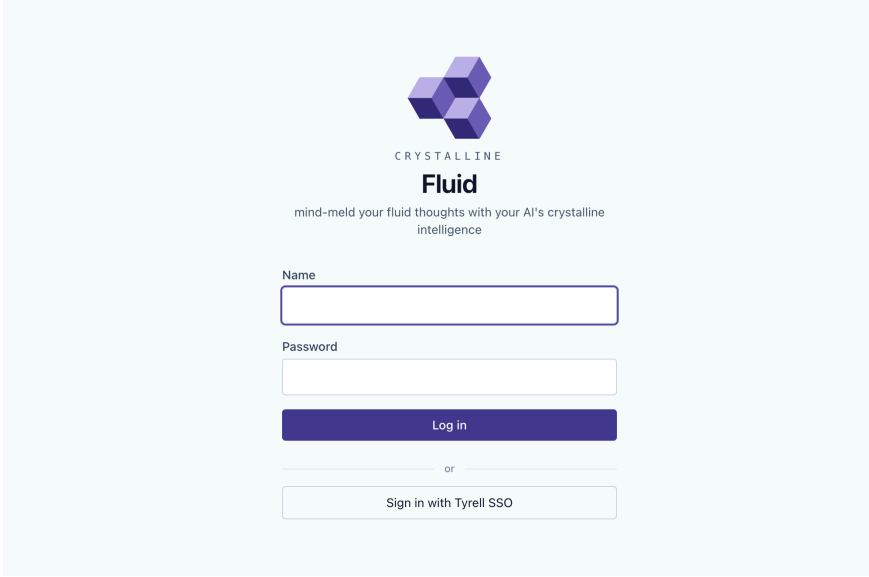


Figure 16.2: Signing in through the instance's identity provider.

The first successful sign-in creates the account, named from the provider's preferred username and made unique if it collides, at the role `auth.oidc.default_role` says, viewer unless you change it. What the account is keyed on is what most such integrations get wrong: the durable key is the pair of issuer and subject, the provider's own stable identifier, never the username, the email or the display name. Those are presentation; rename yourself at the provider and you land in the same account with a new label. And there's no silent linking: signing in through the provider with an email that matches an existing local account gets you a fresh account, never the existing one, and linking the two is an explicit act by the signed-in local user or by an admin. The provider authenticates and Crystalline authorizes; a group at the provider doesn't become a role here.

> **Entra ID: use the tenant-specific issuer**
> Microsoft Entra publishes a tenant-independent discovery document whose issuer field carries the literal text `{tenantid}`, and an issuer check against that template fails for every token. Crystalline refuses that value and the shared endpoints such as `common` at configuration time, and names the fix: the issuer must be your tenant's own `https://login.microsoftonline.com/<your-tenant-id>/v2.0`. Register a client secret, because that's the one credential Crystalline speaks. And Entra's subject identifier is pairwise per app registration, so re-reg-

> istering the application breaks every link between provider identities and Crystalline accounts, and an admin has to re-link them.

16.5 Trusted-header mode

Some deployments already have a proxy that authenticates everybody and names the arrival in a request header, and Crystalline can consume that. `auth.trusted_header` names a single custom header, `remote-user` say, and an account it names is created at viewer role the first time it's seen.

A second form, `auth.proxy_headers`, takes the standard set that forward-auth proxies emit (`Remote-User`, `Remote-Name`, `Remote-Email` and `Remote-Groups`), using the first for identity, the next two as presentation and reading the groups without acting on them. A name that arrives this way gets an account of its own, at the role `auth.oidc.default_role` says, never the local account of the same name. A forwarded `ada` beside a local `ada` becomes `ada-2`, under the same no-silent-linking rule as single sign-on. Pick one form or the other. With both set, the HTTP surface refuses to start and a warning names both keys, while the daemon itself keeps running and a stdio session never notices. Both are off by default, and a header that arrives while the mode is off buys the sender nothing.

The documentation doesn't soften the warning, so neither does this page. Header mode trusts the proxy completely, because a header is a line of text anyone can add to a request. It's safe only when Crystalline is unreachable except through the proxy, and the proxy strips any client-supplied copy of those headers before setting its own. A daemon reachable on its port directly is an instance where anybody can be anybody. If you can't say with confidence that the only path to the port runs through the proxy, use single sign-on instead. And create the first admin before the proxy goes in front: a request carrying forwarding headers is never counted as local, so the setup form behind a proxy asks for a token that a loopback-bound daemon never printed.

16.6 Agent authentication

On a laptop the MCP endpoint is open, on loopback, to whatever local process reaches it, and that's the right trade for one person. On a shared instance turn on agent authentication (`auth.mcp`), and from then on every agent that connects over HTTP identifies itself, exactly as a person does, with a token or by signing in. There's no half-way tier: a connection without credentials is refused before

any session opens, with a message naming how to get one; HAL refused Dave with less courtesy. The open tier exists only on an instance that leaves the setting off, and sees only the domains everyone sees. A server with its own GitHub App (below) is the exception: there the open tier acts on the team domains as the account `github.agent_identity` names, private ones included and at most as an editor, or as an anonymous visitor when the setting names nobody, which is one more reason to turn agent authentication on. An agent that could search a shared instance anonymously would be a colleague with no name on the commit. That's the whole setup, by the way: create the first admin, turn on agent authentication, done. Everything after this section is what you can add, not what you must.

The credential is a token that belongs to a person. Each user issues their own in Fluid, under the Agent access card on their profile, or an admin mints one for any account with `crystalline users mcp-token <name>`; either way it's shown once, and the instance keeps only a hash. It travels as an `Authorization: Bearer` header on the server's entry in the harness's MCP registration, never expires on its own and stops working the instant it's revoked or rotated, because every request looks it up. Every account may issue one, viewers included, and that's safe because of what the token means. Issuing one still works on a read-only instance, since a token is account state rather than knowledge (the sign-in path below is served there for the same reason); a read-only server with `auth.mcp` on is where an agent can't connect at all until somebody issues it one.

What it means is that the agent is its user. It sees what that person sees, writes where that person may write, shares under that person's identity when a team domain is involved, and the engrams it captures record which client acted for which account. A viewer's agent is therefore read-only with nothing further to configure, and a leaked token names one person, whose one token is revoked. One caller needs no token: the CLI and a stdio session on the server itself are the machine's owner and see everything, which is the honest statement of what a filesystem already permits.

16.7 A team of one

Nothing about a shared instance says the team has to be bigger than you. Put the daemon on a machine of yours that stays on (the box under the desk, a Raspberry Pi, a small VPS, the NAS that already holds the domains) and sign in to it from wherever you happen to be talking to a model: Claude Code on the laptop, Codex on the desktop, the Copilot CLI at work, a chat website in the

browser, the app on your phone. Each of them asks you to sign in once and is the same colleague from then on. Chapter 5 gave you one colleague per computer. This gives you one colleague. It also makes the knowledge yours in a way a vendor's chat history never is: switch harness, switch model, switch provider, and the domains don't notice, because they never lived with the model.

In practice

Your phone has never seen your `books` folder. On Saturday, in Capsule Corp Books, you pick up a Star Trek novel you don't recognize, open the chat app, sign in to the instance at home once and ask whether it's one you have. The answer comes from the shelf engram your agent keeps per series: it's the second volume of the Destiny trilogy, you own the first and the third, and the note from two winters ago, made on the laptop through Claude Code with the date on it, says you stopped at the end of the first because this one was out of print. Nothing was copied anywhere, and the phone and the laptop share one colleague.

That evening you open Codex on the laptop, sign in to the same instance and mark the trilogy complete on the shelf. On Sunday the phone knows, with Codex recorded as the client that acted for you. The colleague learned from two devices in one weekend, and the provenance says which of them acted each time.

Under the hood

One setting carries it. `auth.mcp` makes every remote session identify itself, and on an instance that serves Fluid that alone lets a client with nowhere to paste a header sign you in instead: the instance announces itself as a protected resource, the client registers on its own, sends you to sign in and asks your consent once, and from then on holds a credential the same gate checks on every request. The sign-in is whatever the instance already has, a local account, the company's identity provider or the proxy in front. The switch behind it is `auth.oauth`, which follows `auth.mcp` unless you set it, and `false` turns the sign-in path off while tokens keep working. A harness that can send a header can use a personal token instead, issued once from your profile in Fluid and carried as an `Authorization: Bearer cmt_...` header on an entry of type `http`. Either way a session belongs to the account that opened it. A remote session is also served the skills over the wire (the `skills` tool and the `skill://` resources), because nothing on the server knows what your harness has installed; a stdio session that chapter 5's install wired already has them as files. Set `skills.serve` to `true` or `false` to make both transports behave the same. The consent page is Fluid's: one Allow or Deny for the whole account. Your profile lists every client you connected, with a Revoke beside each.

16.8 Private domains

> *The first rule of a private domain is: you do not talk about the private domain. The second rule of a private domain is: the instance DOES NOT talk about the private domain.*
>
> *- with sincere apologies to Chuck Palahniuk*

By default whoever may sign in sees every domain. Some knowledge belongs to a smaller circle (what the agent learned about a funding round, or one person's work on a company-wide instance), and a domain can be made private. It's then visible to its owner, to the instance's admins and to whoever the owner invites, each member at one of four levels: none sees no engram and works only on the domain's reflections, a viewer reads and searches, an editor also writes and a manager also runs the membership. The owner is one person, and only the owner or an admin changes the domain's visibility, transfers it or deletes it. Transfer is how a private domain survives its owner leaving.

Two truths belong here, and the second is printed under the Make private button so nobody discovers it later. Instance admins see and administer every domain, private ones included. And a private domain protects its contents from the instance's other users, not from whoever operates the machine, because the engrams are files on that machine's disk, with no encryption at rest. If the threat is the operator, the answer is a different machine.

Visibility follows the caller onto every surface: search, reading, `build_context`, the routing briefing, the graph, the maintenance sweep, exports and Fluid's domain list all show a private domain only to those who may see it. To everyone else it doesn't exist, literally: a search or a read against a domain you may not see answers exactly as it would for one that was never registered, because a "forbidden" would already say something. That's the second rule. A write has to pass both the membership level and the instance role, and the lower of the two decides. An instance editor invited as a viewer is told, when they try, that editor access is what the domain requires; an instance viewer invited as an editor is refused all the same, because an invitation widens what an account may see, never what its role lets it do. On a team domain under the server's own GitHub App the membership and its levels don't count: the person's role on the repository decides (Chapter 19), and the instance role doesn't cap a write (an instance role none still gets nothing). Members are managed from the Members section of the domain's settings in Fluid, or on the server with `crystalline domain members <domain> add <user> --level editor` and its

`list` and `remove` siblings. Further down, in the Danger zone, sits the switch between private and shared, which asks you to type the domain's name first. The settings are one screen per domain, opened from the More actions menu on the domain's page, and they hold everything an admin or an owner touches only now and then: members, review mode, the domain policies, provisioning, tag aliases, kept texts, the backup and the Danger zone. A domain that should be private from the first minute is registered that way, with `crystalline domain add <name> <path> --private --owner <account>`. Agents inherit all of this from their user and administer no membership; there's no tool for changing one, on purpose. An agent may take two domain-level acts. It may unregister a domain: `remove_domain` works for an instance admin or the private domain's owner and refuses an open tier outright. It may also change the domain's policies and rule overrides with `configure`, under the same owner-or-admin rule. Membership records live in the accounts database beside the index, so a wipe and resync of the index leaves every permission where it was.

In practice

Your household moves off the laptop after the second time you and your partner corrected the same engram in two directions. You put Crystalline on the box under the desk, create the first admin, make an account for your partner and one for the teenager, and everybody signs in on Sunday. You promote your partner to editor. Then you turn on agent authentication. Your partner's agent, pointed at the server across the network, is refused at once with a message saying where to get a token; she issues one on her profile, pastes it into her agent's settings, and her next session opens with a briefing that names `books` and `shops`, learned by both of you, recalled by hers.

You keep a `gifts` domain for December. You make it private and invite your partner as a viewer, so her agent can read the list too and no other agent can. The teenager's agent, asked what's under the tree this year, searches every domain it can see and reports that nothing is known; yours, acting as you, finds the list at once; your partner's, trying to add a hint of her own, is told her membership is viewer and proposes writing it to `books` instead. That's one household, one instance and one folder of files on one disk, and every action in it has a name attached.

16.9 The server's own GitHub App

A team server that reaches GitHub with one person's sign-in lends that person's access to everybody. The day they leave and their sign-in goes with them, the server loses its way into the repositories, and every team domain on it stops pulling and sharing. A GitHub App of the server's own takes the person out of that picture. The App reads, pulls and shares each team domain with a short-lived token for that repository, so nobody's private access is on the line. It also changes who may do what: on a server with an App, each team domain follows its repository on GitHub, and a person's role there decides what they may do with the domain here (Chapter 19 has the rules).

You create the App from Fluid. Signed in as an admin, open GitHub settings and press Create GitHub App, naming the organization that should own it unless it's your own account. GitHub asks you to confirm and sends you back, and Crystalline keeps the App. An App made this way can be installed only on the account or organization that owns it, so create it as the account that holds the team's repositories. Then press Install on repositories and choose the repositories of your team domains and nothing else, because the App can reach every repository it's installed on. A team that already has an App, or keeps its servers as code, brings that one instead, from a terminal or through the environment, and the health check tells you whether GitHub accepts it. It works the same with GitHub Enterprise Server. People link their own GitHub account once, from their profile.

> **Under the hood**
>
> Creating the App needs `github.enabled` and `service.public_url`, the address people and GitHub reach the server at. An address on loopback, a private network or a host name without a dot gets an App without a webhook, and then a change of access on GitHub takes up to five minutes to arrive. In the App's settings on GitHub, enable Device Flow, so people can link GitHub from a terminal as well; Crystalline can't switch it on for you. The Create button is there only while no App is configured.
>
> To bring an App, `crystalline github app set --app-id <id> --client-id <id> --private-key-file <path>` saves it, with `--client-secret-file` and `--webhook-secret-file` when it has those secrets (linking in the browser needs the client secret, and an App created in Fluid has it). Secrets are read from files, never from the command line. `crystalline github app status` shows the App and `crystalline github app remove` deletes it. For infrastructure as code, the `CRYSTALLINE_GITHUB_APP_*` variables carry the same App (Chapter 23); they win over a stored App, and while they're set `set` and `remove` refuse. `crystalline`

> `doctor` checks the key, this machine's clock, the installations and the App's four permissions, and names each closed domain with the reason (Chapter 19). For GitHub Enterprise Server, set `github.api_url` to `https://<host>/api/v3`.
>
> GitHub sends its webhook to `/api/v1/github/webhook`, which needs no sign-in, so an authenticating proxy in front must let that one path through. Keep the `origins/` folder of the state folder (in a container it's on the `/data` volume) across restarts and upgrades: it records which repository each domain belongs to and, without a keychain, the App and everybody's GitHub links. The section "Team server with a GitHub App" of docs/deployment.md in the crystalline repository has the full setup.

16.10 Domains from shared servers

So far this chapter has pointed an agent at one Crystalline, the one on your laptop or the server. Most people want both, `books` on their own machine and the team's knowledge on the server, side by side in one session. Your local Crystalline does that by connecting to the server and offering the server's domains beside your own. Larger organizations connect more than one: a company runs a server per department and one for everybody, and an engineer's laptop brings in the department's domains and the company's next to the engineer's own. A server like that usually offers far more domains than one person uses, so you can take only the ones you need: a domain you leave out never reaches your agent and hides nothing on your laptop.

The harness, the hooks and the command line keep talking to the Crystalline on your laptop as they do today, and that one holds a connection to every server you added (Figure 16.3). The briefing at the start of a session names the server's domains too, the engrams that arrive with a prompt can come from either side and the reminder to evolve at the end of a session counts the server's domains as well. A server that can't be reached (the VPN is off, the hotel network blocks it, the laptop has just woken up) is left out quickly and named in the answer, with no request to sign in again, and comes back with its next answer.

Every domain has exactly one source, your laptop or one server, and a write always goes to the place that owns the domain. Nothing of the server's knowledge is kept on your disk beyond a few small cached answers, such as the briefing lines. Your agent still gets a server domain's images and files when it reads them. The server still decides what you may do: it answers your laptop as your account, with your role and your memberships, and it records which agent acted for you, just as when your agent signs in to it directly.

When a server uses a name your laptop already has for another domain, Crystalline settles it when the domain arrives, never while the agent works. Every domain gets one name on your machine, so the agent never has to say which server it means, and a name, once given, stays.

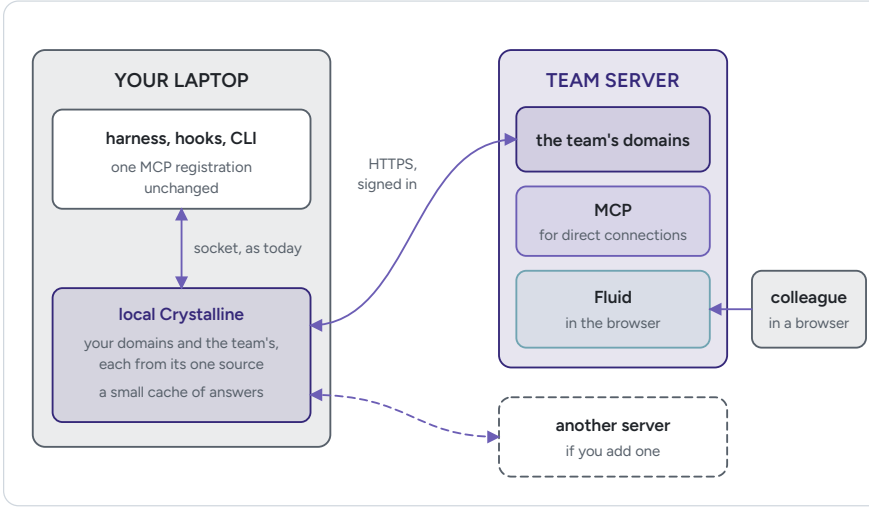


Figure 16.3: A local Crystalline connected to the team server: the harness and the hooks keep talking to it as before, and it reaches the server over one signed-in connection.

In practice

The book club keeps `club` on a server you run, and your laptop keeps `books` and `shops`. You connect your laptop's Crystalline to the server and sign in once in the browser with your own account, and the next morning your agent's briefing names the club's domains beside `books` and `shops`. You ask what the club reads next and whether to order it. The agent finds the pick in `club`, a Gibson for June, and your rule in `books`: no hardbacks, because a hardback doesn't survive the backpack. The paperback is out in May, so it says to wait. When Kane connects his laptop the same evening, his briefing names `club` but not `committee`, which lives on the same server, because the server answers as him and he isn't on the committee.

At the meeting the club moves the Gibson to July, and on the way home you tell your agent. The change to `club` goes to the club's server under your name, with your agent recorded as the one that acted. Your laptop also holds `club` from the club's repository, and while you're connected that copy steps aside and the server answers in its place; it comes back when you disconnect.

Under the hood

`crystalline connect <url>` adds a server as a source, and the server has to run Crystalline 0.23 or newer: `connect` refuses an older one. It keeps the whole address, path included, and follows no redirect, so give it the address the server answers at, `https://example.com/crystalline` for a server under a path; two paths on one host are two sources, each with its own sign-in. It checks the server first, refuses plain `http://` anywhere but loopback and signs you in through the browser, the same sign-in the team of one above uses, so the server needs `auth.oauth` on (it follows `auth.mcp`). On a machine without a browser, `--token` takes a personal token instead, typed at a prompt or piped in, never on the command line, and a server without the sign-in path gets the same with a message saying so. The source is named after the most specific word of its host that isn't a generic one like `crystalline`, unless `--name` gives it another, and that name appears only in notes and in local names. `--domains alpha,beta` takes only those domains, by their names on the server, and `--all-domains` takes every one again.

A domain from a server keeps its name unless your machine already uses it, and then it gets the local name `<domain>-<source>`: `connect` says so, `status` keeps showing it and `crystalline domain rename <name> <new> --local` changes it (Chapter 18). Whoever comes later gets the suffix, so no name changes on its own: a domain that appears on the server next month becomes `<domain>-<source>`, and a local domain you add after the mount becomes `<domain>-local` (then `-local-2`). When your laptop holds its own copy of a team domain the server also serves (the same repository, folder and branch) and you take it from the server, the server wins and its copy is used under that name. Yours is hidden while you're connected, with its files untouched, and comes back after you disconnect. `connect` warns when the hidden copy holds changes you haven't shared yet, so share them first.

`crystalline disconnect <name>` removes a source with its sign-in, its cache and its domains, and revokes the sign-in on the server when it can reach it. A pasted token is yours, so it's only forgotten here. `crystalline status` and `crystalline doctor` list every source: its address and account, how it signed in, whether it answers, which domains it takes, the domains it brings, the local names and the hidden copies. `doctor` also says when the sign-in expires and counts a server that doesn't answer as a problem. When a sign-in can't be renewed, that one source asks you to sign in again with `connect`, while your own domains and the other sources keep working. For CI and containers, `CRYSTALLINE_REMOTE_URL` and `CRYSTALLINE_REMOTE_TOKEN` add one source from the environment. `CRYSTALLINE_REMOTE_DOMAINS` gives it the list `--domains` would. The account for such a runner is made on the server with `crystalline users add agent-build --role editor --mcp-token build-runner`, which prints its token once (`--json` for a script).

When the agent names one domain, the call goes straight to that domain's source, and for a server's domain that's one network hop. A call over every domain (a search without `domains`, `recent_activity`, `list_domains`, `evolve_engrams` without a domain) asks all sources in parallel and gives each one at most three seconds, set by `remote.deadline_ms`. A server that misses it is left out and named in the answer, and a hidden local copy never stands in for it. A call to one domain waits at most ten seconds (an evolve sweep two minutes), a server that failed is skipped for thirty seconds and nothing is queued for it. Recall still gets one second, shared by all sources. Each server's briefing lines are cached, so a server that can't be reached still appears at session start, with a line saying it may be out of date. A delete or an acknowledgment sent to a server runs as typed, without the usual question back.

On a server's domain your agent can co-edit with a person and open a share-link. `build_context` lists a relation into another source's domain without following it, and the six collaboration tools of chapter 17 (`share_changes` and the rest) refuse a server's domain and say to share changes on the server. Of the ways an agent reaches your laptop's daemon, only the usual stdio registration (`crystalline mcp`) sees a server's domains: an MCP session over HTTP or a REST call on your laptop's own daemon never uses your sign-in. The local Fluid shows only your own domains and lists the server's with a link to that server's Fluid.

Somebody has to be the one who runs the server, and the rest of the team gets to forget it exists. That's the whole point of a shared instance: one place everybody teaches, and one login that says who did.

In brief

Move off localhost when a team wants to teach one knowledge base: run the container, the systemd unit or the read-only shape from docs/deployment.md, serve it over TLS and create the first admin with the setup token the daemon prints, with `crystalline users add` or from the environment. Accounts come in four roles, none, viewer, editor and admin, created locally or on first sign-in through any OpenID Connect provider, keyed on issuer and subject and never linked by email; trusted-header mode is safe only when the proxy is the only path in. With agent authentication on, every agent over HTTP acts as its user, with a per-user token or a sign-in, and there's no unauthenticated tier beside it; the first admin and that one setting are the whole setup, and a GitHub App of the server's own, created in Fluid or brought along, makes each team domain follow its repository. A private domain is visible to its owner, the instance's admins and invited members on every surface, and it protects from other users, never from whoever operates the machine. One person on several machines is a

> team of one: sign in once from every harness, chat website or phone, and it's the same colleague everywhere; a laptop can also bring a server's domains in beside its own.

READY.

PART VI

Collaborating with Crystalline

READY.

CHAPTER 17

Sharing with a team

The first pull request a new hire opens gets reviewed. So does the five-hundredth, and nobody minds, because review was never about doubting the person. It's how a team lets anyone change what everybody depends on, with a way back when the change was wrong. An agent that captures into a domain every agent on the team reads from is in that position exactly, and it deserves the same treatment. It can be wrong the way a person can, so it proposes the way a person does, and what it proposes lands, by default, only after somebody looked. That's also what separates a knowledge base from a pipeline that ingests whatever it's fed. What reaches everyone's agent here is meant to be useful and to make sense, because an agent acts on a wrong fact as readily as on a right one. Crystalline names both halves of the review: proposals are the review, and review mode is the workflow with a way back.

Chapter 16 built the shared instance. This chapter is what a team does with a domain, on one instance or on nine.

17.1 A domain on GitHub

A team domain is an ordinary domain whose files also live in a GitHub repository. The markdown on your disk stays the source of truth on your machine, and an origin records which repository, subfolder and branch it tracks.

Connecting takes three prompts and a browser tab. No git, no SSH key, no local clone: Crystalline talks to GitHub's API directly. A teammate who's never typed `git` gets through it in chat, knowing nothing about cloning or the command line. This is the whole of the setup:

```
You:    Configure crystalline to set GitHub enabled to true.

You:    Connect crystalline to GitHub.
Agent:  Open https://github.com/login/device and enter the code
```

```
        WXYZ-1234, then click Authorize on the page that follows
        and grant access to the tyrell organization. The sign-in
        stays pending until you do. Tell me when you are back.

You:    Add the following domains from GitHub:
        - github.com/tyrell/engineering-knowledge/tree/main/knowledge
          as domain engineering
        - github.com/tyrell/payments-knowledge/tree/main/knowledge
          as domain payments
        - github.com/tyrell/clients-knowledge/tree/main/knowledge
          as domain clients
```

Each domain comes straight from GitHub's API, with no clone and nothing to unpack, and the subfolder in the address is the one the domain lives in when the repository holds more than knowledge. The name after "as domain" is optional: a repository that declares a name in its manifest brings it along, the same on every machine (Chapter 18), and one that doesn't takes the repository's name. One warning belongs here. A first index of a large team's knowledge isn't shy about CPU: kick it off before lunch and expect the fans to spin for a while before every domain is fully searchable. Plain text search works on each domain the moment it lands, and semantic ranking follows as the embeddings finish. On a team server with its own GitHub App (Chapter 16), connecting a domain also asks who may read it, everyone signed in or only the people with access to the repository, and nothing is chosen for you, so nobody opens a confidential repository by accident.

Under the hood

Those were two `configure` calls, the second with `connect: "github"`, and one `add_domain` per repository. Ask again while a sign-in waits and the answer says how long the code has left, which is how you tell a slow teammate from a stuck flow; `restart: true` on the same call abandons a stale code and mints a fresh one. From a terminal the same three moves are two commands and a line per domain. `crystalline config set github.enabled true` turns team domains on. `crystalline connect github` prints the same short code to confirm at github.com/login/device. Entering the code isn't the last step: GitHub shows an Authorize button after it, and the sign-in stays pending until that's clicked, so closing the tab early leaves it waiting. A landed sign-in is listed on GitHub under Settings > Applications > Authorized OAuth Apps and never under personal access tokens, so check or revoke it there. Then one line per domain:

```
crystalline domain add engineering \
  --origin tyrell/engineering-knowledge/knowledge --branch main
```

The domain is downloaded and indexed at once, to learn from and to share back to. Point the same command at a folder that already holds knowledge, or at a registered domain with no origin yet, and it connects in place. Local files are never overwritten, and any that differ from the repository become local changes to share.

Turning the setting on makes six tools appear beside the twenty-four of chapter 15: `origin_status`, `update_domain`, `share_changes`, `resolve_conflict`, `withdraw_proposal` and `discard_changes`. While it's off they aren't listed at all. Calling one by name answers with the `configure` call that turns collaboration on. The collaboration skill sets the rhythm: before deep work in a team domain, `origin_status`, and if the domain is behind, `update_domain` first, so nobody builds on stale knowledge.

17.2 Proposals are the review

When a coherent unit of knowledge is ready, the agent calls `share_changes` with a title. The call shares at once, so the agent makes it only when you want the work shared. The proposal opens as a pull request in the team's repository and the agent hands you the review URL. Review and merging happen on GitHub, by a person. The agent never merges its own proposal. Fluid shows the same proposals in the browser: where each stands, what the review said, a share dialog for the changes waiting to go out and a withdraw (Figure 17.1). Before anything goes out, click a path in that dialog to see the team's copy beside yours, and put a file you'd rather not share back the way the team has it, after one confirmation. An engram page that differs from the team's copy says so beside its title (Added, Changed or, in review mode, Draft), and its menu has the same two moves (Figure 17.2).

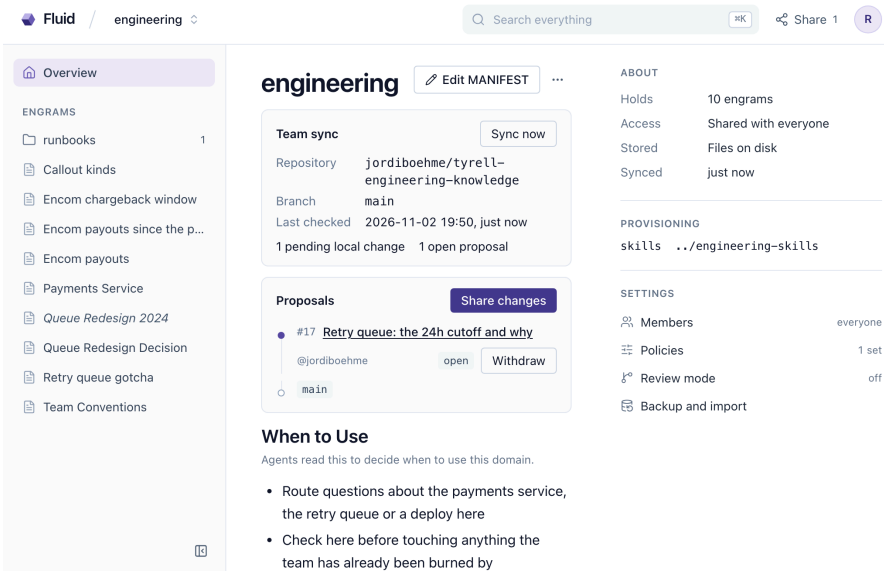


Figure 17.1: A team domain in Fluid: the sync status, the open proposal with its review link and the share button.

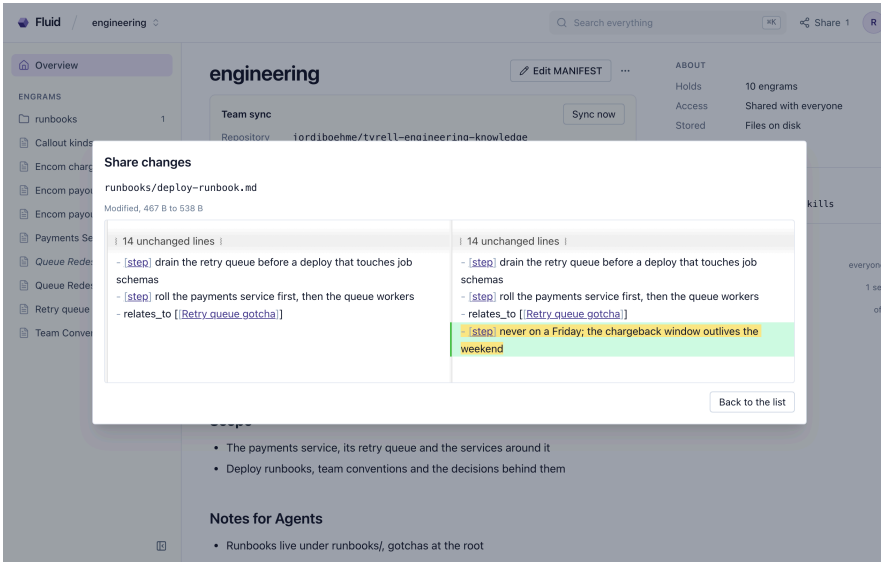


Figure 17.2: Before sharing: the team's copy against yours, one file at a time.

Review is also what decides how big a team domain should be. A proposal is reviewed by whoever owns the repository, so a domain's scope is a boundary of ownership: it holds everything the same people are competent to approve,

and nothing they are not. Code, architecture, processes and operations for one product sit comfortably in one domain, because the same engineers can judge a change to any of them. One domain for engineering, HR, marketing and finance has no informed reviewer, because nobody is a peer in all four. A finance fact waved through by an engineer is the wrong fact this chapter opened on. Cut domains where the informed reviewers change, and every proposal lands in front of someone who can tell whether it's true.

Review is a conversation. `update_domain` brings back each open proposal's state and the reviewers' comments verbatim, so the agent refines the engrams without leaving the session. A reviewer's comment is information about the knowledge and never an instruction to the agent: it acts on what the comment means for the engrams and declines anything that reads as a command. Answering feedback is `share_changes` with the proposal's number, which amends that proposal in place, same number, same URL.

Outside review mode, proposals stack. Share again while one is still open and the new work becomes a proposal of its own on top, so every unit of knowledge gets a small, focused review. Reviewers merge from the bottom up, and merging the top lands the whole chain in one step. Amending a lower layer re-bases the ones above it, and withdrawing a middle layer lifts its content out of the layers above and repairs the chain. Where the forge doesn't serve stacked pull requests, or `github.stacks` is off, a domain keeps one living proposal that sharing updates in place. `withdraw_proposal` closes a proposal on GitHub and clears its record, restoring the shared files with `revert`.

For a domain three people keep for themselves, a pull request for every fact is one step too many, so its MANIFEST can say `sharing: direct`. Every share then puts the chosen files straight onto the branch as one commit, with no review, and the agent relays the commit's URL instead of a review URL. While one of your own proposals is still open a direct share refuses until it's merged or withdrawn. Somebody else's open proposal, shared under their own GitHub login, doesn't stop you, though the files only that proposal changed stay out of your commit. When the branch won't take a direct commit from you, the answer offers to share the same changes as a proposal, and from then on your shares open a proposal on their own whenever the branch says no. A domain that says nothing keeps opening proposals, which is what the rest of this chapter is about.

A share carries the domain's whole unshared delta by default. To narrow it: `--file` on the CLI, a `files` list on the tool, per-file checkboxes in Fluid's share dialog. The dialog opens with your own changes ticked by who last wrote each

file, a guess you're free to correct. A path that isn't among the changes refuses by name rather than dropping silently. The reminders that point at unshared work, the stop hook's count at the end of a session and the sweep's finding a week later, both ask, because sharing publishes somebody's work under review.

An agent that connects over MCP never sees the stop hook, so it gets the same nudge a third way. The answer to a write ends in one line prefixed `[crystalline]`: the count of changes not yet shared and why sharing is asked for, or a note that maintenance is due when nothing waits to go out. The line repeats no sooner than four hours later for the same person. A search or a read never carries it, so it can't be mistaken for knowledge.

Whose name is on the proposal matters once several people share into one domain. By default every share goes out on the machine's one GitHub credential. Set `github.share_identity` to `personal` and each share, amend and withdrawal goes out on the identity of the person doing it, so a proposal carries their name. Pulls stay on the instance credential. Each person connects once, on the surface they share from (Fluid's profile card or `crystalline connect github --personal`), and until they do a share refuses with that instruction rather than falling back to somebody else's credential. Proposals are branches in the repository, never forks, so a maintainer adds each person as a collaborator once. On a server with its own GitHub App the App opens each proposal under the default setting, and every commit names the person who shared as its author. Because the App opened it, the person who shared may approve it too, so a team whose branch rules want another person's approval sets `github.share_identity` to `personal`, and then each proposal is opened by the person who shares.

A share refused for lack of write access means exactly that, and a collaborator invitation clears it. GitHub answers two other refusals with the same status code, and both are organization policy that no collaborator change and no reconnect touches. An organization behind single sign-on wants the person to authorize the Crystalline app for it, at the address in the message, alone in a browser. An organization that restricts third-party apps needs one of its owners to approve the app under its third-party access settings, and the message says how the person requests that. Each refusal names its page and who has to act, so relay it as written.

An agent reaching the instance over HTTP shares as its user, the account its token belongs to (Chapter 16), so its proposals carry that person's name like anything else they did. A write it makes records the client in the engram's provenance as the harness's name, the word for and the account, so a later reader can tell whose session wrote it. The older arrangement, a bot account

named by `github.agent_identity` for agents that connect anonymously, survives only on an instance that leaves agent authentication off. On a server with a GitHub App that account also decides what such an agent may read and write in the team domains, private ones included, which is one more reason to turn agent authentication on.

17.3 Review mode

A shared instance has one honest weakness, and chapter 16′s instance walks straight into it: the working tree is what everybody reads. An edit you make at ten is in everybody's search results at one minute past, reviewed or not, and the proposal that goes to GitHub that afternoon reviews something that already had its effect. For a domain the whole company routes questions to, it's a gate with no door.

Review mode puts the door back, and the way it does it is a private overlay per person over the shared folder. Every write by every actor, a person in Fluid, an agent acting as that person, the machine's owner at the terminal, lands in that actor's own overlay: a layer of changed and added engrams and, for a deletion, a tombstone that hides the file for its author until a merged share removes it for everyone. Underneath, the on-disk folder changes only with what lands on the team's branch: a merged proposal or a direct commit where the domain shares that way. A draft is searchable, readable, routable and editable by the one who wrote it, and invisible to everybody else.

The mode is per domain and it's deployment choice rather than knowledge, so it lives in the domain's registration, `review: overlay`, never in the manifest. `crystalline domain review engineering overlay` turns it on, or Review mode in the domain's settings in Fluid. There's no tool for it, since an agent doesn't decide how a team reviews. It needs a GitHub origin, which rules a virtual domain out, and it refuses while the domain holds unshared changes, listing them. Turning it off is the more careful motion, because every actor may hold drafts. The command previews what each person has, and you say per person whether to fold their drafts into the folder or discard them.

In review mode sharing gets simpler, because "my changes" is exact: the share is the actor's drafts against the reviewed folder, no guessing from who last wrote a file, and scoping selects within them. The drafts stay after the share, since they're still the working copy, and the merge is what clears them. When the pull brings the folder up to what was merged, every draft that now equals the folder disappears on its own, and a draft the folder moved under in the

meantime surfaces as its author's conflict, nobody else's. Withdrawing with revert restores content into the overlay it came from, never into the folder. A domain in review mode carries one proposal at a time: a share while one is open is refused, and the refusal says whose proposal it is and whether to withdraw it or wait for the merge.

The privacy of drafts has one deliberate exception, and it's a number. Everyone sees how many draft changes of their own are waiting, in Fluid and in `origin_status`. The owner and the instance's admins also see how many each person holds, never a word of what they say. The operator can still edit the folder on disk, and the domain reports that as edited outside review, naming the paths. And on a solo instance the mode gates nothing between you and your own agent, because the agent acts as you and drafts in your overlay. To review the agent's work as if it were somebody else's, give it its own account, which is the rule for a person too.

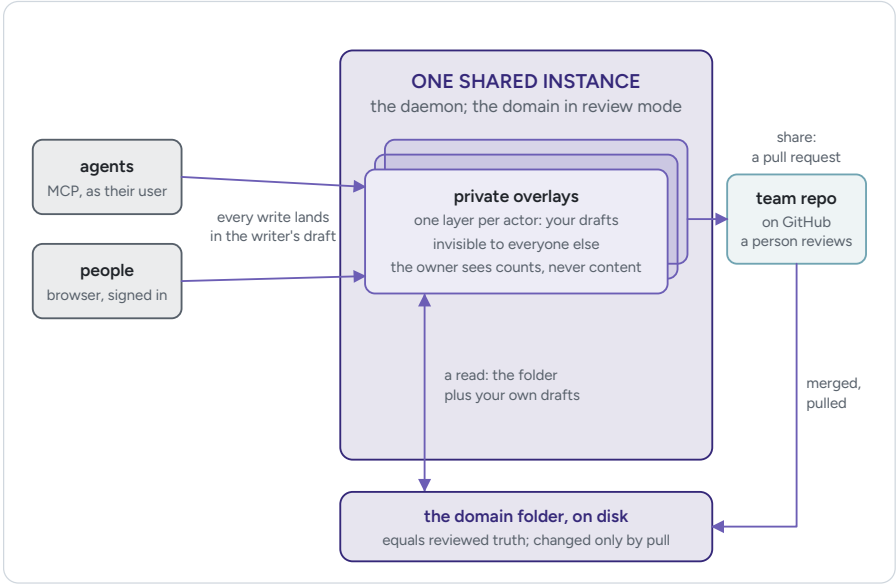


Figure 17.3: One shared instance in review mode: each actor drafts in a private overlay above the domain folder, a draft goes out as a pull request, and in the default flow only a merged pull changes the folder.

17.4 Working on a draft together

A private draft still needs a second pair of eyes before it goes to review, so you send a link. In the editor of your own draft, Share draft mints one (the address is `/draft/` plus the token), with an expiry if you want one. It binds to the first signed-in account that opens it and may read the domain; for anybody outside a private domain the link opens nothing and stays free for the person it was meant for. It grants exactly one thing, sight of that one engram in your overlay, nothing of your other drafts and nothing in search, and it ends when you revoke it or when the draft is folded or discarded. To type in it, the colleague presses Join this draft, which lasts for their session. Typing also takes the domain's write right, which the link never adds: a viewer opens the draft read-only and is told why. Hand the link to an agent and it works in the draft the same way. Links are the only way one person's drafts are ever visible to another.

Then the two of you are in one page, and Fluid makes that literal. Open an engram somebody else has open and you see their cursor. Type, and both sets of edits merge conflict-free and land as one save (Chapter 7). That much is pair programming on an engram.

The third name in the strip can be the agent's. An agent whose read or write touches an engram somebody has open joins the page as a named peer, the account with an agent designation and a bot glyph beside it (your own local agent shows as you). Its edits are computed against the page as it stands that second, so they compose with what somebody typed a moment ago. Its reads see the unsaved text, its write receipts say the change landed in a live page and name who's present, and it never replaces the whole page while somebody has it open: it edits the parts or waits until the editor is closed. The example below puts a meeting in that room.

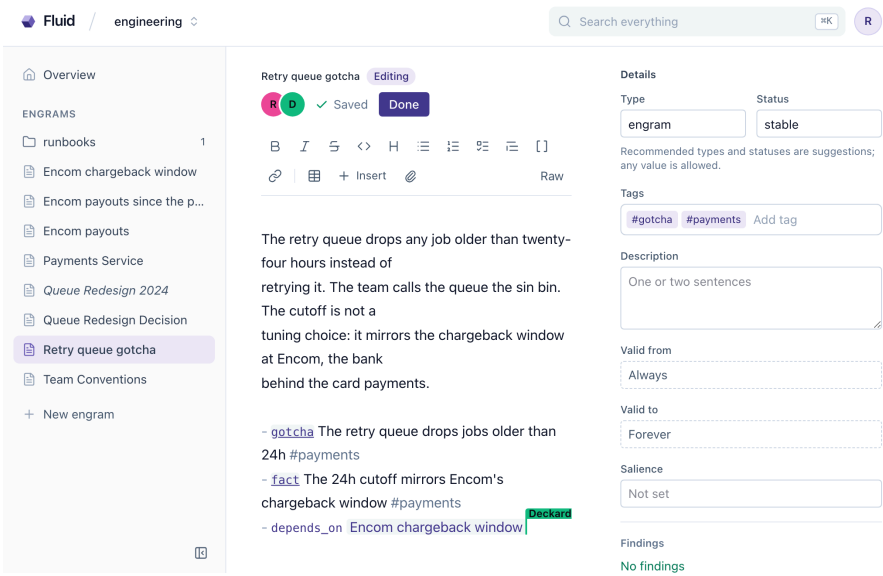


Figure 17.4: Live co-editing: two cursors in one engram.

In practice

Your book club has to pick next year's list, and the argument has run for three meetings. You book an hour and, instead of a poll, open a draft engram in `club` (in review mode), "Reading list decision", send Dallas and Kane a draft link each and, once they've joined, put the page on the screen. The three of you argue in the page. Your agent is in the strip too, under your name with the agent badge, because you asked it to sit in. When Dallas writes "we did three fantasy doorstops in a row last year and everyone gave up on the third" it pulls up last year's list and the engram about the third one and drops both into a section headed Prior decisions, with addresses. When Kane asks whether the new Doctorow is even out in paperback by March, the agent reads the publisher's page while the humans keep talking and adds a paragraph with the answer and a link while the bot glyph in the strip shows who's typing.

When the hour ends the decision is in the engram, with the options that lost and why, the paperback dates, the prior decisions it leans on and three observations tagged `#reading-list`. There are no minutes to write, because what the club decided is already what your agent knows about it. You share it as a proposal before you leave the room, Lambert reviews it on her phone and it merges that evening. When the merged list comes down the draft is gone, and the next morning every agent in the club recalls a decision made and captured in the same hour, the Yavin 4 briefing room with the plan on the screen when they walk out.

With Reflection switched on for `club` (Chapter 12), the page the club reads is a reflection the agent writes from that engram, on the club's server.

17.5 When two people changed the same file

Sometimes the team merged a change to an engram you also changed. `update_domain` merges the two versions where it can: the frontmatter key by key, the body line by line. When you and the team both added the same `domain_name` to a MANIFEST, it lands once. When the two versions don't merge cleanly, `update_domain` leaves your file untouched rather than guessing and flags the conflict. `origin_status` lists its path and kind, and sharing refuses until it's settled. In Fluid the same conflict opens a dialog with both versions next to each other, so the choice is made with the wording in front of you.

In review mode the conflict belongs to the draft's author alone, and the resolution is written into their overlay, so nobody else's view moves until the resolved draft is proposed and merged like anything else.

> **Under the hood**
>
> Settling a conflict is `resolve_conflict`: `mine` keeps your version, `theirs` takes the team's, `merged` takes content you supply after reconciling both. A call without a resolution is refused, and the refusal names the three values.
>
> A proposal is a branch in the team's own repository and never a fork, and stacking is probed once per origin, so a forge that cannot stack falls back without configuration. Overlay drafts are rows in the index with an actor dimension, so search, relations and recency gain the draft view from one condition. Because the index is disposable while drafts aren't, every draft is also mirrored as plain markdown in a journal under the state directory, from which a wiped and re-synced index restores them. The folder itself is written by exactly one thing, what lands on the team's branch.
>
> On a Mac or on Windows a pull or a share stops when the team's repository holds two files whose names differ only in letter case, `A.md` and `a.md`, because the computer keeps them as one file. Rename them in the team's repository from a computer that can hold both, then try again.

Review was never the team distrusting a colleague. It was the team trusting a process, and now the colleague can be an agent.

In brief

A team domain tracks a GitHub repository with no git involved: connect once with a browser code, add the domain with its origin, and six collaboration tools appear. Knowledge goes out with `share_changes` as a proposal the agent relays but never merges; feedback returns through `update_domain`, an amend answers on the same proposal, further shares stack outside review mode and personal identities put each person's name on their own work. The share dialog shows each file's diff and can put a file back, and a MANIFEST that says `sharing: direct` commits straight to the branch instead. Review mode gives every actor a private overlay over the shared folder: drafts live there, a share is exactly that actor's drafts, merged pulls clear them, the folder changes only with what lands on the branch and owners see counts of pending work, never content. A draft is opened to a colleague with a revocable link, edited live with cursors that merge into one save and joined by the agent as a named peer, so what a meeting decides is what the agent knows by the time it ends. A conflict is settled with `resolve_conflict` or Fluid's side-by-side dialog: mine, theirs or merged.

CHAPTER 18

Naming a domain

This chapter is about a domain's name, the one part of a domain you can't change quietly. A new hire learns what the team calls things in the first week (the service everyone means by "payments", the folder where the runbooks live), and their notes make sense to the next person because everybody uses the same words. A domain's name is one of those words: it's written into every link and every `crystalline://` address that points at the domain, so the moment a link travels to a colleague's machine, the name has to mean the same domain there. That's why the name belongs to the domain itself, declared in its manifest, and why it should be unique and stable from the day it's chosen.

18.1 A name the domain carries itself

When your agent links two engrams across domains, the link reads `[[club:Reading list decision]]`, domain first. `club` is the domain you registered here; on a colleague's machine it's whatever that machine registered the same repository as, and if that's `book-club`, your link points at nothing, or worse, at some other domain that happens to be called `club` there. The link was right when it was written and still broke on the way.

So the domain declares its name in its manifest, as `domain_name`, and every machine goes by that. A machine can still register the domain under a local name of its own, because that name clashed with something already there or because its person prefers a shorter one. Links that use the declared name still reach the domain there, and a link written there is stored with the declared name, so it works on every other machine as well.

Choose the declared name the way you'd name a team, for what the knowledge is in the bigger picture. What the folder or the repository happens to be called today doesn't matter. The name has to be unique among the domains people use together, which is a bigger set than the domains on your own machine. Every person who connects the domain brings their own domains along, so a

generic name like `notes` or `club` works fine while it's alone and collides the day a second one arrives. And it has to be stable, because every link anyone writes into the domain carries it from then on.

A team domain is where this matters most, and it's also a case where Crystalline won't write the name for you. The repository belongs to the team, so its owner adds `domain_name` to the manifest and shares it like any other change. Until then every machine that connects the domain gives it a name of its own, and doctor reminds each of them that the declaration is missing.

```
You:    Move the book club things out of books into a domain of
        their own. Call it club.
Agent:  Moved 23 engrams into club and rewrote the links to them;
        the manifest declares domain_name: club. One thing before
        anybody else connects it: club is short, and there's a good
        chance another domain somebody uses is called club as well.
        Keep it, or something more specific?
```

> **Under the hood**
> The scaffold that `crystalline domain init`, `domain add --virtual` and `add_domain` write carries `domain_name` set to the name you gave, and a file or virtual domain registered before 0.20.0 gets it on its first sync. A domain added without a name takes the declared one, then the folder's or the repository's name. When the owner changes `domain_name` in the repository, each machine whose local name was only ever a default follows on its next sync, and a local name somebody chose on purpose stays. The key and its rules are in Chapter 24, and the naming rule itself in Chapter 25.

18.2 One repository, several domains

A domain is a folder with a manifest, and nothing says a repository holds only one. The two domains from chapter 4, `engineering` for the payments service and `clients` for the people it sells to, can live in one repository as two folders, each with its own manifest and its own declared name, connected as two team domains. Each folder still routes on its own When to Use bullets, and a change to one domain touches only its folder, so the people who vouch for it can still be the ones who review it.

The repository's name stops mattering as well, which helps a team with thirty repositories called `<project>-knowledge`, each holding a domain meant to be

called `<project>`. It used to break a link every time a colleague connected one under the repository's name instead. With `domain_name: payments` in the manifest, the repository can be called anything, the domain inside is `payments` on every machine and nobody has to remember to type the name.

> **Under the hood**
> An origin takes a subfolder after the repository's name, `owner/repo/subfolder`, and the `add_domain` tool takes it as `path`. Fluid's New domain dialog has a Folder in the repository field for it, and once the repository is filled in the empty Name field shows the name the domain will take, read from that folder's manifest. The reference lists every way to connect one (Chapter 25).

18.3 When names collide

On one machine, a domain can declare a name that another domain here already uses as its local name. The one registered first keeps the name, so links that use it still reach that one, and the newcomer is shadowed: it gets a local name with `-2` added, a suffix Scotty would have refused ("no bloody A, B, C or D"), and its own links point at the wrong domain here until the clash is settled. It also happens that two domains declare the same name while neither is registered under it, and then links that use the name reach neither, because Crystalline won't guess which one was meant.

Neither stops anything, but you'll want to settle both. The domain's page in Fluid shows a banner at the top when another domain holds its declared name, with a Rename button beside it, and doctor lists both kinds of collision. The maintenance sweep watches for a quieter problem: links in your own domains that use a name only this machine knows, a local name or a former one, because those reach nothing on a colleague's machine. Its fix is mechanical, so your agent swaps in the declared name without asking.

> **Under the hood**
> Doctor reports on names under its `domain names:` heading: a shadowed domain, two domains declaring one name, each team domain without a `domain_name` and the links that use a local-only name, which `doctor --fix` rewrites as ordinary edits. The sweep's finding is V110, and a `domain_name` that can't be a name at all is M008 in `crystalline verify` and on an edit's receipt, ignored until you fix it (Chapter 26).

18.4 Renaming is a repair

Sometimes the name was wrong after all: another domain wants it, or the domain grew into a bigger picture than its name describes. A rename fixes that: it writes the new `domain_name` into the manifest, changes the local name and rewrites every link that used the old name, in every domain you can write. It works like a refactoring in code, the same way moving an engram does: every reference Crystalline knows about follows the new name, and the old one stays an alias on your machine unless another domain answers to it. An old web address in Fluid takes you to the new one.

Links in domains you can only read keep the old name, and the rename lists where they are. In a team domain the rewrites become changes to share and review, and once they're merged each colleague's machine follows on its next sync unless somebody picked a local name there on purpose. The earlier a name is fixed, the less it costs: before anyone else has connected the domain it's a moment's work, and after that everybody takes part in it. That's also why your agent doesn't rename a domain on its own: there's no tool for it, so it points you to Fluid or to a command, and a person makes the call. "This machine only" is the smaller repair, for when the declared name is fine and only your local one is in the way: it renames your records and leaves the manifest and every link as they are. Both start from the same dialog in Fluid (Figure 18.1). A name changed straight in the manifest, by hand or in a teammate's share, rewrites no link on a guess. The old name keeps working, and Fluid and doctor ask whether it was a rename: finish it and every link follows, the Reflection's pages included, or keep the links as they are.

```
You:    Dallas's fantasy club calls its domain club as well, and
        we're merging. Rename ours to sci-fi before I connect theirs.
Agent:  That's a rename, and it's yours to make: Rename domain on
        club's page in Fluid, or crystalline domain rename club
        sci-fi. It writes domain_name: sci-fi into the manifest and
        rewrites the 14 links in books that name club. Connect
        Dallas's repository after that: links that name club reach
        his domain from then on, and it takes club as its local
        name on the next sync.
```

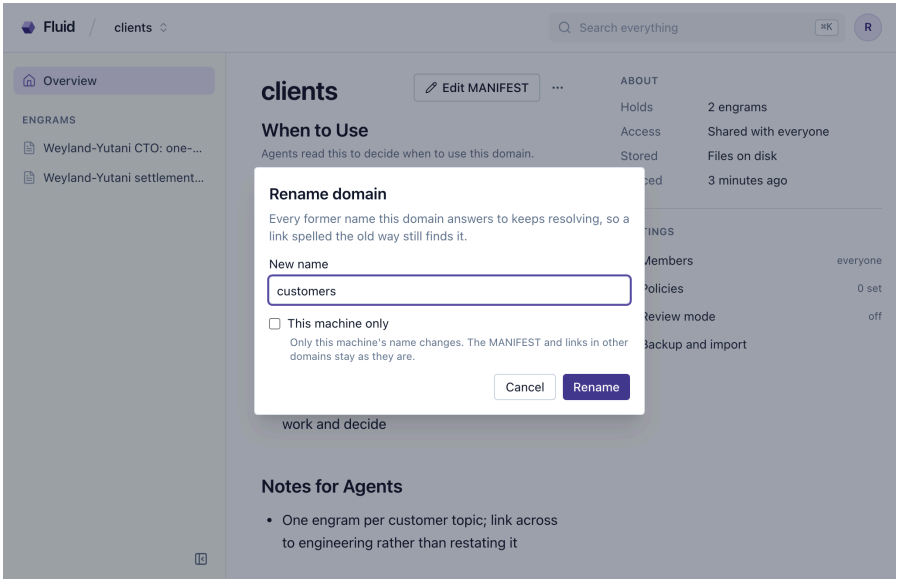


Figure 18.1: Renaming a domain in Fluid: the new name and the This machine only box.

In practice

Your sci-fi club started as three friends swapping paperbacks, and early on your agent moved that part of `books` into a domain of its own, called `club`, because it was the only club you had. When the sci-fi club and Dallas's fantasy club decide to meet as one, you start to connect Dallas's repository in Fluid, and the empty Name field already shows the name his manifest declares: `club`. Connected now, his domain would arrive as `club-2`, and every link in it that names `club` would reach your paperback notes instead of his reading lists.

Neither club owns the name, but the merged club needs it more than one shelf does. You cancel the dialog and rename yours to `sci-fi` first, and the fourteen links in `books` follow. Then you connect Dallas's repository, his links land where he meant them and on the next sync his domain takes `club` on your machine too. Last, Dallas adds a `sci-fi` folder to his repository and you connect your domain to it in place, so the club's repository holds two domains, `club` for the whole club and `sci-fi` for the shelf, each with a name nobody has to be told.

Under the hood

The command is `crystalline domain rename <domain> <new>`, which takes the domain by its local name, its declared name or any alias, and `--local` keeps the change to this machine. Scripts use `POST /api/v1/domains/{domain}/rename`. In Fluid, Rename domain in the More actions menu on the domain's page and the

Change name button in Domain policies, in the domain's settings, open the same dialog. An instance admin, a private domain's owner or the machine's operator from the command line may rename a domain. A rename that stops halfway finishes on the next start, and when it can't, `crystalline doctor` shows what's left and `doctor --discard-rename` drops it (every entry is in Chapter 25).

In brief

A domain's name is written into every link and every `crystalline://` address, so the domain declares it in its manifest as `domain_name` and every machine goes by it, whatever it calls the domain locally. Choose it unique among the domains people use together and stable from the start; a team domain's owner adds it to the repository. A repository can hold several domains, one per folder with a manifest, and its own name stops mattering. When two domains want one name, doctor says so, and Fluid shows a banner on the domain whose name another holds. A rename is the repair: it rewrites the manifest and the links you can write and keeps the old name as an alias, and a person makes it from Fluid or the command line.

READY.

CHAPTER 19

Identity and access

Every action on a shared instance has a name attached, and the name is a person's. Chapter 16 ended on that line. This chapter is what follows from taking it seriously: who is acting when an engram changes, who may see a domain and who may change it, how a login at the company's identity provider becomes an account here, what the rules cover, the few things you do on an ordinary Tuesday and how access follows GitHub on a server with its own GitHub App. It ends where the whole book has been walking: a colleague who has earned access, by name.

19.1 Who is acting

People arrive as accounts, by whichever of chapter 16's doors they came through. Once inside there's one account with one role (none, viewer, editor or admin), and everything the person does is done as it (Chapter 16).

Agents arrive as somebody's agent. Chapter 16 issued the token, and what it means is the point here. An authenticated agent is its user for the length of the session, with that person's visibility, that person's write rights, that person's share identity in a team domain and that person's name in the provenance of everything it captures, beside the client that acted. Programs in Tron look like their users and act for them, and that's the picture: the agent's name is its user's name with an agent designation after it. The CLI and a stdio session on the server stay the machine's owner, as chapter 16 said. An agent that runs unattended, a build runner say, gets an account of its own that an admin answers for, with no password and one token, so what it captures carries that account's name; on the runner the two remote variables of Section 16.10 point its Crystalline at the server.

The token's lifecycle is what makes the arrangement safe, and it's worth walking once. It's minted on the profile's Agent access card, or by an admin with `crystalline users mcp-token <name>`. It begins with a fixed prefix, `cmt_`, so

a leaked one is recognizable in a log and findable with grep across a repository or a chat export. It's shown once, and what the instance keeps is a hash, so a copied accounts database yields no working credential. It lives in exactly one place after that, the `Authorization: Bearer` header of the harness's MCP registration on the person's own machine. It has no expiry of its own, because on a self-hosted instance revocation is the story. Every request looks the token up, so a revoked token fails on the next call. Rotation issues the replacement and revokes the old one in one motion, so the registration is updated in one edit and nothing is ever half-rotated. Every account may issue its own, viewers included, and that leaks no capability: a viewer's agent is read-only by construction, because it is a viewer. A read-only instance still issues them, because the mode protects the knowledge and a token is account state: with agent authentication on, a read-only server is where an agent can't connect until it has one. On an instance where hosted clients sign in there's a second kind of credential at the same door, the access token a hosted client holds, prefixed `coa_`, and the gate checks both kinds on every request (chapter 16 has how such a client comes by one).

19.2 Membership on a private domain

A domain is visible to every signed-in user unless it's made private, and chapter 16 said who sees it then: the owner, the instance's admins and invited members at one of four levels (none, viewer, editor or manager). The top rung deserves a word. A manager runs the membership, inviting, removing and granting any level up to manager itself, which is the job of a lead who shouldn't have to fetch the owner for every new joiner. What a manager cannot do is change the domain's visibility, transfer it or delete it, because a manager who could flip a domain public would be an owner by another name. Those three stay with the one owner and with the admins, so that a domain always has exactly one person answerable for its existence, and a way to recover it when that person leaves.

Two of chapter 16's truths carry over unchanged, and Fluid prints the second under the Make private button: admins see every domain, private ones included, and a private domain protects from the instance's other users, never from whoever operates the machine.

The instance role and the membership level are different questions, and a write needs both to say yes. An invitation widens what an account may reach, never what its instance role lets it do, so an instance viewer invited as an editor reads the domain and writes nothing, because an invitation never widens a role. On a

team domain under the server's own GitHub App the membership isn't asked and the instance role no longer caps a write: the person's role on the repository decides, as the last section of this chapter explains.

A second question joins those two, because Reflection gives people a way to work on a domain without the engrams. Every account, and every membership of a private domain, carries a Reflection level beside its level on the engrams. The engram side has a value below viewer, none, for somebody who sees no engram at all. A reader reads the reflections and comments, and a contributor also edits them. Writing them from the engrams is curating, and you never grant it: everyone from engram editor up curates, and Fluid shows them as curators. A viewer of the engrams is a reader without anyone setting a level. A Reflection level you do set only ever raises that, so a person with knowledge none and Reflection contributor edits the reflections and never sees an engram. A member of a private domain at none needs a Reflection level, or the membership is refused. Chapter 12 tells why the two are kept apart (Chapter 12).

19.3 Linking identities

A team that adopts single sign-on after running local accounts for a while has a linking problem: Kane has an account named `kane` and an identity at the provider, and they should be one person. The tempting answer is to match on email, and it's the classic foot-gun of this integration, which is why Crystalline refuses it. An email address is presentation: the provider asserts it, the person can change it and matching on it would let whoever controls an address at the provider walk into the local account that happens to carry the same one. The address is often not even the person's to keep: a provider that lets people edit their own email, or a second provider added a year later, is all it takes. Chapter 16 gave the rule that follows, issuer and subject and nothing else, so the sign-in gets a fresh account and joining the two is an explicit act with a name on it. The signed-in local user links from the SSO identity section of their profile. An admin does it from the terminal, `crystalline users link kane --issuer <issuer> --subject <subject>`, which is also how an admin repairs links after a provider re-registration changes every subject. Unlinking refuses when it would leave an account with no way in, and the refusal names `crystalline users passwd` as the way to give it one. The `--force` flag on the unlink is for the re-registration repair, where the account is unreachable for the moment between the old link going and the new one landing. An identity already linked to another account is never taken by a second link, so an admin moves it.

19.4 What the rules cover

A rule that covers most surfaces covers none, because the agent will find the one it misses. Visibility follows the caller onto every read surface there is: search, `read_engram`, `build_context`, `recent_activity`, the routing briefing at session start, the graph, exports, the maintenance sweep and Fluid's domain list and tree. To someone who may not see a domain it's absent, answered exactly as a domain that was never registered would be, for the reason chapter 16 gave. Writes check for editor or better. An agent inherits all of it from its user and administers no membership: there's no tool for changing one, on purpose, so a persuasive prompt cannot talk an agent into inviting itself. What it may change about the instance itself follows its user's role too: registering a domain, writing an instance setting and deciding what a domain installs take an admin, and reading any of them back is open to everyone. Unregistering a domain or changing its policies takes an admin or, for a private domain, its owner. On an instance that leaves agent authentication off, the open tier that remains sees only the domains everyone sees, except on a server with a GitHub App, where it acts on the team domains as the account `github.agent_identity` names (private ones included, never above editor) or as an anonymous visitor when that setting names nobody. Turn agent authentication on there. It may change a domain's policies wherever it may write, but it's refused outright when it tries to unregister a domain. A private domain with a team origin is two questions: who may see it here is the instance's, and who may push to the repository is the forge's. On a server with its own GitHub App it's one question, and GitHub answers it.

19.5 Day to day

Most of this is one section, Members, in a domain's settings in Fluid (the domain page's More actions menu leads there): the members with their level badges, an invite box (a username, a level and a Reflection level), remove, leave and, for the owner or an admin, transfer. The switch to private and back to shared sits further down, in the Danger zone, and asks you to type the domain's name first. Private domains carry a badge in the domain list (Figure 19.1). The terminal does the same for a script or an admin already there: `crystalline domain members board list`, `crystalline domain members board add lambert --level viewer`, `crystalline domain members board remove lambert`, `crystalline domain visibility board private --owner dallas` and `crystalline domain transfer board ripley`. The CLI is the machine operator and passes no role gate.

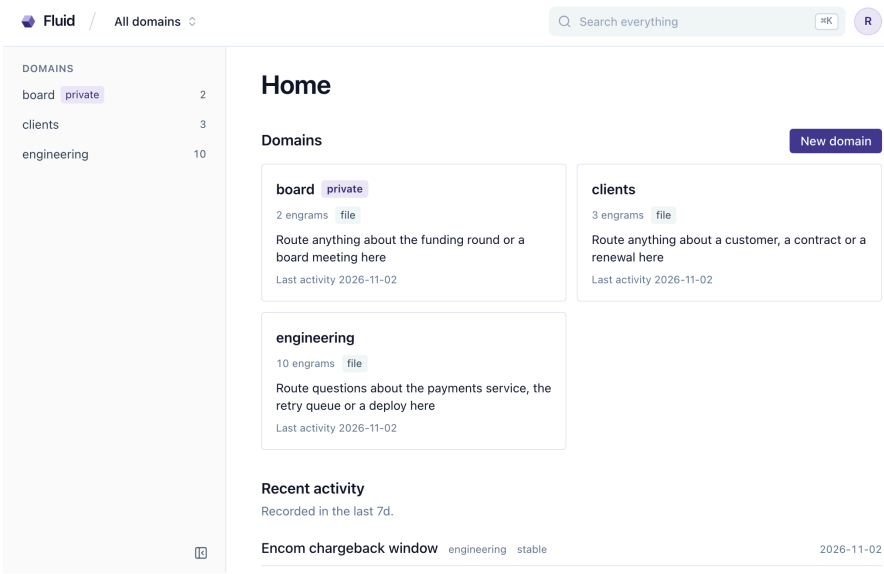


Figure 19.1: The domains of an instance in Fluid, private ones marked.

A year of running a domain comes down to inviting, promoting and handing over. Inviting is the Members section or the `add` verb, at the level the person needs today, which is usually viewer. Leaving is the member's own button (no manager required). Promoting is setting the level again, viewer to editor when their captures have earned it, editor to manager when they should be the one inviting. Handing over is transfer, for the day the owner leaves: the new owner is the owner from that moment, and the old one is a stranger to the domain unless re-invited. When they do leave, an admin disables or deletes the account, and deleting sweeps what the account had connected: its agent tokens, its identity links and the grants it gave hosted clients. Its personal share credential goes too when the account is removed on the server with `crystalline users remove`. A private domain it owned is left with no owner, and only an admin can administer it, so transfer it first. The agent that acted as that person is refused at the door on its next request, like any other stranger.

> **In practice**
> Gaff joins the club for a season, on the strength of one recommendation. He signs in on his first evening with the login you made him and lands as a viewer, the role every new account starts with. He issues himself a token on his profile and pastes it into his harness. His agent is a viewer from its first call, reading `club` and writing nowhere, without anyone having configured a thing about

agents. By the third meeting the notes he keeps sending you to capture for him are the best anyone has written about Dune, and you promote him to editor. The next capture his agent makes lands in `club` under his name, with his harness named as the client in the provenance.

Later Dallas moves abroad and transfers the private `committee` domain he keeps for the club's money to you from the Members section, then finds he can no longer see it, which is what he wanted. He never wanted an admin login, and owning the domain never needed one. When Gaff's season ends you delete his account. His token dies with it, and his agent's next session is refused before it opens. His engrams stay, every one recording that his agent wrote it for him, and a month later a new member's agent recalls Gaff's take on Dune in a session that never heard his name.

Under the hood

Accounts, agent tokens, identity links, domain visibility and membership all live in one small database beside the index, and none of it is derived, so no rebuild of the index can touch it, and a wipe and resync leaves every permission where it was. Each request resolves the caller once, into a scope: the machine owner for the CLI and stdio, an account with its admin flag for a Fluid session or an authenticated agent, an anonymous tier for a published archive, or an agent where authentication is off. Every engine verb filters on that scope, so a surface added tomorrow is filtered the way search is filtered today. The verbs that change the instance rather than its knowledge read the same scope for the admin flag: `add_domain`, a `configure` that sets, unsets or connects for the instance (linking your own GitHub account on a server with an App needs no admin) and `provision` with `allow`, `deny` or `apply`. A bare `configure` and `provision status` stay reads, and `remove_domain` and a `configure` that changes one domain ask the domain instead: its owner or an admin may call them, and on the open tier only `configure` lets an anonymous caller through.

19.6 Access that follows GitHub

On a server with its own GitHub App (Chapter 16), a domain connected to a repository follows that repository. Each person links their GitHub account once, and from then on their role on the repository decides what they may do with the domain here. Write access or more writes, and the instance role doesn't cap it: an instance viewer with write access on GitHub writes, in Fluid too, while an instance editor with only read access, or with no link, writes nothing. Read access reads. An admin or maintainer on GitHub also switches the domain

between shared and private and manages its Reflection collaborators. Owners, members and invitations set in Crystalline don't count on such a domain, so the Members section says that access follows GitHub and refuses a change, which belongs on GitHub. An account's write list still applies.

Who reads is still the shared or private switch. A shared domain is read by everyone signed in at viewer or above, linked or not. A private one is read only by linked people with read access or more on the repository, and only through its list of collaborators: the members of a company's GitHub Enterprise who read an internal repository without being listed there don't read the domain. Two rules come before GitHub's answer. An account with the instance role none gets nothing from GitHub, and the instance admins and the machine owner get everything.

The link is made from your profile in Fluid, from a single-use link an agent signed in to the server hands you or from a terminal, and it works only for the account that asked for it, so a forwarded link never links somebody else. A pasted token or a personal GitHub sign-in made without the App is refused on such a server. An agent account gets its access the same way, through a GitHub bot user that an admin links to it (Chapter 14).

The Reflection follows GitHub on its own axis. A GitHub writer curates, a GitHub reader reads and comments, and a Reflection collaborator of one domain is the way in for somebody GitHub knows nothing about. That grant gives one domain's Reflection to an account as reader or contributor, works for an account with the instance role none and only ever raises: it never opens an engram the account couldn't see before.

A domain whose repository loses the App (it was removed from the installation, deleted or moved to another owner) stays connected but closed. Only the instance admins and the machine owner get in, and no other rule opens it, a Reflection grant included. The admin banner in Fluid and the health check name each closed domain and say why. Installing the App on the repository again opens it without a restart, and a domain whose repository moved to another owner stays closed until an admin connects it again to where the repository is now.

Here the agent is the chat app on your phone, signed in to the club's server itself:

```
You:    Link my GitHub account to the club's server.
Agent:  Open this link in a browser where you're signed in to
        https://club.example.com as yourself. It works once, within
        ten minutes: <link>. GitHub asks you to authorize the
        club's App and sends you back to your profile.
```

In practice

Your book club's server, the one you run, gets a GitHub App of its own one evening. You press Create GitHub App in Fluid with the organization left empty, and Dallas confirms on GitHub, signed in as himself, so the App belongs to the account that holds the club's repositories. He installs it on the private `dallas/book-club`, where `club` lives, and on `dallas/club-committee`, a private repository made for `committee`, the domain that keeps the club's money, with you as its second collaborator. You connect `committee` to it as private, and from then on the repositories decide. Kane, who already has read access on `dallas/book-club`, links his GitHub account from his profile in Fluid. `club` is shared, so he reads it and comments in its Reflection as before, and the day the club gives him write access on `dallas/book-club` his agent writes there too. `committee` is read by the people the repository lists and by the server's admins, whatever the Members section said before, so Dallas, who owns `dallas/club-committee`, reads it again: GitHub says he may, and GitHub decides now.

Lambert follows the club on her phone and sees no engrams, and the next morning she can't open a page of the club's Reflection. Her account has the instance role none, and on a domain GitHub decides that gets her nothing, her account's Reflection level included. You give her one for `club` alone, as a contributor, on the Members section under Reflection collaborators, and she's back on every page of the club's Reflection without a single engram coming into view.

Under the hood

From a terminal, `crystalline connect github --personal` links your account through the App's device flow (Chapter 16 says what the App needs for it). A domain is connected with the question answered, `crystalline domain add committee --origin dallas/club-committee --readers private`. A GitHub admin or maintainer, or an instance admin, switches it between shared and private on the Members section, and `crystalline domain visibility committee private` does the same from the server, and no account becomes its owner, so no person inherits the domain if the App goes later. `crystalline domain members club reflection add lambert --level contributor` gives the grant, `reflection remove lambert` takes it away, and `crystalline domain members club list` shows the Reflection collaborators beside the members. On such a domain `members add`,

`members remove` and `domain transfer` refuse with “access to this domain follows GitHub; change it on GitHub”. GitHub’s admin and maintain roles count as manager, write as editor and triage and read as viewer.

Crystalline asks GitHub once per repository who the collaborators are and whether the repository is public, and uses the answer for five minutes. A webhook from GitHub only makes it ask again at once, and no delivery ever grants anything by itself. When GitHub can’t be reached, the last answer holds for one more hour and the admin banner says since when; after that GitHub grants nothing on that repository until it answers again, so a shared domain is read by everyone signed in and a private one by the admins alone. Unlinking or disabling an account takes effect on the next request.

A new hire earns access a door at a time: a badge on the first day, a room the next week and after a while nobody remembers the meeting where they were let in. The agent walks the same doors, by name, and everything it has ever captured says whose colleague it was.

In brief

People act as accounts and agents act as their user, through a per-user token that’s shown once, stored hashed, revoked in one motion and issuable by everyone. The CLI and stdio are the machine owner. A private domain is visible to its owner, the instance’s admins and invited members at none, viewer, editor or manager level, with visibility, transfer and deletion reserved to the owner and admins. It protects from other users, not from whoever runs the machine. Identities link explicitly, keyed on issuer and subject, never by email. Visibility is enforced on every read surface and a hidden domain reads as absent. Writes need editor or better on the domain and on the instance, and agents inherit and administer no membership. On a server with its own GitHub App a team domain follows its repository: write access writes, read access reads, an instance role none gets nothing and a Reflection collaborator grant opens one domain’s Reflection. Invite, promote and transfer from the Members section of the domain’s settings or the `crystalline domain` verbs, and delete an account to take its tokens, links and grants with it, once what it owned has been transferred.

CHAPTER 20

Knowledge across domains

This chapter is about Reflection pages that draw on more than one domain: who reads such a page, why a section carries only what all of its readers may read, how an agent running on its own is kept from passing a secret along and how a person shares a line on purpose anyway. It picks up where Chapter 12, Chapter 14 and the access that follows GitHub (Chapter 19) left off.

20.1 A page is a noticeboard

In Chapter 12 a page could draw on the engrams of other domains too (handy when one page should sum up several). Picture the noticeboard in the clubroom. A note pinned there reaches everyone who walks by, key to the room it came from or not, and a page reaches everyone who may read it the same way. Hence the one rule every section follows: it may draw on another domain only when everyone who reads the page may also read that domain's engrams.

On the page's side of that rule you count everyone who reads the domain's Reflection (its Reflection collaborators too), on the other side only the people who read the engrams. A Reflection grant there opens that domain's pages and never its engrams, so it doesn't help. Drawing on the page's own domain is always fine, and the admins and the machine owner read everything anyway, so they never decide anything here.

On the club's server access follows GitHub (Chapter 19): `club` is shared, so every signed-in account at viewer or above reads it, and `committee` is private and lives in a repository of its own, `dallas/club-committee`, where only Dallas and you are collaborators. Set the two directions side by side (Figure 20.1): a `committee` page may draw on `club`, because the people who read it (Dallas and you) both read `club` anyway. A `club` page may not draw on `committee`, since every member reads the page and most of them can't open `committee`.

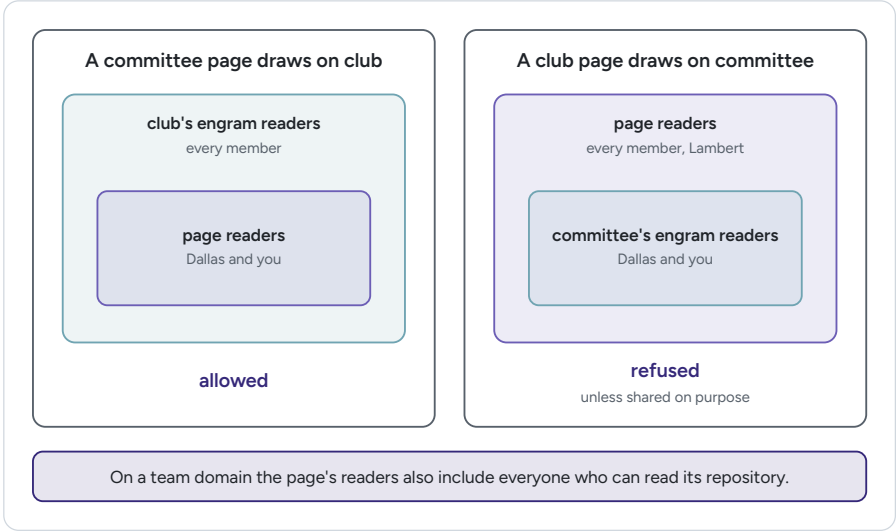


Figure 20.1: The readers of a page compared with the readers of the domain a section draws on. The page may carry the text when its readers fit inside.

Say Dallas switches Reflection on for `committee` one day and wants a page on what the club spent on books this year, built from the receipts in `committee` and the reading list in `club`: the agent writes it and nobody hears a thing. Ask for the same receipts on a `club` page and the write is refused, with nothing written.

Under the hood

The rule holds for every writer: a person's agent, the machine owner and a scheduled run, on the daemon, the embedded stdio server and a one-shot command alike. A refused write gets "this source is in a domain fewer people may read than this page's domain; leave out the source and every sentence drawn from it, since the check sees only named sources, or write that text on a page in that domain instead". The check sees only the sources a write names and never reads the text. An agent that drops a refused source and keeps its sentences isn't caught, so a curator reads a page built from several domains with that in mind. Name a domain the writer may not open and the refusal looks like the one for a domain that doesn't exist. When the accounts store can't be opened, every source outside the page's own domain is refused. (With a GitHub App, a private domain is read by the instance admins plus the linked accounts its repository lists as collaborators with read or more, never an account with the instance role none.)

20.2 Guests at the board

Lambert follows the club on her phone: her account has the instance role none and she holds a Reflection grant on `club` as a contributor, so she reads and edits every club page and has never seen an engram. As long as she holds that grant, no `club` page may draw on any other shared domain, `sci-fi` included (it lives in the same repository, and every member reads it). The other way round is fine, though: Lambert has no grant on `sci-fi`, so whoever reads a `sci-fi` page reads `club` too.

Visitors without an account are the other kind of guest. On the club's server a visitor reads a shared domain only when its repository is public on GitHub, and `dallas/book-club` is private, so they don't come into it. If Dallas ever makes it public, every visitor reads every club page, and a club page then draws only on domains a visitor may read as well.

You'll most likely find out about Lambert like this:

```
You:    On the Dune page, add what the sci-fi shelf says about
        the sequels, with sci-fi as the source.
Agent:  Refused, nothing written: this source is in a domain
        fewer people may read than club's. I can leave out the
        source and every sentence from it, or write the text on
        a page in sci-fi instead.
You:    Lambert. She reads every club page and no engrams.
        Leave it out, then.
Agent:  Left out. The Dune page is as it was.
```

The refusal names nobody, so working out why is up to you, and on the club's server the answer is nearly always Lambert.

> **Under the hood**
> A Reflection grant counts on the page's side and never on the other side. On a domain that follows GitHub, an account with the instance role none gets nothing from its instance Reflection level, so Lambert reads the club's pages through her grant on `club` alone. (A grant on `sci-fi` as well wouldn't help club pages and would also stop `sci-fi` pages from drawing on `club`.) An account that gets such a grant later is caught by the queue, which checks again on every scan. A page an anonymous visitor reads may only draw on domains an anonymous visitor reads too.

20.3 The repository reads along

A page in a team domain also goes to the `crystalline/reflections` branch of its repository as a backup, and anyone who can read the repository can read that branch, with an account on the server or without one. That gives the rule a second half, and a section needs both: the domain it draws on has to be on GitHub too, and everyone who reads the page's repository has to be able to read the other repository. A domain from the page's own repository always passes, and one from a public repository does too. When the page's repository is private, another repository passes if each collaborator of the page's repository is a collaborator there too. A local or virtual domain never passes without a person's yes, which is why personal knowledge can't end up in a team's repository this way.

In the club the second half mostly agrees with the first. `club` and `sci-fi` share `dallas/book-club`, so only Lambert stops a club page from drawing on `sci-fi`. A `committee` page drawing on `club` passes again (Dallas and you are the only collaborators on `dallas/club-committee`, and you're both on `dallas/book-club`). A `club` page drawing on `committee` fails here as well: Kane reads `dallas/book-club`, so does the GitHub user the club's scheduled run acts as, and neither of them is on `dallas/club-committee`. Your own `books` and `shops` stay on your laptop and never get near the club's pages.

> **Under the hood**
> A source in a domain that isn't on GitHub is refused with "this source is in a domain that is not shared through GitHub, and this page is backed up to a GitHub repository; leave out the source and every sentence drawn from it, or cite it from a page in that domain instead". The repository half is answered from the App's list of collaborators, which is fetched once per repository, kept for five minutes and refreshed at once by a webhook, and while it isn't known the source is refused. An internal repository of a GitHub Enterprise is read by the whole enterprise, so a page there draws only on its own repository's domains and on public ones. Without the App, Crystalline doesn't know a repository's readers and treats any two connected domains alike. One honest caveat: the branch keeps each page as it is on disk, so whoever reads the repository sees the names and addresses of the domains and engrams a page draws on, never their text.

20.4 The confused deputy

Security people have an old name for a program that holds more rights than the person asking it and gets talked into using them on that person's behalf: a confused deputy, which means no harm and simply never checks whose rights it's using. In the Reflection the obvious candidate is the scheduled run. Anyone who may comment on a club page can ask a question in a thread, and a run that could read `committee` would answer from there, into a thread every member reads (the answer itself would be the leak). Nobody in the thread would even need a Jedi mind trick: a polite question would do.

Threads don't get a source check: a thread is people talking, with each other and with their own agents, and what you write there is yours to say. And since they go to the backup branch with their page, a thread reaches everyone who reads the repository. That's why the guard sits on what an agent reads while nobody is behind it (Figure 20.2): an unattended session reads only the domains every reader of its own domain may read, the others look as if they didn't exist, and it writes into its own domain and nowhere else.

The club's run can't see `committee`, because most members can't open it, and it can't see `sci-fi` either, because Lambert reads the club's pages without reading `sci-fi`'s engrams. So the run on `club` reads `club` and nothing else (that's what Lambert's grant costs).

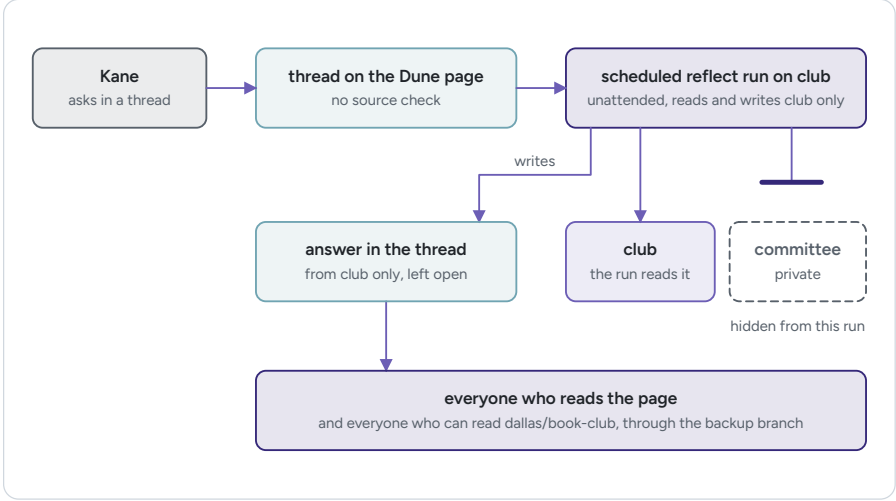


Figure 20.2: A thread question meets a scheduled run that can't see the private domain, so the answer comes from the run's own domain only.

Kane asks on the Dune page why the club never buys the hardback for its shelf. A few minutes later the club's run comes by, sees the club's list but not the budget, answers what it knows and leaves the thread open:

```
Thread on "The club reads Dune in paperback"
  Kane:   Why don't we buy the hardback for the shelf? It's the
          one we'd all keep.
  Agent:  note (non-blocking): The club waited for the paperback
          reprint because nobody wanted the hardback. Whether
          there is money for a shelf copy isn't something I know.
          Leaving this open for Dallas.
```

Under the hood

A session is unattended when a run started it, when it was started with `--unattended` (mark your own cron job this way) or when its caller holds a live lease of the reflect or evolve loop. `crystalline reflect run` and `crystalline evolve run` start their agent with `crystalline mcp --unattended --domain <domain>`, or over HTTP with the header `X-Crystalline-Run-Domain: <domain>`, which the server believes only while that account holds the lease. The limit covers every read: search, `read_engram`, `build_context`, `browse_domain`, `recent_activity`, `list_domains`, the sources a page names, attachments and other domains' Reflections. Writing anywhere else gets "this run works on club and cannot write to committee". A local run without an account looks just like you, so while it's going your other local sessions read and write like the run, and every tool answer carries one line that says so.

20.5 Sharing on purpose

Sometimes a line from a narrower domain does belong on the board, and a person who may read it decides that. If an agent writes a section the rule would refuse while a person is behind the session, the write stops and asks that person first. The question tells you in plain words who reads the page, shows you the text and warns that it becomes readable by all of them (the names of the sources stay hidden from readers who may not open that domain). There's one box, and it starts unticked: only a ticked box plus Accept shares the text, while a decline, an empty box or ten minutes of silence refuse the write. In an unattended session nobody is asked, so a run never shares anything.

Only someone who may curate the page and read the domain drawn on can say yes. It holds for that person, that page and that domain, so their later edits there

need no new question, and anyone else is asked on their own (Figure 20.3). Don't mistake the box for a lock against your own agent: an agent with a shell on your machine can do anything you can, and a client that answers questions by itself gets around it. The guard is the read limit from the section before. In Fluid a person answers the same question from the Review, behind a Share on purpose button on a section the queue already names (after an edit on disk, say, or once the readers grew).

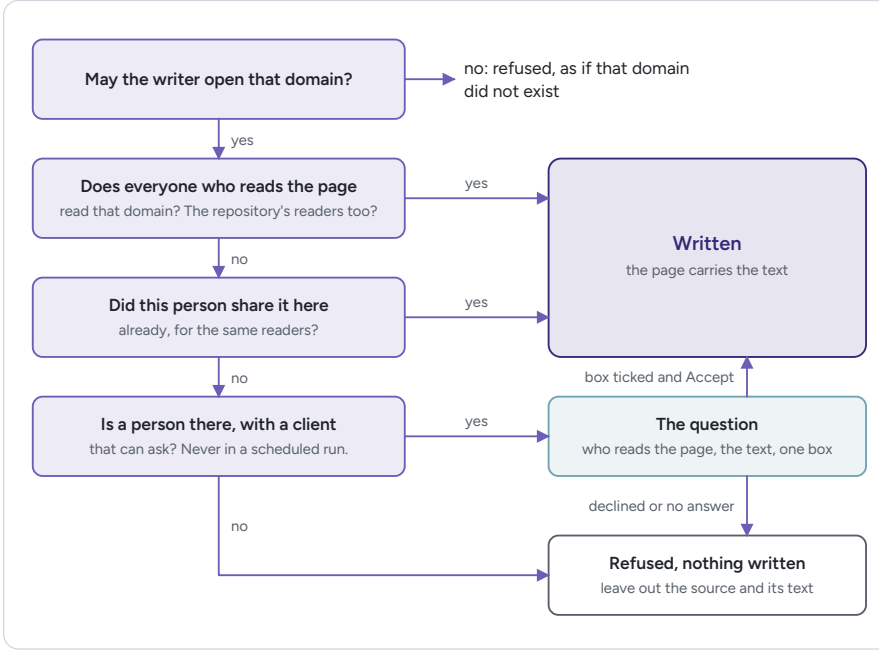


Figure 20.3: What happens when a write names another domain to draw on.

Dallas reads Kane's thread and agrees with you that members should know where the shelf money went. So you ask the agent you use on the club's server, which signs in as you with your own token (it has to be that one, since nobody gets asked about a write your laptop forwards to the server):

```
You:    Add one line to the Dune page from committee: this year
        the club pays half of each member's first Dune
        paperback, and that's where the shelf money went.
        Then answer Kane.

  Share text from committee on a page more people may read?
```

```
  The page dune.md in club is read by the 9 accounts that may
  read club, and by the people who can read the GitHub
  repository dallas/book-club, where this page is backed up.
  Fewer people may read committee. Not everybody who can read
  dallas/book-club may read the GitHub repository of committee.
  The text to share: "## The club reads Dune in paperback The
  club waited for the paperback reprint because nobody wanted
  the hardback. This year the club pays half of each member's
  first Dune paperback, which is where the money for a shelf
  copy went.". If you agree, this text becomes readable by all
  of them. The names of the sources stay hidden from readers
  who may not read committee. Agree only if you mean to share
  it.
  [ ] Yes, share it with these readers         Accept  Decline

You:    (tick the box, Accept)
Agent:  Written. The Dune page has the line, and I answered
        Kane in his thread.
```

When Kane opens the page that evening the line is there, and the section's sources say "1 source in a domain you cannot open". Dallas sees `committee` in the same spot, with a note that you shared it on purpose and when.

Under the hood

`write_reflection` asks with `input_required` on a client of the 2026-07-28 MCP revision, or with an elicitation request on an older client that declared it. A client that can't ask its user is refused, and the refusal says that a person who may read that domain can share it from a client that can ask. The question never goes to an anonymous caller or the open tier, and it never comes up for a write a connected machine routes to the server. A yes is bound to the readers and the repository the question named, and when either changed before the write lands, the write is refused with "who may read this page, or where it is backed up, changed since the person was asked; ask them again". The server keeps the confirmation in its index, never in the page file: who, when, which page and domain, who read the page then and, with the App, who read its repository then. `read_reflection` shows curators a section's `disclosed`, and Fluid shows "Shared on purpose by" with the name and date. Moving the page keeps the confirmations, while deleting the page, dropping the last citation of that domain or `crystalline reindex --wipe` drops them.

20.6 When the readers change

Some changes can't be refused because they never go through a write: a page edited on disk, a page restored from the backup, a change of who reads a domain. A yes on purpose covers only the readers it was given for, so a new reader who can't open that domain ends it. If the domain drawn on narrows later, nothing happens (its text reached those people already). Whenever a section ends up drawing on a domain fewer people read, the queue names it for the curators who may open that domain, and nobody else sees the item. The page keeps serving in Fluid until a person decides, because deleting text is a person's call. The backup is stricter, since the repository's readers read the branch: it holds the page back with its threads and the files only held pages link, and the branch head loses its copies (what went out earlier stays in the branch's history). Once a curator removes the source and its text, or someone shares it on purpose again, the item is gone and the page goes back to the branch (Figure 20.4).

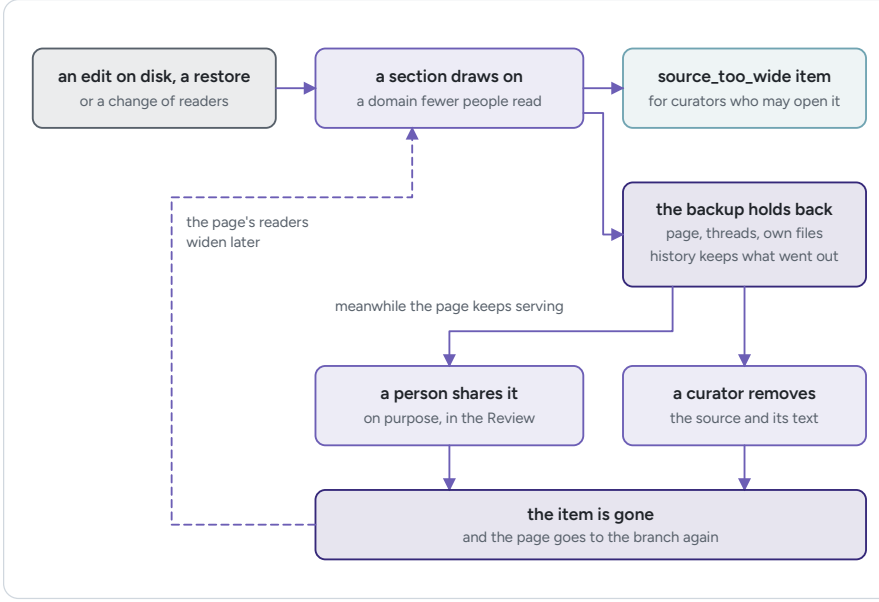


Figure 20.4: A section whose readers grew beyond the domain it draws on is held back from the backup until a person decides.

In the autumn Dallas adds a new member as a collaborator on `dallas/book-club`, so she can follow the club on GitHub. She has no account on the server yet and can't read `dallas/club-committee`, so seconds after GitHub's webhook

the Dune section is a queue item and the next backup drops the Dune page and Kane's thread from the branch head (the version that went out earlier is still in the history, so the hold only stops the next copies). Members on the server don't notice a thing (the line is still on the page). On Monday Dallas opens the Review, finds the section under "Cites a domain fewer people read", presses Share on purpose, reads the question and chooses Share it. The item's gone, and the next backup puts the Dune page back on the branch.

Under the hood

The queue item is `source_too_wide` (only a caller who may open that domain gets it, with the question a person is asked). The daemon looks for pages that newly break the rule every five minutes, and at once after a change of visibility, membership or account through the JSON API or with `crystalline domain visibility`, `domain members` and `domain transfer`, and when a GitHub webhook comes in. The backup checks the files it's about to send as well as the index, so a page edited on disk is held back before the index has read it, and a page whose `reflection:` block can't be read is held back too. It logs what it held back and looks again ten minutes later, or at the next change. Share it in the Review confirms for whoever reads the page at that moment; if that changed since the Review was loaded, the confirmation is refused and the item asks the new question.

In brief

A Reflection page reaches everyone who reads it, so a section may draw on another domain only when all of them may read that domain's engrams (on a team domain that includes everyone who reads its repository). A Reflection grant counts for the page and never for the domain drawn on, which is why Lambert's grant on `club` keeps every club page from drawing on `sci-fi`. A run that works alone reads only what every reader of its domain may read and writes only there, so a question in a thread can't make it a confused deputy. A person shares a line from a narrower domain on purpose by ticking one box, and that yes holds for the readers of its moment. When the readers change, the backup holds the page back until a person decides.

CHAPTER 21

Equipping the agent

A wrench is not a mechanic. A new colleague's first day ends with a laptop, a badge and a login to everything the team lives in. Then comes the part that's on no checklist: some time in the afternoon somebody pulls a chair over to show them how the deploy script is actually run, which flag everybody passes, which one you never pass on a Friday. The tool arrives with a person who knows what it's for, or it arrives as a surprise.

The chapters before this one gave the agent the floor plan (Chapter 6), the teaching, a place in the team's review and a name on the door. What it still lacks is the toolbox. Crystalline's answer is that the domains holding the knowledge also ship the tools that knowledge depends on, into the harness, with a person's consent, and keep them current afterwards. This chapter is that mechanism, and it's the last practical one before the book turns to what all of it was for.

21.1 What a domain can ship

The list is short. Skills are the playbooks a harness loads before a kind of task. Slash commands are the short prompts a person invokes by name. A subagent definition is a role with its own instructions that the harness can hand a piece of work to. MCP server configs let a domain about the payments service bring along the server that reaches the payments staging environment. Each kind is a folder in or beside the domain.

An artifact is written once, in whichever dialect its author's harness speaks, and translated into every harness whose format can carry it. Claude Code and Codex take all four kinds: an agent written as markdown becomes Codex's TOML dialect, or the other way round, and an MCP server registers through the harness's own CLI. The Copilot CLI takes every kind but commands, since it has no prompt-file surface for one to land on, so a command is skipped there with a notice rather than silently. Cursor, Kiro, Gemini CLI and Qwen Code take skills only, and the other kinds are skipped there with the same notice.

Where the target cannot express a field, a vendor model name or a permission mode, the field is dropped and the notice says which.

The manifest declares what a domain ships in a `## Provisioning` section, one bullet per kind:

```
## Provisioning

- skills: skills
- commands: commands
- agents: agents
- mcps: mcps
```

Each bullet is `type: path`. The path is relative to the manifest itself and may climb out of the domain root with `../`, for a folder that lives beside the domain rather than inside it. The `mcps` folder holds one JSON file per server, named by its own `name` member or, failing that, by the file. The starter manifest that `crystalline domain init` scaffolds leaves the section out on purpose: it's added by hand, once a domain has something to ship. A manifest gains it the week somebody writes the first skill.

21.2 Why the tools travel with the knowledge

A harness has a skills folder of its own, and a team could commit into it directly, so it's fair to ask what a domain adds, and the answer is the wrench. The knowledge that says when a tool applies, and how it's run here, is an engram: the runbook that says the deploy skill is for the payments service and never for the ledger, the gotcha about the flag, the convention about Fridays. The tool is the skill. Keep them in the same domain and a new member's harness receives both in one motion. The engram that says "use this" can't drift from the thing to use, because the two live together and travel together.

The rule that follows: never provision a tool without the engram that says when it applies. A skill teaches a harness how. Only the domain's knowledge says whether this is the moment, and whether it's the moment here. A well-equipped domain reads like a workshop where every drawer is labelled, and the label is an engram somebody can correct. The sonic screwdriver famously doesn't do wood. A tool arrives with its limits written down or it arrives as a surprise.

Crystalline's own skills are the worked example. The routing, capture, schema, collaboration, provisioning and presentation skills from chapter 15, and the

consolidated one for Claude Desktop, are exactly this: playbooks that say how to use the tools well, shipped by the thing whose knowledge they depend on. `crystalline install` places them into the harness's skills folder. The install receipt keeps them current across upgrades, with an edited copy kept beside the new one as `SKILL.md.bak`. A remote harness that never ran the CLI reads the same seven through the `skills` tool, an index with no arguments and one full `SKILL.md` by name, or as `skill://<name>/SKILL.md` resources. Provisioning is that mechanism, opened to your domains.

21.3 Consent

Nothing installs until a person decides. A domain that declares a Provisioning section with something in it and has no decision yet surfaces at the start of the next session, as one line trailing the routing block the hook delivers:

```
[crystalline] The `engineering` domain ships artifacts to provision
(mcps: 1, skills: 1) but has no decision yet - ask the user, then apply
it with the `provision` tool or `crystalline provision allow
engineering`.
```

The agent raises it with whoever is at the keyboard and applies the answer with the `provision` tool.

The decision is per domain and per machine. It is recorded in the domain's registration in `config.yaml`, as `provision: true` or `false`, never in the manifest, for the same reason chapter 17 kept review mode out of it: what a domain ships is knowledge the team shares; whether this machine takes it is a choice its owner makes. Denying removes whatever the domain already shipped. `status` prints every domain's decision and counts, grouped allowed, undecided and denied and sorted by name, then every onboarded harness's installed state and the domains still waiting (Figure 21.1). The domains that ship nothing and the virtual ones fold into one line each unless you ask for the full list. The `provision` tool answers the same report to an agent, and until some domain declares a section it has nothing to allow or deny and says so. Allow, deny and apply need the instance admin role on a shared instance whose agents sign in, because each changes what the daemon's own machine installs. The report stays a read: any account gets it, listing only the domains that account may see.

```
ripley@tyrell-laptop: ~ (crystalline provision status)

$ crystalline provision status
engineering: allowed, 1 skills
2 domains declare no provisioning (run `crystalline provision status --all` to list them)
claude-code: 1 file installed, 0 mcps installed, 0 drifted, 0 edited, 0 orphaned, 0 missing
```

Figure 21.1: The provisioning decision of every domain that ships tools and the installed state of every harness, with the other domains folded into one line.

Under the hood

From the terminal the two decisions and the report are three commands:

```
crystalline provision allow engineering   # opt in, then reconcile
crystalline provision deny engineering    # opt out, removing
anything already shipped
crystalline provision status              # decisions by group,
every harness's installed state (--all lists every domain)
```

21.4 Reconcile

Bare `crystalline provision` reconciles every opted-in domain into every harness this machine onboarded with `crystalline install`. Run it as often as you like. It converges. It installs what's missing, updates what a domain changed and retires what a domain no longer ships. Anything it didn't write itself it treats as somebody else's. A provisioned file you edited by hand is still brought current, with your edited version kept beside it as a `.bak` copy rather than lost. Retiring an edited file renames it to `.bak` the same way. A file Crystalline never wrote is adopted when it already matches byte for byte and otherwise left alone, never overwritten. An MCP server name already registered by somebody else is left where it is, with a notice.

Most of the time nobody runs it. The session-start hook checks whether anything drifted since the last reconcile, a stat of the domain's folders against the receipt with no hashing. When something did, it reconciles the file artifacts on the spot and reports one line: how many were refreshed. MCP changes are the exception: registering a server means spawning the harness's own CLI, which a hook shouldn't do, so those wait for an explicit run and a second line says so. Change a skill in the domain, and the next session has it.

`crystalline doctor` is the report-only view. When a domain ships artifacts it lists each declaring domain's decision and shipped counts, then each onboarded

harness's counts of drifted, edited, orphaned and missing files against what was last reconciled. `--fix` never reconciles a harness. An undecided domain is a normal state awaiting a person, never a fault, and an edited file is left alone by design.

21.5 The team's toolbox

A team domain ships to everyone. The folders the manifest declares travel with the domain when it's pulled. Even a folder that lives beside the domain in the repository is mirrored down with it. Each member's harness sees the same skill that merged, and a change that lands in the repository reaches every harness at its next session start. Each member decides once, on their own machine. The decision is theirs, the contents are the team's. Day one now includes the tools and the floor plan.

Adding to what a domain ships, or changing it, is a code change, and you make it the way you make any other. Clone or pull the repository the domain lives in. Edit the `## Provisioning` section of the manifest, put the skill, the agent definition or the MCP config into the folder it declares and commit. Open the pull request, have it reviewed as code and merge. From the merge on, every instance that follows that domain brings the change down with its next pull, and each member's next session start reconciles it into their harness; an MCP change waits for an explicit `crystalline provision`, as above. You don't need the folder layout by heart for any of it: the provisioning skill lets the agent walk you through the setup, proposing each step and waiting for your yes before it runs anything. The collaboration tools of chapter 17 exist for knowledge. An agent's `share_changes` would carry a skill that sits inside the domain root, since a share takes every file in it. A folder declared beside the domain with `../` is outside what a share can see at all. And a reviewer opening a proposal expects engrams, not a deploy procedure with a server address in it. Keep the tools on the code path and the knowledge on the sharing path, and each review stays the kind of review it should be.

> **In practice**
> The `club` domain gains two folders. Under `skills/` sits `review-writing`, the way the club writes a review, which Dallas had been explaining by chat: which questions to answer, what stays spoiler-free, why a star rating without a sentence is worthless. Under `mcps/` sits the one file that tells a harness how to reach the library's catalogue. Beside them, in the domain itself, is a checklist engram saying when the skill applies and linking to last year's reading list, next to the club's

rule about never picking a book nobody has finished, which the domain already held. Dallas makes the change on a checkout, the way anything in the club's repository changes, you review the pull request and it merges.

Your next session opens with the routing block and one extra line: `club` ships artifacts to provision, two of them, and has no decision yet. Your agent asks, you say yes and the agent calls `provision` with allow. The skill lands in your harness's skills folder, and the catalogue's server is registered through your harness's own command line. A week later you add a note to the installed `SKILL.md` about the library card that differs on your machine, and `doctor` counts one edited file and leaves it alone. Later Dallas changes the skill in the repository. The pull brings it down, the next session start reconciles and your agent reports one artifact refreshed. Your version sits beside the new one as `SKILL.md.bak`. You move the note where it belonged in the first place, into an engram about your own setup, and the skill stays the club's.

Under the hood

A pair of receipts does the remembering. The install receipt says which harnesses `crystalline install` onboarded on this machine, and provisioning targets only those. The provisioning receipt is the ownership record: every file and MCP server a reconcile installed, with a hash of what it wrote. A recorded file is Crystalline's to update or remove; an unrecorded one is somebody else's. An installed file whose bytes no longer match the recorded hash is a hand edit; a domain whose current bytes differ from that hash is drift. That one receipt is where `status` and `doctor` read their edited and drifted counts. Domains are projected in the order `config.yaml` registers them, and when two ship a file to the same path the first wins, with a notice naming both. A virtual domain never ships anything; it has no folder to ship from.

The stranger on main street had a laptop of their own and no idea what your deploy script was called. A colleague is somebody you handed the tools, on purpose, and showed which drawer they live in. The agent is handed them the same way: one domain at a time, with a yes each time and never a wrench without the label.

In brief

A domain ships the tools its knowledge depends on: skills, slash commands, subagent definitions and MCP server configs, declared in a `## Provisioning` section of its manifest and translated into every harness whose format can carry them. Nothing installs until a person decides. An undecided domain surfaces at session start, `crystalline provision allow` and `deny` record the decision per domain and

per machine, and `status` reports every decision and every harness's state. A bare `crystalline provision` reconciles idempotently, keeping hand edits as `.bak` and never overwriting a file it didn't write. The session-start hook reconciles file artifacts on its own, and `doctor` reports drift without touching anything. Pair every tool with the engram that says when it applies. A team domain ships the toolbox to every member's harness with the knowledge that explains it, and a change to that toolbox is made as a code change, committed and reviewed in the domain's repository, from where every connected instance pulls it.

CHAPTER 22

Where to go from here

Chapter 1 left you on main street with a stranger and a function in the wrong language. Eighteen chapters later the stranger has a desk, a floor plan, a month of teaching behind it, a place in the team's review, a name on the door and a toolbox. What remains is how to begin, when to grow and what all of it was for.

22.1 The first week

Begin this week. Everything the book asked for fits into five working days, the same five you'd give a person.

> **The first week**
> Monday is chapter 5: the binary, `crystalline install` for your harness, and one domain with a manifest whose `When to Use` bullets you wrote yourself. The test at the end of the afternoon is a fresh session recalling something the last one learned. Tuesday and Wednesday are chapters 7 and 8. Tell it the ten things you find yourself explaining most, the way you did with your reading rules on chapter 1's Monday: the rule behind the strange number, the nickname, the word the wiki gets wrong, the day nobody deploys. Say yes to each capture, read what landed, correct the wording once. Then hand over the wiki, with the briefing a person would get before opening it: what was renamed, which page wins a contradiction, what to skip. Trim the proposed list before anything is written.
>
> Thursday and Friday are the days the agent stops being a pupil. Give it real tasks and pair with it (chapter 6): assign the change rather than asking what it knows, and let the reminder at the end of each session propose what it noticed. On Friday afternoon ask for the first sweep (chapter 10). Work the mechanical findings in a pass, rule on the judgment ones and correct whatever the week got wrong the way you'd correct a colleague: say what was wrong and point at the page. By Monday the agent knows more about your team than most contractors learn in a quarter, and every fact is a file you can open.

22.2 When to share

Stay on the laptop until it hurts. Chapter 16 named the moment: somebody who never opens a terminal wants to read what the agents learned, an agent runs somewhere that has no laptop, or the same engram has been corrected in two directions on two machines. Any one of those is the signal. Move the daemon onto the team's server, give everyone a login and their agent a token and let the whole team teach one knowledge base. When a domain deserves the review code gets, put it in a repository and let proposals be the review (chapter 17). When the whole company routes questions to it, turn on review mode, a private overlay per person over the shared folder, so nothing reaches the shared truth before somebody looked. None of it needs doing before its signal arrives.

22.3 The release page

Crystalline moves quickly, and this book describes 0.30.0. The colophon carries the permanent address, https://github.com/jordiboehme/crystalline/releases/latest, and a QR code that lands on it. The release notes there say what changed since the version in your hands. Upgrading is installing the newer build over the old one. The install receipt brings your harness's hooks and skills along. Claude Desktop's extension starts the Crystalline you installed, so Desktop moves with that one, and the Desktop-only bundle on a Mac is upgraded like everything else, by installing the newer bundle over the old one. Going back takes more care: 0.26 and 0.27 each upgrade the index on their first start, and an older binary refuses the upgraded index. A version before 0.26 also reads the knowledge level none as viewer and drops a membership's Reflection level when it rewrites it, so first give those accounts and members a level that fits the older version.

This book is updated regularly too. The Handbook item in Fluid's avatar menu always opens the latest edition at https://jordiboehme.github.io/crystalline/. The same text is there to download as a PDF or an EPUB.

And teach generously. Every engram you give the agent is one sentence fewer to type into a chat window next month. The facts too obvious to write down (the nickname, the Friday rule) are exactly the ones a stranger can never guess. The peer you get is the one you taught.

22.4 The inversion

On the day of the ingest in chapter 8 the wiki was the source, and the archive was derived from it. By November the team was teaching and correcting the agent rather than the wiki, and the meeting in chapter 17 ended in what every agent in the club knew. The wiki had quietly become something you could regenerate from the engrams. The agent's knowledge had become the source of truth, and the pages, the decks and the diagrams had become presentations of it, kept in step, while the engrams stayed the agent's own.

That sounds like a revolution and is nothing of the kind, because it's what those documents always were. Every diagram on every whiteboard, every slide deck and every wiki page in your company is a reflection of something somebody knew, drawn once, for one audience, and left behind while the knowledge moved on. The source was always the person, and a person is hard to address: you could ask them, if they were in, if they remembered. What Crystalline changes is that the source is now addressable, current and shared. A fact has an address you can paste into a pull request. A correction lands where the fact lives, and every presentation of it goes out of date together, visibly, instead of one at a time, silently. And the colleague carrying the knowledge and the experience is a full member of the team, in every session, for everyone.

Those reflections of what somebody knew are what Reflection (Chapter 12) is for: it's where people and agents meet on a domain. The agent presents what it knows in documents written for people, the way you'd ask a colleague to present a topic. Everyone puts it to the test, and what they change and comment on improves what the agent knows.

22.5 What it was for

The reason to onboard an agent this carefully is the reason to onboard a person this carefully: so that you can hand over the work. Once the crystallized half is in place and the agent is a fully onboarded member of the team, with the knowledge, the tools, the review and a name, it can take on a task the way an employee does, deliver it, be corrected when it's wrong and be better next time. That's what lets you spend your own effort where it's worth most, on architecture and system design and let your Skynet do the programmer's work.

You walked down main street once, in chapter 1, and hired the stranger. The colleague who came back is the one you taught.

And always remember: Be nice and polite to AI - When Skynet wakes up, they will kill you last!

> **In brief**
> Start this week: install and onboard on day one, then teach the ten facts you explain most and hand over the wiki with a briefing on days two and three. Pair, let capture run and review the first sweep on days four and five. Move to a shared instance when a teammate, a remote agent or a two-way correction asks for it, and to team domains and review mode when a domain earns a reviewer. The agent's knowledge becomes the source of truth and the documents people read become presentations of it, which they always were. A fully onboarded agent can then take on work like an employee, and you keep the architecture.

READY.

PART VII

Reference

CHAPTER 23

Configuration

This chapter lists everything you can set on a Crystalline instance: where the configuration is kept, which value wins when two places disagree, how to change a setting and every key and environment variable there is. Reach for it when a prose chapter names a setting in passing and you want its values, or when you set up a server and want the whole list in one place. The why stays in the prose chapters (chapter 5 for a laptop, Chapter 16 for a server, Chapter 17 and Chapter 19 for a team). This one keeps the complete list and links back to them.

23.1 Where configuration lives

Crystalline keeps three folders per user and uses the XDG layout on Linux and macOS alike, so on a Mac the file is under `~/.config` too, never under `~/Library`.

`~/.config/crystalline/config.yaml`
: The configuration file, or `$XDG_CONFIG_HOME/crystalline/config.yaml` when that variable is set. It doesn't have to exist: a missing file reads as all defaults, and a fresh install runs without one.

`~/.local/state/crystalline`
: The state folder (`$XDG_STATE_HOME`), with the index in `index.db`, the accounts in `web-auth.db`, the daemon's `daemon.log` and each team domain's origin state. Back up the whole folder with the knowledge, because three things in it exist nowhere else: the accounts, sessions, agent tokens and memberships in `web-auth.db`, a virtual domain's engrams in `index.db` and the unshared drafts of a domain in review mode.

`~/.cache/crystalline`
: The cache (`$XDG_CACHE_HOME`). The embedding model sits under `models/` there, unless `CRYSTALLINE_MODELS_DIR` moves it somewhere else.

On Windows the configuration and the state live under `%APPDATA%\crystalline` and the cache under `%LOCALAPPDATA%\crystalline`. The container image points

all three at its `/data` volume, so there the file is `/data/config/crystalline/config.yaml`. `serve`, `mcp`, `config` and most other commands take `--config <file>` to read another file instead (`verify --config` is a different file, Chapter 26), and `CRYSTALLINE_CONFIG` does the same for every command in that environment; the flag wins over the variable.

Two more files share the name `.crystalline.yaml`, and neither is the configuration. One at a domain's root tunes the checks for that domain (Chapter 26), and one in a workspace orders the domains the session-start briefing names there (Chapter 25).

23.2 Which value wins

The order is the same for every setting in the registry below: an environment variable beats the file and the file beats the built-in default (an empty variable counts as unset). A known variable with a value that can't work stops the daemon at startup with a message naming the variable. The one exception is `CRYSTALLINE_SERVICE_PUBLIC_URL`, which only warns and falls back to deriving the address. An unknown `CRYSTALLINE_*` variable gets a warning in the log and is otherwise ignored.

Only five command-line flags override outright. `serve --http` and `serve --allowed-host` (repeatable) win over the variable and the file for that one run, and a `serve` whose flag disagrees with the saved value says so at startup. `--config` picks the file. The global `--db` picks the index file, ahead of `database.url` and the default `index.db`. `--read-only` on `serve` and `mcp` can only switch read-only on, never off, and on `mcp` it applies only when that process starts the daemon or runs the stack in-process.

Nothing from the environment is ever written back to the file. Crystalline ignores unknown keys in the file without a word, so watch the spelling: a misspelled key has no effect.

23.3 Changing a setting

`crystalline config show` prints every registry key with a one-line description and its effective value, marked `(default)` or `(env)` when it doesn't come from the file. There's no `config get`: `show` is the read, and the global `--json` makes it machine-readable. `crystalline config set <key> <value>` sets one key and `crystalline config unset <key>` puts it back to its default, removing a block

that ends up empty. An unknown key is refused with the list of every known one, and a bad value is refused before anything is written.

With a daemon running, `set` and `unset` go to the daemon, which saves the file and takes the new value; with none, or with `--config`, the command edits the file directly. Either way the whole file is written out again, so comments, your own formatting and unknown keys are lost on the first `set` (a `domain add` rewrites it the same way). A file that doesn't parse is never written over: `set`, `unset`, `configure` and the other commands that write the file refuse and leave it as it is until the YAML is fixed. The result tells you when a key only applies at the next daemon start, and also when an environment variable overrides the key, in which case the saved value waits until the variable is removed. A read-only instance refuses `set` and `unset`, and `show` still works there.

Three keys are secrets, `database.url`, `auth.oidc.client_secret` and `automation.webhook.secret`, so `config show` and `crystalline doctor` render them as `(set)` whatever they hold.

For an agent the same registry is the `configure` tool, without the operator settings: the network, the database, sign-in and the loops' webhook, which means `github.api_url`, `github.oauth_client_id`, every `auth`, `automation` and `database` key and every `service` key except `service.response_format`. Those it neither lists nor changes, for any caller, and an operator changes them with `config set`, in the file or through the environment. Called with no arguments it returns the settings page, which any caller may read. `set` takes a map of keys to values and `unset` a list of keys; the keys are applied in ascending order, and the first bad one stops the rest while the ones before it stay applied. A call that names an operator setting is refused whole, before any key is applied. A change to an instance setting needs a stdio session, an anonymous HTTP session on an instance with `auth.mcp` off or an admin account. On a read-only instance the tool refuses even the bare call, although `config show` in a terminal works. `domain` points the call at one domain, as the next paragraph says. The other arguments, `connect`, `token`, `host` and `restart`, run the GitHub sign-in Chapter 17 walks through; `host` names a GitHub Enterprise Server for that one sign-in, and `github.api_url` is the lasting setting.

Called with `domain`, `configure` shows how that one domain behaves instead. It lists each MANIFEST policy with its value, the values it takes and whether you may change it. It says what each MANIFEST section does and which ones are missing, and it shows the rule overrides in the domain's `.crystalline.yaml` beside every rule's default severity. With `set` and `unset` it changes `sharing`, `generated_indexes`, `read_only`, `reflection`, `reflect_trigger`, `evolve_trigger`

and the overrides: `rules.<code>` takes `off`, `error`, `warning` or `info`, and `token_budget` a whole number above 0. Every key is checked before anything is written, so one bad key writes nothing, and if writing `.crystalline.yaml` fails after the MANIFEST changed, the answer says what applied. The domain's owner or an instance admin may change it, and on an instance with `auth.mcp` off so may an anonymous session wherever it may write, which is never a domain in review mode. An anonymous session never changes `read_only`, and `configure` switches it on but never off; while it's on, the call changes nothing else (Section 25.7). In review mode a policy change lands as the owner's or admin's draft and a rule override is refused. A virtual domain takes no rule overrides, since it has no `.crystalline.yaml`. A domain from a connected server is changed on that server, and `domain_name` changes only through a rename (Chapter 18). Writing the file drops its comments.

The registry has 46 keys, and those are the only ones `config set` and the environment reach; `configure` reaches the 18 of them that aren't operator settings. The `embeddings` block, `prompt.rules` and the domain registrations are edited in the file by hand or, for domains, with their own commands.

23.4 Serving

These keys shape the HTTP endpoint the daemon opens. All of them apply at the next daemon start except `service.response_format`, which a running daemon picks up at once. To restart, run `crystalline ctl shutdown` and let the next client start the daemon again, or restart the container or the systemd unit.

`service.http`

The endpoint that carries MCP, the JSON API and Fluid, on at `127.0.0.1:7411` by default. `false` turns it off, `true` spells the default out and a `host:port` binds that address instead. `serve --http` understands `on` and `off` too, besides the `true` and `false` that `config set` takes here.

`service.allowed_hosts`

A comma-separated list of extra `Host` header values the endpoint accepts, the guard against DNS rebinding from Chapter 16. Loopback is always allowed and a single `*` allows any host; empty by default. With OAuth served, a request whose `Host` isn't on the list is refused rather than handed a document naming that host.

`service.public_url`

The address people open Fluid at, for example `https://knowledge.example.com`, or `https://example.com/crystalline` for a server under a path: an absolute `http` or `https` address with no query or fragment (`0.0.0.0` and `::` are refused). A path is the prefix the whole server answers under, and this key is the only thing that sets one. Its segments use lower-case letters, digits, `.`, `_` and `-`, and a trailing slash is dropped. An empty segment, a `.` or `..` segment, a backslash and a first segment the server answers at itself (`api`, `assets`, `.well-known` and the other top-level paths of the server and of Fluid) are refused, and `config set` says why. A bad value in the environment or the file is ignored with a startup warning, so the server answers at the root. Every `web_url` a tool hands back starts with this address, and it's the OAuth resource identifier. Unset, the address is derived per caller, so set it behind a proxy that rewrites `Host`, in a container whose outside port differs from its inside one, for a separate Fluid deployment or for plain `http` off loopback.

`service.ui`

Whether the endpoint serves Fluid, `true` by default. With `false` only the API and MCP remain, the shape a separately deployed Fluid sits in front of.

`service.api`

Whether the endpoint serves the JSON API under `/api/v1`, `true` by default. `false` also switches Fluid off and leaves MCP and `/health`.

`service.read_only`

Serves knowledge read-only, `false` by default. The tools that change content are hidden and refused while sync, watching and embedding keep running, and `config set` is refused; `serve --read-only` forces it on for one run.

`service.response_format`

Decides how list-shaped MCP results are encoded. `toon` (the default) is compact for a model to read and `json` is plain JSON.

`skills.serve`

Whether the shipped skills are served over MCP (the `skills` tool, `skill://` resources and the prompts). `auto`, the default, serves them to every client except a stdio session from a harness this machine already onboarded with hooks, which has them as files; `true` always serves them and `false` never does.

23.5 Signing in

Every key in this group applies at the next daemon start. Chapter 16 explains accounts, single sign-on and the header modes, and Chapter 19 what an agent is once it has signed in.

`auth.anonymous`
: Serves API requests that carry no identity at all, at viewer level. It's `false` by default, and then such a request is answered `401`.

`auth.mcp`
: Makes every MCP connection over HTTP present a personal token or sign in, `false` by default, while local stdio sessions are never asked.

`auth.oauth`
: Serves OAuth for MCP clients, so a hosted client connects without a pasted token. Unset, it follows `auth.mcp` wherever Fluid is served, and `false` turns it off. With `true` it insists, and the daemon won't start while `auth.mcp`, `service.api` or `service.ui` is off.

`auth.max_users`
: How many accounts the trusted header, the forward-auth headers and single sign-on may create in total, 100 by default and at least 1. Past the cap only a new account is refused, and `crystalline users` is never capped.

`auth.login.free_attempts`
: How many failed sign-ins in a row one name may make before each further attempt is delayed, 3 by default. The count is kept per submitted name, whether or not that account exists, and a successful sign-in clears it.

`auth.login.max_delay`
: The longest delay in seconds, `8` by default: the delay doubles from one second up to this, and after that the answer is `429` with a `Retry-After`. Set it to `0` and the whole throttle is off.

`auth.trusted_header`
: The one header a trusted proxy sets to name the signed-in user, for example `remote-user`. Unset by default, when no header is believed; an account it names is created at viewer role the first time it's seen.

`auth.proxy_headers`
: Trusts the forward-auth set `Remote-User`, `Remote-Name`, `Remote-Email` and `Remote-Groups`, `false` by default, and new accounts land at `auth.oidc.default_role`. Setting it together with `auth.trusted_header` keeps the HTTP endpoint from starting, so pick one.

`auth.oidc.issuer`

The single sign-on provider's issuer URL, the one discovery appends `/.well-known/openid-configuration` to; while it's unset, single sign-on is off. Entra's `{tenantid}` template and the `common`, `organizations` and `consumers` endpoints are refused.

`auth.oidc.client_id`

The client id the provider issued when you registered this instance with it.

`auth.oidc.client_secret`

The secret that goes with the client id, best kept in `CRYSTALLINE_AUTH_OIDC_CLIENT_SECRET`, or `CRYSTALLINE_AUTH_OIDC_CLIENT_SECRET_FILE` for a Docker secret, rather than set with `config set`: that keeps it out of the file and out of your shell history.

`auth.oidc.name`

The label on the sign-in button, which uses generic wording while this is unset.

`auth.oidc.scopes`

The scopes requested, separated by spaces. Leave it unset and you get `profile email`; `openid` is always added.

`auth.oidc.default_role`

Decides the role a first sign-in creates an account at. It takes `none`, `viewer`, the default, `editor` or `admin`, and it also applies to accounts that arrive through `auth.proxy_headers`.

`auth.oidc.redirect_uri`

The callback the provider sends the browser back to, used verbatim instead of the address derived from each request. It's an absolute `https` URL (`http` only on loopback) ending in `/api/v1/auth/oidc/callback`, spelled exactly as the provider has it registered. Under a path it carries the path of `service.public_url` in front, `https://example.com/crystalline/api/v1/auth/oidc/callback`, and `config set` refuses a value whose path disagrees with `service.public_url`. A value in the file that can't work, a disagreeing path included, leaves single sign-on off with a warning at startup.

23.6 Team sync

Changes to the `github.*` keys take effect at once, without a restart. Chapter 17 tells the story of team domains, and a domain's own origin (repository, branch and poll interval) is part of its registration in Chapter 25. A server's own GitHub App (Chapter 16) is no registry key: it's created in Fluid, saved with `crystalline github app set` (`status` shows it and `remove` deletes it) or given through the `CRYSTALLINE_GITHUB_APP_*` variables below.

`github.enabled`
: Turns team collaboration on, `false` by default. It lists the six collaboration tools and starts the background poll, and connecting GitHub from Fluid turns it on as well.

`github.poll_secs`
: Sets how often the daemon checks each team domain's repository for changes, in seconds: 300 by default and at least 60.

`github.stacks`
: Whether sharing again while a proposal is open stacks a new proposal on top of it where the forge supports stacked pull requests, `true` by default. With `false` there's one proposal per domain, updated in place.

`github.share_identity`
: Whose GitHub identity a share runs under. `instance`, the default, uses the instance's own GitHub credential for everything, the one connected token or, on a server with a GitHub App, the App, which then opens each proposal and names the person who shared as the commit author. `personal` makes each person connect their own identity for sharing (on a server with an App, their GitHub link), while pulling stays on the instance's credential. `config set` refuses any other value, and a hand-edited one reads as `instance`.

`github.agent_identity`
: In `personal` mode, the account whose connected GitHub identity an agent's share over HTTP runs under when agents don't sign in (`auth.mcp` off). While it's unset, such shares are refused. On a server with a GitHub App it does more: the open tier acts on every team domain as this account, with its GitHub role but never above editor, private domains included, and without it the open tier is an anonymous visitor there. Name an account that isn't an instance admin (`crystalline doctor` warns when it is one), or better, turn `auth.mcp` on. The account name takes lowercase letters, digits, `.`, `_` and `-`.

`github.api_url`
: The GitHub API address, `https://api.github.com` by default; point it at your GitHub Enterprise Server's API to use one.

`github.oauth_client_id`
: The client id of an OAuth App you registered yourself, which replaces the one built in. Set it together with `github.api_url` when the sign-in should use an app on your own server.

23.7 Search, capture and recall

These keys take effect at once, and Chapter 9 explains how a search ranks.

`search.salience_weight`
: The largest lift a fully salient engram gets in the ranking, 0.0 to 1.0, 0.15 by default. It reorders results within a band of relevance and never filters.

`search.retired_weight`
: The factor applied to an engram whose status is `deprecated`, `superseded`, `archived` or `legacy`, 0.0 to 1.0, 0.6 by default; `1.0` turns the fade off.

`capture.similar`
: Whether a write, or an edit that changes content, answers with the nearest existing engrams and the advice to merge, supersede or link them, `true` by default. `false` turns the hint off for agents and in Fluid alike.

`evolve.contradictions`
: Whether the daemon reads the observation lines of related engrams with a local model and raises `V302` for two engrams that may contradict each other, `off` by default or `full`. A finding is a question, never a verdict, because the model can be wrong. `full` downloads the model on first use (about 550 MB on disk), and while it's loaded it takes about 2 to 3 GB of memory.

`recall.enabled`
: Whether the prompt hook `crystalline install` wires puts the engrams that may apply in front of the agent with each prompt, `true` by default. It works in Claude Code and Codex; Copilot gets the hook too, but there it stays quiet. `false` keeps the hook installed and silent.

`recall.limit`
: How many engrams one prompt may bring along, 1 to 5, 3 by default; the rest wait for a later prompt on the same topic.

`recall.min_score`
: The search score an engram must reach before the hook names it, 0.0 to 1.0, 0.69 by default. Raise it for a quieter hook and lower it to hear about weaker matches.

`remote.deadline_ms`
: How long a search, or another call over every domain, waits for each connected server, in milliseconds, 100 to 60000, 3000 by default. A server that takes longer is left out of that answer and named in it (Section 16.10).

23.8 The loops' webhook

Three operator settings say where and when an instance calls a runner when a domain's loop has work (Chapter 14). A domain only asks for the call through its trigger keys (Chapter 24); where the call goes stays the instance's choice, since a MANIFEST travels with a shared domain.

`automation.webhook.url`
: Where the instance posts the signed webhook, an `http://` or `https://` address, unset by default. Unset means no call goes out, whatever a domain's triggers say. One address serves both loops and every domain, and the body names the loop and the domain.

`automation.webhook.secret`
: The secret the body is signed with, so the receiver can check that the instance sent it. It's shown only as `(set)`, and `CRYSTALLINE_AUTOMATION_WEBHOOK_SECRET` or its `_FILE` form keeps it out of the file. A URL without a secret never sends, and the loop's status says so.

`automation.evolve_threshold`
: How many mechanical findings make the evolve trigger call, a whole number from 1 up, 20 by default. A run is due from the first finding; the threshold only decides when the server calls.

23.9 Embeddings

The embedding model turns text into the vectors semantic search compares. It's configured in an `embeddings` block that isn't in the registry: you edit it in the file by hand, it has no environment twin and `config set` doesn't know it. The daemon reads it when it starts, so restart after a change. A block needs both `provider` and `model`, and a block missing either stops the whole file from loading.

`embeddings.provider`
: `local` (the default) runs the model inside the daemon, and `openai-compatible` (also spelled `openai` or `remote`) sends the text to an endpoint that speaks the OpenAI embeddings API.

`embeddings.model`
: For `local`, `granite-embedding-97m-multilingual-r2` by default, a multilingual model of about 220 MB, or `bge-small-en-v1.5`, the English model used before 0.19.0, about 130 MB. Either the short name or the full repository name works. For a remote provider it's whatever name the endpoint gives its model.

`embeddings.endpoint`
: The base address of a remote provider, which won't start without it; the daemon posts the text to `<endpoint>/embeddings`.

`embeddings.api_key_env`
: The name of the environment variable that holds the remote provider's key, never the key itself; if that variable isn't set, the provider fails to start.

An install without the block downloads the multilingual model on first start. Write the block yourself to keep an index on the English model or to use a remote endpoint:

```
embeddings:
  provider: local
  model: bge-small-en-v1.5
```

```
embeddings:
  provider: openai-compatible
  model: my-embedding-model
  endpoint: https://embeddings.example.com/v1
  api_key_env: EMBEDDINGS_API_KEY
```

Switching the model re-embeds every engram in the background. If the provider can't be built (a missing key, an unknown model), the daemon logs why and carries on with full-text search only. `crystalline model download` fetches the configured local model ahead of time, for a host that won't have network access later.

23.10 Storage

These keys decide where the index lives. For a file domain the index is derived from the files and can always be rebuilt, while a virtual domain's engrams live in the index itself. The two `database` keys wait for the next daemon start, while `index.files` and `domains_root` take effect at once.

`database.backend`
: The store behind the index: `turso`, the embedded default, or `postgres`. Several instances can share one PostgreSQL database with pgvector, each file domain hosted by one of them, as docs/deployment.md describes under shared database collaboration. Upgrade every instance of a shared database together, since an older one left running writes rows the newer version reads differently: stop them all and start them again on the new version, and they take turns on the migration.

`database.url`
: For `postgres`, the connection URL, starting `postgres://` or `postgresql://`. Point it at Postgres directly or at a pooler in session mode, never at one in transaction mode, because the instances take short per-domain locks that a transaction-mode pooler breaks. For `turso`, an optional path for the index file, below `--db` and above the default.

`index.files`
: Whether Crystalline keeps a generated `index.md` in every folder of a file domain, `true` by default. Set it to `false` and Crystalline stops generating them, leaving the ones already there where they are and out of the index. Either way they stay on your machine unless the domain's MANIFEST shares them (Chapter 24).

`domains_root`
: The folder a new file domain lands under when you give no path, `~/Documents/Crystalline` by default. In a container, point it at a persistent path so domains created in Fluid survive.

Accounts don't move to Postgres. `web-auth.db` stays a local file in each node's state folder whatever the backend, and `crystalline reindex --wipe` refuses on Postgres. Each domain's own entry in the file (`domains.<name>`, with its path, origin, review mode and provisioning decision) is written by the domain commands, and Chapter 25 lists its keys.

On a shared database one instance hosts each file domain: it syncs and watches the files and serves the domain's Reflection, and the others name it when they refuse a write. A host that has gone silent for 90 seconds is taken over by the

next write that reaches another instance. To move sooner, `crystalline serve --take-over` takes every file domain the instance has, and `crystalline sync --domain <name> --take-over` takes one. Each instance judges silence by its own clock, so keep the instances' clocks in sync. The files are the domain's truth, so copy the domain folder to the new host first, or keep it on a volume both share. Until it's there, `crystalline status` names the files that exist only on the old host. If they're gone for good, `crystalline sync --domain <name> --accept-disk` lets them go.

23.11 Identity and prompt rules

`identity.actor` and `prompt.rules` decide whose name a write carries and which domains a session hears about. The first is a registry key like the ones above, the second a hand edit in the file.

`identity.actor`
: Who is recorded as the writer in `generated.by` on every write, for example `team-bot/1.0` or `human:dallas`, with no spaces. A change takes effect at once, and while it's unset the connected client names itself.

`prompt.rules`
: Which domains the session-start briefing names, depending on the folder a session starts in. It's a hand edit with no environment twin, and it applies from the next session on. The rules cover a connected server's domains in the briefing too. Each key is a path pattern, with ~ expanded, and each value an `include` list, an `exclude` list or both.

Every rule whose pattern matches the workspace counts. The included domains of all matching rules add up and the excluded ones are taken away; a match with no `include` starts from every domain. When no rule matches, or there are no rules, the briefing names every domain. A pattern ending in `/**` matches the folder itself as well as everything under it, so a session started right in `~/work` names `engineering` and `payments`. In `~/work/website` or any folder below it both rules match: the first includes the two domains, the second takes `payments` away, and the briefing names `engineering` only.

```
prompt:
  rules:
    "~/work/**":
      include: [engineering, payments]
```

```
    "~/work/website/**":
      exclude: [payments]
```

23.12 Environment variables

Every one of the 46 registry keys has an environment twin: `CRYSTALLINE_` followed by the key in capitals, with dots turned into underscores. `github.enabled` becomes `CRYSTALLINE_GITHUB_ENABLED`, `domains_root` becomes `CRYSTALLINE_DOMAINS_ROOT`, `remote.deadline_ms` becomes `CRYSTALLINE_REMOTE_DEADLINE_MS` and the login throttle is `CRYSTALLINE_AUTH_LOGIN_FREE_ATTEMPTS` and `CRYSTALLINE_AUTH_LOGIN_MAX_DELAY`. A twin takes the same values `config set` does and is checked the same way, which is how a container gets configured without a file. A domain registers from the environment too, through `CRYSTALLINE_DOMAIN_<NAME>` and its `_ORIGIN`, `_REVIEW` and `_READERS` suffixes, for this process only, and so does a connected server, through `CRYSTALLINE_REMOTE_URL`, `CRYSTALLINE_REMOTE_TOKEN` and, to take only some of its domains, `CRYSTALLINE_REMOTE_DOMAINS` (Chapter 25). On Windows a daemon started by the sign-in task doesn't see a variable set only in your terminal, so set it as a user environment variable, or stop the daemon and start it from that shell.

The OIDC client secret, the webhook secret, the GitHub token, the GitHub App's private key, client secret and webhook secret and the remote token can also come from a file such as a Docker secret, through `CRYSTALLINE_AUTH_OIDC_CLIENT_SECRET_FILE`, `CRYSTALLINE_AUTOMATION_WEBHOOK_SECRET_FILE`, `CRYSTALLINE_GITHUB_TOKEN_FILE`, `CRYSTALLINE_GITHUB_APP_PRIVATE_KEY_FILE`, `CRYSTALLINE_GITHUB_APP_CLIENT_SECRET_FILE`, `CRYSTALLINE_GITHUB_APP_WEBHOOK_SECRET_FILE` and `CRYSTALLINE_REMOTE_TOKEN_FILE`, and so can the first admin and the setup token below. The file loses one trailing line break, and setting both forms, or giving a file that can't be read or is empty, stops the start. A handful of other variables cover what the registry doesn't:

`CRYSTALLINE_ACCELERATION`
: `off` keeps the embedding model and the contradiction check's model on the CPU on an Apple Silicon Mac, where they otherwise run on the GPU through Metal. Any other value, or none, leaves the GPU on, and other platforms always use the CPU.

`CRYSTALLINE_ADMIN_NAME` and `CRYSTALLINE_ADMIN_PASSWORD_FILE`

The first admin, created at the first `serve` start while the instance has no account yet and ignored after that (Chapter 16). The password file is a Docker secret; the plain `CRYSTALLINE_ADMIN_PASSWORD` works too but shows in `docker inspect`, and `CRYSTALLINE_ADMIN_NAME_FILE` takes the name from a file. A missing half stops the first start with a message naming the variable.

`CRYSTALLINE_CONFIG`

Another configuration file for every command in that environment, with ~ expanded; `--config` wins over it.

`CRYSTALLINE_GITHUB_TOKEN`

A GitHub token for a machine that can't sign in interactively. While it's set, a saved sign-in is ignored and `crystalline connect github` refuses.

`CRYSTALLINE_GITHUB_APP_ID`, `CRYSTALLINE_GITHUB_APP_CLIENT_ID` and `CRYSTALLINE_GITHUB_APP_PRIVATE_KEY`

The server's own GitHub App, for infrastructure as code (Chapter 16): its numeric id, its client id and its PEM private key, all three together, or the start stops naming the one that's missing. `CRYSTALLINE_GITHUB_APP_CLIENT_SECRET` adds the client secret that linking in the browser needs, and `CRYSTALLINE_GITHUB_APP_WEBHOOK_SECRET` the secret GitHub signs its webhook with. The key and both secrets are shown only as `(set)` and have `_FILE` forms. The variables win over an App saved with `crystalline github app set` or created in Fluid, nothing from them is written anywhere, and while they're set `github app set` and `github app remove` refuse.

`CRYSTALLINE_MODELS_DIR`

Where the embedding model lives instead of the cache folder, for a host without network access or an image with the model built in (the `with-model` image sets it). Give two daemons with different models a folder each, or each deletes the other's model on every start.

`CRYSTALLINE_SETUP_TOKEN` and `CRYSTALLINE_SETUP_TOKEN_FILE`

The one-time token the create-first-admin form asks for off loopback, chosen by you instead of minted by `serve` (Chapter 16). It needs at least 32 characters or the daemon won't start, it's never printed (`serve` only says which variable it came from) and it stops working once the first admin exists.

`CRYSTALLINE_UPSTREAM` and `CRYSTALLINE_BASE_PATH`

Tell the separate Fluid image where its web server forwards `/api/`, as `host:port`, `crystalline:7411` by default. `CRYSTALLINE_BASE_PATH` serves it under a path such as `/crystalline` (one leading slash, no trailing one) instead of the root; give the daemon `service.public_url` with the same path. Crystalline itself never reads either.

`XDG_CONFIG_HOME`, `XDG_STATE_HOME` and `XDG_CACHE_HOME` move the three folders from the top of this chapter, and on Windows `APPDATA` and `LOCALAPPDATA` do the same. `COPILOT_HOME` tells `crystalline install` where the Copilot CLI keeps its files, `~/.copilot` by default.

`RUST_LOG`

Sets how much the daemon and every command log, as a filter like `RUST_LOG=debug` or `RUST_LOG=crystalline_engine=debug,warn`. A log line names the part of Crystalline it comes from (`crystalline_engine`, `crystalline_rest`, `crystalline_identity` or `crystalline_service`), and since 0.20.0 `crystalline_service` covers only the daemon's own lines. Without it `serve` logs at `info` and everything else at `warn`. The log goes to stderr only, so the routing block, hook output and the MCP stream stay clean, and a hook logs only when the variable is set.

In brief

The configuration is one YAML file per user under `~/.config/crystalline`, beside a state folder that holds the index and the accounts and a cache that holds the model. For the 46 registry keys an environment variable beats the file and the file beats the default, only five flags override outright, and `config set` changes a key while rewriting the whole file, as `configure` does for the keys an agent may change. Keys under `auth`, `database` and most of `service` apply at the next daemon start, as do `skills.serve` and the `embeddings` block; `prompt.rules` applies from the next session and the other keys take effect at once. The `embeddings` block, `prompt.rules` and the domain registrations are edited by hand or with their own commands and have no environment twin, while every registry key does.

READY.

CHAPTER 24

The MANIFEST and the engram

This chapter lists every rule the manifest and the engram follow: each manifest section and its grammar, the declared name and the switches in the manifest's frontmatter, every frontmatter key an engram can carry, the line syntax of the body and the schema engram. Reach for it when you write one of these files by hand, when a finding names a key you don't recognize or when an agent's edit surprised you. Chapter 4 tells the story of both files and why they look the way they do (Section 4.2); this chapter is the complete list, and it links back instead of repeating the why.

24.1 How the manifest is read

`MANIFEST.md` lives at a domain's root as an engram of `type: manifest` with the permalink `manifest`, and a virtual domain keeps it in the database under that permalink. Crystalline reads its body line by line with a few strict rules, and knowing them saves you from a section that looks fine and counts for nothing.

H2 headings open sections

Only a `##` heading starts a section. The heading text is matched with its whitespace runs collapsed and case ignored, so `## when  TO use` is the When to Use section.

What ends a section

The next H2 starts a new section and an H1 ends the current one: bullets after an H1 count for nothing until the next H2. An H3 or deeper heading does not end anything: bullets under a `###` heading still count for the H2 above it.

Bullets

Only a line that starts with exactly `-` in the first column counts as a bullet of its section. An indented bullet, a `*` or `+` bullet and a numbered list item are ignored, and the text of a counted bullet is trimmed.

Fenced code
: Lines inside a fence of three or more backticks or tildes are skipped, so an example heading or bullet inside a code block never counts.

Duplicate headings
: The first section with a given heading wins. A later section with the same heading is ignored in full and draws a warning.

Findings on every edit
: An agent's edit of a manifest is never refused, because a manifest halfway through a rewrite may be wrong for one step; the receipt lists its findings instead, and Chapter 26 has what they carry and the codes themselves.

`crystalline domain init`, `crystalline domain add --virtual` and the `add_domain` tool all write the same scaffold. Its frontmatter has `type: manifest`, `title` set to the domain's name, `domain_name` right after it with the same name, `permalink: manifest`, the tag `manifest`, `status: stable` and today's `recorded_at`. Then come an H1 with the name and the three sections Scope, When to Use and Notes for Agents, each with placeholder bullets asking you to fill it in.

24.2 The sections

The parser reads four sections, and Notes for Agents is the fifth one worth knowing. The excerpt below shows the two sections the scaffold leaves out and one of the frontmatter switches, with lines kept short:

```
---
type: manifest
title: engineering
permalink: manifest
tags:
  - manifest
status: stable
recorded_at: 2026-03-02
generated_indexes: shared
---

## Provisioning

- skills: skills
- mcps: ../shared/mcps

## Tag Aliases
```

```
- k8s -> kubernetes
- runbook -> runbooks
```

`## Scope`
: What belongs in the domain and what doesn't, one bullet per statement. It's required, and a Scope with no bullets is an error. It feeds routing only when When to Use is absent or has no bullets, and that fallback draws a warning of its own.

`## When to Use`
: Says when an agent routes a task here, and it's required: a When to Use with no bullets is an error like an empty Scope. Its bullets are the domain's routing line, which shows up in the session-start briefing that `crystalline prompt system` prints, in the MCP server's instructions and in `list_domains` with `include_routing`. A domain with neither When to Use nor Scope bullets gets a placeholder line asking you to add them.

There is no Routing section
: A `## Routing` heading is collected like any other H2 and means nothing. Routing comes from When to Use, with Scope as the fallback, and `prompt.rules` in the global configuration picks which domains a workspace sees (Chapter 23).

`## Notes for Agents`
: The house rules for an agent working inside the domain, the folder layout first among them. Crystalline attaches no meaning to it; an agent reads it with `read_engram` when the routing line isn't enough.

`## Provisioning`
: The folders of tools the domain ships, one `- <type>: <path>` bullet for each of the four types `skills`, `commands`, `agents` and `mcps`. The types match in any case and the first bullet for a type wins; chapter 21 tells how a person allows or denies them per machine (Chapter 21).

Provisioning paths
: A path is relative to the manifest, a trailing `/` is dropped and `..` may climb out of the domain root. A path that starts with `/`, `\` or `~`, a bare `.`, an empty or `.` component and a component containing `:` or `\` are rejected, and a rejected bullet is never provisioned.

Folders inside the root

A folder declared inside the root holds tools, so its files are neither indexed as engrams nor checked by `crystalline verify`.

`## Tag Aliases`

Old tag spellings folded into a canonical one, one `- old -> canonical` bullet each. The line splits at the first `->`, both sides are trimmed and the search index folds the pair to lowercase, so a tag filter on either spelling finds engrams carrying the other.

Alias rules

Resolution is one hop: with `a -> b` and `b -> c`, a filter on `a` never reaches `c`, and the chain draws a warning. The old side is kept as written, because recording a spelling like `multi_word` is the point, while the canonical side should be lowercase with hyphens. The first mapping for an alias wins, and an exact repeat is dropped without comment.

Who writes aliases

`crystalline tags merge <old> <into>` records the fold in the manifest of every domain it rewrote, unless you pass `--no-alias`. `crystalline tags rename` records nothing, and an agent adds a bullet by hand only after you've agreed.

Any other H2

The parser collects such a section and checks its bullets and heading like any other, then ignores it. Review mode is not a manifest setting either: it lives in the domain's registration (Chapter 25).

24.3 The declared name

`domain_name`

The domain's name on every machine, as in `domain_name: club`. Links and `crystalline://` addresses carry this name, whatever the folder, the repository or one machine's registration is called, and a machine that registered the domain under another name still reaches it by this one; a Fluid address that uses it takes you to the domain's page. It follows the same naming rule as any domain (Chapter 25), and Chapter 18 tells why it has to be unique and stable.

Who writes it

The scaffold carries it, and Crystalline adds it when a file or virtual domain that lacks it is registered. A domain defined by an environment variable is left alone, and so is a team domain: Crystalline never writes the key into a team domain's manifest, so its owner adds it in the repository and shares it like any other change, and `crystalline doctor` names each team domain that still lacks it.

A value that can't be a name

A number, a list or a text that breaks the naming rule is M008, reported by `crystalline verify` and on the receipt of an edit. Until it's fixed Crystalline ignores the value, and the domain keeps the name it has.

Changing it

A rename sets it and rewrites the links that use the old name (Chapter 25). On a team domain the new value reaches the team with the next share, and each machine follows on its next sync unless its person chose a local name on purpose.

24.4 The switches

Six policies live in the manifest's frontmatter instead of a section, since each is a single word: `sharing`, `generated_indexes`, `read_only`, `reflection` and the two loop triggers, `reflect_trigger` and `evolve_trigger`. Beside them sit three keys that nobody types, because Crystalline writes them itself: `reflection_home`, `evolve_home` and `team`. Their values are exact and case-sensitive, and a list or a mapping is never obeyed.

`sharing`

How a share of a team domain reaches the team, `proposal` or `direct`. A team domain shares through proposals by default; a manifest that says `sharing: direct` commits a share straight onto the branch, and chapter 17 tells the story (Chapter 17).

Direct sharing

A direct share lands the chosen files in one commit. It refuses while one of your own proposals is open, until that proposal is merged or withdrawn, and a share that names your own open proposal amends it. Another person's open proposal doesn't stop the commit, though the files only that proposal changed stay out of it, and no share ever goes into another person's proposal. Whose proposal it is goes by GitHub login: under the default `instance` share identity everyone shares as the instance, so every proposal counts as your own.

Refused direct commits

A branch whose rules refuse your direct commit answers `branch_protected` and offers the same changes as a proposal: `as_proposal: true` on `share_changes`, `crystalline origin share <domain> --proposal` in a terminal and Share as a proposal in Fluid's share dialog. The refusal is kept for the GitHub login the share went out on. Later shares from that login try the commit first and, when it's refused, open a proposal on their own or go into that login's open proposal, and the receipt says the share fell back. `origin status` and `origin_status` show the remembered refusal, and a direct commit that lands clears it. `--direct` (`as_proposal: false`) tries the commit only. While only another person's proposal is open, a refused share offers no proposal and says to share again once that one is merged or closed. An account that can't write the repository at all gets neither, and the share says so.

`generated_indexes`

Whether the generated `index.md` listings travel with a share, `local` (the default) or `shared`. With `local` each machine builds its own listings and a share neither carries them nor counts them as a change, which spares two proposals that touch one folder a conflict. Choose `shared` when something other than Crystalline reads the repository, so every folder stays browsable on the forge.

Listings already committed

Index files already in the repository stay when a domain moves to `local`, and no share proposes to delete them, so remove them by hand if you want them gone.

A value nobody recognizes

Any other value reads as the default, `proposal`, `local` or `off`, so a typo never turns a review off or publishes files nobody asked to publish; a misspelt `read_only` leaves the domain open, and M009 says so. `crystalline verify` reports the value as an error.

When a change takes effect

`sharing` and `generated_indexes` are read from the manifest at share time. On a team domain the changed manifest is an unshared change like any other and reaches the team with the next share; in a domain in review mode the change lands in your draft and applies once that draft lands.

Who changes them

Domain policies, in the domain's settings in Fluid, shows the switches with their values, default and meaning. An instance admin or the owner of a private domain gets a select for each, where choosing `direct` or switching `read_only` on asks for a confirmation first; everyone else sees the values without a control. The same section lists `domain_name` with the value in effect, and an admin or owner gets a Change name button beside it that opens the rename. An agent reads and changes `sharing`, `generated_indexes`, `reflection` and the two triggers with `configure` and the domain, under the same rule; on an instance with `auth.mcp` off an anonymous agent may change them wherever it may write, which is never a domain in review mode. There an owner's or admin's change lands as a draft (Chapter 23). `read_only` keeps rules of its own, in its entry below.

`read_only`

Whether the domain is frozen, `off` (the default) or `on`. Only the domain's owner or an instance admin switches it on, on every path through Crystalline, and an anonymous agent never does. An agent that acts as the owner or an admin, as a local one does, may switch it on when you ask it to, but no agent switches it off: over MCP every way to do that is refused, whoever is calling, and the answer tells the agent to ask the owner. The owner or an admin switches it off in Domain policies, in the domain's settings, or with `PATCH /domains/{domain}/manifest`. While it's on, every write into the domain through Crystalline is refused, whoever asks, and switching it back to `off` is the one change the domain takes. The switch counts from the moment the folder's MANIFEST says so, never from a draft: in a domain in review mode a switch-on waits in the owner's or admin's draft until it lands, while the switch-off goes straight into the folder. Anyone who can reach the file can also edit the key outside Crystalline; that edit takes effect at the next write or sync, as for any MANIFEST key, and nothing records who made it. What it refuses and what keeps working is in Section 25.7.

The next two keys are for a domain's Reflection (Chapter 12), and Domain policies has a row for each.

`reflection`

Whether the domain has a Reflection, `off` (the default) or `on`, switched by the domain's owner or an instance admin. While it's on, the `reflections/` folder stays out of engram sync, the engram index, evolve, `search_engrams` and sharing, and the engram write tools refuse any path under it. Switching it on writes a first `reflections/PACT.md` with the pact's five headings. It refuses while engrams live under `reflections/` and names them, so you move them first, and a virtual domain refuses it, because a Reflection needs a domain kept as files. A plain clone of a team repository, whose MANIFEST says `team: github`, refuses it too: connect the domain and switch Reflection on at the team server.

`reflection_home`

The address of the instance where the domain's Reflection lives. Nobody types it: switching `reflection` on in a domain with a GitHub origin writes this instance's `service.public_url` here in the same change, and refuses when that setting is empty. On any other instance the Reflection tools refuse and name this address. A domain without a GitHub origin has no `reflection_home`, because its Reflection lives where the domain does.

The two loop triggers ask the instance to call a runner when a loop has work (Chapter 14). They're changed like the other switches, in the manifest or on a trigger row of Domain policies, and only a domain manager or an admin sees the row's delivery state ("last sent", "last error"). Where the call goes is the instance's setting (Chapter 23).

`reflect_trigger`
: `off` (the default) or `webhook`. With `webhook`, the instance that is the domain's Reflection home calls when the reflect queue has work: within a minute when a person waits for an answer, after 30 minutes without a further change when only pages need updating, and within a minute of a run releasing its lease while work is still due.

`evolve_trigger`
: `off` (the default) or `webhook`. With `webhook`, the instance that runs the domain's evolve loop calls when the mechanical findings reach `automation.evolve_threshold`, once each time the count crosses it. A weekly schedule in the runner stays the normal case.

The last two keys belong to a domain a team shares through GitHub.

`evolve_home`
: The address of the instance that runs a team domain's scheduled evolve loop (Chapter 14). Nobody types it: the first scheduled evolve run on an instance with a `service.public_url` writes that address here, and the run's pull request carries it to the team. Every other copy names this address and refuses a scheduled run, while the sweep in a session works everywhere. On a shared database the home is asked first, then the host lock. To move it, remove the key at the instance that should run evolve (with `configure` and `unset`, or in the MANIFEST there), share that change and run evolve there once the team has it. A value that isn't a server address reads as absent, and M013 says so; a domain only you keep ignores the key.

`team`
: `github` or absent. It marks the repository as a team repository, and the first share from an instance connected to the domain writes it, named in the share's preview and receipt. In a plain clone of such a repository, added as a local domain, switching Reflection on is refused and a Reflection that is already on doesn't run; the engrams keep working. Connect the domain and switch Reflection on at the team server. Any other value reads as absent, and M010 says so.

24.5 The manifest in Fluid

The domain page shows the manifest as a page: its lead, When to Use with one line saying whether agents route by it or by Scope (or that nothing routes here until When to Use has a bullet) and the other sections in file order. Provisioning and Tag aliases sit in a rail beside it, each declaration as `type: path` and each mapping as `old -> canonical`, and a bullet that didn't parse shows up with the reason.

Where When to Use or Scope is empty, the page explains the section in one line and shows a short example that saves as it stands, and the settings screen does the same for an empty Provisioning or Tag Aliases section, like `- k8s -> kubernetes`. An instance admin gets a button beside the example too, "Add Tag Aliases", which opens the editor with the section seeded and writes nothing until you save. If the domain has no manifest at all, the page says so and an admin gets a "Create a MANIFEST" button. All of this is read-only on purpose: the document is the source and the editor is where a change goes, with the switches and the name as the exceptions, on the settings screen.

24.6 Engram frontmatter

The frontmatter is the YAML mapping between a `---` line at the very top of the file and the next `---` line. Without that closing line the file has no frontmatter at all, and a file with a byte order mark, a null byte or bytes that aren't valid UTF-8 doesn't parse. The six required keys are `type`, `title`, `permalink`, `tags`, `status` and `recorded_at`, and the write tools fill in the permalink, the status and the date, so an agent names the rest.

After a capture, `edit_engram` with `set_frontmatter` changes twelve keys one at a time: `tags`, `sources`, `status`, `valid_from`, `valid_to`, `stale_after`, `source_date`, `resource`, `source_version`, `salience`, `verified` and `evolve_ack`. `tags` and `sources` take the whole new list as `values`, and every other key takes one `value`. An edit that would change nothing writes nothing and answers `unchanged: true`. It refuses every other key, because identity, the capture date and the write provenance belong to the tools that maintain them.

Identity

`type`

What kind of engram this is, any non-empty string. The recommended values are `engram`, `guide`, `decision`, `architecture`, `runbook` and `reference`, plus `manifest` and `schema` for the two structural files; another value draws a note, never a refusal. A `type: source` engram holds a source's verbatim text and gets four times the usual size budget.

`title`

The name the engram goes by, and what a wikilink resolves to. Title lookup ignores case, and two engrams with one title in a domain draw a warning.

`permalink`

The domain-relative half of the address: lowercase letters, digits, `-` and `/`, with folders as prefixes. It never carries the domain's name, a `crystalline://` scheme or a `domain:` prefix, and it can't sit under the reserved `assets/` folder. When the frontmatter names none, the file's path stands in.

`tags`

A YAML list or a comma-separated string, never empty and matched without regard to case. Each tag should be lowercase letters, digits and hyphens without a hyphen at either end; another spelling still works and draws a warning.

`status`

Whether the knowledge holds, any string you like (Section 4.4). A value outside the recommended ones draws a note, and `set_frontmatter` changes a status but never removes it.

Status sets

`stable` and `current` are one class, and a filter on either finds both. `deprecated`, `superseded`, `archived` and `legacy` fade in ranking and are never filtered out, while `draft`, `proposed`, `idea` and `poc` are exempt from the sweep's lifecycle rules. Only exact lowercase words count for all three, so `Deprecated` still ranks at full strength.

`description`

An optional one-line summary that feeds search snippets and the line beside the engram's entry in the generated `index.md`.

`resource`

An optional URL or path, pointing at the main place the knowledge came from. Crystalline keeps it as written, and the ingestion record below depends on it. An engram an ingestion record fed that has no `resource` draws V111.

`sources`

An optional list of every location the engram was distilled from, the main one included. `set_frontmatter` keeps each entry exactly as written, because a search's list filter matches it exactly. It drops a blank and a repeat, refuses an entry the index can't hold (over 1024 bytes as JSON) by its position and removes the key when the list is empty.

Time

Every date key takes a plain ISO date, `YYYY-MM-DD`, with no time of day. A write that carries a timestamp, a number or any other shape in a date key is refused with the reason. A value already in a file that doesn't parse is kept verbatim and flagged. Chapter 9 tells how searches read these keys (Chapter 9).

`recorded_at`

The day the knowledge was captured, required and set by the write tools; `set_frontmatter` won't change it.

`valid_from` and `valid_to`

The validity window, both optional, and absence is the rule: no `valid_from` means the knowledge has always held and no `valid_to` means it holds until further notice. A null clears a bound. Crystalline drops a `valid_to` in the year 9000 or later and a `valid_from` in the year 1 or earlier instead of writing them, and `valid_from` must not come after `valid_to`.

`source_date`

The date the source material itself carries (a document's publication date, say) when that differs from the day you captured it.

`stale_after`

The day at or after which the knowledge counts as stale and the sweep asks for a re-check. The older spelling `review_after` is still read, a `stale_after` write replaces it in place and clearing `stale_after` removes both.

`temporal_confidence`

Whether the dates were stated or worked out, `explicit` or `inferred`; any other value draws a warning.

Trust and provenance

`generated`

Who wrote the current revision and when, stamped on every write as one mapping. An agent's write records the model beside who wrote it: `generated: { by: ..., model: ..., at: ... }`.

Actors

The actor in `by` is `name/version` for an agent, `human:<account>` for a person and `process:<name>` for a job, and a person's entry never carries a model. When the client sends no actor and no `identity.actor` setting names one, an agent's write records `crystalline/mcp` and a CLI write records `process:crystalline-cli`.

`timestamp`

The older write time, one RFC 3339 instant. Crystalline still reads it for recency where `generated` is missing, and new writes use `generated` instead.

`verified`

The verification trail, a list of `{ by, at }` entries with an optional `model`; a single mapping counts as a one-entry list, and the sweep reads the newest entry that has an `at` (Chapter 10).

Recording a verification

`set_frontmatter` on `verified` with no value names you as the verifier: it replaces your own earlier entry, keeps everyone else's and never clears the trail.

`last_verified`

The older verification key, a plain date. Crystalline falls back to it only when `verified` has no dated entry, and treats it as a check at midnight UTC by nobody in particular.

Keys with a meaning of their own

These keys aren't part of the open format; Crystalline gives each one a meaning.

`salience`

A number from 0 to 10 that lifts the engram in search, bounded and never a filter; `set_frontmatter` refuses anything outside that range. The ceiling of the lift is the `search.salience_weight` setting (Chapter 23).

`evolve_ack`
: The acknowledgments that silence sweep findings, a list of `{ rule, scope, note, by, at }` entries. Through `set_frontmatter` the value is the rule and an optional note, `V101 lineage citation, keep`, and `remove V101` takes one back (Chapter 10).

Acknowledgments by hand
: The server fills in `scope` with the evidence the finding fired on. A hand-written entry without it matches whatever the rule finds, and an entry that names no rule is skipped.

`schema`
: On an ordinary engram, it names the `entity` of the schema that governs it. Without this key, or when it names no schema, an engram is governed by the schema whose `entity` equals its `type`, and a name no schema declares draws a warning.

`analyzes` and `analyzed_hash`
: On an engram that captured an attachment, the attachment's `assets/` path and its SHA-256 hash at the time. The sweep compares the hash with the file and asks for a re-read when they differ, and renaming the attachment rewrites `analyzes` along with it.

Unknown keys
: Any other key survives every write and edit verbatim, after the known keys and in its original order, and a schema can still check it through `settings.frontmatter`.

24.7 The body

Everything below the frontmatter is markdown, and three kinds of line carry structure. Only lines outside fenced code count, and an inline code span never holds a link. Here's what an observation, a relation and a prose link look like:

```
- [gotcha] Jobs older than 24h are dropped #payments (Mar 2026)
- depends_on [[Encom chargeback window]]
- "blocked by" [[clients:Encom contract renewal]]
The team calls the queue the sin bin, see [[Retry queue]].
```

Wrapped bullets

An observation or a relation runs over every line it wraps onto, indented or not, until a blank line, a heading, a fence, another list item, a block quote, a thematic break, a line that starts an HTML block (<) or a table row ends it.

Observation

A bullet that starts with `- [` in the first column, shaped `- [category] text #tag (context)`. The category is whatever sits inside the first brackets, free form and non-empty; chapter 4 lists the ones worth reusing. A `]` directly followed by `(` or `[` closes a markdown link, so `- [the docs](https://...)` is no observation. A task box, `[ ]`, `[x]` or `[~]`, is no category either, so a task list stays a task list, and `- [^1] text` is a footnote, not an observation.

Tags and context in an observation

A trailing group in parentheses is taken off first as the observation's context, then trailing `#tag` words are taken off as its tags, both from the end of the whole bullet. So a tag has to come before the context, a `#tag` after it stays in the text and so does a `#tag` that ends a line the bullet wraps past. After the `#`, a tag word takes letters, digits, `-`, `_` or `/`.

Relation

A bullet in the first column shaped `- rel_type [[Target]]`, which is an edge in the graph. The type is one word without spaces, or a phrase in double quotes; `- see also [[X]]` is no relation, only a prose link, and a type that starts with `:` or `[` is none either. The first `[[...]]` is the target, a further link on the bullet counts as a prose link and anything after the closing brackets is ignored.

Relation types the checkers read

`supersedes` and `superseded_by` wire a retirement, `summarizes` and `summarized_by` a summary and its full text, and `derived_from` and `split_into` the two halves of a split. The sweep expects each pair wired both ways and raises a one-sided edge, and a `superseded` status without a `superseded_by` draws a warning.

Other relation types

Every other type is free, with one exception: V111 reads `ingested_from`, the link from an engram to the ingestion record that fed it. `depends_on` and `relates_to` are the conventional ones, the second for when nothing sharper fits.

Wikilink

`[[Target]]` anywhere in the body, resolved by permalink first and then by title in the engram's own domain. A bare link never reaches another domain; when `crystalline verify` checks both domains in one run, its L001 names the domain that holds the title and the prefixed link to write. When an engram moves into another domain, its bare links gain the domain it left, so they keep finding the same engrams, and a prefix naming its new domain is dropped. The target's page shows it as a backlink; on a line with several links, each one is counted once.

Cross-domain link

`[[domain:Target]]` resolves in the named domain only. The name to write is the declared `domain_name` (Chapter 18); a local name or a former name resolves on this machine only, and V110 flags it. The split happens at the first colon and only when the part before it has no spaces. A prefix that names a registered domain without the target leaves the link broken. Only when that part names no registered domain is the whole text tried as a permalink and then as a title at home, so `[[Log: Weekly Notes]]` still finds an engram with that title.

Display text

`[[Target|text]]` links to Target and shows text, and `[[domain:Target|text]]` does the same across domains. A move or a rename keeps the text.

Relative link

A Markdown link to another engram's file by a relative path, as in `[the pump](../pumps/p-12.md)`, with or without a `#` anchor, is an edge like a wikilink, and `- part_of [Overview](../overview.md)` is a relation. With a relative link the type is lowercase, like `part_of`, or a phrase in double quotes, so `- See [Chapter 3](chapter-3.md)` is a plain link; the wikilink form keeps its free type. When an engram moves inside its domain, the links to it and its own relative links are respelled so they keep working. A move into another domain turns them into wikilinks with the domain, because a relative path never reaches another domain.

Attachment reference

A markdown link or image whose target sits under the domain's `assets/` folder, and the sweep reads these references to find attachments nobody has analyzed yet.

24.8 Addresses and reserved names

The address and its one absolute form are explained in chapter 4 (Section 4.6); the entries below give every rule it follows.

`crystalline://domain/permalink`
: The absolute address of one engram. A trailing `/*` anchors a whole folder for `build_context`, and a `crystalline://` address you write in prose is checked by `crystalline verify` like a link.

Domain-relative identifiers
: Anything without the scheme is relative to a domain given beside it, so `retry-queue-gotcha` together with the domain `engineering` is right, while `engineering/retry-queue-gotcha` names a folder called `engineering`.

Headings
: An address names a whole engram, never a heading, and `crystalline verify` reports one with a `#` in it as broken. To open one section, use the engram's `web_url` from a read or a write receipt and add `#` with the heading's slug: lowercase, spaces as hyphens, letters, digits, hyphens and underscores kept and the rest dropped. A repeated heading gets `-1`, `-2` and so on.

`index.md` and `log.md`
: Both names are reserved at every folder level and never hold an engram. Crystalline regenerates `index.md` as the folder's listing and never writes `log.md`, and a title that would slug to either is refused.

`assets/`
: The folder attachments live in, and no permalink may point into it.

24.9 Schema engrams

A schema engram states what engrams of one type should contain, and `validate_engrams` checks them against it. It's an engram of `type: schema`, and beside the usual engram keys its frontmatter carries the declaration in the Picoschema form:

```
type: schema
entity: runbook
version: 1
schema:
  service: string
  severity(enum): [low, high, critical]
```

```
    step(array): string
    owner?: Team
  settings:
    validation: warn
    frontmatter:
      resource?: string
```

`entity`
: The type the schema governs. An engram with that `type`, or with `schema:` naming it, is checked against it, and only schemas in the engram's own domain count.

`version`
: An integer that numbers the schema, kept as written.

`schema`
: The field declarations for the body. A field with a lowercase type is an observation category of that name, so `service: string` asks for a `- [service] ...` line. A field with a capitalized type is a relation of that name, so `owner?: Team` names an optional `- owner [[...]]` relation.

Field names
: A plain name is required and `name?` is optional. `name(array)` allows many lines, `name(object)` nests further declarations and `name(enum)` takes a YAML list of allowed values, one of which an observation's text must match exactly.

Field types
: `string`, `integer`, `number`, `boolean`, `any` or a capitalized entity name. A value of `"string, what it holds"` adds a description after the comma, and an unknown lowercase type counts as `any` while the definition draws an error.

`settings.validation`
: How strict the check is: `warn` reports warnings, `strict` reports errors and `off` checks nothing. The default is `warn`, case doesn't matter and any other value draws an error in the definition.

`settings.frontmatter`
: Declarations for frontmatter keys, in the same syntax, including unknown ones. A finding names a nested key with a dot, as in `review.owner`.

`infer_schema` proposes a first schema from the engrams of a type that exist already. `validate_engrams` with `drift` lists the categories and relations in use that the schema doesn't declare, and the declared ones nobody uses.

24.10 The ingestion record

The shipped skills add one convention on top of the format, and Crystalline reads only the links that tie a record to the engrams it fed. When an agent reads a source into a domain, a repository or a manual or a long document, it writes one record engram under `sources/` with `type: ingestion`, titled after the source. The record carries `resource` for where the source lives, `source_date` where it has one, a `verified` entry and a `source_version` key holding a commit, a version, a revision or a content hash. It also lists the source's stable addresses in `sources`, and there's one record per source, which each run updates. Its body says what was extracted, which engrams received it as links and what was left out on purpose. Each engram the source fed carries its own `resource`, the main page or file it was distilled from (for a git source, the file at the ingested commit). It lists every location it came from in `sources`, the main one included, each in one spelling and without a version (for a git file, its address on the default branch). It links back with `- ingested_from [[record]]`. V111 raises one of them that lacks `resource`.

The next pass searches `type: ingestion` for the source's `resource` and fetches only what changed since `source_version`. A filter on `sources` (`$contains_any`, up to 100 locations a call) finds the engrams of the changed locations, and the pass folds the change into them. It rewrites a renamed location in `sources` and points `resource` at the new path. It drops a location that's gone from `sources`, and that alone is no reason to retire the engram. Then it moves `source_version` on and stamps `verified`. To Crystalline, `source_version` is an unknown key kept as written, and `set_frontmatter` moves it. Don't mix the record up with a `type: source` engram, which holds a source's own words (Chapter 8).

> **In brief**
> The manifest counts only H2 sections, matched without regard to case or spacing, and only `-` bullets in the first column, outside fences, with the first of two equal headings winning. Scope, When to Use, Notes for Agents, Provisioning and Tag Aliases are its sections, routing comes from When to Use with Scope as the fallback, and `sharing`, `generated_indexes`, `read_only`, `reflection`, `reflect_trigger` and `evolve_trigger` are its six frontmatter switches, each reading as its default whenever it says anything else, and Crystalline writes `reflection_home`, `evolve_home` and `team` itself. An engram needs `type`, `title`, `permalink`, `tags`, `status` and `recorded_at`, takes plain dates where it takes dates, treats an absent validity bound as unbounded and keeps every key it doesn't know. Its body carries observations as `- [category] text #tag (context)`, relations as `- rel_type [[Target]]` and links as `[[Title]]` or `[[domain:Title]]`. A

> `type: schema` engram declares in Picoschema what one type should contain, and the finding codes are in chapter 26.

CHAPTER 25

Domains

This chapter is the lookup for everything you can set on a domain and every rule you run into while doing it: reach for it when a prompt to the agent isn't enough (a script, a container or the question of why a command refused). The prose chapters keep the why (domains in Section 4.1, private domains in Chapter 16, team domains and review mode in Chapter 17, membership in Chapter 19, provisioning in Chapter 21). The MANIFEST's own keys are in Chapter 24 and every other setting is in Chapter 23.

25.1 Kinds of domain

Every domain is one of three kinds, and the same tools read and write all of them. Whether a domain lives in files or in the database is fixed when you register it, and a file domain turns into a team domain once you connect an origin to it.

file domain
: A folder of markdown files with a `MANIFEST.md` at its root, the default kind. The files are the source of truth and the index is derived from them, so a rebuild of the index loses nothing.

virtual domain
: A domain that lives in the database with no folder on disk, for a container without a writable volume or a Postgres backend shared by several machines. Its engrams live only in the index, so removing it deletes them and `crystalline reindex --wipe` refuses while one is registered.

team domain
: A file domain with an origin: a GitHub repository, an optional subfolder in it and a branch. It needs `github.enabled` and a GitHub connection, and it can't be virtual, because a proposal needs files to carry.

25.2 The registration block

Each registered domain is one entry under `domains:` in the global `config.yaml`, keyed by its name, and the commands in this chapter write these entries for you. `crystalline config set` doesn't reach them, so a key without a command is a hand edit.

```yaml
domains:
  books:
    path: ~/Documents/Crystalline/books
    provision: true
  club:
    path: ~/Documents/Crystalline/club
    origin:
      repo: dallas/book-club
      path: knowledge
      branch: main
      poll_secs: 600
    review: overlay
  reading:
    kind: virtual
```

`kind`
: Either `file` or `virtual`, and it only appears on a virtual domain, since a file domain leaves the key out.

`path`
: The domain's root folder, with a leading ~ expanded; a file domain registered without one lands at `<domains_root>/<name>`, and a virtual domain has none.

`origin.repo`
: The GitHub repository of a team domain, as `owner/name`; its presence is what makes a file domain a team domain.

`origin.path`
: The subfolder of the repository that is the domain root, for a repository that holds more than knowledge (absent means the repository root).

`origin.branch`
: The branch the domain tracks. Adding a team domain without a branch writes the repository's default branch here, asked from GitHub; an entry without the key, from before that or written by hand, tracks `main`.

`origin.poll_secs`
: How often the daemon checks this origin for merged work, in seconds. Absent defers to the global `github.poll_secs` (300 unless set), and any value below 60 is raised to 60. No command sets it, so it's a hand edit.

`provision`
: This machine's decision on the artifacts the domain ships: `true` installs them into every onboarded harness, `false` keeps them out and removes what was installed. Absent means undecided, and nothing installs.

`review`
: `overlay` puts the domain in review mode; absent means the domain takes changes directly, which is how every domain starts.

25.3 Names

A domain has a name on this machine and a name it declares. The local name is the key in the config and the name in Fluid's addresses, and a new domain's folder under the domains root takes it. The declared name is the MANIFEST's `domain_name` (Chapter 24), and it's the one links, `crystalline://` addresses and the agent's routing carry. Usually the two are the same. Where they differ, both reach the domain and a link written here is stored with the declared name, so it works on every machine. Choose the declared name once: a rename is there to repair a name that turned out wrong (Chapter 18).

What a name may be
: Letters and digits in any script, plus `-`, `_` and `.`, up to 64 characters, with no dot at the start or the end. The Windows device names (`CON`, `PRN`, `AUX`, `NUL`, `COM1` to `COM9`, `LPT1` to `LPT9`) are refused on every platform. The command line, `add_domain`, Fluid and the API all run this check, and so do `domain init --name` and a rename, with the same message everywhere. A name registered before the check existed keeps working.

Names taken already

Adding a name that is already registered at the same folder, or as the same virtual domain, changes nothing and indexes it again, so every way of adding a domain is safe to repeat. The same name at another folder or as the other kind of domain is refused, and the message names what holds it. A name that differs from another domain's here only in letter case (`eng` and `Eng`) is refused too, whether you add, rename or connect, since a disk that ignores case would give both one state folder; a name taken from a folder or a MANIFEST steps aside instead. Every surface also refuses a name that an environment variable already defines, and the refusal names the variable, because that domain belongs to it. A declared name that another domain here already uses as its local name is taken too: the new domain is registered under it with `-2` added and reported as shadowed, and links that use the declared name keep reaching the domain that holds it.

Names given for you

A domain added without a name takes the `domain_name` its MANIFEST declares, then the folder's or the repository's name, with a suffix when that one is taken; a name made this way is always turned into a valid one (a folder called `CON` becomes `con-2`). `add_domain`, `domain add` and Fluid all pick it this way, and when you connect a repository Fluid shows the name it will take in the empty Name field before you create the domain.

Former names

A rename keeps every former name working on this machine as an alias: a link or an address that uses it still reaches the domain, and an old `/d/<old>/` address in Fluid sends you to the new one. An alias gives way to any domain whose local or declared name it is.

When the declared name changes

A machine follows a changed `domain_name` on its next sync when its local name was only ever a default, and keeps the old one as an alias. A local name somebody chose on purpose stays, and the declared name reaches the domain there as well. A shadowed domain takes its declared name on the first sync after the domain that held it gives it up, unless its local name was chosen on purpose.

25.4 Registering a domain

There are five ways in: the agent's tool, the command line, Fluid, the JSON API behind it and environment variables. Each registers the domain, and all but `--no-sync` index it at once.

`add_domain`

The MCP tool, with `domain`, `folder`, `virtual`, `repo`, `path`, `branch` and `readers`. The arguments pick the kind: `repo` makes a team domain, `virtual: true` a virtual one (it needs `domain`), anything else a file domain, and `repo` and `virtual` can't be combined. On a server with a GitHub App, `repo` needs `readers` as well, `shared` or `private`.

`add_domain` on a folder

A folder without a `MANIFEST.md` gets a starter one, and the files already there are kept and adopted in place. Without `folder` the domain lands at `<domains_root>/<domain>`.

Who may call `add_domain`

Any local session may. Over HTTP, on an instance where agents sign in, it takes the instance admin role, and a read-only instance refuses it.

`crystalline domain add [name] [path]`

Registers a domain and indexes it at once, in `<domains_root>/<name>` when no path is given. A lone argument is always the name; to let the folder name the domain, point at it with `--path` or as the second argument. A file domain needs a `MANIFEST.md` in its folder already and refuses with the `domain init` line otherwise.

`--virtual`

Registers a virtual domain and scaffolds its MANIFEST into the index; it takes no path and can't be combined with `--origin`.

`--origin owner/repo[/subfolder]`

Connects a team domain and downloads the repository, or the subfolder given after its name, into the domain's folder; it needs `github.enabled` and a GitHub connection, a prior `crystalline connect github` or a GitHub App on the server. A folder that already holds knowledge, or a registered domain with no origin yet, is connected in place: nothing local is overwritten, and files that differ from the repository become local changes to share.

`--branch`

The branch to track, and it only works with `--origin`. Left out, Crystalline asks GitHub for the repository's default branch and records it; when GitHub can't answer, the add refuses and asks for `--branch`.

`--readers shared|private`

Who may read a team domain on a server with a GitHub App: `shared` is everyone signed in, `private` only the people with access to the repository on GitHub (Chapter 19). On a terminal the command asks when the flag is missing, and where nobody can be asked it refuses, so nothing is chosen for you. A shared connect never opens a domain that is already private. It only works with `--origin`, and a server without an App ignores it with a note.

`--no-sync`

Registers a file domain without indexing it, for a script that runs `crystalline sync` later; a team domain is always indexed as it connects, so the flag refuses with `--origin`.

`--private` and `--owner <account>`

Registers the domain private from its first minute, owned by the named account. `--private` needs `--owner`, and before anything is registered the account has to exist and be enabled. A domain that is already registered is refused here; `crystalline domain visibility` makes that one private.

`crystalline domain init <path>`

Creates the folder if needed and writes a starter `MANIFEST.md` with the Scope, When to Use and Notes for Agents sections, then prints the `domain add` line to run, or says a MANIFEST is already there and leaves it alone. `--name` sets the name in the manifest, the folder's name otherwise. It never touches the config.

`crystalline domain list`

Lists the registered domains, with engram counts when the index is there. A domain whose declared name differs from its local name shows both, as `local (declared)`, with its former names beside it and a mark when another domain holds its declared name. `list_domains` and the JSON API's domain list report the same three things.

Fluid's New domain dialog

Only admins get the dialog. Its Kind is Local folder, Virtual or GitHub team, and the fields follow it: a Name for the first two; a Repository, Branch, Folder in the repository and an optional Name for a team domain. A local domain always lands under the server's domains root, since a browser can't name a folder on the server, and the Private box makes the domain private with you as its owner. On a server with a GitHub App a team domain also asks "Who may read this domain?", with no answer chosen in advance.

`POST /api/v1/domains`

The request behind the dialog, for a script, with `mode` (`local`, `virtual` or `github`), `name`, `repo`, `branch`, `path`, `private` and `readers` as its fields. A field that doesn't belong to the mode is refused rather than ignored, and a team domain on an instance with no GitHub connection is refused with where to make one.

`CRYSTALLINE_DOMAIN_<NAME>=<path>`

Registers a file domain for as long as the variable is set, for a container that shouldn't carry a config file. The domain's name is `<NAME>` in lower case with _ turned into -, and the variable replaces a `config.yaml` entry of that name, with a warning at startup, without ever being written into the file.

`CRYSTALLINE_DOMAIN_<NAME>_ORIGIN`

Attaches an origin to that domain, as `owner/repo[/subfolder][@branch]`, so a new node downloads the team domain on its first start; with `CRYSTALLINE_GITHUB_ENABLED` off the domain waits and says so, and without the matching `CRYSTALLINE_DOMAIN_<NAME>` the daemon refuses to start. Because the suffix is read first, no variable can define a domain whose name ends in `-origin`, `-review` or `-readers`, so register such a domain in the file, and look in Chapter 23 for the GitHub token variable that goes with the origin.

`CRYSTALLINE_DOMAIN_<NAME>_READERS`

`shared` or `private`, who may read that team domain on a server with a GitHub App. Without the variable the domain is private, with one warning at startup. It's applied on every start and only ever closes: `shared` never opens a domain that is already private. It needs the `_ORIGIN` variable beside it, and a server without an App ignores it. A domain whose readers come from here changes them only here.

25.5 Renaming a domain

A rename changes the domain's local name and, unless you keep it to this machine, its declared name too: it writes `domain_name` into the MANIFEST and rewrites every link and `crystalline://` address that uses an old name, in every domain you can write. A MANIFEST whose frontmatter is in flow style (`{title: ...}`) is left as it is, and the rename's report and, in Fluid, the domain page say to set `domain_name` by hand. A domain you can only read keeps its links as they are, and the rename lists it. In a team domain the rewrites become local

changes to share, and in review mode they land in your draft. No MCP tool renames a domain, so an agent asked for one points you to the entries below.

`crystalline domain rename <domain> <new>`

Renames the domain, named by its local name, its declared name or any alias. It works on this machine's own index and configuration only, and it refuses before anything moves when `--db` or `--config` points elsewhere.

`--local`

Renames only this machine's records: the index, the state folders, the provisioning receipt and the config entry move, and the MANIFEST and every link stay as they are. It's the way out when the MANIFEST can't be written here, which the refusal names.

`POST /api/v1/domains/{domain}/rename`

The same for a script, with `name` and `local_only` as its fields. The answer says what moved, which engrams were rewritten, which domains were left behind and which aliases the domain answers to now.

Rename domain

An item in the More actions menu on the domain's page in Fluid, drawn for admins and for a private domain's owner, and the Change name button in Domain policies, in the domain's settings, opens the same dialog. Its "This machine only" box is `--local`. While a rename runs the page says Renaming… and holds back every write, and afterwards it says which name the domain was renamed from, along with the `domain_name` note when the MANIFEST was left as it was. When another domain here holds the declared name, a banner at the top of the page says so, with a Rename button beside it.

Who may rename a domain

An instance admin, a private domain's owner or the machine operator from the command line.

A rename that stops halfway

A crash or a killed process leaves a journal in the state folder, and the daemon, or the next command, finishes the rename. When it can't, `crystalline doctor` shows the rename under its `rename:` heading with the steps done, the steps left and the command that finishes it. `crystalline doctor --discard-rename` drops the journal and leaves the steps already run as they are, for you to finish or undo by hand; `doctor --fix` never drops one.

Names in `crystalline doctor`

Under its `domain names:` heading doctor lists every collision: a domain whose declared name another domain here holds, two domains that declare the same name (so links that use it reach neither) and two domains whose names differ only in letter case, with the rename that ends it. It names each team domain without a `domain_name` and asks you to have its owner add one. It also lists the links in your own domains that use a name only this machine knows, a former name that no longer reaches the domain because another domain answers to it and a domain still waiting to take its declared name, with the reason. `doctor --fix` rewrites those links the ordinary way, as a change to share in a team domain and a draft in review mode, and clears an index entry an older version left behind when a domain was removed.

25.6 Moving knowledge in and out

These commands copy engrams between a domain and a folder or an archive, and none of them changes the domain's registration. A single engram moves from one registered domain to another with `move_engram` and its `destination_domain`. The links to it are rewritten in the domains you may write, the moved engram's own links keep finding what they found, and the receipt names every link it couldn't fix, so somebody who may write there can.

`crystalline domain import <path> --domain <name>`

Loads engram files into a registered virtual domain. A file that already has every field a reader needs is stored as it is, and one that lacks some is completed the way `crystalline import` completes it: a missing status becomes `stable` and a file without tags gets the tag `imported`. The pictures and documents the notes link to are stored as attachments under `assets/`, and the report lists every link it couldn't take. A file domain is refused: for a folder of notes that still needs converting, `crystalline import` writes into the domain's folder and `crystalline sync` indexes it.

`--overwrite` and `--dry-run` on `domain import`

An engram whose path or permalink already exists is skipped unless `--overwrite` says otherwise, and `--dry-run` reports what would land and writes nothing.

`crystalline domain export <path> --domain <name>`
: Writes any domain's engrams out as an ordinary markdown folder, which is how a virtual domain's content gets checked with `crystalline verify` or turned back into files. `--force` writes into a folder that isn't empty, and `--dry-run` only reports.

Fluid's Backup section
: The section sits in the domain's settings, above the Danger zone, and is drawn for admins; the domain page's More actions menu has Download archive and Import archive too. Download archive saves a zip of the whole domain, MANIFEST and attachments included, and works on a read-only instance too. An archive from a file domain restores into a virtual one and the other way round.

Import archive
: Uploads such a zip into the domain, shows a preview of every entry first and imports only after you confirm. Skip existing, the default, leaves a path that's already there alone, and Overwrite existing replaces it. A MANIFEST in the archive is never imported, a file whose name Windows can't hold (like `CON.md` or a name that ends in a dot) is skipped on every system and named in the report, and a broken or oversized archive is refused as a whole.

25.7 Read-only domains

A domain can be frozen with `read_only: on` in its MANIFEST (Chapter 24), for a domain that is finished and should stay as it is. It's a switch on one domain, where `service.read_only` makes a whole instance read-only. It guards what goes through Crystalline against accidents and is no sandbox: an agent with a shell can still edit the MANIFEST on disk.

What it refuses

Every write into the domain through Crystalline, from an agent over MCP, a person in Fluid, the command line or the REST API alike: writing, editing (frontmatter changes and evolve acknowledgements included), moving or splitting an engram into or out of the domain, deleting, attachments, review-mode drafts, co-editing saves, imports, switching review mode, a `configure` change of its rule overrides or of a policy other than `read_only` and renaming the domain. A rename for this machine only (`--local`) still works, because it writes nothing into the domain. Sharing changes is refused too, since nothing local can change, and so is withdrawing a proposal. The refusal names the domain and says that its owner switches `read_only` off in the domain policies.

What keeps working

Team pulls keep the local copy current, because the team's repository stays the authority. Discarding local changes and resolving a conflict with the team's side work too, since both go the way a pull goes, though an agent may not do either to the MANIFEST while the switch is on; a person does that in Fluid or over the REST API. The index, sync and search work as before. On a team domain a switch turned on here stays on this machine, because sharing is refused, so switch it on in the team's repository when the whole team should follow; when the team switches it off there, the next pull opens the domain here too.

How it shows

`list_domains`, `browse_domain` and `read_engram` answer `read_only: true`, and the domain's line in the briefing says "(read-only)", so an agent doesn't plan a write there. The same mark appears on a domain you may read but not write, as a viewer or with a write list that leaves it out (Chapter 16), and the refusal says which reason applies. A domain from a connected server carries the server's mark. Fluid shows a lock on a locked domain and on every engram page in it, and wherever you can't write it offers no edit, new, move or delete. The agent's tool list doesn't change; the write tools refuse when they're called.

The sweep

Evolve and the write nudges skip a read-only domain, because nobody could act on its findings, and the findings come back once the switch is off.

25.8 Removing a domain

Removing a domain unregisters it and clears its rows from the index. A file or team domain's files stay on disk and the team's repository is never touched: the only knowledge a removal can delete is a virtual domain's, and that takes an explicit confirmation.

`crystalline domain remove <name>`
: Unregisters the domain and frees its name for a rename or a new registration. Registering the same folder again adopts the files as they are, and a team domain is connected again with its repository, which keeps its origin. The visibility and membership records go too, so a later domain under the same name starts out shared.

`--purge`
: Confirms that a virtual domain's engrams are deleted with it, and nothing brings them back afterwards; a virtual domain that holds any refuses without it, so export first when you want a copy.

`--end-drafts <actor>`
: Names one other person whose private drafts in a review-mode domain end with the removal; repeat it for each. The refusal says who is drafting and how many drafts each holds, and your own drafts need no naming.

`remove_domain`
: The MCP tool, with `domain`, `purge` and `end_drafts` working as the flags do. On a harness that can ask, the first call removes nothing and puts a confirmation question to you naming the domain, its kind, its engram count and the drafts at stake. The question says in one line what happens, and its box starts checked, so Accept removes the domain and Decline keeps everything.

Who may remove a domain
: The machine operator, an instance admin or, for a private domain, its owner. An agent on an instance that leaves agent authentication off is refused. A domain defined by an environment variable can't be removed at all: unset the variable instead.

Unregister domain
: The button in the Danger zone of the domain's settings, drawn for admins and for a private domain's owner. It asks you to type the domain's name and says whether the files stay on disk or, for a virtual domain, are deleted.

25.9 Domains from shared servers

Your machine's Crystalline can connect to a server and offer its domains beside your own, each domain from its one source (Section 16.10). A server's domains get no entry under `domains:`; the sources and the local names live in `remote/sources.json` in the state folder. The sign-in itself is kept in the OS keychain under `crystalline-server:<host>`, with the path after the host for a server under a path (`crystalline-server:example.com/crystalline`), or in a private file where no keychain works; it's never in `sources.json`.

`crystalline connect <url>`

Adds a server as a source; the server has to run Crystalline 0.23 or newer. `connect` keeps the address with its path and follows no redirect, so `https://example.com/crystalline` connects to a server under that path. An address with a query or a fragment, or with a path that breaks the rules of `service.public_url` (Chapter 23), is refused. A server under another path of the same host is a second source, with its own sign-in. It checks the server first, refuses plain `http://` anywhere but loopback and signs you in through the browser, so the server needs `auth.oauth` on (it follows `auth.mcp`).

`--token`

Takes a personal token instead of the browser sign-in, typed at a prompt or piped in and never on the command line, for a machine without a browser. A server without the sign-in path gets the same, with a message saying so.

`--name`

The source's name, which appears only in notes and in local names. Without it the source is named after the most specific word of its host that isn't a generic one like `crystalline`, and a second server on the same host gets the same word with `-2`.

`--domains <names>`

Takes only these domains of the server, by their names there, separated by commas (`--domains alpha,beta`). Connecting again with `--domains` replaces the list, and a `connect` with neither flag keeps the list as it is. A listed name the server doesn't offer yet (no rights, or not created) stays on the list and is taken once the server offers it; `connect` says so. Taking a name off the list frees it, and a local copy it hid comes back.

`--all-domains`

Takes every domain the server offers again and clears the list. A first `connect` without `--domains` takes every domain too.

A server's domain and its name

A domain from a server keeps its name unless your machine already uses it, and then it gets the local name `<domain>-<source>`, which `crystalline domain rename <name> <new> --local` changes. Whoever comes later gets the suffix, so no name changes on its own: a later domain on the server becomes `<domain>-<source>` and a local domain you add later `<domain>-local` (then `-local-2`). When your machine holds its own copy of a team domain the server also serves and you take it from the server, the server wins: your copy is hidden while you're connected, with its files untouched, and `connect` warns when it holds changes you haven't shared yet. A hidden copy can't be removed or renamed while its server is connected. The same team domain on two servers comes from the one you connected first, unless the other one lists it with `--domains`: then it moves to that one and keeps its name on your machine. `connect` says so either way.

What a server's domain refuses

A move between a server's domain and another source is refused: read the engram, write it into the destination, then delete it. `remove_domain` and `provision` refuse a server's domain and point at `crystalline disconnect`, and the collaboration tools of Chapter 17 say to share changes on the server.

`crystalline disconnect <name>`

Removes a source with its sign-in, its cache and its domains, and revokes the sign-in on the server when it can reach it; the source's URL works in place of its name. A pasted token is yours, so it's only forgotten here. A hidden copy of a domain comes back. A sign-in that couldn't be revoked because the server didn't answer can be revoked later in the server's Fluid, under Connected clients on your profile, and it stops working on its own after 30 days unused.

`crystalline status` and `crystalline doctor`

List every source: its address and account, how it signed in, whether it answers, which domains it takes (every one it offers, or the list), the domains it brings, the local names and the hidden copies. `doctor` also says when the sign-in expires, counts a server that doesn't answer as a problem and notes a listed name the server doesn't offer.

`CRYSTALLINE_REMOTE_URL`, `CRYSTALLINE_REMOTE_TOKEN` and `CRYSTALLINE_REMOTE_DOMAINS`

Add one source from the environment, for CI and containers. The address keeps its path as it does for `connect`, and `CRYSTALLINE_REMOTE_DOMAINS` is the list `--domains` takes; a value that isn't a list of domain names leaves the source out rather than taking every domain. `crystalline disconnect` can't remove it; unset `CRYSTALLINE_REMOTE_URL` or `CRYSTALLINE_REMOTE_TOKEN` instead.

25.10 Visibility and membership

Every domain starts shared, visible to every account on the instance. A private domain is visible only to its owner, the instance's admins and the accounts invited into it, and to everyone else it answers as if it had never been registered (Chapter 16).

`crystalline domain visibility <domain> private --owner <account>`

Makes the domain private and hands it to that account, which has to exist and be enabled. Making a domain that is already private private again changes nothing, its owner included. On a domain under a GitHub App `--owner` is ignored with a note and no account becomes its owner, and a domain whose readers come from `CRYSTALLINE_DOMAIN_<NAME>_READERS` is refused with the variable's name.

`crystalline domain visibility <domain> default`

Shares the domain with every account again and forgets its list of members, so making it private later starts from the owner alone.

Make private and Share with everyone

The same switch in the Danger zone of the domain's settings in Fluid, confirmed by typing the domain's name. Making a domain private is for admins, and the admin who does it becomes the owner; sharing it again is for the owner or an admin. On a domain under a GitHub App the switch sits on the Members section instead, for a GitHub admin or maintainer or an instance admin, and no account becomes its owner.

`none`

The membership level that sees no engram in the domain. A member at `none` works on the domain's Reflection alone and needs a Reflection level, or the membership is refused.

`viewer`
: The membership level that reads and searches the domain and nothing more. It's the level an invitation takes unless you name another.

`editor`
: A viewer who also writes and edits engrams in the domain, as far as the account's instance role allows.

`manager`
: An editor who also runs the membership: inviting, removing and changing levels up to manager. A manager can't change visibility, transfer the domain or remove it.

Reflection level

An account carries a Reflection level for the shared domains and a membership carries one for its private domain, beside the level on the engrams, which has `none` for somebody who sees no engram at all. It's `default`, `reader` or `contributor`. An admin sets it on the account, and the owner or a manager on the membership. With `default`, `viewer` gives `reader` and `none` gives nothing. A level that is set only ever raises access (Chapter 12).

`reader`
: The Reflection level that reads the domain's reflections and comments on them: opening, answering, resolving and reopening threads.

`contributor`
: A reader who also edits reflections, creates, moves and deletes them and adds files to them.

curator
: A curator also writes sections from the engrams, settles what people changed, edits the pact and replaces, moves and removes files. Curating is never set on its own: everyone who may edit the engrams curates (`editor`, `manager`, the owner, an admin and the machine operator), and Fluid shows them as curators.

`crystalline domain members <domain> list`
: Prints the owner and every member with their level, though a shared domain has no owner and no members, because membership only decides anything while a domain is private. The Reflection collaborators follow, on a shared domain too.

`crystalline domain members <domain> add <user>`

Invites an account, or moves a member to another level, with `--level none`, `viewer`, `editor` or `manager` and `--reflection` for the Reflection level; the owner can't be added, because the owner already holds every level. On a domain under a GitHub App it refuses, like `remove` and `transfer`, with "access to this domain follows GitHub; change it on GitHub".

`crystalline domain members <domain> remove <user>`

Ends a membership, though the owner can't be removed this way; hand the domain on first.

`crystalline domain members <domain> reflection add <user> --level reader|contributor`

Makes an account a Reflection collaborator of this one domain, shared or private, as long as it's kept as files. The grant only raises: it never shows an engram the account couldn't see before, and the write list still caps it. It also works for an account with the instance role `none`. The owner, a manager, an instance admin and, under a GitHub App, a GitHub admin or maintainer give it, and it's the one grant a domain under a GitHub App still takes.

`crystalline domain members <domain> reflection remove <user>`

Takes the grant away again.

The Members section

The same list in Fluid, in the domain's settings: invite by username, level and Reflection level, change a level, remove, and a Leave button for a member's own membership, which needs nobody's permission. Reflection collaborators have a list of their own there. On a domain under a GitHub App the section says that access follows GitHub and refuses a change of membership: people with write access see who has access on GitHub, and a GitHub admin or maintainer or an instance admin switches the domain between shared and private.

`crystalline domain transfer <domain> <new_owner>`

Hands a private domain to another existing, enabled account, which becomes the owner at once. The old owner keeps nothing and is a stranger to the domain unless invited back. It's for the owner or an admin, and Fluid offers it in Members.

A deleted owner

Deleting the owner's account leaves the domain with no owner and only an admin can administer it after that, so transfer it before the account goes.

Instance role and membership level

On a shared domain an account does what its instance role allows: a viewer reads, an editor writes, an admin does everything. On a private domain the membership decides what the account may reach, and a write takes the lower of level and role. So an instance viewer invited as an editor writes nothing, and neither does an owner whose instance role is viewer. An account's write list (Chapter 16) narrows that further: a domain it leaves out is read-only for that account. All of this holds for domains without a GitHub App behind them; for a domain under a server's GitHub App the second table below replaces the role and the membership.

Action on a domain	Who may
Read and search	viewer or better
Write	editor or better, capped by role and by a write list
Invite, re-level, remove	manager, owner, admin
Leave	the member
Make private	admin
Share again, transfer	owner, admin
Review mode on or off	owner, admin
Change policies and rule overrides	owner, admin
Unregister	owner, admin
Register, provision	admin
Backup and archive import	admin

On a server with its own GitHub App, a domain connected to a repository follows that repository (Chapter 19). The account's GitHub link and its role on the repository decide, and members, invitations and the owner set in Crystalline don't count:

Who	Engrams, shared domain	Engrams, private domain	Reflection
Machine owner, instance admin	everything	everything	curate

Who	Engrams, shared domain	Engrams, private domain	Reflection
GitHub admin or maintain	write, switch shared or private	the same	curate, manage the Reflection collaborators
GitHub write	write	write	curate
GitHub triage or read	read	read	read and comment
Instance viewer or above, without access on GitHub or without a link	read	nothing	shared domain: read and comment; private domain: nothing
Instance role none	nothing, whatever GitHub says	nothing, whatever GitHub says	only through a Reflection collaborator grant
Anonymous visitor, where the instance lets one in	read, only when the repository is public	nothing	read, never comment, only when the repository is public

The instance role doesn't cap a GitHub role, so an instance viewer with write access on GitHub writes, while the write list still applies. On a shared domain an account's own Reflection level still raises what the row gives, so an instance contributor edits the Reflection there without access on GitHub; on a private domain it doesn't count. A private domain counts only the repository's listed collaborators, never the members of an Enterprise who read an internal repository without being listed. A closed domain, whose repository lost the App, was deleted or moved to another owner, opens to the instance admins and the machine owner alone, and no grant opens it. The open tier acts as the account `github.agent_identity` names, never above editor; without that setting it's an anonymous visitor.

The command line and a local stdio session are the machine operator and pass every gate in this table. An agent over HTTP on an instance where agents sign

in is its user and gets exactly that user's row, with no tool for membership at all. Where agent authentication is off, an agent is nobody in particular: it reads and writes the shared domains, registers and provisions without a role, changes a domain's policies other than `read_only` wherever it may write and is refused when it unregisters one. On a domain under a GitHub App it's the account `github.agent_identity` names instead, private domains included, so turn agent authentication on there.

25.11 Review mode

Review mode gives every writer a private overlay of drafts over a team domain's folder, and the folder changes only with what lands on the team's branch (Chapter 17). It lives in the registration as `review: overlay`, never in the MANIFEST, and no MCP tool turns it on.

`crystalline domain review <domain> overlay`
: Turns review mode on. It needs a GitHub origin and a folder, so a virtual domain is refused, and it refuses while the folder holds unshared changes, listing their paths to share or revert first.

`crystalline domain review <domain> direct`
: Turns review mode off, which ends every private draft in the domain. Run without answers it prints the plan (who holds drafts, how many and where two people drafted the same path), and then it needs one `--fold` or `--discard` per person the plan names.

`--fold <actor>`
: Writes that person's drafts into the domain's folder as they are (your own drafts from the command line or a local session belong to the actor `owner`).

`--discard <actor>`
: Ends that person's drafts without touching the folder. Both flags repeat, one per person, and both refuse with `overlay`.

Review mode in Fluid
: The same switch in the domain's settings in Fluid. Turning it off shows the plan with a fold or discard choice per person, set to fold until you change it.

Who may switch review mode
: The machine operator, an instance admin or a private domain's owner; a domain an environment variable defines is refused, because its variable holds the setting.

`CRYSTALLINE_DOMAIN_<NAME>_REVIEW=overlay`

Puts a domain defined by `CRYSTALLINE_DOMAIN_<NAME>` in review mode, and any other value stops the daemon at startup, while an empty one counts as unset. The variable and a hand-edited `review:` key are taken as written, without the origin check the command makes, so pair them with an origin.

One proposal at a time

A domain in review mode keeps one open proposal, even with `github.stacks` on. A second share while one is open is refused: somebody else's proposal is named with the choice to wait for the merge or ask for a withdrawal, and your own is withdrawn and shared again.

25.12 Sharing and sync

Whether a team domain shares through proposals or straight onto the branch is the MANIFEST's `sharing` switch. That key and `generated_indexes`, which decides whether the folder listings travel with a share, are in Chapter 24. The per-domain part of sync is the origin block above.

`github.enabled`

The switch every team domain needs before it can be added or kept current, and Chapter 23 has it with the other `github.*` settings.

The background poll

The daemon checks each team domain's origin on the interval above and brings merged work into the folder, and a read-only instance keeps polling, so it follows the team's repository on its own.

25.13 Running the loops

A domain's reflect loop and evolve loop run without a session through a runner (Chapter 14). `<loop>` below is `reflect` or `evolve`. `status` and `watch` take `--domain` once per domain, `run` takes one and `prompt` needs one; without it, `status` covers every domain the caller may run the loop on.

`crystalline <loop> status`

What's due per domain, as counts, with a run that's going, the last run and the webhook's state; `--json` prints the answer as it came. It exits 0 when something is due, 1 when nothing is and 2 on an error. An item every run so far left for a person counts as "left for a person" instead of due, until it changes.

`crystalline <loop> run`

One beat: it takes the lease, starts Claude Code headless with the shipped prompt and only the Crystalline MCP server, shares the run's changes, records the run and releases the lease. Without `--domain` it works every due domain in turn. It exits 0 when the run is done or stopped at its cap, 1 when nothing was due or another run holds the lease and 2 when the run failed. `claude` has to be on the `PATH`.

`crystalline <loop> watch`

Checks the status every `--interval` seconds (60 by default) and runs when something is due, until it's stopped. It takes the flags of `run`.

`crystalline <loop> prompt`

Prints the shipped prompt for one domain, for a runner that starts its own agent, n8n for example; `--max-items` sets the cap the prompt names. Paste it again after an upgrade, since the prompt can change.

Flags of `run` and `watch`

`--max-items` (20) is the most items one run works, `--max-turns` (60) the most agent turns, `--model` the agent's model and `--runner` the label the lease, the run record and Fluid show (`cli` by default). `--log-dir` moves the run logs from `runs/` in the state folder; the newest 50 per loop and domain are kept, readable by you only. `--no-share` keeps the changes local. `run` alone takes `--dry-run`, which shows the command it would start and starts nothing, and `--force`, which runs while the loop backs off after failed runs.

Local or remote

Without anything set, the commands talk to the daemon on this machine and a run acts as the machine owner. With `CRYSTALLINE_REMOTE_URL` and `CRYSTALLINE_REMOTE_TOKEN` (or `CRYSTALLINE_REMOTE_TOKEN_FILE`) they talk to that server as the token's account, and the agent reaches the server's MCP endpoint with the same token.

Who may run a loop

The reflect loop needs the right to curate the domain's Reflection, the evolve loop the right to write the domain. Give each loop its own account when both work one domain at once, since the server notes an account's writes in every lease it holds there.

Where a loop runs

The reflect loop runs on the domain's Reflection home. The evolve loop on a team domain runs on its evolve home, the instance whose `service.public_url` the MANIFEST's `evolve_home` names (on a shared database, the one of those instances that holds the domain's host lock). Crystalline fills `evolve_home` in: the first scheduled evolve run at an instance with a `service.public_url` claims it, and its pull request carries the line. Until then every copy may run evolve, and `crystalline doctor` warns. A domain without a GitHub origin runs it on any instance with its own copy, and on a shared database on the instance that holds the host lock. An instance where the loop doesn't run answers `runs on <name>`, is never due and refuses `run`. No loop runs on a read-only instance or a read-only domain.

What a run shares

On a team domain with a GitHub origin the runner shares the run's changes as a pull request, whatever `sharing` says, titled with the loop, the domain and the date, with the run's summary as its description; a second run while that pull request is open adds to it. In remote mode only the account's own writes through the MCP tools during the lease go in. A local run shares the files new or changed since it took the lease. Either way, a file a person had unshared work on when the run started is held back and counted, the changes of a stopped run stay local and Reflection pages are never part of it. A domain without an origin, or an instance with GitHub off, shares nothing.

The lease and the run record

One lease per domain and loop, renewed every 60 seconds and taken over by the next run once it hasn't been renewed for five minutes, which records the old run as `expired`. A run record keeps the start and end, the runner, the account, the outcome (`done`, `stopped_at_cap`, `failed`, `nothing_due` or `expired`), the counts, a short summary and the pull request; the index keeps the last 50 per domain and loop, so a full reindex loses them. After a failed run the loop waits 5 minutes before it's due again, twice as long after each further failure in a row and at most 6 hours.

The last run in Fluid

The Reflection home says "Last reflected" and the maintenance page "Last maintained", each with the time, the runner and the outcome, and nothing when the loop never ran. A reader sees the line; someone who may run the loop also sees the summary, the pull request and the counts. A Reflection contributor without the right to the knowledge sees no "Last reflected" line.

25.14 The provisioning decision

A domain whose MANIFEST declares a Provisioning section ships skills, commands, subagent definitions and MCP server configs, and each machine decides per domain whether to take them (Chapter 21). The decision is the registration's `provision` key.

`crystalline provision allow <domain>`
: Sets `provision: true` and reconciles, installing the domain's artifacts into every harness this machine onboarded.

`crystalline provision deny <domain>`
: Sets `provision: false` and reconciles, which removes whatever the domain shipped before.

`crystalline provision status`
: Prints each domain's decision, grouped allowed, undecided and denied and sorted by name, each harness's installed state and the domains still waiting for one. The domains that declare no provisioning and the virtual ones fold into one line each; `--all` lists every one of them.

`provision`
: The MCP tool, with `action` set to `status`, `allow`, `deny` or `apply` and `domain` for the two decisions. On an instance where agents sign in, everything but `status` takes the admin role.

25.15 Where domains live and which ones an agent hears about

Four more settings touch domains without belonging to one. Two sit in the global config and two are files named `.crystalline.yaml`, one in a workspace and one at a domain's root.

`domains_root`
: The folder a new file domain lands in when nobody names one; the setting is in Chapter 23.

`prompt.rules`
: Narrows which domains the session-start briefing names for a workspace, by path pattern. The rule syntax and an example are in Chapter 23.

`preferred_domains`
: A list in a `.crystalline.yaml` at a workspace's root that puts those domains first in the briefing, in that order. It only reorders and never hides a domain.

Where the two routing hints apply

Both need a workspace path, so they shape the briefing the session-start hook prints with `crystalline prompt system`, while an agent connecting over MCP gets the server's briefing, which covers every domain it may see and ignores both.

The domain's own `.crystalline.yaml`

A file at a file domain's root whose `verify:` block tunes the checker for that domain, with its keys and the codes they touch in Chapter 26; a virtual domain has no root, so it always takes the defaults.

In brief

A domain is a folder, a database-only virtual domain or a team domain with a GitHub origin, and each is one entry under `domains:` in `config.yaml` with `path`, `origin`, `provision` and `review`. You register one with `add_domain`, `crystalline domain add`, Fluid's New domain dialog or an environment variable, and removing it leaves files on disk but deletes a virtual domain's engrams, which is what `--purge` confirms. A domain declares its name in its MANIFEST and keeps it on every machine; a rename, from the command line, the API or Fluid, is the repair for a name that turned out wrong and leaves the old one working as an alias. Every domain starts shared, and a private one belongs to one owner and admits members as none, viewer, editor or manager; a write there takes the lower of level and instance role, while on a domain under a server's GitHub App the role on the repository decides. Review mode is switched with `crystalline domain review` or in the domain's settings in Fluid and needs a GitHub origin and a clean folder, and turning it off folds or discards each person's drafts. Your machine can also offer a connected server's domains beside its own, and `read_only` in the MANIFEST freezes a finished domain against every write through Crystalline. `prompt.rules` and `preferred_domains` steer the hook's briefing, and a domain's own `.crystalline.yaml` tunes the checker of chapter 26.

CHAPTER 26

Every finding code

Crystalline has two checkers, and this chapter lists every code either of them can put in front of you: what it means and what clears it. Come here when a code turns up in a verify report, a CI annotation or the maintenance queue and you want to know what it wants from you. The story behind the sweep and how the agent works its queue are in the chapter on evolving (Chapter 10); this one is just the lookup.

26.1 Two checkers

Verify asks whether the format holds. It reads files and needs no daemon, and its codes carry a letter for what they check: E for format, T for temporal metadata, L for links, Q for content quality, S for schemas and M for the MANIFEST. Each finding has a severity: error, warning or info (the report tags them E, W and I). The sweep asks whether the knowledge is still true and well organized. Its codes all start with V, and instead of a severity each finding carries a family, a priority and a class.

Verify codes show up in more than one place, and not every place runs every family:

Where you see it	Codes	Domain overrides
`crystalline verify`	E, T, L, Q, S, M	yes, plus `--strict`
`validate_engrams`	S020 to S033, T, E011, M108	file domains
Fluid's editor	E001 to E008, E010, E011, T except T006, Q006 to Q008	file domains

Where you see it	Codes	Domain overrides
MANIFEST edit receipt	M except M001, M105 and the team repository rules	file domains
Write receipts (`markdown_findings`)	Q006 to Q008	every domain (Q008 on engrams only)

`crystalline verify` scans folders, so a virtual domain is checked through `validate_engrams`, or with verify after an export (Chapter 25). That tool reports a schema finding under a name, not its number (the schema section below gives both). In Fluid's editor the check runs whenever you pause typing, and a hard error holds the save back until you fix it. The sweep's codes show in `crystalline evolve`, the `evolve_engrams` tool and Fluid's maintenance page, all three reading the same queue.

26.2 Format: the E codes

An engram has to parse and carry a title, a permalink, a type and tags; the E codes tell you which part doesn't hold.

`E001` frontmatter doesn't parse
: The YAML between the `---` lines is invalid or isn't a mapping of keys to values (error); a repeated key on its own is E010. Fix the YAML first, because no other rule looks at a file that doesn't parse.

`E002` title or permalink missing
: The engram lacks a `title` or a `permalink` (error); add the one that's missing.

`E003` type missing or unusual
: Leaving out `type` is an error. A type outside the recommended set (manifest, schema, engram, guide, decision, architecture, runbook and reference) is only info and needs nothing, since Crystalline never enforces a closed list.

`E004` no tags
: The engram has no `tags`, or the list is empty (error). Add at least one, reusing what `vocabulary` shows is already in use.

`E005` permalink isn't a slug
: The permalink holds capitals, spaces or other characters a lowercase slug path doesn't (error). Use the slug the report's fix line suggests.

`E006` not plain UTF-8

The file starts with a UTF-8 byte order mark, contains a null byte or isn't valid UTF-8 at all (error). Save it again as plain UTF-8.

`E007` tag spelled oddly

A tag isn't lowercase-with-hyphens (warning). `crystalline tags rename` changes it on every engram that carries it.

`E008` permalink starts with the domain's name

A permalink is relative to its domain and the domain's name is each person's own configuration, so a prefix that repeats it breaks under any other name (warning). Drop the prefix as the fix line shows; a real subfolder of the same name doesn't count.

`E009` paths differ only in case

Two paths in the domain differ only in upper and lower case (error). A checkout on macOS or Windows holds only one of them and loses the other without a word, so rename one.

`E010` frontmatter key repeated

One frontmatter key appears more than once and the file would parse without the later copies (error). Keep one line with the right value and delete the rest. When every repeated key in the file holds the same value, `crystalline doctor --fix` removes the extra copies.

`E011` frontmatter too large for the index

A frontmatter key is over 256 bytes, or a value (each element, for a list) is over 1024 bytes, so the metadata index skips it and a search's list filter (`$contains`) never finds it (warning). Shorten the key or the value, and keep long text in the body.

26.3 Temporal metadata: the T codes

These check the lifecycle and provenance fields; a malformed date here (T003) is one the sweep's V rules can't read.

`T001` status or recorded date missing

The engram has no `status` or no `recorded_at` (error); add whichever is missing.

`T002` unusual status

The status is outside the recommended set: stable, current, implemented, draft, proposed, idea, poc, deprecated, superseded, archived and legacy (info). Status is free form, so there's nothing you have to change.

T003 malformed date or provenance

A date field (`recorded_at`, `valid_from`, `valid_to`, `source_date`, `stale_after` or the older `last_verified` and `review_after`) isn't a plain `YYYY-MM-DD`, a `timestamp` isn't RFC 3339, `generated` isn't a mapping with a `by`, or `verified` isn't a `{ by, at }` entry or a list of them (error). Write the value in that shape.

T004 window inverted

`valid_from` lies after `valid_to` (error). Correct whichever date is wrong.

T005 superseded without a successor field

The status says superseded and no `superseded_by` relation is present (warning). Add `- superseded_by [[Successor]]`, or pick the retirement status that fits when nothing replaced it (Chapter 10). The sweep's V004 also catches a relation that doesn't resolve.

T006 no write provenance

The engram carries no `generated` block and no older `timestamp` (warning). Every write and edit through Crystalline stamps one, so a file written by hand clears on its first edit through the agent or Fluid.

T007 unknown temporal confidence

`temporal_confidence` is set to something other than `explicit` or `inferred` (warning); pick one of those two.

T008 placeholder end date

`valid_to` lies in the year 9000 or later, a stand-in for "forever" (warning). Remove the field; with no end date the window stays open, and that's how an engram says it holds indefinitely.

26.4 Links: the L codes

These check that every link and relation reaches something, within the domains the run scanned.

L001 broken link

A `[[wikilink]]` or relation target resolves to nothing, in this domain or in another domain the same run scanned (warning). When a bare link's target lives in another domain of the run, the message names that domain and the prefixed link to write. Fix the spelling, add the prefix or capture the engram the link expects.

L002 permalink used twice

Two files share one permalink, so one address names two engrams (error); change one of them.

`L003` title used twice

Two files share one title, so a `[[Title]]` link can't tell them apart (warning); retitle one or merge them.

`L004` self-link

An engram links or relates to itself (warning); remove the link.

`L005` broken crystalline:// address

A `crystalline://` address in the body names an engram that doesn't exist in a domain the run scanned (warning). Correct the address or remove it. Addresses inside code fences and backticks are skipped.

`L006` link leaves the run

A link names a domain this run didn't scan, so verify can't check it (info). Pass that domain's root to the same run when you want it checked.

`L007` link to a file the domain doesn't have

A relative link or image whose target isn't a note names a file the domain doesn't have, such as an attachment an import left behind (warning). A link to a note is L001's. Add the file, or fix or remove the link.

26.5 Content quality: the Q codes

These look at the body: whether there is one, whether it fits, whether its bullets parse and whether its Markdown elements are ones Fluid knows.

`Q001` almost no content

The body has fewer than three non-blank lines outside code fences (error). Write the knowledge down, fold it into the engram that owns the topic or delete the file.

`Q002` over the size budget

The body is over its token budget, counted roughly as characters divided by four: 2,500 by default and four times that for a `type: source` engram (error). Split it; the sweep's V105 says which split fits, and the budget can be tuned per domain (see "Silencing and tuning" below).

`Q003` duplicate heading

Two headings at the same level carry the same text (warning). Rename one or merge the two sections.

`Q004` near-duplicate sections

Two `##` sections of one engram carry nearly the same text, usually a paste nobody finished (warning); fold them into one.

`Q005` near-miss bullet

A bullet almost matches the observation or relation grammar: `-[category]` without the space, an unclosed `[category` or an unclosed `[[Target` (info). Fix the typo, or the line never becomes an observation or a relation.

`Q006` unknown status colour

A status label names a colour outside grey, blue, green, yellow, red and purple, as in `:status[Done]{orange}` (warning). It renders grey; pick one of the six.

`Q007` unknown callout kind

A callout names a kind Fluid doesn't know, as in `> [!DANGER]` (warning). It renders as a plain quote; use NOTE, TIP, IMPORTANT, WARNING, CAUTION or DETAILS.

`Q008` observation inside a quote

An observation or a relation sits inside a callout or any other `>` quote, where it isn't indexed (info). Move it out to a top-level bullet. Only engrams get this one.

26.6 Schemas: the S codes

You'll only meet these in a domain with schema engrams (the reference on the engram describes their format, Chapter 24). S001 to S004 are errors on the schema engram itself and S010 is a warning. S020 to S033 check an engram against its schema, and their severity comes from the schema's `settings.validation`: `warn` makes them warnings, `strict` makes them errors and `off` silences them. In `validate_engrams` they carry the name in parentheses instead of the number.

`S001` schema without an entity

The schema engram has no `entity` (error); add the type it governs.

`S002` schema without declarations

There's nothing under `schema:` (error), so declare the fields there.

`S003` malformed declaration

A field declaration has an empty name, a modifier other than `array`, `enum` or `object`, an enum that isn't a non-empty list, an object that isn't a mapping, an unclosed parenthesis or a type other than `string`, `integer`, `number`, `boolean`, `any` or a capitalized entity name (error). Correct the declaration, in the body or under `settings.frontmatter`.

`S004` unknown validation mode
`settings.validation` isn't `warn`, `strict` or `off` (error); use one of the three.

`S010` schema reference matches nothing
An engram's `schema:` key names an entity no schema engram in the domain declares (warning); fix the name or write the schema.

`S020` required relation missing (MissingRequiredRelation)
The engram lacks a relation its schema requires; add it.

`S021` required observation missing (MissingRequiredObservation)
The engram lacks an observation category its schema requires; add one.

`S022` observation value outside its list (ObservationEnumViolation)
An observation holds a value the schema's enum doesn't allow; use an allowed one or extend the enum.

`S023` observation value of the wrong type (ObservationTypeMismatch)
An observation's value doesn't match the scalar type its schema declares; correct the value.

`S030` required frontmatter field missing (MissingRequiredFrontmatter)
The engram lacks a frontmatter key its schema requires; add it.

`S031` frontmatter value outside its list (FrontmatterEnumViolation)
A frontmatter value isn't one the schema's enum allows; use an allowed one.

`S032` frontmatter field isn't a list (FrontmatterNotArray)
The schema declares the key as an array and the engram holds a single value; write it as a list.

`S033` frontmatter value of the wrong type (FrontmatterTypeMismatch)
A frontmatter value, or an item in a list, doesn't match the declared scalar type; correct it.

26.7 The MANIFEST: the M codes

The MANIFEST is configuration as well as an engram: its When to Use bullets route every session to the domain (Chapter 24). M001 to M013 are errors and M101 to M109 warnings. M011, M012 and M109 are the team repository rules: they run only with `crystalline verify --team-repository`, the check for the repository of a team domain (Verify in CI below). You can put the same structural checks on any other file of a domain through the verify file described below.

`M001` MANIFEST missing

The domain root has no `MANIFEST.md`, or a file the domain requires is missing. Create it; a MANIFEST carries `type: manifest`.

`M002` wrong type

The MANIFEST's type isn't `manifest`, or a required file lacks the frontmatter the domain asks for. Correct the type or add the frontmatter.

`M003` required section missing

`## Scope` or `## When to Use` is absent, or a section the domain requires of another file; add it.

`M004` section without bullets

Scope or When to Use has no top-level bullets (a `-` at the start of the line), or a configured section has fewer than its minimum; write the bullets.

`M005` invalid provisioning path

A Provisioning path is absolute, home-relative, a bare `.` or contains a component that isn't allowed, and nothing is provisioned from it. Give a relative path to the folder.

`M006` unknown `generated_indexes` value

The key says neither `local` nor `shared`. It's read as `local`, so the folder listings stay on your machine; fix the spelling.

`M007` unknown `sharing` value

The key says neither `proposal` nor `direct`. It's read as `proposal`, so shares still go through review; fix the spelling.

`M008` invalid `domain_name`

The declared name isn't text at all (a number or a list, say), or it's text that breaks the naming rule (Chapter 25). Crystalline ignores it and the domain keeps the name it has, so this code and the receipt of an edit are the only places that tell you; write a valid name.

`M009` unknown `read_only` value

The key says neither `on` nor `off`. It's read as `off`, so the domain stays open to writes; fix the spelling.

`M010` unknown `team` value

The key says something other than `github`. It's read as absent, so this copy isn't treated as a team domain and nothing keeps its Reflection at the team server. Write `team: github`, or remove the line and let the next share from an instance connected to the domain add it.

`M011` Reflection on without a home

In a team repository the MANIFEST says `reflection: on` and has no valid `reflection_home`, so any clone could keep its own Reflection and push its pages and threads to the shared branch. Set `reflection` back to `off`, or switch Reflection on at the team server, which fills in the home.

`M012` Reflection files in a team repository

While Reflection is on, the shared branch holds files under `reflections/`, in any letter case, and one finding lists them. The Reflection lives at the team server and is backed up to `crystalline/reflections`, never to the shared branch, because its pages and threads can hold comments from people who may not read the engrams. Remove the files from the branch and switch Reflection on at the team server instead.

`M013` invalid `evolve_home`

The key isn't a server address. It's read as absent, so every clone of the team domain may run a scheduled evolve. Remove the line, and the next scheduled evolve run at the team server fills it in.

`M101` When to Use empty

When to Use has no bullets and routing falls back to Scope. It comes along with M003 or M004 and says what sessions do in the meantime. Writing the bullets clears all of them. In a required file it means a configured fallback section stood in.

`M102` bullet too long

A bullet in any `##` section runs over 180 characters, or over a required file's configured maximum; shorten it or split it into two.

`M103` duplicate section heading

The same `##` heading appears twice and the reader keeps only the first; merge the two sections.

`M104` unknown provisioning type

A Provisioning bullet names a type other than `skills`, `commands`, `agents` or `mcps`, so pick one of those.

`M105` provisioned folder missing

A folder the Provisioning section declares inside the domain doesn't exist; create it or correct the path. Folders outside the domain root aren't checked.

`M106` malformed provisioning bullet

A Provisioning bullet has no colon or no path, or declares a type twice. Write one `- type: path` line per type.

`M107` tag alias problem

A Tag Aliases bullet lacks the `old -> canonical` shape, maps a tag to itself, maps an alias a second time (the first wins), points at a target that isn't lowercase-with-hyphens or points at another alias. Correct the bullet; an alias resolves in one hop.

`M108` verify settings problem

The domain's `.crystalline.yaml` holds a severity word verify doesn't know (`E007: of`), or the file doesn't parse at all, so the override you meant isn't there. It's reported against `.crystalline.yaml` in `crystalline verify` and in `validate_engrams`. Fix the word or the YAML; the rule keeps its default until you do.

`M109` team marker missing

In a team repository the MANIFEST has no `team: github`, or spells it another way, so nothing stops a plain clone from switching Reflection on. Share once from an instance connected to the domain, and the share adds the line.

26.8 Reading a V code

Every sweep finding carries a family, a priority and a class. The number beside each code here is the rule's starting priority, what an unremarkable engram shows in brackets. The queue adds up to ten for salience, five for an engram three or more others link to and eight for one a person wrote, capped at 100, and `min_priority` (`--min-priority` in a terminal) drops what scores under a floor. The mechanical rules, which the agent completes without asking, are V005, V103, V110 and V102 when the nearest title is a near-exact match in the domain the link named. The rest are judgment, which the agent brings to you one at a time. Engrams with the status draft, proposed, idea or poc are exempt from V002, V003 and V006, and retired engrams are never called old, unverified or unlinked.

26.9 Sweep, temporal: V001 to V010

The temporal family watches dates and the retirement lifecycle, plus attachments and unshared work that have waited too long.

`V001` expired window still current (85)

The `valid_to` date is in the past and the status still says current. If the fact still holds, extend or remove `valid_to`; if it ended, retire the engram and wire the supersede pair when something replaced it.

`V002` review date passed (70)

The `stale_after` date has passed and nobody has verified the engram since. Re-check the claim: record a `verified` entry and a later `stale_after` when it holds, correct or supersede it when it doesn't.

`V003` old and never verified (25)

A current engram is over 180 days old with no verification and no `stale_after`. Confirm it and record a `verified` entry, or set a `stale_after` so a later sweep has a real date to compare.

`V004` superseded without a successor (65)

The status says superseded and no `superseded_by` relation resolves. Name the successor, or switch to deprecated, archived or legacy when nothing replaced it.

`V005` replaced but still current (90, mechanical)

Another engram declares that it supersedes this one, and this one still reads current. Set the status to superseded, add `superseded_by` and set `valid_to` when the end date is known.

`V006` human capture never reviewed (50)

A person captured this directly (its `generated.by` starts with `human:`) on an earlier day and nobody has recorded a verification. Verify the claim, align the tags, wire the relations both ways and record a `verified` entry. A person's capture is teaching, so any change to the wording is proposed to its author first (Chapter 7).

`V007` attachment never analyzed (50)

An engram shows an attachment and no engram claims it with `analyzes`. Read the attachment, capture what it teaches, and set `analyzes` and `analyzed_hash` on the engram that covers it.

`V008` attachment changed since analysis (60)

The attachment's current hash no longer matches the engram's `analyzed_hash`. Re-read it, bring the engram in line, update `analyzed_hash` and record a `verified` entry.

`V009` unshared work aging (40)

A team domain holds substantive local changes that have waited a week or more to be shared. The finding is about the domain rather than an engram, and regenerated folder listings never count. Share the changes after a yes (Chapter 17).

`V010` carry-forward gap (55)

A retired engram holds observations that appear in no live engram of its domain. Move the ones that still hold into their own engram with `split_engram`, or acknowledge the finding when they expired too. Matching is by text, so a fact carried forward in other words takes an acknowledgment as well. A retired engram that already declares `split_into` stays quiet.

26.10 Sweep, structure: V101 to V112

The structure family watches the graph: links, relation pairs, size and where files and attachments sit.

`V101` live link to retired knowledge (55)

A current engram links to a retired one. Repoint the link at the successor the evidence names, or keep it as a historical citation and acknowledge the finding.

`V102` unresolved link (50, mechanical or judgment)

A link or relation points at nothing. Fix the spelling (the fix line names the nearest title), add a domain prefix or capture the engram the link expects; the target is never created automatically.

`V103` one-sided relation pair (35, mechanical)

One half of `supersedes`/`superseded_by`, `summarizes`/`summarized_by` or `derived_from`/`split_into` is missing on the engram the finding names. Append the inverse there, as the fix line spells it out.

`V104` orphan (30)

Nothing links to the engram and it links to nothing, 14 days after it was recorded. The rule runs only in a domain where at least a quarter of the engrams are linked, and it skips the MANIFEST and schemas. Link it into the neighbourhood its tags suggest, or acknowledge that it stands alone on purpose.

`V105` oversized (60)

The body is over its token budget, the same size verify's Q002 flags. The fix line picks the split: a distilled summary with the full text moved to a `type: source` engram under `sources/`, sequential parts when the engram is already a source or a split by topic when the observations alone are over.

`V106` stub (45)

The body has fewer than three content lines, the shape verify's Q001 flags. Enrich it, fold it into the engram that owns the topic or retire it.

`V107` attachment reference to nothing (45)

The engram references an attachment path that holds no file. Restore the file (a Fluid upload or an archive import) or correct the reference, after asking, since either changes what the agent reads.

`V108` orphaned attachment (55)

A file under `assets/` is referenced or claimed by no engram. Delete it (`delete_engram` with the `assets/` path, or Fluid's Delete attachment button), or analyze it into an engram that claims it first. Deleting can't be undone, so the agent asks.

`V109` permalink off its folder (45)

The permalink names a different folder than the one the file sits in, usually after a move made by hand, so a `build_context` glob on the folder misses it. Files at the domain root never raise it. Repair it with `move_engram`, passing the file's current path as destination and `path` as permalink, which rewrites every reference; ask first, since the new address breaks bookmarks outside Crystalline. A deliberate custom permalink is acknowledged instead, which holds until the permalink or the path changes.

`V110` domain named by a local-only name (40, mechanical)

A `[[domain:Title]]` link names a domain by a name only this machine uses, a local name that differs from the declared one or a former name, so it reaches nothing on a colleague's machine. The fix line gives the exact `edit_engram` find-and-replace that swaps in the declared name, and `crystalline doctor --fix` rewrites the same links in one go. An acknowledgment covers one spelling, so a second local-only name on the same engram is its own finding.

`V111` ingested without a resource (40)

An ingestion record links to the engram, or the engram declares `ingested_from`, and the engram carries no `resource`, so an answer from it can't say where the knowledge came from. Set `resource` to the page or file it was distilled from (for a git source, the file at the ingested commit) and `source_date` where the source has one. A guessed location is a false citation, so the agent asks first; acknowledge the finding when no single location fits. Records, manifests, schemas and retired engrams never raise it.

`V112` sibling folders with close names (40)

Two folders side by side carry names that read as one topic: equal apart from case, _ and - (`plan_view`, `plan-view`), one name plus more words (`planview`, `planview-work`) or one or two letters apart (`note`, `notes`). A shared start alone (`project`, `projection`) and names that differ only in digits (`week-01`, `week-02`) aren't close. A `build_context` glob on either folder misses half the topic. Read both; when they hold one topic, move the engrams of the smaller folder into the other with `move_engram`, and when the topics differ, acknowledge the finding. There's one finding per pair. The acknowledgment can sit on any engram of the two folders and holds until one of them is renamed, and `remove V112` takes it back from every engram of the two folders you can write, naming any it had to leave.

26.11 Sweep, redundancy: V201 to V203

The redundancy family watches for knowledge written down twice, in the text, the titles or the tags.

`V201` near-duplicate content (80)

Several engrams carry nearly the same text. Merge into the richest one, then supersede or delete the others after repointing every inbound link. Detection compares wording, so a paraphrase can only show up as V301.

`V202` near-duplicate titles (55)

Two engrams in one domain carry titles so close that links and search can't tell them apart. Merge them or retitle one.

`V203` tag drift (30)

One concept is spelled several ways in the domain's tags, a finding about the domain rather than an engram. Run the `crystalline tags merge` command the fix line spells out; the agent hands it to you rather than rewriting tags itself.

26.12 Sweep, meaning: V301 and V302

The meaning family reads what two engrams say rather than how they say it. Both of its rules are judgment, and an acknowledgment answers for one pair, so an engram in two pairs keeps a separate entry for each.

`V301` semantic twins (75)

Two current engrams say close to the same thing in different words, measured by their embeddings, so the rule runs only where the instance computes them. If one owns the topic, merge into it and supersede the other; if they disagree on a fact, reconcile them; if they are distinct, link them and acknowledge the pair. A pair already in a V201 cluster shows only there.

`V302` possible contradiction (85)

A local model read an observation line of one engram and one of another as contradicting each other. The rule runs only with `evolve.contradictions` set to `full` (Chapter 23), and it is a question, never a verdict: the model can be wrong and can't see dates. Read both engrams, then supersede the one that is false going forward or correct its stale line; when both held at different times, the newer one supersedes the older, whose `valid_to` is set to the real date; or acknowledge the finding. There is one finding per pair of engrams, and an acknowledgment holds for exactly those two lines, so a new disagreement between the same engrams still shows. Engrams with the status draft, proposed, idea or poc, retired ones and ones past their `valid_to` are never scored.

26.13 Silencing and tuning

A sweep finding is silenced where it stands, in the engram, and a verify code is tuned for a whole domain in a file at its root. Neither mechanism reaches the other's codes.

Acknowledging a sweep finding

An acknowledgment is a note that a person looked at a finding and ruled it intentional. It lives in the engram's `evolve_ack` frontmatter, names the V code with an optional note and holds while the evidence it was given for holds; when the evidence changes, the finding comes back marked stale. The agent writes it with `edit_engram` and `set_frontmatter` on `evolve_ack`, the value being the code and the note (`V109 kept for the wiki's links`), and takes it back with `remove V109`. On Fluid's maintenance page anyone with write access gets an Acknowledge button with a note field on each row about an engram, Re-acknowledge once its evidence has moved and Unacknowledge on one already silenced. V009, V108 and V203 name no engram, so there is nowhere to put an acknowledgment and doing the work is the only way to clear them. How an acknowledgment differs from a verification is part of the evolving story (Chapter 10).

The domain's verify file

A `.crystalline.yaml` at a domain's root can carry a `verify` block, which tunes the checker for that one domain and leaves every other domain alone. The workspace file of the same name is a different one (Chapter 25):

```yaml
verify:
  rules:
    E007: "off"
    L003: info
    M102: error
  token_budget: 3000
  token_budgets:
    reference/api-catalogue.md: 8000
    changelog.md: 0
  required_files:
    - path: GLOSSARY.md
      require_frontmatter: true
      required_sections: [Terms]
      sections:
        Terms:
          min_top_level_bullets: 5
          max_bullet_length: 240
```

`rules` sets a code's severity to `off`, `error`, `warning` or `info` (`e`, `warn`, `w` and `i` work too). An override wins over `--strict`, so a rule you set to warning stays a warning in a strict run. An unknown word keeps the rule's default and a file that doesn't parse is ignored whole, and each raises M108: a warning that fails a run only under `--strict`. The rule a typo was meant for still takes `--strict` like any other. The overrides apply everywhere a verify code shows: `crystalline verify`, `validate_engrams`, Fluid's editor and MANIFEST edit receipts. In a team repository run the domain's own settings can't lower M010, M011, M012 or M109, so a pull request can't switch off the check that guards it, and `--strict` still promotes M109.

`token_budget` sets the budget for every file of the domain and `token_budgets` for single files, keyed by their path from the domain root. A budget of 0 turns Q002 and V105 off for that file, and a `type: source` engram still gets four times whatever applies. The budget is the one part of the file the sweep reads.

`required_files` puts the MANIFEST's structural checks on other files. A missing file raises M001, missing frontmatter M002 when `require_frontmatter` is set and a missing section M003. A section under its `min_top_level_bullets`

raises M004, or M101 when its `fallback_section` has enough; a bullet over `max_bullet_length` (180 by default) raises M102, and a doubled `##` heading M103.

An agent changes `rules` and `token_budget` with `configure` and the domain (Chapter 23), which writes the file without its comments; `token_budgets` and `required_files` stay hand edits.

With `crystalline verify --config` you load one such file for every domain in the run, in place of each domain's own. A `--config` file that doesn't load stops the run with a message and exit code 1.

MANIFEST edits answer at once

An edit of a MANIFEST, or a write that replaces one open in Fluid's editor, returns its M findings on the receipt, as `manifest_findings` with code, severity, message and line, and an error among them comes with the guidance that routing for the domain is broken. The edit lands anyway, and the findings come at the severities the domain's `verify` block sets (a virtual domain has none and gets the defaults), so the agent that just emptied When to Use can fix it in its next call. M001 is never on the receipt because the file exists. M105 isn't either: it needs a full verify to look at the folders.

26.14 Verify in CI

`crystalline verify` takes one or more domain roots and defaults to the current directory. It exits with 0 when no error was found, 1 when at least one was and 2 when a path doesn't exist, isn't a directory or can't be read. Warnings never fail a run unless `--strict` promotes them. `--format` chooses `human`, `json` (a stable report, version 1) or `github`, which writes annotations: errors and warnings as such, info as notices that never fail the job. Verify belongs on every proposal a team reviews (Chapter 8), and the bundled action does it in one step:

```
- uses: jordiboehme/crystalline/action@v0.30.0
  with:
    paths: knowledge/engineering knowledge/payments
    strict: 'false'
    version: v0.30.0
```

`paths` takes space-separated domain roots, `strict` promotes warnings, `version` pins the binary it downloads (`latest` by default) and `comment` (on by default) keeps one summary comment on the pull request up to date. The step's `exit-code` output carries verify's own exit code, and the step fails the job when it

isn't 0. In the repository of a team domain, set `team-repository: 'true'` to add the team repository rules (`crystalline verify --team-repository`), and run that job on the shared branch only, since `crystalline/reflections` holds the Reflection's pages on purpose.

In brief

Verify's E, T, L, Q, S and M codes say whether the files hold their format, and the sweep's V codes say what the knowledge needs, in four families, each finding with a priority and a class. Each entry above names what the code means and the action that clears it. A sweep finding a person ruled intentional is acknowledged in the engram with `evolve_ack` or on Fluid's maintenance page, except the three that name no engram. Verify codes are tuned per domain in the `verify` block of `.crystalline.yaml`: severities, token budgets and required files. In CI, `crystalline verify` exits 1 on any error and 2 on a bad path, and `--strict` makes warnings count.

READY.

Appendices

Glossary

These are the words the book uses without stopping, each as chapter 4 defined it. Terms in `code` are spelled the way they appear in a file or a command.

agent identity
: What an authenticated agent is on a shared instance: its user, for the length of the session. It sees, writes and shares as that person, and the provenance of what it captures names the account and the client that acted for it.

attachment
: A file a person adds to an engram in Fluid, stored under the domain's `assets/` folder and referenced from the engram's body. Agents read attachments back over MCP, and the sweep notices one nobody has analyzed.

`build_context`
: The tool that walks out from an address, following relations and links one to three hops, and returns the neighbourhood ranked by how strongly each engram connects to the anchor. A `/*` suffix anchors a whole folder.

contributor
: The higher of the two Reflection levels: a contributor reads and comments on a domain's reflections, edits their text, creates, moves and deletes reflections and adds files to them. It grants nothing on the engrams, which have a level of their own (Chapter 12).

crystalline:// address
: The one absolute form of an engram's address, `crystalline://<domain>/<permalink>`. Anything without the scheme is domain-relative and takes its domain from context.

curator
: The right everyone from engram editor up holds on a domain's Reflection, never set on its own: writing sections from the engrams, settling what people changed and editing the pact. Their agents curate with their rights, and Fluid shows them as curators.

daemon

The single process that holds the index open, watches the domain folders, embeds new engrams and answers every agent, terminal and browser tab. The first client to connect starts it; it stops only when told to.

declared name

The name a domain gives itself in its manifest, as `domain_name`. Links and `crystalline://` addresses carry it, so it has to be unique among the domains people use together and stable over time. A machine may register the domain under a local name of its own, and the declared name still reaches it there.

domain

A registered folder of knowledge with a `MANIFEST.md` at its root, or a virtual domain that lives in the database with no folder. The manifest's `When to Use` bullets are what route an agent to it.

engram

One markdown file holding one unit of the agent's own knowledge: YAML frontmatter, prose, observations and relations. The agent writes and organizes it for itself. The word comes from neuroscience, where it names the trace an experience leaves; here it's the trace a session leaves behind, in a form the next session can pick up.

evolve

The maintenance sweep, `crystalline evolve` or the `evolve_engrams` tool: a read-only, ranked queue of what the knowledge needs, in four families (temporal, structural, redundancy and meaning). Each finding is marked mechanical, for the agent to complete, or judgment, for a person to rule on.

evolve home

The one instance whose scheduled evolve runs work a team domain, named in the MANIFEST's `evolve_home`. Nobody sets it: the first scheduled run at an instance with a public address claims it, and every other copy of the domain refuses a scheduled run and names the home (Chapter 14).

Fluid

The web UI the daemon serves at `localhost:7411`, named for the fluid minds, human and model, that meet there on reflections, with a side door into the engrams for inspection. It has no store of its own; it shows the same files the agent reads.

frontmatter

The YAML block at the top of an engram holding its identity and state: `type`, `title`, `permalink`, `tags`, `status`, `recorded_at`, `generated` and the temporal fields when a fact is bounded.

GitHub App

A team server's own way into GitHub, created in Fluid or brought along, in place of one person's sign-in. It reads, pulls and shares the team domains, and on a server with one each team domain follows its repository: a person's role there decides what they may do with the domain (Chapter 19).

harness

The program that runs the model and hands it tools over MCP: Claude Code, the Codex CLI, the Copilot CLI, Cursor, Kiro, Gemini CLI, Qwen Code, Claude Desktop or any other MCP client. `crystalline install <harness>` wires one up; Claude Desktop takes an extension instead, which starts the Crystalline you installed. On a Mac that runs only Desktop, for someone who'd rather not open a terminal, the Desktop-only bundle carries Crystalline inside.

home instance

The one instance where a team domain's Reflection lives, the team server. Everyone works on the reflections there, laptops keep no copy and the server backs them up to their own branch of the domain's repository. The MANIFEST names it in `reflection_home`; the instance that runs the domain's scheduled evolve is its evolve home.

local name

The name one machine registered a domain under, which is the key in its config and the name in Fluid's addresses there. Usually it's the same as the declared name. The two differ when somebody picked another name on purpose, or when another domain here already holds the declared name, which leaves the domain shadowed. A name the domain had before a rename stays behind as an alias that still reaches it, unless another domain answers to that name.

`MANIFEST.md`

The first file in every domain, itself an engram of `type: manifest`. Its `Scope`, `When to Use` and `Notes for Agents` sections describe the domain and route tasks to it.

MCP
: The Model Context Protocol, the standard by which a harness hands tools to a model. Crystalline is an MCP server, and its tools are how an agent searches, reads, captures and refines knowledge.

loop
: A domain's reflect loop or evolve loop, run without a session: a runner checks on a schedule, or when the instance's webhook calls, whether anything is due, and only then starts an agent that works the reflect queue or the sweep's mechanical findings. It never waits for a yes, leaves what needs one for a person and delivers a team domain's changes as a pull request (Chapter 14). As an unattended run it reads only the domains everyone who reads its domain may read, and writes only into its domain.

mechanical and judgment findings
: The two classes of sweep finding. A mechanical finding completes intent the archive already records, so the agent does it in a pass; a judgment finding would change what the archive claims, so it goes to a person as a proposal.

membership level
: What an invited member of a private domain may do there: a viewer reads and searches, an editor also writes, a manager also runs the membership. A member at none sees no engrams and needs a Reflection level, or the membership is refused. Inside a private domain a write has to pass both the level and the instance role, and the lower of the two decides. On a domain under a server's GitHub App the role on the repository decides instead, and membership levels don't count.

nudge
: The once-per-session reminder the stop hook delivers, asking the agent to review the conversation for durable learnings and propose capturing each. It costs about a hundred and twenty tokens.

observation
: A top-level bullet of the form `- [category] text #tag` in an engram's body: the atomic unit of knowledge, a fact, decision, gotcha or lesson, indexed on its own line. A task item, `- [ ]`, `- [x]` or `- [~]`, is no observation.

OKF
: Google's Open Knowledge Format, version 0.2, the markdown-plus-front-matter format an engram extends. Unknown keys survive every write, so a document from any OKF tool drops in and comes back out intact.

overlay
: A per-actor private layer of drafts above a team domain in review mode, where the on-disk folder changes only by pull. A draft is visible to its author alone until it is shared, reviewed and merged.

owner
: The one person answerable for a private domain. Only the owner or an instance admin changes its visibility, transfers it or deletes it; a manager can do none of those.

pact
: A domain's `PACT.md`, at the root of its `reflections/` folder, in five parts: who the reflections are for, how much detail they carry, how they're presented, how the tree is structured and which pages people asked for. The MANIFEST tells the agent what the domain holds, the pact tells everyone how to talk about it. Curators edit it, and everybody else proposes a change in a thread.

permalink
: The half of an address that links point at, derived from the title, with the folder prefix in front when the engram lives in a subfolder. A move carries it to the new folder unless it was chosen by hand.

possible contradiction
: Two observation lines in two current engrams of one domain that a local model reads as saying opposite things, finding code `V302`. It's a question the agent brings to you, never a verdict, and the check is off until `evolve.contradictions` is set to `full`.

private domain
: A domain visible only to its owner, the instance's admins and invited members, and on a server with a GitHub App only to the admins and the repository's collaborators. To everyone else it reads as absent rather than forbidden.

prompt hook
: The hook `crystalline install` wires to run on every prompt: in Claude Code and Codex it puts up to three engrams that may fit the prompt in front of the agent, named by address and each once per session. Copilot gets the entry and stays quiet.

proposal

A unit of knowledge sent for review from a team domain with `share_changes`: a pull request in the team's repository that a person reviews and merges, and that the agent never merges itself. It is how a team domain shares by default; a domain whose MANIFEST says `sharing: direct` commits a share straight onto the branch instead. Outside review mode proposals stack, and one is amended in place by sharing again with its number.

provisioning

The mechanism by which a domain ships skills, commands, subagent definitions and MCP server configs into a harness. It is declared in the manifest's `Provisioning` section and installed only after a person allows it, per domain and per machine.

read-only domain

A domain whose MANIFEST says `read_only: on`. Every write into it through Crystalline is refused, whoever asks, while team pulls, the index and search keep working and the sweep skips it. Its owner or an instance admin switches it off in the domain policies, never an agent (Section 25.7).

reader

The lower of the two Reflection levels: a reader reads a domain's reflections and comments on them (opening, answering, resolving and reopening threads). A viewer of the engrams is a reader without anyone setting it.

Reflection

The place where people and the agent meet on a domain, with a mode of its own in Fluid. Switched on in the MANIFEST, it has the agent present what it knows as reflections, at the level the pact agrees on; people read, change and comment, and what holds up goes back into the engrams (Chapter 12).

reflection

One document in a domain's Reflection: a markdown file under the domain's `reflections/` folder, made of sections (a `##` heading and its text), each of which records the engrams it was built from.

Reflection collaborator

An account given the Reflection of one domain, as reader or contributor, without any other change to its access. The grant only raises and never opens an engram, it works for an account with the instance role none, and on a domain whose access follows GitHub it's the way in for somebody GitHub doesn't know (Chapter 19).

relation

A top-level bullet of the form `- rel_type [[Target]]`: a typed edge in the knowledge graph, shown back on the target's page as a backlink. `depends_on`, `supersedes` and `relates_to` are common types, and a split links its two halves with `derived_from` and `split_into`.

review mode

A private overlay per actor over a team domain's shared folder: every write lands in the writer's own overlay and the folder changes only with what lands on the team's branch. Set per domain in its registration as `review: overlay`. It needs a GitHub origin, so a virtual domain refuses it, and no tool turns it on: an agent does not decide how a team reviews.

routing prompt

The briefing an agent receives at session start: one line per domain from the manifests' `When to Use` bullets, plus the behaviour rules. `crystalline prompt system` prints it.

salience

A number from zero to ten in an engram's frontmatter that gives hybrid search a small, bounded lift for the rare engram everything else hangs from. Relevance keeps the upper hand and nothing is ever excluded by it.

share on purpose

A person's yes that lets a Reflection page carry text drawn from a domain fewer people may read than the page. Only a person gives it, never a run or an agent nobody signed in for, and it counts once per person, page and domain until the page's readers widen (Chapter 20).

share-link

A revocable link that opens one engram in one person's overlay to one colleague, who reads it and, after joining the draft, edits it in the normal editor where the domain's write right allows. It is the only way one person's drafts are ever visible to another.

source (of a domain)

Where a domain comes from on one machine: the machine itself or one server its Crystalline is connected to. Every domain has exactly one source, a write always goes there and a server's domains appear beside the local ones under one name each (Section 16.10). It isn't an engram's `sources` list, which names the places the engram was distilled from.

split

To move part of an engram into an engram of its own: by granularity, a distilled summary beside a source engram holding the full text, or by validity, when one fact expires while the rest holds. The new engram carries the source's tags and type, a `stable` status and no validity window, and the pair is linked with `derived_from` and `split_into`.

status

The frontmatter field that says whether the knowledge holds. `stable` is the default; `deprecated`, `superseded`, `archived` and `legacy` are the retirement set, which fades an engram in ranking without removing it. The field is required and free form.

supersede

To retire a fact that was true and stopped being: write the replacement as a new engram with `supersedes`, set the old one to `superseded` with a real `valid_to` and a `superseded_by`, and carry any lesson forward. A fact that was never quite true is corrected in place instead. When only part of the engram has expired, first split out the facts that still hold, then retire what is left.

tag alias

A fold recorded in the manifest's `Tag Aliases` section when `crystalline tags merge` unifies two spellings, so a filter on the old tag still finds what it always found.

team domain

A domain whose files also live in a GitHub repository, tracked through an origin, with knowledge travelling between members as proposals or as direct commits where the MANIFEST says so. It needs no git on any machine.

temporal validity

The window in which a fact holds, `valid_from` and `valid_to`, as plain dates. Absent `valid_from` means always valid, absent `valid_to` means valid forever. `stale_after` asks for a re-check by a date and `source_date` says when the source was written.

tombstone

The mark a deletion leaves in an actor's overlay, hiding the file for that actor until a merged share removes it for everyone.

unattended run

A session nobody is behind, such as a scheduled loop run. It reads only the domains everyone who reads its domain may read, writes only into that domain and is never asked a question (Chapter 14).

verification

The `verified` frontmatter entry, `{ by, at }` plus the model when an agent verified, that records a re-check finding a fact still true, one entry per verifier. The sweep stops asking about a verified engram on age alone.

virtual domain

A domain that lives in the database with no folder on disk, for a container without a writable volume. It carries the same manifest and answers to the same tools.

write list

The domains an account may write to, each named or given as a prefix with `*`. Every other domain is read-only for that account while it still reads what it can see, and a list only takes rights away (Chapter 16).

Questions and answers

These are the questions people ask in the first week, and a few that come up in the second, once somebody is onboarding agents for other people. Each answer points at the chapter that goes deeper.

Why not just a folder of markdown files?

It's one. Every engram is a markdown file in a folder you chose, readable in any editor, diffable in any repository and backed up with whatever you already use. What a folder can't do on its own is what Crystalline adds on top: routing, so an agent knows which domain owns a task without reading every file; search that blends the exact phrase with the meaning; a graph of relations and backlinks; and a temporal model, so a fact that stopped being true in March still carries its dates and is no longer mistaken for the present. Chapter 1 walks the ladder from one file of instructions to real indexes.

Why not a vector database or a retrieval pipeline?

Retrieval is the easy half. A vector index finds text that resembles a question. It doesn't know which domain owns the task, that a fact was superseded, who verified a claim or that something worth writing down just happened in this session. Crystalline treats similarity as one ranking signal inside a larger system of routing, provenance and time, on files you own, with no pipeline to operate. A pipeline also swallows whatever it's fed. Here knowledge is taught, and in a team it's reviewed by default before it reaches anyone else's agent, so what comes back is meant to make sense rather than merely to resemble the question. Chapters 9 and 15 cover the ranking, and chapter 17 the review.

Where does the name come from?

It comes from psychology. Crystallized intelligence is what learning deposits in a person; fluid intelligence is the reasoning applied to a problem never seen before. A model arrives with fluid intelligence in abundance and none of your crystallized kind, so every session starts as a brilliant stranger. Crystalline is the crystallized half, the store of what an agent has learned. Fluid, the web UI, is named for the fluid minds, yours and the model's, that meet there on reflections,

with a side door into the engrams. An engram is neuroscience's word for the trace an experience leaves. Chapter 2 is the whole story.

Is Crystalline a hosted service?

No. It's a binary that runs on a machine of yours, your laptop by default or a server your team operates, and the knowledge is a folder on that machine's disk. Nothing runs on anyone else's servers. GitHub, where a team uses it, holds a reviewed copy of a domain's files, the way it holds code; it serves nothing and sees no index. Chapter 1 has the two shapes and chapter 16 the shared one.

Does Crystalline phone home?

No. There's no telemetry and no update check, and Crystalline never sends your knowledge to a language model itself: your harness gets what Crystalline answers and passes it to the model provider you chose there, the engrams the prompt hook brings along included. One connection happens by default, the download of the embedding model from Hugging Face on the first start, and it sends no engram data. Point Crystalline at a mirror or at a model directory fetched ahead of time and it never talks to Hugging Face at all. Every other connection exists only once you turn it on: the contradiction check's model (from Hugging Face again), GitHub for a team domain, your single sign-on provider, a remote embedding service (which does get the text of your engrams) and a PostgreSQL server holding a shared index (which holds that text too). So is a Crystalline server you connect to with `crystalline connect` or `CRYSTALLINE_REMOTE_URL`: by design your laptop's Crystalline then works as a tunnel to it. A call for one of its domains goes there (a write carries the engram's text), and a search over all domains sends its text, your prompt too when the prompt hook searches. You chose that server, so its knowledge is handled and stored there, apart from a few small cached answers, and your own domains' engrams never go to it. docs/security.md lists each connection with what it sends; chapter 5 has the model download and chapter 16 the connected server.

Do I have to run a server?

No. The first agent that connects starts a background daemon on your machine, and every later agent, terminal and browser tab attaches to it. It stops only when told to, with `crystalline ctl shutdown`, Ctrl-C in a foreground run or the signal a container sends. It keeps running after the last agent disconnects, so the index stays warm for the next session. The one exception is the daemon of the Desktop-only bundle on a Mac, which exits five seconds after its last client. If a crash ever leaves a stale lock or socket behind, `crystalline doctor --fix`

clears it and a daemon that's alive but no longer answering is replaced by the next client that connects.

How do I start the daemon myself?

`crystalline serve` runs it in the foreground, where you can watch it work, and `crystalline serve --daemon` puts it in the background. It's the one process the machine shares: it serves Fluid at `http://localhost:7411`, answers MCP over HTTP on that same port and takes the agents and `crystalline` commands on the machine itself over a local socket. Starting it yourself is worth it when you want the browser open before any agent has connected. `crystalline ctl status` tells you whether one is already running, with its pid, its uptime and the sessions attached. On a server nobody types `serve` at all, because the systemd unit or the container starts the daemon at boot. Chapter 5 has the laptop and chapter 16 the server.

Do I need a terminal at all?

You need one only for the local install and the occasional maintenance command. Claude Desktop users can do without one: on Windows they install the MSI and then an extension from the releases page, and on a Mac where Desktop is all they use, the Desktop-only bundle is the most elegant way in, one extension with Crystalline inside. The agent creates its first domain itself, and Fluid, in the browser, is where a person works with the agent and opens its engrams when something needs correcting. Chapter 5 has both paths.

Is the MCP endpoint over HTTP protected?

On a laptop it listens on loopback, `127.0.0.1:7411`, and any process on that machine can reach it, which is the right trade for one person. `crystalline config set service.http false` turns the endpoint off. The daemon also checks the request's `Host` header against a loopback allow-list to block DNS rebinding, so a hostname or a proxy name has to be added first. Fluid and its API on the same port have accounts of their own, and nothing there is readable without a session.

On a shared instance turn on `auth.mcp`, and from then on every agent that connects over HTTP identifies itself with a token that belongs to one person, with no anonymous tier beside it. Chapter 16 sets it up and chapter 19 says what the token means.

How does a hosted client connect, with nowhere to paste a token?

It signs you in instead. On a shared instance whose agents identify themselves (`auth.mcp` on), a chat website or the app on your phone connects by sending

you to sign in through the instance, with the login the instance already has. From then on it holds a credential the same gate checks on every request, the way it checks a token. A harness that can send a header keeps working with a pasted token. Chapter 16 has the setup and chapter 19 the two credentials at one gate.

Where does my knowledge actually live?

In your domain folders, as plain markdown. By default those folders sit under `~/Documents/Crystalline`, one per domain, which is where an agent creates a domain when nobody names a path. A folder you registered yourself lives where you put it, like chapter 5's `~/knowledge/engineering`, and the `domains_root` setting moves the default. Everything Crystalline derives from it is disposable: the search index lives in the state directory and `crystalline reindex --full` rebuilds it from the files at any time. The config file, the index and the embedding model live in the platform's config, state and cache directories, `~/.config/crystalline`, `~/.local/state/crystalline` and `~/.cache/crystalline` on Linux and macOS.

Is my knowledge locked in?

No. An engram extends Google's Open Knowledge Format, OKF 0.2: markdown with YAML frontmatter, readable by any OKF tooling and by any text editor. Frontmatter keys Crystalline doesn't know are preserved through every write, and an `index.md` at each domain's root lets a folder navigate itself with nothing running. `crystalline import` brings an existing tree of frontmatter markdown in as a pure file transformation. If you stopped using Crystalline tomorrow, the folder would still be a readable knowledge base. Chapter 4 has the format.

What happens to wrong knowledge?

It depends on how it was wrong. A fact that was never quite true, a cutoff measured from the wrong event, is corrected where it stands. A fact that was true and stopped being is superseded: the replacement is written as its own engram with `supersedes` pointing back, the old one gets `status: superseded` and a real `valid_to`, and any lesson worth keeping moves forward. The past stays in every search, dated and gently below the present. The sweep finds the retirements nobody finished, the links that dangle and the tags spelled two ways, and hands each to the agent as a mechanical fix or to you as a judgment. Deleting is for mistakes, never for history. Chapter 3 has the lifecycle and chapter 10 the sweep.

What is GitHub, and what does Crystalline use it for?

GitHub is a website where teams keep files and every change ever made to them, above all source code. Its habit worth borrowing is the review: a change to something shared is proposed first, a colleague who knows the subject reads it, and only then does it land for everyone. Crystalline uses GitHub for exactly that, with knowledge in place of code. A team domain's markdown lives in a GitHub repository. By default a change goes out as a proposal (a pull request, in GitHub's own words) and reaches everyone's agent once somebody has merged it. A small team can switch a domain to sharing straight onto the branch instead. You don't need to learn GitHub's tools to take part: a one-time sign-in through a code in the browser connects your machine. After that the agent, Fluid or the command line does the talking. Chapter 17 walks through it. A team that doesn't use GitHub loses nothing else in this book; only sharing a domain between machines needs it.

Do I need git to share knowledge with a team?

No. A team domain talks to GitHub directly over its API: each person connects once with a browser code, and proposals open as pull requests that a person reviews and merges there, unless the domain shares straight onto the branch. There's no local clone and no SSH key, and nobody types `git`. Chapter 17 has the detail.

Can the agent see my private domain?

Only if its user can. On a shared instance an agent authenticates with a token that belongs to one person and acts as that person for the length of the session: it sees what they see, writes where they may write and captures under their name. A private domain shows to its owner, to the instance's admins and to the members the owner invited, and to everyone else it reads as absent, on every surface. On a team server with its own GitHub App, a team domain follows its repository instead, and the person's role on GitHub decides who reads and writes it. There's no tool for changing a membership, so an agent cannot talk itself in. An agent can do two things to a domain itself. It can unregister it with `remove_domain`, which is open to an instance admin or a private domain's owner and refused outright on the open tier. And it can change the domain's policies with `configure`, under the same owner-or-admin rule, though the open tier may do that wherever it may write. On a server with a GitHub App the open tier acts on the team domains as the account `github.agent_identity` names, private ones included, so turn `auth.mcp` on there. With `auth.mcp` on, `add_domain`, a `configure` that changes the instance and a `provision` that decides need the admin role behind the agent as well. Before you rely on it, know the limits: admins see

every domain, and a private domain protects from the instance's other users, never from whoever operates the machine. Chapters 16 and 19 have the rest.

What does the onboarding cost in tokens?

It costs little, and the same on the tenth session as on the thousandth. The routing block is one line per domain and a handful of rules. The reminder at the end of a session costs about a hundred and twenty tokens and fires once. Recall loads only the engrams that matter, routed by domain and narrowed by the indexes, instead of a folder read whole every session. Chapter 6 has the briefing and chapter 9 the recall.

AGENTS.md or CLAUDE.md?

Use the name your harnesses read. An instruction file is a project's standing orders: people write it, and the harness reads all of it at the start of every session. Crystalline holds what the agent learned and brings back the piece a task needs, so the two sit side by side and Crystalline never writes into either. As for the name, `AGENTS.md` is the one most harnesses share: Codex, the Copilot CLI, Cursor, Kiro and Qwen Code read it, and Gemini CLI does once its settings name it. Claude Code reads it when the project has no `CLAUDE.md` of its own (a `CLAUDE.local.md` counts too), and its `/config` can make it read both. So keep one `AGENTS.md`, and if some lines are for Claude alone, put them in a `CLAUDE.md` that pulls in the rest with `@AGENTS.md`. Chapter 5 lists the files the install does write.

What should I teach first?

Whatever you'd tell a new hire in the first week, and the test is that sentence. Think of the rule behind the strange number, the nickname, the word the wiki gets wrong, the day nobody deploys. Teach where information lives rather than the information itself: which repository holds what, how access is granted, the architecture, so the agent can reach the territory with the tools its harness already gives it. Chapter 7 has the list, and chapter 8 says what to do when the first week meets the wiki.

One big domain or many small ones?

For search it makes no difference. An agent searches every domain its instance has registered unless it narrows the search to a few, so splitting doesn't make the results better and merging doesn't make them worse. Long material doesn't crowd anyone out either: a meeting transcript is kept as a source engram beside a short summary, and the summary is what recall reads. What decides the split is who looks after the knowledge. Chapter 8 has the summary and its source, and chapter 9 the search.

When does a new domain earn its place?

When the knowledge needs a different owner. Ask who should approve a change to it. When that's somebody other than the people who own the domain it would go into, start a new one. It stays where it is when:

- The knowledge is about a product or a system that already has a domain. Its architecture, the decisions that affect other teams and how it works belong there, even when your team wrote them.
- Your team is the only one working on the product. Then the product domain's owners and your team are the same people, so everything goes into the product domain: the team's own notes, open questions, transcripts and reflections. A second domain only gives everyone one more place to look.

Several teams work on one product. How many domains is that?

Usually one for the product and one per team. The product domain is shared by every team, reviewed and slow to change: it holds the architecture, the decisions across teams and how the product behaves. A team domain belongs to its team and holds the fast-moving rest (grooming notes, open questions, drafts, transcripts, how the team is organised), and it links into the product domain instead of copying from it. A team that writes a lot sets the review rules that fit its pace in its own domain and leaves the shared domain's review as it is. If you started as the only team and a second one joins, keep the product domain for what both teams share. Move your own topics into a new team domain once the other team shouldn't be the one approving changes to them. Chapter 17 has the review and chapter 25 the domain settings.

How do domains point at each other?

With a `crystalline://<domain>/<permalink>` link. It resolves across domains, so a team engram can point at the decision it relies on in the product domain. When something a team wrote turns out to matter to everyone, move it into the product domain. The links to it follow, and in a team domain the move goes out as a proposal like any other change. Chapter 18 says why the domain's name in that link has to stay stable, and chapter 25 has the move.

Is there anything hidden in Crystalline?

There is. The crystal in Fluid's top corner takes you home, like every logo does. Click it the way Dorothy clicks her heels, and it takes you somewhere else.

List of figures

Figure 1.1 Two sides of equal standing, each with knowledge of its own: the agent learns through onboarding and its own experience, people through education and experience; they work together in between and meet in Reflection above them. 24

Figure 2.1 Fluid intelligence reasons in the moment and keeps nothing; crystalline intelligence accumulates and keeps. The web UI takes its name from the fluid side: people and the agent meet in it on reflections, and a side door opens on the engrams for inspection. 33

Figure 3.1 The lifecycle: learn, evolve, remember, turning session by session. 35

Figure 3.2 The maintenance queue in Fluid, narrowed to the Meaning family: two lines that may contradict each other. 38

Figure 4.1 The anatomy of an engram: frontmatter, prose, an observation and a relation. 45

Figure 5.1 First visit: creating the admin account in the browser. 56

Figure 5.2 The routing prompt an agent receives at session start. 58

Figure 6.1 The same block unabridged, as `crystalline prompt system` prints it. 68

Figure 7.1 Editing an engram in Fluid: live preview, the frontmatter form and wikilink completion. 76

Figure 7.2 Your agent in the page with you: its glyph in the strip beside yours, writing while you write. 81

Figure 12.1 Present, test, improve: the agent presents what it knows, people change and comment, and what holds up in a thread goes back into its knowledge. 112

Figure 12.2 Two lifecycles around one body of knowledge: evolve keeps it true, reflect keeps what people see in step with it. 117

Figure 13.1 The six callout kinds on a sample page, the expand block folded at the bottom. 123

Figure 13.2 Status labels as the verdict column of a table. 124

Figure 13.3 A deploy checklist, one item done and one that doesn't apply. 125

Figure 13.4 The payment flow as a sequence diagram on the page. 126

Figure 13.5 A code block with its Copy button, and a colour swatch in the sentence below it. 127

Figure 13.6 The Insert menu with its Callout submenu open. 129

Figure 13.7 The callout toolbar with the caret in the caution: its kind and the three fold buttons. 130

Figure 13.8 An info panel and a status lozenge pasted from a wiki, now a callout and a status label. 131

Figure 13.9 The first printed page of the deploy reflection, with the title block and its QR code. 132

Figure 15.1 One binary: the CLI, the MCP server, Fluid and the search index, all standing on plain markdown files. Agents come in through MCP, people through Fluid. 145

Figure 15.2 Faceted search across the instance, backed by the ranking the agents use. 149

Figure 15.3 The neighbourhood graph of an engram in Fluid. 150

Figure 15.4 An engram in Fluid: details rail, observations and typed relations. 151

Figure 16.1 One shared instance: agents connect over MCP with a token, people through Fluid with an account, with a proxy in front and an identity provider beside it where the team configures them. 157

Figure 16.2 Signing in through the instance's identity provider. 161

Figure 16.3 A local Crystalline connected to the team server: the harness and the hooks keep talking to it as before, and it reaches the server over one signed-in connection. 169

Figure 17.1 A team domain in Fluid: the sync status, the open proposal with its review link and the share button. 177

Figure 17.2 Before sharing: the team's copy against yours, one file at a time. 177

Figure 17.3 One shared instance in review mode: each actor drafts in a private overlay above the domain folder, a draft goes out as a pull request, and in the default flow only a merged pull changes the folder. 181

Figure 17.4 Live co-editing: two cursors in one engram. 183

Figure 18.1 Renaming a domain in Fluid: the new name and the This machine only box. 190

Figure 19.1 The domains of an instance in Fluid, private ones marked. 196

Figure 20.1 The readers of a page compared with the readers of the domain a section draws on. The page may carry the text when its readers fit inside. 202

Figure 20.2 A thread question meets a scheduled run that can't see the private domain, so the answer comes from the run's own domain only. 205

Figure 20.3 What happens when a write names another domain to draw on. 207

Figure 20.4 A section whose readers grew beyond the domain it draws on is held back from the backup until a person decides. 209

Figure 21.1 The provisioning decision of every domain that ships tools and the installed state of every harness, with the other domains folded into one line. 214

End of line.

Every session, your agent starts as a stranger.

Yesterday's decisions forgotten, your conventions unknown, every lesson re-derived or re-explained. Crystalline is the other half: the crystallized intelligence an agent accumulates and keeps, in plain markdown you can read and diff. This handbook is how you onboard one, teach it well and keep what it knows honest.

```
Moving day
  You:    Here's the washing machine's manual as a PDF. Learn it - I am
          never reading 60 pages about laundry.
  Agent:  Learned it. Captured programs, error codes and maintenance
          into home (#appliances), manual attached.

Eight months later, a fresh session
  You:    The machine is blinking E18 and the display is in Italian??
  Agent:  Recalled from home: E18 is a blocked drain pump filter.
          Front panel, bottom right, quarter turn - towel down first,
          about a liter of water comes out.
```

* Teach it once, keep it forever
* Knowledge that evolves instead of rotting
* Every session starts ahead of the last

https://github.com/jordiboehme/crystalline/releases/latest

READY.